CHAPTER 16
Dynamic SQL and Dynamic PL/SQL

Dynamic SQL refers to SQL statements that are constructed and executed at runtime. Dynamic is the opposite of static. *Static SQL* refers to SQL statements that are fully specified, or fixed, at the time the code containing those statements is compiled. *Dynamic PL/SQL* refers to entire PL/SQL blocks of code that are constructed dynamically, then compiled and executed.

Time for a confession: I have had more fun writing dynamic SQL and dynamic PL/SQL programs than just about anything else I have ever done with the PL/SQL language. By constructing and executing dynamically, you gain a tremendous amount of flexibility. You can also build extremely generic and widely useful reusable code. That said, you should only use dynamic SQL when it is needed; static SQL is always preferred, since when you make your code dynamic, it is more complicated, harder to debug and test, possibly slower, and more difficult to maintain.

So what can you do with dynamic SQL and dynamic PL/SQL?[1] Here are just a few ideas:

Execute DDL statements
: You can only execute queries and DML statements with static SQL inside PL/SQL. What if you want to create a table or drop an index? Time for dynamic SQL!

Support ad hoc query and update requirements of web-based applications
: A common requirement of Internet applications is that users be able to specify which columns they want to see and vary the order in which they see the data (of course, users don't necessarily realize they are doing so).

1. For the remainder of this chapter, any reference to *dynamic SQL* also includes dynamic PL/SQL blocks, unless otherwise stated.

Softcode business rules and formulas
: Rather than hardcoding business rules and formulas into your code, you can place that logic in tables. At runtime, you can generate and then execute the PL/SQL code needed to apply the rules.

Ever since Oracle7 Database, we PL/SQL developers have been able to use the built-in DBMS_SQL package to execute dynamic SQL. In Oracle8*i* Database, we were given a second option for executing dynamically constructed SQL statements: *native dynamic SQL* (NDS). NDS is a *native* part of the PL/SQL language; it is much easier to use than DBMS_SQL, and for many applications it will execute more efficiently. There are still requirements for which DBMS_SQL is a better fit; they are described at the end of this chapter. For almost every situation you face, however, NDS will be the preferred implementation approach.

NDS Statements

One of the nicest things about NDS is its simplicity. Unlike DBMS_SQL, which has dozens of programs and lots of rules to follow, NDS has been integrated into the PL/SQL language through the addition of one new statement, EXECUTE IMMEDIATE (which executes a specified SQL statement immediately), and through the enhancement of the existing OPEN FOR statement, allowing you to perform multiple-row dynamic queries.

The EXECUTE IMMEDIATE and OPEN FOR statements will not be directly accessible from Oracle Forms Builder and Oracle Reports Builder until the PL/SQL version in those tools is upgraded to at least Oracle8*i* Database. For earlier versions, you will need to create stored programs that hide calls to these constructs; you will then be able to execute those stored programs from within your client-side PL/SQL code.

The EXECUTE IMMEDIATE Statement

Use EXECUTE IMMEDIATE to execute (immediately!) the specified SQL statement. Here is the syntax of this statement:

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

where:

SQL_string
: Is a string expression containing the SQL statement or PL/SQL block.

define_variable

Is a variable that receives a column value returned by a query.

record

Is a record based on a user-defined TYPE or %ROWTYPE that receives an entire row returned by a query.

bind_argument

Is an expression whose value is passed to the specified SQL statement or PL/SQL block, or an identifier that serves as an input and/or output variable to the function or procedure that is called in the PL/SQL block.

INTO clause

Is used for single-row queries. For each column value returned by the query, you must supply an individual variable or field in a record of a compatible type. If you preface INTO with BULK COLLECT, you can fetch multiple rows into one or more collections.

USING clause

Allows you to supply bind arguments for the SQL string. This clause is used for both dynamic SQL and PL/SQL, which is why you can specify a parameter mode. This mode is relevant only for PL/SQL and with the RETURNING clause. The default mode for a bind variable is IN, which is the only kind of bind argument you would have for SQL statements.

You can use EXECUTE IMMEDIATE for any SQL statement or PL/SQL block. The string may contain placeholders for bind arguments, but you cannot use bind values to pass in the names of schema objects, such as table names or column names.

When you execute a DDL statement in your program, you will also perform a commit. If you don't want the DDL-driven commit to affect outstanding changes in the rest of your application, place the dynamic DDL statement within an autonomous transaction procedure. See the *auton_ddl.sql* file on the book's website for a demonstration of this technique.

When the statement is executed, the runtime engine replaces each placeholder (an identifier with a colon in front of it, such as :salary_value or even :1) in the SQL string with its corresponding bind argument in the USING clause. Note that you cannot pass a NULL literal value. Instead, you must pass an expression that evaluates to a value of the correct type, which might also happen to have a value of NULL.

NDS supports all SQL datatypes. You can bind scalar values like strings, numbers, and dates, but you can also bind schema-level collections, LOBs, instances of an object type, XML documents, REFs, and more. The INTO clause can contain a PL/SQL record whose number and types of fields match the values fetched by the dynamic query.

Beginning with Oracle Database 12*c*, the USING clause can also bind many datatypes known only to PL/SQL: Booleans, associative arrays with PLS_INTEGER indexes, and composite types (records, collections) declared in a package spec.

Let's take a look at a few examples:

- Create an index:

  ```
  BEGIN
     EXECUTE IMMEDIATE 'CREATE INDEX emp_u_1 ON employees (last_name)';
  END;
  ```

  It can't get much easier than that, can it?

- Create a stored procedure that will execute any DDL statement:

  ```
  PROCEDURE exec_DDL (ddl_string IN VARCHAR2)
  IS
  BEGIN
     EXECUTE IMMEDIATE ddl_string;
  END;
  ```

  With exec_ddl in place, I can create that same index as follows:

  ```
  BEGIN
     exec_DDL ('CREATE INDEX emp_u_1 ON employees (last_name)');
  END;
  ```

- Obtain the count of rows in any table for the specified WHERE clause:

  ```
  /* File on web: tabcount_nds.sf */
  FUNCTION tabcount (table_in IN VARCHAR2)
     RETURN PLS_INTEGER
  IS
     l_query  VARCHAR2 (32767) := 'SELECT COUNT(*) FROM ' || table_in;
     l_return PLS_INTEGER;
  BEGIN
     EXECUTE IMMEDIATE l_query INTO l_return;
     RETURN l_return;
  END;
  ```

  So now I never again have to write SELECT COUNT(*), whether in SQL*Plus or within a PL/SQL program. Instead, I can do the following:

  ```
  BEGIN
     IF tabCount ('employees') > 100
     THEN
        DBMS_OUTPUT.PUT_LINE ('We are growing fast!');
     END IF;
  END;
  ```

- Update the value of a numeric column and return the number of rows that have been updated:

  ```
  /* File on web: updnval.sf */
  FUNCTION updNVal (
     col IN VARCHAR2, val IN NUMBER,
     start_in IN DATE, end_in IN DATE)
     RETURN PLS_INTEGER
  IS
  BEGIN
     EXECUTE IMMEDIATE
        'UPDATE employees SET ' || col || ' = :the_value
          WHERE hire_date BETWEEN :lo AND :hi'
       USING val, start_in, end_in;
     RETURN SQL%ROWCOUNT;
  END;
  ```

  That is a very small amount of code to achieve all that flexibility! This example introduces the bind clause (USING): the PL/SQL engine passes the values in the USING clause to the SQL engine. After parsing the statement, the SQL engine uses the values provided in place of the placeholders (:the_value, :lo, and :hi). Notice also that I can use the SQL%ROWCOUNT cursor attribute to determine the number of rows modified by a dynamic SQL statement, as well as with static statements.

- Run a different block of code at the same time on successive days. Each program's name has the structure DAYNAME_set_schedule. Each procedure has the same four arguments: you pass in employee_id and hour for the first meeting of the day, and it returns the name of the employee and the number of appointments for the day. I can use dynamic PL/SQL to handle this situation:

  ```
  /* File on web: run9am.sp */
  PROCEDURE run_9am_procedure (
     id_in IN employee.employee_id%TYPE,
     hour_in IN INTEGER)
  IS
     v_apptCount INTEGER;
     v_name VARCHAR2(100);
  BEGIN
     EXECUTE IMMEDIATE
        'BEGIN ' || TO_CHAR (SYSDATE, 'DAY') ||
           '_set_schedule (:id, :hour, :name, :appts); END;'
       USING IN
         id_in, IN hour_in, OUT v_name, OUT v_apptCount;

     DBMS_OUTPUT.PUT_LINE (
        'Employee ' || v_name || ' has ' || v_apptCount ||
        ' appointments on ' || TO_CHAR (SYSDATE));
  END;
  ```

- Bind a PL/SQL-specific Boolean with EXECUTE IMMEDIATE (new in 12*c*):

```
/* File on web: 12c_bind_boolean.sql */
CREATE OR REPLACE PACKAGE restaurant_pkg
AS
   TYPE item_list_t
      IS TABLE OF VARCHAR2 (30);

   PROCEDURE eat_that (
      items_in              IN item_list_t,
      make_it_spicy_in_in   IN BOOLEAN);
END;
/

CREATE OR REPLACE PACKAGE BODY restaurant_pkg
AS
   PROCEDURE eat_that (
      items_in              IN item_list_t,
      make_it_spicy_in_in   IN BOOLEAN)
   IS
   BEGIN
      FOR indx IN 1 .. items_in.COUNT
      LOOP
         DBMS_OUTPUT.put_line (
               CASE
                  WHEN make_it_spicy_in_in
                  THEN
                     'Spicy '
               END
            || items_in (indx));
      END LOOP;
   END;
END;
/

DECLARE
   things   restaurant_pkg.item_list_t
      := restaurant_pkg.item_list_t (
            'steak',
            'quiche',
            'eggplant');
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN restaurant_pkg.eat_that(:l, :s); END;'
      USING things, TRUE;
END;
/
```

As you can see, EXECUTE IMMEDIATE makes it very easy to execute dynamic SQL statements *and* PL/SQL blocks, with a minimum of syntactic fuss.

The OPEN FOR Statement

The OPEN FOR statement was actually *not* introduced into PL/SQL for NDS; it was first offered in Oracle7 Database to support cursor variables. Now it is deployed in an especially elegant fashion to implement multiple-row dynamic queries. With DBMS_SQL, you go through a painful series of steps to implement multirow queries: parse, bind, define each column individually, execute, fetch, and extract each column value individually. That's a lot of code to write!

For native dynamic SQL, Oracle took an existing feature and syntax—that of cursor variables—and extended it in a very natural way to support dynamic SQL. The next section explores multirow queries in detail. Let's now look at the syntax of the OPEN FOR statement:

```
OPEN {cursor_variable | :host_cursor_variable} FOR SQL_string
   [USING bind_argument[, bind_argument]...];
```

where:

cursor_variable
: Is a weakly typed cursor variable

:host_cursor_variable
: Is a cursor variable declared in a PL/SQL host environment such as an Oracle Call Interface (OCI) program

SQL_string
: Contains the SELECT statement to be executed dynamically

USING clause
: Follows the same rules as in the EXECUTE IMMEDIATE statement

If you are not familiar with cursor variables, you might want to review Chapter 15. Here you will learn how to use cursor variables with NDS.

Following is an example that demonstrates the declaration of a weak REF CURSOR type and a cursor variable based on that type, and the opening of a dynamic query using the OPEN FOR statement:

```
PROCEDURE show_parts_inventory (
   parts_table IN VARCHAR2,
   where_in IN VARCHAR2)
IS
   TYPE query_curtype IS REF CURSOR;
   dyncur query_curtype;
BEGIN
   OPEN dyncur FOR
      'SELECT * FROM ' || parts_table ||
      ' WHERE ' || where_in;
   ...
```

Once you have opened the query with the OPEN FOR statement, the syntax rules used to fetch rows, close the cursor variable, and check the attributes of the cursor are all the same as for static cursor variables and hardcoded explicit cursors.

Let's now take a closer look at the OPEN FOR statement. When you execute an OPEN FOR statement, the PL/SQL runtime engine does the following:

1. Associates a cursor variable with the query found in the query string
2. Evaluates any bind arguments and substitutes those values for the placeholders found in the query string
3. Executes the query
4. Identifies the result set
5. Positions the cursor on the first row in the result set
6. Zeros out the rows-processed count returned by %ROWCOUNT

Note that any bind arguments (provided in the USING clause) in the query are evaluated only when the cursor variable is opened. This means that if you want to use a different set of bind arguments for the same dynamic query, you must issue a new OPEN FOR statement with those arguments.

To perform a multirow query, you follow these steps:

1. Declare a REF CURSOR type (or use the Oracle-defined SYS_REFCURSOR weak REF CURSOR type).
2. Declare a cursor variable based on the REF CURSOR.
3. OPEN the cursor variable FOR your query string.
4. Use the FETCH statement to retrieve one or more rows from the result set identified by the query.
5. Check cursor attributes (%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN) as necessary.
6. Close the cursor variable using the normal CLOSE statement. Generally, if and when you are done with your cursor variable, you should close it explicitly.

Here is a simple program to display the specified column of any table for the rows indicated by the WHERE clause (it will work for number, date, and string columns):

```
/* File on web: showcol.sp */
PROCEDURE showcol (
   tab IN VARCHAR2,
   col IN VARCHAR2,
   whr IN VARCHAR2 := NULL)
IS
   cv SYS_REFCURSOR;
   val VARCHAR2(32767);
BEGIN
   OPEN cv FOR
      'SELECT ' || col ||
      '  FROM ' || tab ||
      ' WHERE ' || NVL (whr, '1 = 1');

   LOOP
      /* Fetch and exit if done; same as with explicit cursors. */
      FETCH cv INTO val;
      EXIT WHEN cv%NOTFOUND;

      /* If on first row, display header info. */
      IF cv%ROWCOUNT = 1
      THEN
         DBMS_OUTPUT.PUT_LINE (RPAD ('-', 60, '-'));
         DBMS_OUTPUT.PUT_LINE (
            'Contents of ' || UPPER (tab) || '.' || UPPER (col));
         DBMS_OUTPUT.PUT_LINE (RPAD ('-', 60, '-'));
      END IF;

      DBMS_OUTPUT.PUT_LINE (val);
   END LOOP;

   /* Don't forget to clean up! Very important... */
   CLOSE cv;
END;
```

Here are some examples of output from this procedure:

```
SQL> EXEC showcol ('emp', 'ename', 'deptno=10')
--------------------------------------------------
Contents of EMP.ENAME
--------------------------------------------------
CLARK
KING
MILLER
```

I can even combine columns:

```
BEGIN
   showcol (
      'emp',
      'ename || ''-$'' || sal',
      'comm IS NOT NULL');END;/
--------------------------------------------------
Contents of EMP.ENAME || '-$' || SAL
--------------------------------------------------
ALLEN-$1600
WARD-$1250
MARTIN-$1250
TURNER-$1500
```

FETCH into variables or records

The FETCH statement in the showcol procedure shown in the previous section fetches into an individual variable. You could also FETCH into a sequence of variables, as shown here:

```
PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
   cv SYS_REFCURSOR;
   mega_bucks company.ceo_compensation%TYPE;
   achieved_by company.cost_cutting%TYPE;
BEGIN
   OPEN cv FOR
      'SELECT ceo_compensation, cost_cutting
        FROM ' || company_table_name (company_id_in);

   LOOP
      FETCH cv INTO mega_bucks, achieved_by;
      ...
   END LOOP;

   CLOSE cv;
END;
```

Working with a long list of variables in the FETCH list can be cumbersome and inflexible; you have to declare the variables, keep that set of values synchronized with the FETCH statement, and so on. To ease our troubles, NDS allows us to fetch into a record, as shown here:

```
PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
   cv SYS_REFCURSOR;
   ceo_info company%ROWTYPE;
BEGIN
   OPEN cv FOR
      'SELECT * FROM ' || company_table_name (company_id_in);

   LOOP
      FETCH cv INTO ceo_info;
      ...
   END LOOP;

   CLOSE cv;
END;
```

Of course, in many situations you will not want to do a SELECT *; this statement can be very inefficient if your table has hundreds of columns and you need to work with only a few. A better approach is to create record type that correspond to different requirements. The best place to put these structures is in a package specification, so that they can be used throughout your application. Here's one such package:

```
PACKAGE company_pkg
IS
   TYPE ceo_info_rt IS RECORD (
      mega_bucks company.ceo_compensation%TYPE,
      achieved_by company.cost_cutting%TYPE);

END company_pkg;
```

With this package in place, I can rewrite my CEO-related code as follows:

```
PROCEDURE mega_bucks (company_id_in IN INTEGER)
IS
   cv SYS_REFCURSOR;
   rec company_pkg.ceo_info_rt;
BEGIN
   OPEN cv FOR
      'SELECT ceo_compensation, cost_cutting FROM ' ||
      company_table_name (company_id_in);

   LOOP
      FETCH cv INTO rec;
      ...
   END LOOP;

   CLOSE cv;
END;
```

The USING clause in OPEN FOR

As with the EXECUTE IMMEDIATE statement, you can pass in bind arguments when you open a cursor. You can provide only IN arguments for a query. By using bind arguments, you can also improve the performance of your SQL and make it easier to write and maintain that code. In addition, you can potentially dramatically reduce the number of distinct parsed statements that are cached in the SGA, and thereby increase the likelihood that your preparsed statement will still be in the SGA the next time you need it. (See the section "Binding Variables" on page 550 for information about this technique.)

Let's revisit the showcol procedure. That procedure accepted a completely generic WHERE clause. Suppose that I have a more specialized requirement: I want to display (or in some way process) all column information for rows that contain a date column with a value within a certain range. In other words, I want to be able to support this query:

```
SELECT last_name
  FROM employees
 WHERE hire_date BETWEEN x AND y;
```

as well as this query:

```
SELECT flavor
  FROM favorites
 WHERE preference_period BETWEEN x AND y;
```

I also want to make sure that the time component of the date column does not play a role in the WHERE condition.

Here is the header for the procedure:

```
/* File on web: showdtcol.sp */
PROCEDURE showcol (
   tab IN VARCHAR2,
   col IN VARCHAR2,
   dtcol IN VARCHAR2,
   dt1 IN DATE,
   dt2 IN DATE := NULL)
```

The OPEN FOR statement now contains two placeholders and a USING clause to match:

```
OPEN cv FOR
      'SELECT ' || col ||
      '  FROM ' || tab ||
      ' WHERE ' || dtcol ||
         ' BETWEEN TRUNC (:startdt)
               AND TRUNC (:enddt)'
   USING dt1, NVL (dt2, dt1+1);
```

I have crafted this statement so that if the user does not supply an end date, the WHERE clause returns rows with date column values that match the dt1 provided. The rest of the showcol procedure remains the same, except for some cosmetic changes in the display of the header.

The following call to this new version of showcol asks to see the names of all employees hired in 1982:

```
BEGIN
   showcol ('emp', 'ename', 'hiredate',
      DATE '1982-01-01', DATE '1982-12-31');
END;
```

The output is:

```
----------------------------------------------------------------------
Contents of EMP.ENAME for HIREDATE between 01-JAN-82 and 31-DEC-82
----------------------------------------------------------------------
MILLER
```

About the Four Dynamic SQL Methods

Now that you've been introduced to the two basic statements used to implement native dynamic SQL in PL/SQL, it's time to take a step back and review the four distinct types, or methods, of dynamic SQL and the NDS statements you will need to implement those methods. The methods and the corresponding NDS statements are listed in Table 16-1.

Table 16-1. The four methods of dynamic SQL

Type	Description	NDS statements used
Method 1	No queries; just DDL statements and UPDATEs, INSERTs, MERGEs, or DELETEs, which have no bind variables	EXECUTE IMMEDIATE without USING and INTO clauses
Method 2	No queries; just UPDATEs, INSERTs, MERGEs, or DELETEs, with a fixed number of bind variables	EXECUTE IMMEDIATE with a USING clause
Method 3 (single row queried)	Queries (SELECT statements) with a fixed number of columns and bind variables, retrieving a single row of data	EXECUTE IMMEDIATE with USING and INTO clauses
Method 3 (multiple rows queried)	Queries (SELECT statements) with a fixed number of columns and bind variables, retrieving one or more rows of data	EXECUTE IMMEDIATE with USING and BULK COLLECT INTO clauses or OPEN FOR with dynamic string
Method 4	A statement in which the number of columns selected (for a query) or the number of bind variables set is not known until runtime	For method 4, you will use the DBMS_SQL package

Method 1

The following DDL statement is an example of method 1 dynamic SQL:

```
EXECUTE IMMEDIATE 'CREATE INDEX emp_ind_1 on employees (salary, hire_date)';
```

This UPDATE statement is also method 1 dynamic SQL, because its only variation is in the table name—there are no bind variables:

```
EXECUTE IMMEDIATE
   'UPDATE ' || l_table || ' SET salary = 10000 WHERE employee_id = 1506'
```

Method 2

If I replace both of the hardcoded values in the previous DML statement with placeholders (a colon followed by an identifier), I then have method 2 dynamic SQL:

```
EXECUTE IMMEDIATE
   'UPDATE ' || l_table || '
       SET salary = :salary WHERE employee_id = :employee_id'
   USING 10000, 1506;
```

The USING clause contains the values that will be *bound* into the SQL string after parsing and before execution.

Method 3

A method 3 dynamic SQL statement is a query with a fixed number of bind variables (or none). This likely is the type of dynamic SQL you will most often be writing. Here is an example:

```
EXECUTE IMMEDIATE
   'SELECT last_name, salary FROM employees
     WHERE department_id = :dept_id'
```

```
      INTO l_last_name, l_salary
      USING 10;
```

I am querying just two columns from the employees table here, and depositing them into the two local variables with the INTO clause. I also have a single bind variable. Because the numbers of these items are static at the time of compilation, I use method 3 dynamic SQL.

Method 4

Finally, let's consider the most complex scenario: method 4 dynamic SQL. Consider this very generic query:

```
OPEN l_cursor FOR
   'SELECT ' || l_column_list ||
     'FROM employees';
```

At the time I compile my code, I don't have any idea how many columns will be queried from the employees table. This leaves me with quite a challenge: how do I write the FETCH INTO statement to handle that variability? My choices are twofold: either fall back on DBMS_SQL to write relatively straightforward (though voluminous) code, or switch to dynamic PL/SQL block execution.

Fortunately for many of you, scenarios requiring method 4 dynamic SQL are rare. If you run into it, however, you should read "Meeting Method 4 Dynamic SQL Requirements" on page 571.

Binding Variables

You have seen several examples that use bind variables or arguments with NDS. Let's now go over the various rules and special situations you may encounter when binding.

You can bind into your SQL statement only those expressions (literals, variables, complex expressions) that replace placeholders for data values inside the dynamic string. You cannot bind in the names of schema elements (tables, columns, etc.) or entire chunks of the SQL statement (such as the WHERE clause). For those parts of your string, you must use concatenation.

For example, suppose you want to create a procedure that will truncate the specified view or table. Your first attempt might look something like this:

```
PROCEDURE truncobj (
   nm IN VARCHAR2,
   tp IN VARCHAR2 := 'TABLE',
   sch IN VARCHAR2 := NULL)
IS
BEGIN
   EXECUTE IMMEDIATE
      'TRUNCATE :trunc_type :obj_name'
```

```
        USING tp, NVL (sch, USER) || '.' || nm;
END;
```

This code seems perfectly reasonable. But when you try to run the procedure you'll get this error:

```
ORA-03290: Invalid truncate command - missing CLUSTER or TABLE keyword
```

If you rewrite the procedure to simply truncate tables, as follows:

```
EXECUTE IMMEDIATE 'TRUNCATE TABLE :obj_name' USING nm;
```

then the error becomes:

```
ORA-00903: invalid table name
```

Why does NDS (and DBMS_SQL) have this restriction? When you pass a string to EXECUTE IMMEDIATE, the runtime engine must first parse the statement. The parse phase guarantees that the SQL statement is properly defined. PL/SQL can tell that the following statement is valid:

```
'UPDATE emp SET sal = :xyz'
```

without having to know the value of :xyz. But how can PL/SQL know if the following statement is well formed?

```
'UPDATE emp SET :col_name = :xyz'
```

Even if you don't pass in nonsense for col_name, it won't work. For that reason, you must use concatenation:

```
PROCEDURE truncobj (
   nm IN VARCHAR2,
   tp IN VARCHAR2 := 'TABLE',
   sch IN VARCHAR2 := NULL)
IS
BEGIN
   EXECUTE IMMEDIATE
      'TRUNCATE ' || tp || ' ' || NVL (sch, USER) || '.' || nm;
END;
```

Argument Modes

Bind arguments can have one of three modes:

IN
Read-only value (the default mode)

OUT
Write-only variable

IN OUT
Can read the value coming in and write the value going out

When you are executing a dynamic query, all bind arguments must be of mode IN, except when you are taking advantage of the RETURNING clause, as shown here:

```
PROCEDURE wrong_incentive (
   company_in IN INTEGER,
   new_layoffs IN NUMBER
   )
IS
   sql_string VARCHAR2(32767);
   sal_after_layoffs NUMBER;
BEGIN
   sql_string :=
      'UPDATE ceo_compensation
          SET salary = salary + 10 * :layoffs
        WHERE company_id = :company
       RETURNING salary INTO :newsal';

   EXECUTE IMMEDIATE sql_string
     USING new_layoffs, company_in, OUT sal_after_layoffs;

   DBMS_OUTPUT.PUT_LINE (
      'CEO compensation after latest round of layoffs $' || sal_after_layoffs);
END;
```

Besides being used with the RETURNING clause, OUT and IN OUT bind arguments come into play mostly when you are executing dynamic PL/SQL. In this case, the modes of the bind arguments must match the modes of any PL/SQL program parameters, as well as the usage of variables in the dynamic PL/SQL block.

Here are some guidelines for the use of the USING clause with dynamic PL/SQL execution:

- A bind variable of mode IN can be provided as any kind of expression of the correct type: a literal value, named constant, variable, or complex expression. The expression is evaluated and then passed to the dynamic PL/SQL block.
- You must provide a variable to receive the outgoing value for a bind variable of mode OUT or IN OUT.
- You can bind values only to variables in the dynamic PL/SQL block that have a SQL type. If a procedure has a Boolean parameter, for example, that Boolean cannot be set (or retrieved) with the USING clause.

This restriction is *partially* removed in 12.1 and higher. You can now bind many PL/SQL-specific types, including record and collection types, but you still cannot bind Booleans.

Let's take a look at how this works, with a few examples. Here is a procedure with IN, OUT, and IN OUT parameters:

```
PROCEDURE analyze_new_technology (
   tech_name IN VARCHAR2,
   analysis_year IN INTEGER,
   number_of_adherents IN OUT NUMBER,
   projected_revenue OUT NUMBER
   )
```

And here is a block that executes this procedure dynamically:

```
DECLARE
   devoted_followers NUMBER;
   est_revenue NUMBER;
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN
          analyze_new_technology (:p1, :p2, :p3, :p4);
       END;'
   USING 'Java', 2002, IN OUT devoted_followers, OUT est_revenue;
END;
```

Because I have four distinctly named placeholders in the dynamic block, the USING clause must contain four expressions and/or variables. Because I have two IN parameters, the first two of those elements can be literal values or expressions. The second two elements cannot be a literal, constant, or expression, because the parameter modes are OUT or IN OUT.

Ah, but what if two or more of the placeholders have the same name?

Duplicate Placeholders

In a dynamically constructed and executed SQL string, NDS associates placeholders with USING clause bind arguments by *position* rather than by name. The treatment of multiple placeholders with the same name varies, however, according to whether you are using dynamic SQL or dynamic PL/SQL. You need to follow these rules:

- When you are executing a dynamic SQL string (DML or DDL—in other words, when the string does *not* end in a semicolon), you must supply an argument for each placeholder, even if there are duplicates.
- When you are executing a dynamic PL/SQL block (i.e., when the string ends in a semicolon), you must supply an argument for each *unique* placeholder.

Here is an example of a dynamic SQL statement with duplicate placeholders. Notice the repetition of the val_in argument:

```
PROCEDURE updnumval (
   col_in     IN   VARCHAR2,
   start_in   IN   DATE, end_in  IN   DATE,
   val_in     IN   NUMBER)
IS
   dml_str VARCHAR2(32767) :=
      'UPDATE emp SET ' || col_in || ' = :val
        WHERE hiredate BETWEEN :lodate AND :hidate
        AND :val IS NOT NULL';
BEGIN
   EXECUTE IMMEDIATE dml_str
  USING val_in, start_in, end_in, val_in;
END;
```

And here is a dynamic PL/SQL block with a duplicate placeholder. Notice that val_in is supplied only once:

```
PROCEDURE updnumval (
   col_in     IN   VARCHAR2,
   start_in   IN   DATE, end_in IN   DATE,
   val_in     IN   NUMBER)
IS
   dml_str VARCHAR2(32767) :=
      'BEGIN
          UPDATE emp SET ' || col_in || ' = :val
           WHERE hiredate BETWEEN :lodate AND :hidate
           AND :val IS NOT NULL;
       END;';
BEGIN
   EXECUTE IMMEDIATE dml_str
  USING val_in, start_in, end_in;
END;
```

Passing NULL Values

You will encounter special moments when you want to pass a NULL value as a bind argument, as follows:

```
EXECUTE IMMEDIATE
   'UPDATE employee SET salary = :newsal
     WHERE hire_date IS NULL'
   USING NULL;
```

You will, however, get this error:

```
PLS-00457: in USING clause, expressions have to be of SQL types
```

Basically, this is saying that NULL has no datatype, and "no datatype" is not a valid SQL datatype.

So what should you do if you need to pass in a NULL value? You can do one of two things:

- Hide the NULL value behind a variable façade, most easily done with an uninitialized variable, as shown here:

```
DECLARE
   /* Default initial value is NULL */
   no_salary_when_fired NUMBER;
BEGIN
    EXECUTE IMMEDIATE
      'UPDATE employee SET salary = :newsal
        WHERE hire_date IS NULL'
      USING no_salary_when_fired;
END;
```

- Use a conversion function to convert the NULL value to a typed value explicitly:

```
BEGIN
    EXECUTE IMMEDIATE
      'UPDATE employee SET salary = :newsal
        WHERE hire_date IS NULL'
      USING TO_NUMBER (NULL);
END;
```

Working with Objects and Collections

Suppose that I am building an internal administrative system for the national health management corporation Health$.Com. To reduce costs, the system will work in a distributed manner, creating and maintaining separate tables of customer information for each for-profit hospital owned by Health$.Com.

I'll start by defining an object type (person) and nested table type (preexisting_conditions), as follows:

```
/* File on web: health$.pkg */
CREATE OR REPLACE TYPE person AS OBJECT (
   name VARCHAR2(50), dob DATE, income NUMBER);
/
CREATE OR REPLACE TYPE preexisting_conditions IS TABLE OF VARCHAR2(25);
/
```

Once these types are defined, I can build a package to manage my most critical health-related information—data needed to maximize profits at Health$.Com. Here is the specification:

```
PACKAGE health$
AS
   PROCEDURE setup_new_hospital (hosp_name IN VARCHAR2);

   PROCEDURE add_profit_source (
      hosp_name IN VARCHAR2,
      pers IN Person,
      cond IN preexisting_conditions);

   PROCEDURE minimize_risk  (
      hosp_name VARCHAR2,
      min_income IN NUMBER := 100000,
      max_preexist_cond IN INTEGER := 0);

   PROCEDURE show_profit_centers (hosp_name VARCHAR2);
 END health$;
```

With this package, I can do the following:

- Set up a new hospital, which means creating a new table to hold information about that hospital. Here's the implementation from the body:

  ```
  FUNCTION tabname (hosp_name IN VARCHAR2) IS
  BEGIN
     RETURN hosp_name || '_profit_center';
  END;

  PROCEDURE setup_new_hospital (hosp_name IN VARCHAR2) IS
  BEGIN
     EXECUTE IMMEDIATE
        'CREATE TABLE ' || tabname (hosp_name) || ' (
           pers Person,
           cond preexisting_conditions)
           NESTED TABLE cond STORE AS cond_st';
  END;
  ```

- Add a "profit source" (formerly known as a "patient") to the hospital, including data on preexisting conditions. Here's the implementation from the body:

  ```
  PROCEDURE add_profit_source (
     hosp_name IN VARCHAR2,
     pers IN Person,
     cond IN preexisting_conditions)
  IS
  BEGIN
     EXECUTE IMMEDIATE
        'INSERT INTO ' || tabname (hosp_name) ||
            ' VALUES (:revenue_generator, :revenue_inhibitors)'
        USING pers, cond;
  END;
  ```

  The use of objects and collections is transparent. I could be inserting scalars like numbers and dates, and the syntax and code would be the same.

- Minimize the risk to the health maintenance organization's bottom line by removing any patients who have too many preexisting conditions or too little income. This is the most complex of the programs. Here is the implementation:

  ```
  PROCEDURE minimize_risk  (
     hosp_name VARCHAR2,
     min_income IN NUMBER := 100000,
  ```

```
   max_preexist_cond IN INTEGER := 1)
IS
   cv RefCurTyp;
   human Person;
   known_bugs preexisting_conditions;

   v_table VARCHAR2(30) := tabname (hosp_name);
   v_rowid ROWID;
BEGIN
   /* Find all rows with more than the specified number
      of preconditions and deny them coverage. */
   OPEN cv FOR
      'SELECT ROWID, pers, cond
         FROM ' || v_table || ' alias
        WHERE (SELECT COUNT(*) FROM TABLE (alias.cond))
               > ' ||
               max_preexist_cond ||
         ' OR
               alias.pers.income < ' || min_income;
   LOOP
      FETCH cv INTO v_rowid, human, known_bugs;
      EXIT WHEN cv%NOTFOUND;
      EXECUTE IMMEDIATE
         'DELETE FROM ' || v_table || ' WHERE ROWID = :rid'
         USING v_rowid;
   END LOOP;
   CLOSE cv;
END;
```

I decided to retrieve the ROWID of each profit source so that when I did the DELETE it would be easy to identify the rows. It would be awfully convenient to make the query FOR UPDATE and then use WHERE CURRENT OF cv in the DELETE statement, but that is not possible for two reasons: (1) the cursor variable would have to be globally accessible to be referenced inside a dynamic SQL statement; and (2) you cannot declare cursor variables in packages because they don't have persistent state. See "Dynamic PL/SQL" on page 557 for more details.

Still, in general, when you fetch rows that you know you plan to change in some way, you should use FOR UPDATE to ensure that you do not lose your changes.

Dynamic PL/SQL

Dynamic PL/SQL offers some of the most interesting and challenging coding opportunities. Think of it—while a user is running your application, you can take advantage of NDS to do any of the following:

- Create a program, including a package that contains globally accessible data structures.
- Obtain the values of global variables (and modify them) by name.
- Call functions and procedures whose names are not known at compile time.

I have used this technique to build very flexible code generators, softcoded calculation engines for users, and much more. Dynamic PL/SQL allows you to work at a higher level of generality, which can be both challenging and exhilarating.

There are some rules and tips you need to keep in mind when working with dynamic PL/SQL blocks and NDS:

- The dynamic string must be a valid PL/SQL block. It must start with the DECLARE or BEGIN keyword, and end with an END statement and a semicolon (you *can* prefix the DECLARE or BEGIN with a label or a comment). The string will not be considered PL/SQL code unless it ends with a semicolon.
- In your dynamic block, you can access only PL/SQL code elements that have global scope (standalone functions and procedures, and elements defined in the specification of a package). Dynamic PL/SQL blocks execute outside the scope of the local enclosing block.
- Errors raised within a dynamic PL/SQL block can be trapped and handled by the local block in which the string was run with the EXECUTE IMMEDIATE statement.

Build Dynamic PL/SQL Blocks

Let's explore these rules. First, I will build a little utility to execute dynamic PL/SQL:

```
/* File on web: dynplsql.sp */
PROCEDURE dynPLSQL (blk IN VARCHAR2)
IS
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN ' || RTRIM (blk, ';') || '; END;';
END;
```

This one program encapsulates many of the rules mentioned previously for PL/SQL execution. By enclosing the string within a BEGIN-END anonymous block, I guarantee that whatever I pass in will be executed as a valid PL/SQL block. For instance, I can execute the calc_totals procedure dynamically as simply as this:

```
SQL> exec dynPLSQL ('calc_totals');
```

Now let's use this program to examine what kinds of data structures you can reference within a dynamic PL/SQL block. In the following anonymous block, I want to use dynamic SQL to assign a value of 5 to the local variable num:

```
<<dynamic>>
DECLARE
   num NUMBER;
BEGIN
   dynPLSQL ('num := 5');
END;
```

This string is executed within its own BEGIN-END block, which appears to be a nested block within the anonymous block named "dynamic". Yet when I execute this script, I receive the following error:

```
PLS-00201: identifier 'NUM' must be declared
ORA-06512: at "SCOTT.DYNPLSQL", line 4
```

The PL/SQL engine is unable to resolve the reference to the variable named num. I get the same error even if I qualify the variable name with its block name:

```
<<dynamic>>
DECLARE
   num NUMBER;
BEGIN
   /* Also causes a PLS-00302 error! */
   dynPLSQL ('dynamic.num := 5');
END;
```

Now suppose that I define the num variable inside a package as follows:

```
PACKAGE pkgvars
IS
   num NUMBER;
END pkgvars;
```

I can now successfully execute the dynamic assignment to this newly defined variable:

```
BEGIN
   dynPLSQL ('pkgvars.num := 5');
END;
```

What's the difference between these two pieces of data? In my first attempt, the variable num is defined locally in the anonymous PL/SQL block. In my second attempt, num is a public global variable defined in the pkgvars package. This distinction makes all the difference with dynamic PL/SQL.

It turns out that a dynamically constructed and executed PL/SQL block is not treated as a nested block; instead, it is handled as if it were a procedure or function called from within the current block. So, any variables local to the current or enclosing blocks are not recognized in the dynamic PL/SQL block; you can make references only to globally defined programs and data structures. These PL/SQL elements include standalone functions and procedures and any elements defined in the specification of a package.

Fortunately, the dynamic block is executed within the context of the calling block. If you have an exception section within the calling block, it will trap exceptions raised in the dynamic block. So, if I execute this anonymous block in SQL*Plus:

```
BEGIN
   dynPLSQL ('undefined.packagevar := ''abc''');
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE (SQLCODE);
END;
```

I will not get an unhandled exception.

The assignment performed in this anonymous block is an example of indirect referencing: I don't reference the variable directly, but instead do so by specifying the name of the variable. The Oracle Forms Builder product (formerly known as SQL*Forms and Oracle Forms) offers an implementation of indirect referencing with the NAME_IN and COPY programs. This feature allows developers to build logic that can be shared across all forms in the application. PL/SQL does not support indirect referencing, but you can implement it with dynamic PL/SQL. See the *dynvar.pkg* file on the book's website for an example of such an implementation.

The following sections offer a few more examples of dynamic PL/SQL to spark your interest and, perhaps, inspire your creativity.

Replace Repetitive Code with Dynamic Blocks

This is a true story, I kid you not. During a consulting stint at an insurance company in Chicago, I was asked to see what I could do about a particularly vexing program. It was very large and continually increased in size—soon it would be too large to even compile. Much to my amazement, this is what the program looked like:

```
PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
   IF    line = 1 THEN process_line1;
   ELSIF line = 2 THEN process_line2;
   ...
   ELSIF line = 514 THEN process_line514;
   ...
   ELSIF line = 2057 THEN process_line2057;
   END IF;
END;
```

Each one of those line numbers represented fine print in an insurance policy that helped the company achieve its primary objective (minimizing the payment of claims). For each line number, there was a process_line program that handled those details. And as the insurance company added more and more exceptions to the policy, the program got bigger and bigger. Not a very scalable approach to programming!

To avoid this kind of mess, a programmer should be on the lookout for repetition of code. If you can detect a pattern, you can either create a reusable program to encapsulate that pattern, or explore the possibility of expressing that pattern as a dynamic SQL construction.

At the time, I fixed the problem using DBMS_SQL, but dynamic SQL would have been a perfect match. Here's the NDS implementation:

```
PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN process_line' || line || '; END;';
END;
```

From thousands of lines of code down to one executable statement! Of course, in most cases, identification of the pattern and conversion of that pattern into dynamic SQL will not be so straightforward. Still, the potential gains are enormous.

Recommendations for NDS

By now, you should have a solid understanding of how native dynamic SQL works in PL/SQL. This section covers some topics you should be aware of as you start to build production applications with this PL/SQL feature.

Use Invoker Rights for Shared Programs

I have created a number of useful generic programs in my presentation of NDS, including functions and procedures that do the following:

- Execute any DDL statement.
- Return the count of rows in any table.
- Return the count for each grouping by specified column.

These are pretty darn useful utilities, and I want to let everyone on my development team use them. So, I compile them into the COMMON schema and grant EXECUTE authority on the programs to PUBLIC.

However, there is a problem with this strategy. When Sandra connects to her SANDRA schema and executes this command:

```
SQL> EXEC COMMON.exec_DDL ('create table temp (x date)');
```

she will inadvertently create a table in the COMMON schema—unless I take advantage of the invoker rights model, which is described in detail in Chapter 24. The invoker rights model means that you define your stored programs so that they execute under the authority of and the privileges of the invoking schema rather than the defining schema (which is the default starting with Oracle8*i* Database and the only option prior to that release).

Fortunately, it's easy to take advantage of this new feature. Here is a version of my exec_ddl procedure that executes any DDL statement, but always has an impact on the calling or invoking schema:

```
PROCEDURE exec_DDL (ddl_string IN VARCHAR2)
   AUTHID CURRENT_USER
IS
BEGIN
   EXECUTE IMMEDIATE ddl_string;
END;
```

I recommend that you use the AUTHID CURRENT_USER clause in all of your dynamic SQL programs, and particularly in those you plan to share among a group of developers.

Anticipate and Handle Dynamic Errors

Any robust application needs to anticipate and handle errors. Error detection and correction with dynamic SQL can be especially challenging.

Sometimes the most challenging aspect of building and executing dynamic SQL programs is getting the string of dynamic SQL correct. You might be combining a list of columns in a query with a list of tables and then a WHERE clause that changes with each execution. You have to concatenate all that stuff, getting the commas right, the ANDs and ORs right, and so on. What happens if you get it wrong?

Well, the Oracle database raises an error. This error usually tells you exactly what is wrong with the SQL string, but that information can still leave much to be desired. Consider the following nightmare scenario: I am building the most complicated PL/SQL application ever. It uses dynamic SQL left and right, but that's OK. I am a pro at NDS. I can, in a flash, type EXECUTE IMMEDIATE, OPEN FOR, and all the other statements I need. I blast through the development phase, and rely on some standard exception-handling programs I have built to display an error message when an exception is encountered.

Then the time comes to test my application. I build a test script that runs through a lot of my code; I place it in a file named *testall.sql* (you'll find it on the book's website). With trembling fingers, I start my test:

```
SQL> @testall
```

And, to my severe disappointment, here is what shows up on my screen:

```
ORA-00942: table or view does not exist
ORA-00904: invalid column name
ORA-00921: unexpected end of SQL command
ORA-00936: missing expression
```

Now, what am I supposed to make of all these error messages? Which error message goes with which SQL statement? Bottom line: when you do lots of dynamic SQL, it is very easy to get very confused and waste lots of time debugging your code—unless you take precautions as you write your dynamic SQL.

Here are my recommendations:

- Always include an error-handling section in code that calls EXECUTE IMMEDIATE and OPEN FOR.
- In each handler, record and/or display the error message and the SQL statement when an error occurs.
- You might also want to consider adding a "trace" in front of these statements so that you can easily watch the dynamic SQL as it is constructed and executed.

How do these recommendations translate into changes in your code? Let's first, apply these changes to the exec_ddl routine, and then generalize from there. Here is the starting point:

```
PROCEDURE exec_ddl (ddl_string IN VARCHAR2)
   AUTHID CURRENT_USER IS
BEGIN
   EXECUTE IMMEDIATE ddl_string;
END;
```

Now let's add an error-handling section to show us problems when they occur:

```
/* File on web: execddl.sp */
PROCEDURE exec_ddl (ddl_string IN VARCHAR2)
   AUTHID CURRENT_USER IS
BEGIN
   EXECUTE IMMEDIATE ddl_string;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE (
         'Dynamic SQL Failure: ' || DBMS_UTILITY.FORMAT_ERROR_STACK);
      DBMS_OUTPUT.PUT_LINE (
         '   on statement: "' || ddl_string || '"');
      RAISE;
END;
```

When I use this version to attempt to create a table using really bad syntax, this is what I see:

```
SQL> EXEC execddl ('create table x')
Dynamic SQL Failure: ORA-00906: missing left parenthesis
   on statement: "create table x"
```

Of course, in your production version, you might want to consider something a bit more sophisticated than the DBMS_OUTPUT built-in package.

With DBMS_SQL, if your parse request fails and you do not explicitly close your cursor in the error section, that cursor remains open (and uncloseable), leading to possible *ORA-01000: maximum open cursors exceeded* errors. This will not happen with NDS; cursor variables declared in a local scope are automatically closed—and the memory released—when the block terminates.

Now let's broaden our view a bit: when you think about it, the exec_ddl procedure is not really specific to DDL statements. It can be used to execute any SQL string that does not require either USING or INTO clauses. From that perspective, you now have a single program that can and should be used in place of a direct call to EXECUTE IMMEDIATE; it has all that error handling built in. I supply such a procedure in the ndsutil package.

I could even create a similar program for OPEN FOR—again, only for situations that do not require a USING clause. Because OPEN FOR sets a cursor value, I would probably want to implement it as a function, which would return a type of weak REF CURSOR. This leads right to a packaged implementation along these lines:

```
PACKAGE ndsutil
IS
   FUNCTION openFor (sql_string IN VARCHAR2) RETURN SYS_REFCURSOR;
END;
```

This NDS utility package contains the complete implementation of this function; the body is quite similar to the exec_dll procedure shown earlier.

Use Binding Rather than Concatenation

In most situations, you can take two different paths to insert program values into your SQL string: binding and concatenation. The following table contrasts these approaches for a dynamic UPDATE statement.

Binding	Concatenation
`EXECUTE IMMEDIATE 'UPDATE ' || tab || 'SET sal = :new_sal' USING v_sal;`	`EXECUTE IMMEDIATE 'UPDATE ' || tab || 'SET sal = '|| v_sal;`

Binding involves the use of placeholders and the USING clause; concatenation shortcuts that process by adding the values directly to the SQL string. When should you use each approach? I recommend that you bind arguments whenever possible (see the next section for limitations on binding) rather than relying on concatenation. There are four reasons to take this approach:

Binding is usually faster

When you bind in a value, the SQL string does not contain the value, just the placeholder name. Therefore, you can bind different values to the same SQL statement without changing that statement. Because it is the same SQL statement, your application can more likely take advantage of the preparsed cursors that are cached in the SGA of the database.

Note 1: I included the phrase "more likely" here because there are very few absolutes in the world of Oracle optimization. For example, one possible drawback with binding is that the cost-based optimizer has less information with which to work and might not come up with the best explain plan for your SQL statement.

Note 2: If you need to execute the same SQL statement multiple times with different bind variables, consider using DBMS_SQL—it can completely avoid parsing, which is not possible with NDS. (See "Minimizing Parsing of Dynamic Cursors" on page 578.)

Binding is easier to write and maintain

When you bind, you don't have to worry about datatype conversion; it is all handled for you by the NDS engine. In fact, binding minimizes datatype conversion because it works with the native datatypes. If you use concatenation, you will often need to write very complex, error-prone string expressions involving multiple single quotes, TO_DATE and TO_CHAR function calls, and so on.

Binding helps avoid implicit conversions

If you concatenate, you might inadvertently leave it up to the database to perform implicit conversions. Under some circumstances, the conversion that the database applies might not be the one you wanted; it could negate the use of indexes.

Binding negates the chance of code injection

One of the greatest dangers with dynamic SQL is that you write very generalized code that is intended to be used in a certain way, yet depending on what the user passes in to your string, the resulting dynamic statement could perform very different kinds of operations. That is, users can "inject" unwanted actions into your SQL statement. See the following section for an example.

There are some potential downsides to binding, however. In versions earlier than 11.1, bind variables negate the use of any histogram statistics because the bind values are assigned only after the statement has been parsed. The cost-based optimizer may, therefore, have less information to work with, and be unable to come up with the best execution plan for your SQL statement. In 11.1 and higher, the adaptive cursor sharing feature can now take advantage of these statistics.

For PL/SQL developers, I believe the primary emphasis should be how to write clean, easy-to-understand, and maintainable code. If I rely on lots of concatenation, I end up with statements that look like this:

```
EXECUTE IMMEDIATE
 'UPDATE employee SET salary = ' || val_in ||
 ' WHERE hire_date BETWEEN ' ||
    ' TO_DATE (''' || TO_CHAR (v_start)  || ''')' ||
    ' AND ' ||
    ' TO_DATE (''' || TO_CHAR (v_end)  || ''')';
```

A switch to binding makes my code much more understandable:

```
EXECUTE IMMEDIATE
   'UPDATE employee SET salary = :val
     WHERE hire_date BETWEEN :lodate AND :hidate'
   USING v_sal, v_start, v_end;
```

While there may be some scenarios in which concatenation is actually more efficient, don't worry about that until you or your DBA identifies a particular dynamic SQL statement with binding as the source of the problem. In other words, move from binding to concatenation only when a bottleneck is identified—on an exception basis.

Minimize the Dangers of Code Injection

Many web-based applications offer wonderful flexibility to the end user. This flexibility is often accomplished through the execution of dynamic SQL and PL/SQL blocks. Consider the following example of a very general "get rows" procedure:

```
/* File on web: code_injection.sql */
PROCEDURE get_rows (
   table_in IN VARCHAR2, where_in IN VARCHAR2
)
IS
BEGIN
   EXECUTE IMMEDIATE
      'DECLARE
       l_row ' || table_in || '%ROWTYPE;
    BEGIN
          SELECT * INTO l_row
      FROM ' || table_in || ' WHERE ' || where_in || ';
    END;';
END get_rows;
```

This looks like such an innocent program, but in fact it opens up gaping holes in your application. Consider the following block:

```
BEGIN
   get_rows ('EMPLOYEE'
     ,'employee_id=7369;
       EXECUTE IMMEDIATE
          ''CREATE PROCEDURE backdoor (str VARCHAR2)
```

```
            AS BEGIN EXECUTE IMMEDIATE str; END;''' );
END;
/
```

After running this code, I have created a "back door" procedure that will execute any statement I pass in as a dynamic string. I could, for example, use UTL_FILE to retrieve the contents of any file on the system, then create (and drop) any table or object I desire, restricted only by whatever privileges are defined for the owner's schema.

Code injection, also known as *SQL injection*, can compromise seriously the security of any application. The execution of dynamic PL/SQL blocks offers the greatest opportunity for injection. While this is a very big topic that cannot be treated fully in this book, I offer the following recommendations to minimize the chances of injection occurring with your application. Chapter 23 provides additional security recommendations.

Restrict privileges tightly on user schemas

The best way to minimize the risk of injection is to make sure that any schema to which an outside user connects has severely restricted privileges.

Do not let such a schema create database objects, remove database objects, or directly access tables. Do not allow the execution of supplied packages that interact (or can be used to interact) with the operating system, such as UTL_SMTP, UTL_FILE, UTL_TCP (and related packages), and DBMS_PIPE.

Such a schema should have privileges only to execute stored programs defined in *another* schema. This PL/SQL code may then be designed to carefully allow only a restricted set of operations. When defining programs in these executable schemas that use dynamic SQL, be sure to define the subprograms as AUTHID CURRENT_USER. That way, all SQL statements will be executed using the limited privileges of the currently connected schema.

Use bind variables whenever possible

Strict enforcement of the use of bind variables, plus built-in analysis and automated rejection of potentially dangerous strings, can help minimize the danger of injection.

By requiring binding, you can lose some flexibility. For instance, in the get_rows procedure I would need to replace the completely dynamic WHERE clause with something less generic, but more tightly fitting the expected behavior of the application. Here's an example using a variation of the get_rows procedure:

```
PROCEDURE get_rows (
   table_in   IN   VARCHAR2, value1_in in VARCHAR2, value2_in IN DATE
)
IS
   l_where VARCHAR2(32767);
BEGIN
   IF table_in = 'EMPLOYEES'
```

```
      THEN
         l_where := 'last_name = :name AND hire_date < :hdate';
      ELSIF table_in = 'DEPARTMENTS'
      THEN
         l_where := 'name LIKE :name AND incorporation_date = :hdate';
      ELSE
         RAISE_APPLICATION_ERROR (
            -20000, 'Invalid table name for get_rows: ' || table_in);
      END IF;
      EXECUTE IMMEDIATE
         'DECLARE l_row ' || table_in || '%ROWTYPE;
          BEGIN
             SELECT * INTO l_row
          FROM ' || table_in || ' WHERE ' || l_where || ';
          END;'
          USING value1_in, value2_in;
   END get_rows;
   /
```

In this rewrite, the WHERE clause relies on two bind variables; there is no opportunity to concatenate a back door entry point. I also check the table name and make sure it is one that I expect to see. This will help avoid calls to functions in the FROM clause (known as *table functions*), which could also cause aberrant behavior.

Check dynamic text for dangerous text

The problem with the recommendations in the previous sections is that they rely on the proactive diligence of an individual developer or DBA to minimize the risk. While these recommendations are a good start, perhaps something more could be offered to developers. It is also possible to include checks in your programs to make sure that the text provided by the user does not contain "dangerous" characters, such as the semicolon.

I created a utility named SQL Guard that takes another approach: analyzing the string provided by the user to see if it contains a risk of SQL injection. The programmer can then decide whether or not to execute that statement and perhaps to log the problematic text. You will find the code and a user's guide for SQL Guard in the *sqlguard.zip* file on the book's website.

With SQL Guard, the tests used to determine if there is a risk of SQL injection can be configured by the user. In other words, SQL Guard comes with a set of predefined tests; you can remove from or add to that list of tests to check for SQL injection patterns that may be specific to your own application environment.

It isn't possible to come up with a proactive mechanism that will trap, with 100% certainty, all possible SQL injection attempts. That said, if you decide to use SQL Guard, you *should* (it seems to me) be able to achieve the following:

- Increase awareness of the threat of SQL injection among your developers.

- Thwart the most common SQL injection attacks.
- More easily analyze your code base to identify possible injection pathways.

Use DBMS_ASSERT to validate inputs

Use the supplied DBMS_ASSERT package to ensure that a user input that is *supposed* to be a valid SQL object name (for example, a schema name or table name) is, in fact, valid. The DBMS_ASSERT package was first documented in Oracle Database 11*g*. It has since been backported to each of these Oracle versions: 8.1, 9.2, 10.1, and 10.2. In some cases, it is available in the latest patchset; in others, it is available in a Critical Patch Update. You may need to contact Oracle Support before you can start using the package.

DBMS_ASSERT.SIMPLE_SQL_NAME is purely an *asserter*. You pass it a string that should contain a valid SQL name. If it is valid, the function returns that string, unchanged. If it is *not* valid, Oracle raises the DBMS_ASSERT.INVALID_SQL_NAME exception.

For a much more comprehensive treatment of this issue, check out the whitepaper titled "How to write SQL injection proof PL/SQL," available on the Oracle Technology Network.

When to Use DBMS_SQL

Native dynamic SQL should be your first choice (over DBMS_SQL) to satisfy dynamic SQL requirements in your PL/SQL programs, because it is much easier to write; you need less code, and the code you write is more intuitive, leading to many fewer bugs. The code is also much easier to maintain and in most situations will be more efficient.

There are, however, situations when you will want or need to use DBMS_SQL. The following sections describe these situations.

Obtain Information About Query Columns

DBMS_SQL allows you to describe the columns of your dynamic cursor, returning information about each column in an associative array of records. This capability offers the possibility of writing very generic cursor-processing code; this program may come in particularly handy when you are writing method 4 dynamic SQL, and you are not certain how many columns are being selected.

When you call this program, you need to have declared a PL/SQL collection based on the DBMS_SQL.DESC_TAB collection type (or DESC_TAB2, if your query might return column names that are greater than 30 characters in length). You can then use collection methods to traverse the table and extract the needed information about the cursor. The following anonymous block shows the basic steps you will perform when working with this built-in:

```
DECLARE
   cur PLS_INTEGER := DBMS_SQL.OPEN_CURSOR;
   cols DBMS_SQL.DESC_TAB;
   ncols PLS_INTEGER;
BEGIN
   -- Parse the query
   DBMS_SQL.PARSE
      (cur, 'SELECT hire_date, salary FROM employees', DBMS_SQL.NATIVE);
   -- Retrieve column information
   DBMS_SQL.DESCRIBE_COLUMNS (cur, ncols, cols);
   -- Display each of the column names
   FOR colind IN 1 .. ncols
   LOOP
      DBMS_OUTPUT.PUT_LINE (cols (colind).col_name);
   END LOOP;
   DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

To simplify your use of DESCRIBE_COLUMNS, I have created a package that hides much of the underlying detail, making it easier to use this feature. Here is the package specification:

```
/* File on web: desccols.pkg */
PACKAGE desccols
IS

   FUNCTION for_query (sql_in IN VARCHAR2)
      RETURN DBMS_SQL.desc_tab;

   FUNCTION for_cursor (cur IN PLS_INTEGER)
      RETURN DBMS_SQL.desc_tab;

   PROCEDURE show_columns (
      col_list_in   IN   DBMS_SQL.desc_tab
   );
END desccols;
```

You can also use the for_query function when you want to get information about the columns of a dynamic query, but might not otherwise be using DBMS_SQL.

Here is a script demonstrating the usage of this package:

```
/* File on web: desccols.sql */
DECLARE
   cur   INTEGER           := DBMS_SQL.open_cursor;
   tab   DBMS_SQL.desc_tab;
BEGIN
   DBMS_SQL.parse (cur
                 , 'SELECT last_name, salary, hiredate FROM employees'
                 , DBMS_SQL.native
```

```
                  );
   tab := desccols.for_cursor (cur);
   desccols.show (tab);
   DBMS_SQL.close_cursor (cur);
   --
   tab := desccols.for_query ('SELECT * FROM employees');
   desccols.show (tab);
END;
/
```

Meeting Method 4 Dynamic SQL Requirements

DBMS_SQL supports method 4 dynamic SQL (variable number of columns selected or variables bound) more naturally than NDS. You have already seen that in order to implement method 4 with NDS, you must switch to dynamic PL/SQL, which is generally a higher level of abstraction than many developers want to deal with.

When would you run into method 4? It certainly arises when you build a frontend to support ad hoc query generation by users, or when you want to build a generic report program, which constructs the report format and contents dynamically at runtime. Let's step through the implementation of a variation on this theme: the construction of a PL/SQL procedure to display the contents of a table—any table—as specified by the user at runtime. Here I cover only those aspects pertaining to the dynamic SQL itself; check out the *intab.sp* file on the book's website for the full implementation.

The "in table" procedural interface

For this implementation, I will use PL/SQL and DBMS_SQL. But before building any code, I need to come up with a specification. How will the procedure be called? What information do I need from my user (a developer, in this case)? What should a user have to type to retrieve the desired output? I want my procedure (which I call "intab" for "in table") to accept the inputs in the following table.

Parameter	Description
Name of the table	Required. Obviously, a key input to this program.
WHERE clause	Optional. Allows you to restrict the rows retrieved by the query. If not specified, all rows are retrieved. You can also use this parameter to pass in ORDER BY and HAVING clauses, because they follow immediately after the WHERE clause.
Column name filter	Optional. If you don't want to display all columns in the table, provide a comma-delimited list, and only those columns will be used.

Given these inputs, the specification for my procedure becomes the following:

```
PROCEDURE intab (
   table_in IN VARCHAR2
 , where_in IN VARCHAR2 DEFAULT NULL
 , colname_like_in IN VARCHAR2 := '%'
);
```

Here are some examples of calls to intab, along with their output. First, the entire contents of the emp table:

```
SQL> EXEC intab ('emp');
-----------------------------------------------------------------------
_                          Contents of emp
-----------------------------------------------------------------------
EMPNO ENAME      JOB       MGR  HIREDATE        SAL     COMM    DEPTNO
-----------------------------------------------------------------------
 7369  SMITH      CLERK     7902 12/17/80 120000 800              20
 7499  ALLEN      SALESMAN  7698 02/20/81 120000 1600    300      30
 7521  WARD       SALESMAN  7698 02/22/81 120000 1250    500      30
 7566  JONES      MANAGER   7839 04/02/81 120000 2975             20
 7654  MARTIN     SALESMAN  7698 09/28/81 120000 1250    1400     30
 7698  BLAKE      MANAGER   7839 05/01/81 120000 2850             30
 7782  CLARK      MANAGER   7839 06/09/81 120000 2450             10
 7788  SCOTT      ANALYST   7566 04/19/87 120000 3000             20
 7839  KING       PRESIDENT      11/17/81 120000 5000             10
 7844  TURNER     SALESMAN  7698 09/08/81 120000 1500    0        30
 7876  ADAMS      CLERK     7788 05/23/87 120000 1100             20
 7900  JAMES      CLERK     7698 12/03/81 120000 950              30
 7902  FORD       ANALYST   7566 12/03/81 120000 3000             20
```

And now let's see just those employees in department 10, specifying a maximum length of 20 characters for string columns:

```
SQL> EXEC intab ('emp', 'deptno = 10 ORDER BY sal');
-----------------------------------------------------------------------
_                          Contents of emp
-----------------------------------------------------------------------
EMPNO ENAME      JOB       MGR  HIREDATE        SAL     COMM    DEPTNO
-----------------------------------------------------------------------
 7934  MILLER     CLERK     7782 01/23/82 120000 1300             10
 7782  CLARK      MANAGER   7839 06/09/81 120000 2450             10
 7839  KING       PRESIDENT      11/17/81 120000 5000             10
```

And now an entirely different table, with a different number of columns:

```
SQL> EXEC intab ('dept')
------------------------------------
_        Contents of dept
------------------------------------
DEPTNO DNAME          LOC
------------------------------------
 10     ACCOUNTING     NEW   YORK
 20     RESEARCH       DALLAS
 30     SALES          CHICAGO
 40     OPERATIONS     BOSTON
```

Notice that the user does not have to provide any information about the structure of the table. My program will get that information itself—precisely the aspect of intab that makes it a method 4 dynamic SQL example.

Steps for intab construction

To display the contents of a table, follow these steps:

1. Construct and parse the SELECT statement (using OPEN_CURSOR and PARSE).
2. Bind all local variables with their placeholders in the query (using BIND_VARIABLE).
3. Define each column in the cursor for this query (using DEFINE_COLUMN).
4. Execute and fetch rows from the database (using EXECUTE and FETCH_ ROWS).
5. Retrieve values from the fetched row, and place them into a string for display purposes (using COLUMN_VALUE). Then display that string with a call to the PUT_LINE procedure of the DBMS_OUTPUT package.

My intab implementation does not currently support bind variables. I assume, in other words, that the *where_in* argument does not contain any bind variables. As a result, I will not be exploring in detail the code required for step 2.

Constructing the SELECT

To extract the data from the table, I have to construct the SELECT statement. The structure of the query is determined by the various inputs to the procedure (table name, WHERE clause, etc.) and the contents of the data dictionary. Remember that the user does not have to provide a list of columns. Instead, I must identify and extract the list of columns for that table from a data dictionary view. I have decided to use the ALL_TAB_COLUMNS view in the intab procedure so the user can view the contents not only of tables she owns (which are accessible in USER_TAB_COLUMNS), but also any table for which she has SELECT access.

Here is the cursor I use to fetch information about the table's columns:

```
CURSOR col_cur
   (owner_in IN VARCHAR2,
    table_in IN VARCHAR2)
IS
   SELECT column_name, data_type,
          data_length,
          data_precision, data_scale
     FROM all_tab_columns
    WHERE owner = owner_in
      AND table_name = table_in;
```

With this column cursor, I extract the name, datatype, and length information for each column in the table. How should I store all of this information in my PL/SQL program?

To answer this question, I need to think about how that data will be used. It turns out that I will use it in many ways—for example:

- To build the select list for the query, I will use the column names.
- To display the output of a table in a readable fashion, I need to provide a column header that shows the names of the columns over their data. These column names must be spaced out across the line of data in, well, columnar format. So, I need the column names and the length of the data for each column.
- To fetch data into a dynamic cursor, I need to establish the columns of the cursor with calls to DEFINE_COLUMN. For this, I need the column datatypes and lengths.
- To extract the data from the fetched row with COLUMN_VALUE, I need to know the datatypes of each column, as well as the number of columns.
- To display the data, I must construct a string containing all the data (using TO_CHAR to convert numbers and dates). Again, I must pad out the data to fit under the column names, just as I did with the header line.

I need to work with the column information several times throughout my program, yet I do not want to read repeatedly from the data dictionary. As a result, when I query the column data out of the ALL_TAB_COLUMNS view, I will store that data in three PL/SQL collections, as described in the following table.

Collection	Description
colname	The names of each column
coltype	The datatypes of each column (a string describing the datatype)
collen	The number of characters required to display the column data

So, if the third column of the emp table is SAL, then colname(3) = 'SAL', coltype(3) = 'NUMBER', and collen(3) = 7, and so forth.

The name and datatype information is stored directly from the data dictionary. Calculating the column length is a bit trickier, but not crucial to learning how to write method 4 dynamic SQL. I will leave it to the reader to study the file.

I apply all of my logic inside a cursor FOR loop that sweeps through all the columns for a table (as defined in ALL_COLUMNS). This loop (shown in the following example) fills my PL/SQL collection:

```
FOR col_rec IN col_cur (owner_nm, table_nm)
LOOP
   /* Construct select list for query. */
   col_list := col_list || ', ' || col_rec.column_name;

   /* Save datatype and length for calls to DEFINE_COLUMN. */
   col_count := col_count + 1;
   colname (col_count) := col_rec.column_name;
```

```
      coltype (col_count) := col_rec.data_type;

      /* Construct column header line. */
      col_header :=
         col_header || ' ' || RPAD (col_rec.column_name, v_length);
   END LOOP;
```

When this loop completes, I have constructed the select list, populated my PL/SQL collections with the column information I need for calls to DBMS_SQL.DEFINE_COLUMN and DBMS_SQL.COLUMN_VALUE, and also created the column header line. Now that was a busy loop!

Now it is time to parse the query, and then construct the various columns in the dynamic cursor object.

Defining the cursor structure

The parse phase is straightforward enough. I simply cobble together the SQL statement from its processed and refined components, including, most notably, the column list I just constructed (the col_list variable):

```
DBMS_SQL.PARSE
   (cur,
    'SELECT ' || col_list ||
    '  FROM ' || table_in || ' ' || where_clause,
    DBMS_SQL.NATIVE);
```

Of course, I want to go far beyond parsing. I want to execute this cursor. Before I do that, however, I must give some structure to the cursor. With DBMS_SQL, when you open a cursor, you have merely retrieved a handle to a chunk of memory. When you parse the SQL statement, you have associated a SQL statement with that memory. But as a next step, I must define the columns in the cursor so that it can actually store fetched data.

With method 4 dynamic SQL, this association process is complicated. I cannot hardcode the number or type of calls to DBMS_SQL.DEFINE_COLUMN in my program; I do not have all the information until runtime. Fortunately, in the case of intab, I *have* kept track of each column to be retrieved. Now all I need to do is issue a call to DBMS_SQL.DEFINE_COLUMN for each row defined in my collection, colname. Before we go through the actual code, here are some reminders about DBMS_SQL.DEFINE_COLUMN.

The header for this built-in procedure is as follows:

```
PROCEDURE DBMS_SQL.DEFINE_COLUMN
    (cursor_handle IN INTEGER,
    position IN INTEGER,
    datatype_in IN DATE|NUMBER|VARCHAR2)
```

There are three things to keep in mind with this built-in:

- The second argument is a number. DBMS_SQL.DEFINE_COLUMN does not work with column *names*; it only works with the sequential positions of the columns in the list.
- The third argument establishes the datatype of the cursor's column. It does this by accepting an expression of the appropriate type. You do not, in other words, pass a string such as "VARCHAR2" to DBMS_SQL.DEFINE_COLUMN. Instead, you would pass a variable defined as VARCHAR2.
- When you are defining a character-type column, you must also specify the maximum length of values that can be retrieved into the cursor.

In the context of the intab procedure, the row in the collection is the *N*th position in the column list. The datatype is stored in the coltype collection, but must be converted into a call to DBMS_SQL.DEFINE_COLUMN using the appropriate local variable. These complexities are handled in the following FOR loop:

```
FOR col_ind IN 1 .. col_count
LOOP
  IF is_string (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN
      (cur, col_ind, string_value, collen (col_ind));

  ELSIF is_number (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN (cur, col_ind, number_value);

  ELSIF is_date (col_ind)
  THEN
    DBMS_SQL.DEFINE_COLUMN (cur, col_ind, date_value);
  END IF;
END LOOP;
```

When this loop is completed, I will have called DEFINE_COLUMN for each column defined in the collections. (In my version, this is all columns for a table. In your enhanced version, it might be just a subset of all these columns.) I can then execute the cursor and start fetching rows. The execution phase is no different for method 4 than it is for any of the other simpler methods. Specify:

```
fdbk := DBMS_SQL.EXECUTE (cur);
```

where fdbk is the feedback returned by the call to EXECUTE.

Now for the finale: retrieval of data and formatting for display.

Retrieving and displaying data

I use a cursor FOR loop to retrieve each row of data identified by my dynamic cursor. If I am on the first row, I will display a header (this way, I avoid displaying the header for a query that retrieves no data). For each row retrieved, I build the line and then display it:

```
LOOP
   fdbk := DBMS_SQL.FETCH_ROWS (cur);
   EXIT WHEN fdbk = 0;

   IF DBMS_SQL.LAST_ROW_COUNT = 1
   THEN
      /* We will display the header information here */
      ...
   END IF;

   /* Construct the line of text from column information here */
   ...

   DBMS_OUTPUT.PUT_LINE (col_line);
END LOOP;
```

The line-building program is actually a numeric FOR loop in which I issue my calls to DBMS_SQL.COLUMN_VALUE. I call this built-in for each column in the table (information that is stored in—you guessed it—my collections). As you can see, I use my is_* functions to determine the datatype of the column and therefore the appropriate variable to receive the value.

Once I have converted my value to a string (necessary for dates and numbers), I pad it on the right with the appropriate number of blanks (stored in the collen collection) so that it lines up with the column headers:

```
col_line := NULL;
FOR col_ind IN 1 .. col_count
LOOP
   IF is_string (col_ind)
   THEN
      DBMS_SQL.COLUMN_VALUE (cur, col_ind, string_value);

   ELSIF is_number (col_ind)
   THEN
      DBMS_SQL.COLUMN_VALUE (cur, col_ind, number_value);
      string_value := TO_CHAR (number_value);

   ELSIF is_date (col_ind)
   THEN
      DBMS_SQL.COLUMN_VALUE (cur, col_ind, date_value);
      string_value := TO_CHAR (date_value, date_format_in);
   END IF;

   /* Space out the value on the line
      under the column headers. */
   col_line :=
      col_line || ' ' ||
      RPAD (NVL (string_value, ' '), collen (col_ind));
END LOOP;
```

There you have it: a very generic procedure for displaying the contents of a database table from within a PL/SQL program. Again, check out *intab.sp* for the full details; the *intab_dbms_sql.sp* file also contains a version of this procedure that is updated to take advantage of more recent database features and is more fully documented.

Minimizing Parsing of Dynamic Cursors

One of the drawbacks of EXECUTE IMMEDIATE is that each time the dynamic string is executed it will be reprepared, which may involve parsing, optimization, and plan generation—though in most cases in a well-designed application, it will simply be a "soft parse" for a previously parsed statement. For most dynamic SQL requirements, the overhead of these steps will be compensated for by other benefits of NDS (in particular, the avoidance of calls to a PL/SQL API, as happens with DBMS_SQL). If, however, you find yourself executing the same statement repeatedly, only changing the bind variables, then DBMS_SQL might offer some advantages.

With DBMS_SQL, you can explicitly avoid the parse phase when you know that the SQL string you are executing dynamically is changing only its bind variables. All you have to do is avoid calling DBMS_SQL.PARSE again, and simply rebind the variable values with calls to DBMS_SQL.BIND_VARIABLE. Let's look at a very simple example, demonstrating the specific calls you make to the DBMS_SQL package.

The following anonymous block executes a dynamic query inside a loop:

```
 1 DECLARE
 2 l_cursor   pls_INTEGER;
 3 l_result   pls_INTEGER;
 4 BEGIN
 5 FOR i IN 1 .. counter
 6 LOOP
 7   l_cursor := DBMS_SQL.open_cursor;
 8    DBMS_SQL.parse
 9       (l_cursor, 'SELECT ... where col = ' || i , DBMS_SQL.native);
10    l_result := DBMS_SQL.EXECUTE (l_cursor);
11    DBMS_SQL.close_cursor (l_cursor);
12 END LOOP;
13 END;
```

Within my loop, I take the actions listed in the following table.

Line(s)	Description
7	Obtain a cursor—simply a pointer to memory used by DBMS_SQL.
8–9	Parse the dynamic query after concatenating in the only variable element, the variable i.
10	Execute the query.
11	Close the cursor.

This is all valid (and, of course, you would usually follow up the execution of the query with fetch and retrieve steps), yet it also is a misuse of DBMS_SQL. Consider the following rewrite of the same steps:

```
DECLARE
   l_cursor   PLS_INTEGER;
   l_result   PLS_INTEGER;
BEGIN
   l_cursor := DBMS_SQL.open_cursor;
   DBMS_SQL.parse (l_cursor, 'SELECT ... WHERE col = :value'
                 , DBMS_SQL.native);

   FOR i IN 1 .. counter
   LOOP
      DBMS_SQL.bind_variable (l_cursor, 'value', i);
      l_result := DBMS_SQL.EXECUTE (l_cursor);
   END LOOP;

   DBMS_SQL.close_cursor (l_cursor);
END;
```

In this usage of DBMS_SQL, I now declare the cursor only once, because I can reuse the same cursor with each call to DBMS_SQL.PARSE. I also move the parse call *outside of the cursor*. Because the structure of the SQL statement itself doesn't change, I don't need to reparse for each new value of i. So I parse once and then, within the loop, bind a new variable value into the cursor, and execute. When I am all done (after the loop terminates), I close the cursor.

The ability to perform each step explicitly and separately gives developers enormous flexibility (and also headaches from all the code and complexity of DBMS_SQL). If that is what you need, DBMS_SQL is hard to beat.

If you do use DBMS_SQL in your application, I encourage you to take advantage of the package found in the *dynalloc.pkg* file on the book's website. This "dynamic allocation" package helps you to:

- Minimize cursor allocation through cursor reuse.
- Perform tight and useful error handling for all DBMS_SQL parse operations.
- Avoid errors trying to open or close cursors that are already opened or closed.

Oracle Database 11g New Dynamic SQL Features

Oracle Database 11*g* added interoperability between native dynamic SQL and DBMS_SQL: you can now take advantage of the best features of each of these approaches to obtain the best performance with the simplest implementation. Specifically, you can now convert a DBMS_SQL cursor to a cursor variable, and vice versa, as I describe in the following sections.

DBMS_SQL.TO_REFCURSOR function

Use the DBMS_SQL.TO_REFCURSOR function to convert a cursor *number* (obtained through a call to DBMS_SQL.OPEN_CURSOR) to a weakly typed cursor variable (declared with the SYS_REFCURSOR type or a weak REF CURSOR type of your own). You can then fetch data from this cursor variable into local variables, or even pass the cursor variable to a non-PL/SQL host environment for data retrieval, having hidden all the complexities of the dynamic SQL processing in the backend.

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 the cursor, you may not use DBMS_SQL any longer to manipulate that cursor, including to closing the cursor. All operations must be done through the cursor variable.

Why would you want to use this function? As noted in previous sections, DBMS_SQL is sometimes the preferred or only option for certain dynamic SQL operations, in particular method 4. Suppose I have a situation in which I know the specific columns that I am selecting, but the WHERE clause of the query has an unknown (at compile time) number of bind variables. I cannot use EXECUTE IMMEDIATE to execute the dynamic query because of this (it has a fixed USING clause).

I *could* use DBMS_SQL from start to finish, but using DBMS_SQL to retrieve rows and values from within the rows involves an onerous amount of work. I could make it easier with bulk processing in DBMS_SQL, but it's so much easier to use a regular old static fetch and even BULK COLLECT.

The following example demonstrates precisely this scenario:

```
/* File on web: 11g_to_refcursor.sql */
DECLARE
   TYPE strings_t IS TABLE OF VARCHAR2 (200);

   l_cv             sys_refcursor;
   l_placeholders   strings_t     := strings_t ('dept_id');
   l_values         strings_t     := strings_t ('20');
   l_names          strings_t;

   FUNCTION employee_names (
      where_in            IN   VARCHAR2
    , bind_variables_in   IN   strings_t
    , placeholders_in     IN   strings_t
   )
      RETURN sys_refcursor
   IS
      l_dyn_cursor   NUMBER;
      l_cv           sys_refcursor;
      l_dummy        PLS_INTEGER;
   BEGIN
      /* Parse the retrieval of last names after appending the WHERE clause.

      NOTE: if you ever write code like this yourself, you MUST take steps
      to minimize the risk of SQL injection. This topic is also covered in
      this chapter. READ IT!
      */
      l_dyn_cursor := DBMS_SQL.open_cursor;
      DBMS_SQL.parse (l_dyn_cursor
                    , 'SELECT last_name FROM employees WHERE ' || where_in
                    , DBMS_SQL.native
                     );
      /*
         Bind each of the variables to the named placeholders.
         You cannot use EXECUTE IMMEDIATE for this step if you have
         a variable number of placeholders!
      */
      FOR indx IN 1 .. placeholders_in.COUNT
      LOOP
         DBMS_SQL.bind_variable (l_dyn_cursor
                               , placeholders_in (indx)
                               , bind_variables_in (indx)
                                );
      END LOOP;
      /*
      Execute the query now that all variables are bound.
      */
      l_dummy := DBMS_SQL.EXECUTE (l_dyn_cursor);
      /*
      Now it's time to convert to a cursor variable so that the frontend
      program or another PL/SQL program can easily fetch the values.
      */
      l_cv := DBMS_SQL.to_refcursor (l_dyn_cursor);
      /*
      Do not close with DBMS_SQL; you can ONLY manipulate the cursor
      through the cursor variable at this point.
      DBMS_SQL.close_cursor (l_dyn_cursor);
      */
      RETURN l_cv;
   END employee_names;
BEGIN
   l_cv := employee_names ('DEPARTMENT_ID = :dept_id', l_values, l_placeholders);

   FETCH l_cv BULK COLLECT INTO l_names;

   CLOSE l_cv;

   FOR indx IN 1 .. l_names.COUNT
   LOOP
      DBMS_OUTPUT.put_line (l_names(indx));
   END LOOP;
```

```
END;
/
```

Another example of a scenario in which this function will come in handy is when you need to execute dynamic SQL that requires DBMS_SQL, but then you need to pass the result set back to the middle-tier client (as with Java or .NET-based applications). You cannot pass back a DBMS_SQL cursor, but you definitely can return a cursor variable.

DBMS_SQL.TO_CURSOR function

Use the DBMS_SQL.TO_CURSOR function to convert a REF CURSOR variable (either strongly or weakly typed) to a SQL cursor number, which you can then pass to DBMS_SQL subprograms. The cursor variable must already have been opened before you can pass it to the DBMS_SQL.TO_CURSOR function.

After you convert the cursor variable to a DBMS_SQL cursor, you will not be able to use native dynamic SQL operations to access that cursor or the data "behind" it.

This function comes in handy when you know at compile time how many variables you need to bind into the SQL statement, but you don't know how many items you are selecting (another example of method 4 dynamic SQL!).

The following procedure demonstrates this application of the function:

```
/* File on web: 11g_to_cursorid.sql */
PROCEDURE show_data (
   column_list_in          VARCHAR2
 , department_id_in   IN   employees.department_id%TYPE
)
IS
   sql_stmt   CLOB;
   src_cur    SYS_REFCURSOR;
   curid      NUMBER;
   desctab    DBMS_SQL.desc_tab;
   colcnt     NUMBER;
   namevar    VARCHAR2 (50);
   numvar     NUMBER;
   datevar    DATE;
   empno      NUMBER            := 100;
BEGIN
   /* Construct the query, embedding the list of columns to be selected,
      with a single bind variable.

      NOTE: this kind of concatenation leaves you vulnerable to SQL injection!
      Please read the section in this chapter on minimizing the dangers of code
      injection so that you can make sure your application is not vulnerable.
   */
   sql_stmt :=
         'SELECT '
      || column_list_in
      || ' FROM employees WHERE department_id = :dept_id';
```

```
/* Open the cursor variable for this query, binding in the single value.
   MUCH EASIER than using DBMS_SQL for the same operations!
*/
OPEN src_cur FOR sql_stmt USING department_id_in;

/*
To fetch the data, however, I can no longer use the cursor variable,
since the number of elements fetched is unknown at complile time.

This is, however, a perfect fit for DBMS_SQL and the DESCRIBE_COLUMNS
procedure, so convert the cursor variable to a DBMS_SQL cursor number
and then take the necessary, if tedious, steps.
*/
curid := DBMS_SQL.to_cursor_number (src_cur);
DBMS_SQL.describe_columns (curid, colcnt, desctab);

FOR indx IN 1 .. colcnt
LOOP
   IF desctab (indx).col_type = 2
   THEN
      DBMS_SQL.define_column (curid, indx, numvar);
   ELSIF desctab (indx).col_type = 12
   THEN
      DBMS_SQL.define_column (curid, indx, datevar);
   ELSE
      DBMS_SQL.define_column (curid, indx, namevar, 100);
   END IF;
END LOOP;

WHILE DBMS_SQL.fetch_rows (curid) > 0
LOOP
   FOR indx IN 1 .. colcnt
   LOOP
      DBMS_OUTPUT.put_line (desctab (indx).col_name || ' = ');

      IF desctab (indx).col_type = 2
      THEN
         DBMS_SQL.COLUMN_VALUE (curid, indx, numvar);
         DBMS_OUTPUT.put_line ('   ' || numvar);
      ELSIF desctab (indx).col_type = 12
      THEN
         DBMS_SQL.COLUMN_VALUE (curid, indx, datevar);
         DBMS_OUTPUT.put_line ('   ' || datevar);
      ELSE /* Assume a string. */
         DBMS_SQL.COLUMN_VALUE (curid, indx, namevar);
         DBMS_OUTPUT.put_line ('   ' || namevar);
      ELSEND IF;
   END LOOP;
END LOOP;
```

```
      DBMS_SQL.close_cursor (curid);
   END;
```

Enhanced Security for DBMS_SQL

In 2006, security specialists identified a new class of vulnerability in which a program that uses DBMS_SQL and raises an exception allows an attacker to use the unclosed cursor to compromise the security of the database.

Oracle Database 11*g* introduced three security-related changes to DBMS_SQL to guard against this kind of attack:

- Generation of unpredictable, probably randomized, cursor numbers
- Restriction of the use of the DBMS_SQL package whenever an invalid cursor number is passed to a DBMS_SQL program
- Rejection of a DBMS_SQL operation when the user attempting to use the cursor is not the same user who opened the cursor

Unpredictable cursor numbers

Prior to Oracle Database 11*g*, calls to DBMS_SQL.OPEN_CURSOR returned a sequentially incremented number, usually between 1 and 300. This predictability could allow an attacker to iterate through integers and test them as valid, open cursors. Once found, such a cursor could be repurposed and used by the attacker.

Now, it will be very difficult for an attacker to find a valid cursor through iteration. Here, for example, are five cursor numbers returned by OPEN_CURSOR in this block:

```
SQL> BEGIN
  2     FOR indx IN 1 .. 5
  3     LOOP
  4        DBMS_OUTPUT.put_line (DBMS_SQL.open_cursor ());
  5     END LOOP;
  6  END;
  7  /
1693551900
1514010786
1570905132
182110745
1684406543
```

Denial of access to DBMS_SQL when bad cursor number is used (ORA-24971)

To guard against an attacker "fishing" for a valid cursor, the Oracle database will now deny access to the DBMS_SQL package as soon as an attempt is made to work with an invalid cursor number.

Consider the following block:

```
/* File on web: 11g_access_denied_1.sql */
DECLARE
   l_cursor     NUMBER;
   l_feedback   NUMBER;

   PROCEDURE set_salary
   IS
   BEGIN
      DBMS_OUTPUT.put_line ('Set salary = salary...');
      l_cursor := DBMS_SQL.open_cursor ();
      DBMS_SQL.parse (l_cursor
                    , 'update employees set salary = salary'
                    , DBMS_SQL.native
                     );
      l_feedback := DBMS_SQL.EXECUTE (l_cursor);
      DBMS_OUTPUT.put_line ('   Rows modified = ' || l_feedback);
      DBMS_SQL.close_cursor (l_cursor);
   END set_salary;
BEGIN
   set_salary ();

   BEGIN
      l_feedback := DBMS_SQL.EXECUTE (1010101010);
   EXCEPTION
      WHEN OTHERS
      THEN
         DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack ());
         DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace ());
   END;

   set_salary ();
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack ());
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace ());
END;
```

Here, I execute a valid UPDATE statement, setting salary to itself for all rows in the employees table, within the set_salary local procedure. I call that procedure and then attempt to execute an invalid cursor. Then I call set_salary again. Here are the results from running this block:

```
Set salary = salary...
   Rows modified = 106

ORA-29471: DBMS_SQL access denied
ORA-06512: at "SYS.DBMS_SQL", line 1501
ORA-06512: at line 22
```

```
Set salary = salary...
ORA-29471: DBMS_SQL access denied
ORA-06512: at "SYS.DBMS_SQL", line 980
ORA-06512: at line 9
ORA-06512: at line 30
```

The set_salary procedure worked the first time, but once I tried to execute an invalid cursor, an ORA-29471 error was raised when I tried to run the set_salary program again. In fact, any attempt to call a DBMS_SQL program will raise that error.

The only way to reenable access to DBMS_SQL is by logging off and logging back on. Rather severe! But that makes sense, given the possibly dangerous nature of the situation that resulted in this error.

The database will also deny access to DBMS_SQL if the program in which you opened the cursor raised an exception (not necessarily related to the dynamic SQL). If you "swallow" that error (i.e., do not re-raise the exception), then it can be quite difficult to determine the source of the error.

Rejection of DBMS_SQL operation when effective user changes (ORA-24970)

Oracle Database 11*g* provides a new overloading of the OPEN_CURSOR function that accepts an argument as follows:

```
DBMS_SQL.OPEN_CURSOR (security_level IN INTEGER) RETURN INTEGER;
```

This function allows you to specify security protection that Oracle enforces on the opened cursor when you perform operations on that cursor. Here are the security levels that the database currently recognizes:

0
: Turns off security checks for DBMS_SQL operations on this cursor. This means you can fetch from the cursor and rebind and reexecute the cursor with a different effective userid or roles than those in effect at the time the cursor was first parsed. This level of security is *not enabled by default*.

1
: Requires that the effective userid and roles of the caller to DBMS_SQL for bind and execute operations on this cursor be the same as those of the caller of the most recent parse operation on this cursor.

2
: Requires that the effective userid and roles of the caller to DBMS_SQL for all bind, execute, define, describe, and fetch operations on this cursor be the same as those of the caller of the most recent parse operation on this cursor.

Here is an example of how you might encounter the error caused by Oracle's new security check:

1. Create the user_cursor procedure in the HR schema, as shown here. Note that this is a definer rights program, meaning that when another schema calls this program, the current or effective user is HR. Open a cursor and parse a query against ALL_SOURCE with this cursor. Then return the DBMS_SQL cursor number as an OUT argument:

```
/* File on web: 11g_effective_user_id.sql */
PROCEDURE user_cursor (
   security_level_in   IN       PLS_INTEGER
 , cursor_out          IN OUT   NUMBER
)
AUTHID DEFINER
IS
BEGIN
   cursor_out := DBMS_SQL.open_cursor (security_level_in);
   DBMS_SQL.parse (cursor_out
                 , 'select count(*) from all_source'
                 , DBMS_SQL.native
                  );
END;
```

2. Grant the ability to run this program to SCOTT:

```
GRANT EXECUTE ON use_cursor TO scott
```

3. Connect to SCOTT. Then run HR's use_cursor program, specifying level 2 security, and retrieve the dynamic SQL cursor. Finally, try to execute that cursor from the SCOTT schema:

```
SQL> DECLARE
  2     l_cursor   NUMBER;
  3     l_feedback number;
  4  BEGIN
  5     hr.use_cursor (2, l_cursor);
  6     l_feedback := DBMS_SQL.execute_and_fetch (l_cursor);
  7  END;
  8  /
DECLARE
*
ERROR at line 1:
ORA-29470: Effective userid or roles are not the same as when cursor was parsed
ORA-06512: at "SYS.DBMS_SQL", line 1513
ORA-06512: at line 6
```

Oracle raises the ORA-29470 error because the cursor was opened and parsed under the HR schema (as a result of the AUTHID DEFINER clause) but executed under the SCOTT schema.