**BULK COLLECT and FORALL**

**Part 9 in a series of articles on understanding and using PL/SQL**

In the previous article in this series, I introduced readers to PL/SQL collections. These data structures come in very handy when implementing algorithms that manipulate lists of program data, but they are also key elements in some of the powerful performance optimization features in PL/SQL.

In this article, I will cover the two most important of these features: BULK COLLECT and FORALL.

* BULK COLLECT: SELECT statements that retrieve multiple rows with a single fetch, improving the speed of data retrieval
* FORALL: INSERTs, UPDATEs, and DELETEs that use collections to change multiple rows of data very quickly

You may be wondering what *very quickly* might mean—how much impact do these features really have? Actual results will vary, depending on the version of Oracle Database you are running and the specifics of your application logic. You can [download](http://www.oracle.com/technetwork/issue-archive/2012/12-sep/o52plsql-1676762.zip) and run the script to compare the performance of row-by-row inserting with FORALL inserting. On my laptop running Oracle Database 11*g* Release 2, it took 4.94 seconds to insert 100,000 rows, one at a time. With FORALL, those 100,000 were inserted in 0.12 seconds. Wow!

|  |
| --- |
| **Answer to the Challenge**  The PL/SQL Challenge question in last issue’s “[Working with Collections](http://www.oracle.com/technetwork/issue-archive/2012/12-may/o32plsql-1578019.html)” article tested your knowledge of iterating through the contents of a sparsely populated collection. Choice (c) is the only correct choice, and offers the simplest algorithm for accomplishing this task:     DECLARE  l\_names DBMS\_UTILITY.maxname\_array;  BEGIN  l\_names (1) := ‘Strawberry’;  l\_names (10) := ‘Blackberry’;  l\_names (2) := ‘Raspberry’;  DECLARE  indx PLS\_INTEGER := l\_names.FIRST;  BEGIN  WHILE (indx IS NOT NULL)  LOOP  DBMS\_OUTPUT.put\_line (l\_names (indx));  indx := l\_names.NEXT (indx);  END LOOP;  END;  END;  / |

Given that PL/SQL is so tightly integrated with the SQL language, you might be wondering why special features would be needed to improve the performance of SQL statements inside PL/SQL. The explanation has everything to do with how the runtime engines for both PL/SQL and SQL communicate with each other—through a *context switch*.

**Context Switches and Performance**

Almost every program PL/SQL developers write includes both PL/SQL and SQL statements. PL/SQL statements are run by the PL/SQL statement executor; SQL statements are run by the SQL statement executor. When the PL/SQL runtime engine encounters a SQL statement, it stops and passes the SQL statement over to the SQL engine. The SQL engine executes the SQL statement and returns information back to the PL/SQL engine (see Figure 1). This transfer of control is called a context switch, and each one of these switches incurs overhead that slows down the overall performance of your programs.

|  |
| --- |
| **o52plsql-f1  Figure 1: Switching between PL/SQL and SQL engines** |

Let’s look at a concrete example to explore context switches more thoroughly and identify the reason that FORALL and BULK COLLECT can have such a dramatic impact on performance.

Suppose my manager asked me to write a procedure that accepts a department ID and a salary percentage increase and gives everyone in that department a raise by the specified percentage. Taking advantage of PL/SQL’s elegant cursor FOR loop and the ability to call SQL statements natively in PL/SQL, I come up with the code in Listing 1.

**Code Listing 1:** increase\_salary procedure with FOR loop

PROCEDURE increase\_salary (

department\_id\_in IN employees.department\_id%TYPE,

increase\_pct\_in IN NUMBER)

IS

BEGIN

FOR employee\_rec

IN (SELECT employee\_id

FROM employees

WHERE department\_id =

increase\_salary.department\_id\_in)

LOOP

UPDATE employees emp

SET emp.salary = emp.salary +

emp.salary \* increase\_salary.increase\_pct\_in

WHERE emp.employee\_id = employee\_rec.employee\_id;

END LOOP;

END increase\_salary;

Suppose there are 100 employees in department 15. When I execute this block,

BEGIN

increase\_salary (15, .10);

END;

the PL/SQL engine will “switch” over to the SQL engine 100 times, once for each row being updated. Tom Kyte, of AskTom ([asktom.oracle.com](http://asktom.oracle.com/)), refers to row-by-row switching like this as “slow-by-slow processing,” and it is definitely something to be avoided.

I will show you how you can use PL/SQL’s bulk processing features to escape from “slow-by-slow processing.” First, however, you should always check to see if it is possible to avoid the context switching between PL/SQL and SQL by doing as much of the work as possible *within SQL*.

Take another look at the increase\_salary procedure. The SELECT statement identifies all the employees in a department. The UPDATE statement executes for each of those employees, applying the same percentage increase to all. In such a simple scenario, a cursor FOR loop is not needed at all. I can simplify this procedure to nothing more than the code in Listing 2.

**Code Listing 2:** Simplified increase\_salary procedure without FOR loop

PROCEDURE increase\_salary (

department\_id\_in IN employees.department\_id%TYPE,

increase\_pct\_in IN NUMBER)

IS

BEGIN

UPDATE employees emp

SET emp.salary =

emp.salary

+ emp.salary \* increase\_salary.increase\_pct\_in

WHERE emp.department\_id =

increase\_salary.department\_id\_in;

END increase\_salary;

Now there is just a single context switch to execute one UPDATE statement. All the work is done in the SQL engine.

Of course, in most real-world scenarios, life—and code—is not so simple. We often need to perform other steps prior to execution of our data manipulation language (DML) statements. Suppose that, for example, in the case of the increase\_salary procedure, I need to check employees for eligibility for the increase in salary and if they are ineligible, send an e-mail notification. My procedure might then look like the version in Listing 3.

**Code Listing 3:** increase\_salary procedure with eligibility checking added

PROCEDURE increase\_salary (

department\_id\_in IN employees.department\_id%TYPE,

increase\_pct\_in IN NUMBER)

IS

l\_eligible BOOLEAN;

BEGIN

FOR employee\_rec

IN (SELECT employee\_id

FROM employees

WHERE department\_id =

increase\_salary.department\_id\_in)

LOOP

check\_eligibility (employee\_rec.employee\_id,

increase\_pct\_in,

l\_eligible);

IF l\_eligible

THEN

UPDATE employees emp

SET emp.salary =

emp.salary

+ emp.salary

\* increase\_salary.increase\_pct\_in

WHERE emp.employee\_id = employee\_rec.employee\_id;

END IF;

END LOOP;

END increase\_salary;

I can no longer do everything in SQL, so am I then resigned to the fate of “slow-by-slow processing”? Not with BULK COLLECT and FORALL in PL/SQL.

**Bulk Processing in PL/SQL**

The bulk processing features of PL/SQL are designed specifically to reduce the number of context switches required to communicate from the PL/SQL engine to the SQL engine.

Use the BULK COLLECT clause to fetch multiple rows into one or more collections with a single context switch.

Use the FORALL statement when you need to execute the same DML statement repeatedly for different bind variable values. The UPDATE statement in the increase\_salary procedure fits this scenario; the only thing that changes with each new execution of the statement is the employee ID.

I will use the code in Listing 4 to explain how these features affect context switches and how you will need to change your code to take advantage of them.

**Code Listing 4:** Bulk processing for the increase\_salary procedure

1 CREATE OR REPLACE PROCEDURE increase\_salary (

2 department\_id\_in IN employees.department\_id%TYPE,

3 increase\_pct\_in IN NUMBER)

4 IS

5 TYPE employee\_ids\_t IS TABLE OF employees.employee\_id%TYPE

6 INDEX BY PLS\_INTEGER;

7 l\_employee\_ids employee\_ids\_t;

8 l\_eligible\_ids employee\_ids\_t;

9

10 l\_eligible BOOLEAN;

11 BEGIN

12 SELECT employee\_id

13 BULK COLLECT INTO l\_employee\_ids

14 FROM employees

15 WHERE department\_id = increase\_salary.department\_id\_in;

16

17 FOR indx IN 1 .. l\_employee\_ids.COUNT

18 LOOP

19 check\_eligibility (l\_employee\_ids (indx),

20 increase\_pct\_in,

21 l\_eligible);

22

23 IF l\_eligible

24 THEN

25 l\_eligible\_ids (l\_eligible\_ids.COUNT + 1) :=

26 l\_employee\_ids (indx);

27 END IF;

28 END LOOP;

29

30 FORALL indx IN 1 .. l\_eligible\_ids.COUNT

31 UPDATE employees emp

32 SET emp.salary =

33 emp.salary

34 + emp.salary \* increase\_salary.increase\_pct\_in

35 WHERE emp.employee\_id = l\_eligible\_ids (indx);

36 END increase\_salary;

|  |  |
| --- | --- |
| **Lines** | **Description** |
| 5–8 | Declare a new nested table type and two collection variables based on this type. One variable, l\_employee\_ids, will hold the IDs of all employees in the department. The other, l\_eligible\_ids, will hold the IDs of all those employees who are eligible for the salary increase. |
| 12–15 | Use BULK COLLECT to fetch all the IDs of employees in the specified department into the l\_employee\_ids collection. |
| 17–28 | Check for salary increase eligibility: If ineligible, an e-mail is sent. (Note: Implementation of check\_eligibility is not included in this article.) If eligible, add the ID to the l\_eligible\_ids collection. |
| 30–35 | Use a FORALL statement to update all the rows identified by employee IDs in the l\_eligible\_ids collection. |

Listing 4 also contains an explanation of the code in this new-and-improved increase\_salary procedure. There are three phases of execution:

1. Fetch rows with BULK COLLECT into one or more collections. A single context switch is needed for this step.
2. Modify the contents of collections as required (in this case, remove ineligible employees).
3. Change the table with FORALL using the modified collections.

Rather than move back and forth between the PL/SQL and SQL engines to update each row, FORALL “bundles up” all the updates and passes them to the SQL engine with a single context switch. The result is an extraordinary boost in performance.

I will first explore BULK COLLECT in more detail, and then cover FORALL.

**About BULK COLLECT**

To take advantage of bulk processing for queries, you simply put BULK COLLECT *before* the INTO keyword and then provide one or more collections *after* the INTO keyword. Here are some things to know about how BULK COLLECT works:

* It can be used with all three types of collections: associative arrays, nested tables, and VARRAYs.
* You can fetch into individual collections (one for each expression in the SELECT list) or a single collection of records.
* The collection is always populated densely, starting from index value 1.
* If no rows are fetched, then the collection is emptied of all elements.

Listing 5 demonstrates an example of fetching values for two columns into a collection of records.

**Code Listing 5:** Fetching values for two columns into a collection

DECLARE

TYPE two\_cols\_rt IS RECORD

(

employee\_id employees.employee\_id%TYPE,

salary employees.salary%TYPE

);

TYPE employee\_info\_t IS TABLE OF two\_cols\_rt;

l\_employees employee\_info\_t;

BEGIN

SELECT employee\_id, salary

BULK COLLECT INTO l\_employees

FROM employees

WHERE department\_id = 10;

END;

If you are fetching lots of rows, the collection that is being filled could consume too much session memory and raise an error. To help you avoid such errors, Oracle Database offers a LIMIT clause for BULK COLLECT. Suppose that, for example, there could be tens of thousands of employees in a single department and my session does not have enough memory available to store 20,000 employee IDs in a collection.

Instead I use the approach in Listing 6.

**Code Listing 6:** Fetching *up to* the number of rows specified

DECLARE

c\_limit PLS\_INTEGER := 100;

CURSOR employees\_cur

IS

SELECT employee\_id

FROM employees

WHERE department\_id = department\_id\_in;

TYPE employee\_ids\_t IS TABLE OF

employees.employee\_id%TYPE;

l\_employee\_ids employee\_ids\_t;

BEGIN

OPEN employees\_cur;

LOOP

FETCH employees\_cur

BULK COLLECT INTO l\_employee\_ids

LIMIT c\_limit;

EXIT WHEN l\_employee\_ids.COUNT = 0;

END LOOP;

END;

With this approach, I open the cursor that identifies all the rows I want to fetch. Then, inside a loop, I use FETCH-BULK COLLECT-INTO to fetch *up to* the number of rows specified by the c\_limit constant (set to 100). Now, no matter how many rows I need to fetch, my session will never consume more memory than that required for those 100 rows, yet I will still benefit from the improvement in performance of bulk querying.

**About FORALL**

Whenever you execute a DML statement inside of a loop, you should convert that code to use FORALL. The performance improvement will amaze you and please your users.

The FORALL statement is *not* a loop; it is a declarative statement to the PL/SQL engine: “Generate all the DML statements that *would* have been executed one row at a time, and send them all across to the SQL engine with one context switch.”

As you can see in Listing 4, lines 30 through 35, the “header” of the FORALL statement looks just like a numeric FOR loop, yet there are no LOOP or END LOOP keywords.

Here are some things to know about FORALL:

* Each FORALL statement may contain just a single DML statement. If your loop contains two updates and a delete, then you will need to write three FORALL statements.
* PL/SQL declares the FORALL iterator (indx on line 30 in Listing 4) as an integer, just as it does with a FOR loop. You do not need to—and you should not—declare a variable with this same name.
* In at least one place in the DML statement, you need to reference a collection and use the FORALL iterator as the index value in that collection (see line 35 in Listing 4).
* When using the IN low\_value . . . high\_value syntax in the FORALL header, the collections referenced inside the FORALL statement must be densely filled. That is, every index value between the low\_value and high\_value must be defined.
* If your collection is not densely filled, you should use the INDICES OF or VALUES OF syntax in your FORALL header.

**FORALL and DML Errors**

Suppose that I’ve written a program that is supposed to insert 10,000 rows into a table. After inserting 9,000 of those rows, the 9,001st insert fails with a DUP\_VAL\_ON\_INDEX error (a unique index violation). The SQL engine passes that error back to the PL/SQL engine, and if the FORALL statement is written like the one in Listing 4, PL/SQL will terminate the FORALL statement. The remaining 999 rows will not be inserted.

If you want the PL/SQL engine to execute as many of the DML statements as possible, even if errors are raised along the way, add the SAVE EXCEPTIONS clause to the FORALL header. Then, if the SQL engine raises an error, the PL/SQL engine will save that information in a pseudocollection named SQL%BULK\_EXCEPTIONS, and continue executing statements. When all statements have been attempted, PL/SQL then raises the ORA-24381 error.

You can—and should—trap that error in the exception section and then iterate through the contents of SQL%BULK\_EXCEPTIONS to find out which errors have occurred. You can then write error information to a log table and/or attempt recovery of the DML statement.

Listing 7 contains an example of using SAVE EXCEPTIONS in a FORALL statement; in this case, I simply display on the screen the index in the l\_eligible\_ids collection on which the error occurred, and the error code that was raised by the SQL engine.

**Code Listing 7:** Using SAVE EXCEPTIONS with FORALL

BEGIN

FORALL indx IN 1 .. l\_eligible\_ids.COUNT SAVE EXCEPTIONS

UPDATE employees emp

SET emp.salary =

emp.salary + emp.salary \* increase\_pct\_in

WHERE emp.employee\_id = l\_eligible\_ids (indx);

EXCEPTION

WHEN OTHERS

THEN

IF SQLCODE = -24381

THEN

FOR indx IN 1 .. SQL%BULK\_EXCEPTIONS.COUNT

LOOP

DBMS\_OUTPUT.put\_line (

SQL%BULK\_EXCEPTIONS (indx).ERROR\_INDEX

|| ‘: ‘

|| SQL%BULK\_EXCEPTIONS (indx).ERROR\_CODE);

END LOOP;

ELSE

RAISE;

END IF;

END increase\_salary;

**From SQL to PL/SQL**

This article talks mostly about the context switch from the PL/SQL engine to the SQL engine that occurs when a SQL statement is executed from within a PL/SQL block. It is important to remember that a context switch also takes place when a user-defined PL/SQL function is invoked from within an SQL statement.

Suppose that I have written a function named betwnstr that returns the string between a start and end point. Here’s the header of the function:

FUNCTION betwnstr (

string\_in IN VARCHAR2

, start\_in IN INTEGER

, end\_in IN INTEGER

)

RETURN VARCHAR2

I can then call this function as follows:

SELECT betwnstr (last\_name, 2, 6)

FROM employees

WHERE department\_id = 10

If the employees table has 100 rows and 20 of those have department\_id set to 10, then there will be 20 context switches from SQL to PL/SQL to run this function.

You should, consequently, play close attention to all invocations of user-defined functions in SQL, especially those that occur in the WHERE clause of the statement. Consider the following query:

SELECT employee\_id

FROM employees

WHERE betwnstr (last\_name, 2, 6) = 'MITHY'

In this query, the betwnstr function will be executed 100 times—and there will be 100 context switches.

|  |
| --- |
| **Next Steps**  **DOWNLOAD** [**Oracle Database 11*g***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html) [**script for this article**](http://www.oracle.com/technetwork/issue-archive/2012/12-sep/o52plsql-1676762.zip)  **TEST**[**your PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ**[**PL/SQL 101, Parts 1-8**](http://www.oracle.com/technetwork/issue-archive/index-087690.html)  **READ more about**[**INDICES OF and VALUES OF**](http://bit.ly/MoOcgH) |

**FORALL with Sparse Collections**

If you try to use the IN *low\_value .. high\_value* syntax with FORALL and there is an undefined index value within that range, Oracle Database will raise the “ORA-22160: element at index [N] does not exist” error.

To avoid this error, you can use the INDICES OF or VALUES OF clauses. To see how these clauses can be used, let’s go back to the code in Listing 4. In this version of increase\_salary, I declare a second collection, l\_eligible\_ids, to hold the IDs of those employees who are eligible for a raise.

Instead of doing that, I can simply *remove* all ineligible IDs from the l\_employee\_ids collection, as follows:

FOR indx IN 1 .. l\_employee\_ids.COUNT

LOOP

check\_eligibility (l\_employee\_ids (indx),

increase\_pct\_in,

l\_eligible);

IF NOT l\_eligible

THEN

l\_employee\_ids.delete (indx);

END IF;

END LOOP;

But now my l\_employee\_ids collection may have gaps in it: index values that are undefined between 1 and the highest index value populated by the BULK COLLECT.

No worries. I will simply change my FORALL statement to the following:

FORALL indx IN INDICES OF l\_employee\_ids

UPDATE employees emp

SET emp.salary =

emp.salary

+ emp.salary \*

increase\_salary.increase\_pct\_in

WHERE emp.employee\_id =

l\_employee\_ids (indx);

Now I am telling the PL/SQL engine to use only those index values that are defined in l\_employee\_ids, rather than specifying a fixed range of values. Oracle Database will simply skip any undefined index values, and the ORA-22160 error will not be raised.

This is the simplest application of INDICES OF. Check the [documentation](http://bit.ly/MoOcgH) for more-complex usages of INDICES OF, as well as when and how to use VALUES OF.

**Bulk Up Your Code!**

Optimizing the performance of your code can be a difficult and time-consuming task. It can also be a relatively easy and exhilarating experience—if your code has not yet been modified to take advantage of BULK COLLECT and FORALL. In that case, you have some low-hanging fruit to pick!

|  |
| --- |
| **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.    I create and populate my employees table as follows:  CREATE TABLE plch\_employees  (     employee\_id   INTEGER,     last\_name     VARCHAR2 (100)  )  /  BEGIN     INSERT INTO plch\_employees          VALUES (100, ‘Picasso’);     INSERT INTO plch\_employees          VALUES (200, ‘Mondrian’);     INSERT INTO plch\_employees          VALUES (300, ‘O’’Keefe’);     COMMIT;  END;  /      **Question**  Which of these blocks will uppercase the last names of all employees in the table?    **a.**  DECLARE     TYPE ids\_t IS TABLE OF plch\_employees.employee\_id%TYPE;     l\_ids   ids\_t := ids\_t (100, 200, 300);  BEGIN     FORALL indx IN 1 .. l\_ids.COUNT     LOOP        UPDATE plch\_employees           SET last\_name = UPPER (last\_name)         WHERE employee\_id = l\_ids (indx);     END LOOP;  END;  /  **b.**  DECLARE     TYPE ids\_t IS TABLE OF plch\_employees.employee\_id%TYPE;     l\_ids   ids\_t := ids\_t (100, 200, 300);  BEGIN     FORALL indx IN 1 .. l\_ids.COUNT        UPDATE plch\_employees           SET last\_name = UPPER (last\_name)         WHERE employee\_id = l\_ids (indx);  END;  /  **c.**  BEGIN     UPDATE plch\_employees         SET last\_name = UPPER (last\_name);  END;  /  d.   DECLARE     TYPE ids\_t IS TABLE OF plch\_employees.employee\_id%TYPE;     l\_ids   ids\_t := ids\_t (100, 200, 300);  BEGIN     FORALL indx IN INDICES OF l\_ids        UPDATE plch\_employees           SET last\_name = UPPER (last\_name)         WHERE employee\_id = l\_ids (indx);  END;  / |