**Working with Cursors**

**Part 12 in a series of articles on understanding and using PL/SQL**

The central purpose of the Oracle PL/SQL language is to make it as easy and efficient as possible to query and change the contents of tables in a database. You must, of course, use the SQL language to access tables, and each time you do so, you use a *cursor* to get the job done. A cursor is a pointer to a private SQL area that stores information about the processing of a SELECT or data manipulation language (DML) statement (INSERT, UPDATE, DELETE, or MERGE). Cursor management of DML statements is handled by Oracle Database, but PL/SQL offers several ways to define and manipulate cursors to execute SELECT statements. This article focuses on the most-common ways programmers execute SELECT statements in PL/SQL, namely

* Using the SELECT-INTO statement
* Fetching from an explicit cursor
* Using a cursor FOR loop
* Using EXECUTE IMMEDIATE INTO for dynamic queries
* Using cursor variables

At the end of the article, I offer some quick tips to help you figure out which of these techniques you should use for different scenarios.

**SELECT-INTO**

SELECT-INTO offers the fastest and simplest way to fetch a single row from a SELECT statement. The syntax of this statement is

|  |
| --- |
| **Answer to the Challenge**  The PL/SQL Challenge question in last issue’s “Wrap Up Your Code in a Neat Package” article focused on how the values assigned to package-level variables persist in your session and asked, “Which of the choices will display ‘3’ after execution?” In the print version of the article, only choice a was correct. In the online version (which also matches the quiz offered at [PL/SQL Challenge](http://www.plsqlchallenge.com/), both a and b are correct. Visit PL/SQL Challenge to read a complete explanation of the answers to this quiz. |

*SELECT select\_list INTO variable\_list FROM remainder\_of\_query;*

*where remainder\_of\_query* contains the list of tables or views, the WHERE clause, and other clauses of the query. The number and types of elements in the *variable\_list* must match those of the *select\_list*.

If the SELECT statement identifies more than one row to be fetched, Oracle Database will raise the TOO\_MANY\_ROWS exception. If the statement doesn’t identify any rows to be fetched, Oracle Database will raise the NO\_DATA\_FOUND exception.

Here are some examples of using SELECT-INTO:

Get the last name for a specific employee ID (the primary key in the employees table):

DECLARE

l\_last\_name employees.last\_name%TYPE;

BEGIN

SELECT last\_name

INTO l\_last\_name

FROM employees

WHERE employee\_id = 138;

DBMS\_OUTPUT.put\_line (

l\_last\_name);

END;

If there is a row in the employees table with ID 138, this block will display the last name of that employee. If there is no such row, the block will fail with an unhandled NO\_DATA\_FOUND exception. Assuming that a unique index is defined on the employee\_id column, this block will never raise the TOO\_MANY\_ROWS exception.

Fetch an entire row from the employees table for a specific employee ID:

DECLARE

l\_employee employees%ROWTYPE;

BEGIN

SELECT \*

INTO l\_employee

FROM employees

WHERE employee\_id = 138;

DBMS\_OUTPUT.put\_line (

l\_employee.last\_name);

END;

Again, if an employee exists for that ID, the last name will be displayed. In this case, I declare a record based on the employees table and fetch all columns (with a SELECT \*) into that record for the specified row.

Fetch columns from different tables:

DECLARE

l\_last\_name

employees.last\_name%TYPE;

l\_department\_name

departments.department\_name%TYPE;

BEGIN

SELECT last\_name, department\_name

INTO l\_last\_name, l\_department\_name

FROM employees e, departments d

WHERE e.department\_id=d.department\_id

AND e.employee\_id=138;

DBMS\_OUTPUT.put\_line (

l\_last\_name ||

' in ' ||

l\_department\_name);

END;

In this case, I need more than one column value but not *all* the column values in either or both of the tables. So I declare two variables and fetch the two column values into those variables.

What happens if the list of variables in the INTO clause does not match the SELECT list of the query? You will see one of the error messages shown in Table 1.

|  |  |
| --- | --- |
| ORA-00947: not enough values | The INTO list contains fewer variables than the SELECT list. |
| ORA-00913: too many values | The INTO list contains more variables than the SELECT list. |
| ORA-06502: PL/SQL: numeric or value error | The number of variables in the INTO and SELECT lists matches, but the datatypes do not match and Oracle Database was unable to convert implicitly from one type to the other. |

**Table 1:** Possible error messages if INTO and SELECT lists do not match

**Fetching from Explicit Cursors**

A SELECT-INTO is also referred to as an implicit query, because Oracle Database implicitly opens a cursor for the SELECT statement, fetches the row, and then closes the cursor when it finishes doing that (or when an exception is raised).

You can, alternatively, explicitly declare a cursor and then perform the open, fetch, and close operations yourself.

Suppose I need to write a block that fetches employees in ascending salary order and gives them a bonus from a total pool of funds by calling the assign\_bonus procedure, whose header is

PROCEDURE assign\_bonus (

employee\_id\_in IN

employees.employee\_id%TYPE,

bonus\_pool\_io IN OUT INTEGER)

Each time assign\_bonus is called, the procedure subtracts the bonus given from the total and returns that reduced total. When that bonus pool is exhausted, it stops fetching and commits all changes.

Listing 1 includes a block that uses an explicit cursor to implement this logic, and it describes the operations in the block at specified line numbers.

**Code Listing 1:** Block and description of explicit cursor implementation

1 DECLARE

2 l\_total INTEGER := 10000;

3

4 CURSOR employee\_id\_cur

5 IS

6 SELECT employee\_id

7 FROM plch\_employees

8 ORDER BY salary ASC;

9

10 l\_employee\_id employee\_id\_cur%ROWTYPE;

11 BEGIN

12 OPEN employee\_id\_cur;

13

14 LOOP

15 FETCH employee\_id\_cur INTO l\_employee\_id;

16 EXIT WHEN employee\_id\_cur%NOTFOUND;

17

18 assign\_bonus (l\_employee\_id, l\_total);

19 EXIT WHEN l\_total <= 0;

20 END LOOP;

21

22 CLOSE employees\_cur;

23 END;

|  |  |
| --- | --- |
| **Line(s) Description** | |
| 4–8 | The explicit cursor declaration. Move the query from the executable section (where the SELECT-INTO must reside), and use the CURSOR keyword to declare (give a name to) that query. |
| 10 | Declare a record based on the row of data returned by the query. In this case, there is just a single column value, so you could just as easily have declared l\_employee\_id as employees.employee\_id%TYPE. But whenever you use an explicit cursor, it is best to declare a record by using %ROWTYPE, so that if the SELECT list of the cursor ever changes, that variable will change with it. |
| 12 | Open the cursor, so that rows can now be fetched from the query. Note: This is a step Oracle Database performs with the SELECT-INTO statement. |
| 14 | Start a loop to fetch rows. |
| 15 | Fetch the next row for the cursor, and deposit that row’s information into the record specified in the INTO clause.  Note: This is a step Oracle Database performs with the SELECT-INTO statement. |
| 16 | If the FETCH does not find a row, exit the loop. |
| 18 | Call assign\_bonus, which applies the bonus and also decrements the value of the l\_total variable by that bonus amount. |
| 19 | Exit the loop if all the bonus funds have been exhausted. |
| 22 | Close the cursor.  Note: This is a step Oracle Database performs with the SELECT-INTO statement. |

Here are some things to keep in mind when working with explicit cursors:

* If the query does not identify any rows, Oracle Database will *not* raise NO\_DATA\_FOUND. Instead, the cursor\_name%NOTFOUND attribute will return TRUE.
* Your query can return more than one row, and Oracle Database will not raise TOO\_MANY\_ROWS.
* When you declare a cursor in a package (that is, *not* inside a subprogram of the package) and the cursor is opened, it will stay open until you explicitly close it or your session is terminated.
* When the cursor is declared in a declaration section (and not in a package), Oracle Database will also automatically close it when the block in which it is declared terminates. It is still, however, a good idea to explicitly close the cursor yourself. If the cursor is moved to a package, you will have the *now necessary* CLOSE already in place. And if it is local, then including a CLOSE statement will also show other developers and your manager that you are paying attention.

**Using the Cursor FOR Loop**

The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. With a numeric FOR loop, the body of the loop executes once for every integer value between the low and high values specified in the range. With a cursor FOR loop, the body of the loop is executed for each row returned by the query.

The following block uses a cursor FOR loop to display the last names of all employees in department 10:

BEGIN

FOR employee\_rec IN (

SELECT \*

FROM employees

WHERE department\_id = 10)

LOOP

DBMS\_OUTPUT.put\_line (

employee\_rec.last\_name);

END LOOP;

END;

You can also use a cursor FOR loop with an explicitly declared cursor:

DECLARE

CURSOR employees\_in\_10\_cur

IS

SELECT \*

FROM employees

WHERE department\_id = 10;

BEGIN

FOR employee\_rec

IN employees\_in\_10\_cur

LOOP

DBMS\_OUTPUT.put\_line (

employee\_rec.last\_name);

END LOOP;

END;

The nice thing about the cursor FOR loop is that Oracle Database opens the cursor, declares a record by using %ROWTYPE against the cursor, fetches each row into a record, and then closes the loop when all the rows have been fetched (or the loop terminates for any other reason).

Best of all, Oracle Database automatically optimizes cursor FOR loops to perform similarly to BULK COLLECT queries (covered in “[Bulk Processing with BULK COLLECT and FORALL](http://www.oracle.com/technetwork/issue-archive/2012/12-sep/o52plsql-1709862.html),” in the September/October 2012 issue of *Oracle Magazine*). So even though your code looks as if you are fetching one row at a time, Oracle Database will actually fetch 100 rows at a time—and enable you to work with each row individually.

**Dynamic Queries with EXECUTE IMMEDIATE**

Dynamic SQL means that at the time you write (and then compile) your code, you do not have all the information you need for parsing a SQL statement. Instead, you must wait for runtime to complete the SQL statement and then parse and execute it.

Oracle Database makes it easy to execute SQL statements (and PL/SQL blocks) dynamically with the EXECUTE IMMEDIATE statement. And querying data is the easiest dynamic SQL operation of all!

You can fetch a single row or multiple rows. Here is a generic function that fetches the value of a numeric column in any table, for the specified WHERE clause:

CREATE OR REPLACE FUNCTION

single\_number\_value (

table\_in IN VARCHAR2,

column\_in IN VARCHAR2,

where\_in IN VARCHAR2)

RETURN NUMBER

IS

l\_return NUMBER;

BEGIN

EXECUTE IMMEDIATE

'SELECT '

|| column\_in

|| ' FROM '

|| table\_in

|| ' WHERE '

|| where\_in

INTO l\_return;

RETURN l\_return;

END;

As you can see, instead of SELECT-INTO, I use EXECUTE IMMEDIATE-INTO and construct the SELECT statement from the arguments passed to the function. Here’s an example of calling the function:

BEGIN

DBMS\_OUTPUT.put\_line (

single\_number\_value (

'employees',

'salary',

'employee\_id=138'));

END;

As with SELECT-INTO, EXECUTE IMMEDIATE-INTO will raise NO\_DATA\_FOUND if no rows are found and TOO\_MANY\_ROWS if more than one row is found.

You can also use EXECUTE IMMEDIATE to fetch multiple rows of data, which means that you will populate a collection, so you must use BULK COLLECT. The following is a procedure that will display the values of any numeric column for all rows specified in the WHERE clause:

CREATE OR REPLACE PROCEDURE

show\_number\_values (

table\_in IN VARCHAR2,

column\_in IN VARCHAR2,

where\_in IN VARCHAR2)

IS

TYPE values\_t IS TABLE OF NUMBER;

l\_values values\_t;

BEGIN

EXECUTE IMMEDIATE

'SELECT '

|| column\_in

|| ' FROM '

|| table\_in

|| ' WHERE '

|| where\_in

BULK COLLECT INTO l\_values;

FOR indx IN 1 .. l\_values.COUNT

LOOP

DBMS\_OUTPUT.put\_line

(l\_values (indx));

END LOOP;

END;

And when I call the procedure for the standard employees table

BEGIN

show\_number\_values (

'employees',

'salary',

'department\_id = 10

order by salary desc');

END;

I see the following two rows of output:

4400

3200

A general note of caution regarding dynamic SQL and the preceding examples in particular: whenever you concatenate text to execute a dynamically executed statement, you run the risk of *SQL injection*. This occurs when a malicious user “injects,” or inserts into the statement, code that changes the behavior of that SQL statement.

For advice on avoiding SQL injection–based security breaches, check out “[How to Write SQL Injection Proof PL/SQL](http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf)".

**Cursor Variables**

A cursor variable is, as you might guess from its name, a variable that points to a cursor or a result set. Unlike with an explicit cursor, you can pass a cursor variable as an argument to a procedure or a function. There are several excellent use cases for cursor variables, including the following:

* Pass a cursor variable back to the host environment that called the program unit—the result set can be “consumed” for display or other processing.
* Construct a result set inside a function, and return a cursor variable to that set. This is especially handy when you need to use PL/SQL, in addition to SQL, to build the result set.
* Pass a cursor variable to a pipelined table function—a powerful but quite advanced optimization technique. A full explanation of cursor variables, including the differences between strong and weak REF CURSOR types, is beyond the scope of this article.
* Instead, I will show the basic syntax for working with cursor variables and identify situations in which you might consider using this feature.

Cursor variables can be used with either embedded (static) or dynamic SQL. Listing 2 includes the names\_for function, which returns a cursor variable that fetches either employee or department names, depending on the argument passed to the function.

**Code Listing 2:**Block and description of the names\_for function, which returns a cursor variable

1 CREATE OR REPLACE FUNCTION names\_for (

2 name\_type\_in IN VARCHAR2)

3 RETURN SYS\_REFCURSOR

4 IS

5 l\_return SYS\_REFCURSOR;

6 BEGIN

7 CASE name\_type\_in

8 WHEN 'EMP'

9 THEN

10 OPEN l\_return FOR

11 SELECT last\_name

12 FROM employees

13 ORDER BY employee\_id;

14 WHEN 'DEPT'

15 THEN

16 OPEN l\_return FOR

17 SELECT department\_name

18 FROM departments

19 ORDER BY department\_id;

20 END CASE;

21

22 RETURN l\_return;

23 END names\_for;

|  |  |
| --- | --- |
| **Line(s) Description** | |
|  |  |
| 3 | Return a piece of data whose type is SYS\_REFCURSOR. |
| 5 | Declare a cursor variable to be returned by the function. |
| 7 | Use a CASE statement driven by the value of name\_type\_in to determine which query should be opened. |
| 10–13 | Open a cursor variable for a query from the employees table. |
| 16–19 | Open a cursor variable for a query from the departments table. |

Here is a block that uses the names\_for function to display all the names in the departments table:

DECLARE

l\_names SYS\_REFCURSOR;

l\_name VARCHAR2 (32767);

BEGIN

l\_names := names\_for ('DEPT');

LOOP

FETCH l\_names INTO l\_name;

EXIT WHEN l\_names%NOTFOUND;

DBMS\_OUTPUT.put\_line (l\_name);

END LOOP;

CLOSE l\_names;

END;

As you can see, all the information about the query being opened is “hidden” behind the function header. You simply ask to get the “names for” a given table. The function picks the appropriate SELECT statement, opens the cursor variable for that statement, and then returns the variable pointing to that result set.

Once the cursor variable has been opened and passed back to the block, I use the same code with a cursor variable that I would use with an explicit cursor:

1. FETCH from the cursor (variable) INTO one or more variables (I can even FETCH-BULK COLLECT INTO with a cursor variable, populating a collection with multiple rows)
2. Check the %NOTFOUND attribute of the cursor variable to see if I am done fetching all rows
3. CLOSE the cursor variable when done

The OPEN-FOR statement is unique to cursor variables and enables me to specify at runtime, without having to switch to dynamic SQL, which data set will be fetched through the cursor variable.

|  |
| --- |
| **Next Steps**  **DOWNLOAD** [**Oracle Database 11*g***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html)  **TEST**[**your PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ**[**PL/SQL 101, Parts 1-11**](http://www.oracle.com/technetwork/issue-archive/index-087690.html)  **READ more about** [**BULK COLLECT and FORALL**](http://www.oracle.com/technetwork/issue-archive/2012/12-sep/o52plsql-1709862.html) [**SQL injection**](http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf) |

Nevertheless, you *can* use OPEN-FOR with a dynamic SELECT statement. Here is a very simple example:

CREATE OR REPLACE FUNCTION

numbers\_from (

query\_in IN VARCHAR2)

RETURN SYS\_REFCURSOR

IS

l\_return SYS\_REFCURSOR;

BEGIN

OPEN l\_return FOR query\_in;

RETURN l\_return;

END numbers\_from;

And here is a block—virtually identical to the one that calls names\_for, above—that displays all the salaries for employees in department 10:

DECLARE

l\_salaries SYS\_REFCURSOR;

l\_salary NUMBER;

BEGIN

l\_salaries :=

numbers\_from (

'select salary

from employees

where department\_id = 10');

LOOP

FETCH l\_salaries INTO l\_salary;

EXIT WHEN l\_salaries%NOTFOUND;

DBMS\_OUTPUT.put\_line (l\_salary);

END LOOP;

CLOSE l\_salaries;

END;

**Choosing the Right Way to Query**

This article has shown that the PL/SQL language offers many different ways, ranging from the simplest SELECT-INTO implicit query to the much more complicated cursor variable, to use cursors to fetch data from relational tables into local variables.

Here are some guidelines to help you decide which technique to use:

* When fetching a single row, use SELECT-INTO or EXECUTE IMMEDIATE-INTO (if your query is dynamic). Do not use an explicit cursor or a cursor FOR loop.
* When fetching *all* the rows from a query, use a cursor FOR loop *unless* the body of the loop executes one or more DML statements (INSERT, UPDATE, DELETE, or MERGE). In such a case, you will want to switch to BULK COLLECT and FORALL.
* Use an explicit cursor when you need to fetch with BULK COLLECT, but *limit* the number of rows returned with each fetch.
* Use an explicit cursor when you are fetching multiple rows but might conditionally exit before all rows are fetched.
* Use a cursor variable when the query you are fetching from varies at runtime (but isn’t necessarily dynamic) and especially when you need to pass a result back to a non-PL/SQL host environment.
* Use EXECUTE IMMEDIATE to query data *only* when you cannot fully construct the SELECT statement while writing your code.

|  |
| --- |
| **Move SELECT-INTOs into Functions**  PL/SQL developers frequently need to retrieve data for a single row in a table, specified (usually) by a primary key value, and often find themselves writing the same primary key lookup again and again. A much better approach is to move each of your SELECT-INTO queries into a function whose sole purpose is to serve up the requested row. So instead of this:  DECLARE  l\_employee employees%ROWTYPE;  BEGIN  SELECT \*  INTO l\_employee  FROM employees  WHERE employee\_id = 138;  DBMS\_OUTPUT.put\_line (  l\_employee.last\_name);  END;  you would first create a function:  CREATE OR REPLACE FUNCTION row\_for\_employee\_id (  employee\_id\_in IN employees.employee\_id%TYPE)  RETURN employees%ROWTYPE  IS  l\_employee employees%ROWTYPE;  BEGIN  SELECT \*  INTO l\_employee  FROM employees e  WHERE e.employee\_id =  row\_for\_employee\_id.employee\_id\_in;  RETURN l\_employee;  EXCEPTION  WHEN NO\_DATA\_FOUND  THEN  RETURN NULL;  END;  Then the anonymous block for your primary key lookup would be DECLARE l\_employee employees%ROWTYPE; BEGIN l\_employee := row\_for\_employee\_id (138); DBMS\_OUTPUT.put\_line ( l\_employee.last\_name); END;  Best of all, the *next time* you need to get a row from the employees table for an ID, you’ll just call the function.  There are two big advantages to this approach:   1. Your productivity increases, because you can write less code and rely on prebuilt, pretested, reusable programs. 2. If you ever need to change the way you look up that single row, you’ll make the change in one place (the “single point of definition”) and all programs that call the function will immediately use the improved version.   Note that I included in the function an exception handler that traps NO\_DATA\_FOUND and simply returns a NULL record. During execution of a SELECT-INTO, the absence of data is often not actually an error but, rather, just a data condition. So it is quite common to trap the exception and return an indicator that no row was found. (NULL is usually, but not necessarily, a good indicator of this state of affairs.) The programmer who calls the function gets to decide how to treat the NO\_DATA\_FOUND condition. |

|  |
| --- |
| **Take the Challenge**  Each PL/SQL 101 article offers a quiz to test your knowledge of the information provided in it. The quiz appears below and also at [PL/SQL Challenge](http://www.plsqlchallenge.com/), a Website that offers online quizzes on the PL/SQL language as well as SQL and Oracle Application Express.    Here is your quiz for this article:  The plch\_employees table has a single-column primary key named employee\_id. I need to write a function with the following header: FUNCTION plch\_one\_employee (  employee\_id\_in IN PLS\_INTEGER)  RETURN plch\_employees%ROWTYPE  that returns a record that contains the row of information for the specified primary key. Which of the choices provides an implementation of PLCH\_ONE\_EMPLOYEE that meets this requirement?  **a.**  IS  l\_return plch\_employees%ROWTYPE;  BEGIN  SELECT \*  INTO l\_return  FROM plch\_employees  WHERE employee\_id = employee\_id\_in;  RETURN l\_return;  END plch\_one\_employee;    **b.**  IS  CURSOR one\_emp\_cur  IS  SELECT \*  FROM plch\_employees  WHERE employee\_id = employee\_id\_in;  l\_return one\_emp\_cur%ROWTYPE;  BEGIN  OPEN one\_emp\_cur;  FETCH one\_emp\_cur INTO l\_return;  CLOSE one\_emp\_cur;  RETURN l\_return;  END plch\_one\_employee;    **c.**  IS  l\_return plch\_employees%ROWTYPE;  BEGIN  FOR rec IN (SELECT \*  FROM plch\_employees  WHERE employee\_id = employee\_id\_in)  LOOP  l\_return := rec;  END LOOP;  RETURN l\_return;  END plch\_one\_employee;    **d.**  IS  l\_cursor SYS\_REFCURSOR;  l\_return plch\_employees%ROWTYPE;  BEGIN  OPEN l\_cursor FOR  SELECT \*  FROM plch\_employees  WHERE employee\_id = employee\_id\_in;  FETCH l\_cursor INTO l\_return;  CLOSE l\_cursor;  RETURN l\_return;  END plch\_one\_employee; |