PL SQL 101, Database Developer, Database, SQL and PL SQL, DBA

Bulk data processing with BULK COLLECT and FORALL in PL/SQL

November 4, 2020 | 17 minute read

Steven Feuerstein
Developer Advocate for PL/SQL

Part 9 in a series of articles on understanding and using PL/SQL for accessing Oracle Database

PL/SQL is one of the core technologies at Oracle and is essential to leveraging the full potential of Oracle Database. PL/SQL combines the relational data access capabilities of the Structured Query Language with a flexible embedded procedural language, and it executes complex queries and programmatic logic run inside the database engine itself. This enhances the agility, efficiency, and performance of database-driven applications.

Steven Feuerstein, one of the industry's best-respected and most prolific experts in PL/SQL, wrote a 12-part tutorial series on the language. Those articles, first published in 2011, have been among the most popular ever published on the Oracle website and continue to find new readers and enthusiasts in the database community. Beginning with the first installment, the entire series is being updated and republished; please enjoy!

In the previous article in this series, "Working with collections in PL/SQL," you met collections, important data structures that come in very handy when implementing algorithms that manipulate lists of program data. Collections 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:** These are SELECT statements that retrieve multiple rows with a single fetch, thereby improving the speed of data retrieval.
>
> **FORALL:** These are INSERT, UPDATE, and DELETE operations 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 and run the script to compare the performance of row-by-row inserting with FORALL. On my laptop running Oracle Database 11g 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!

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

Nearly every PL/SQL program 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.

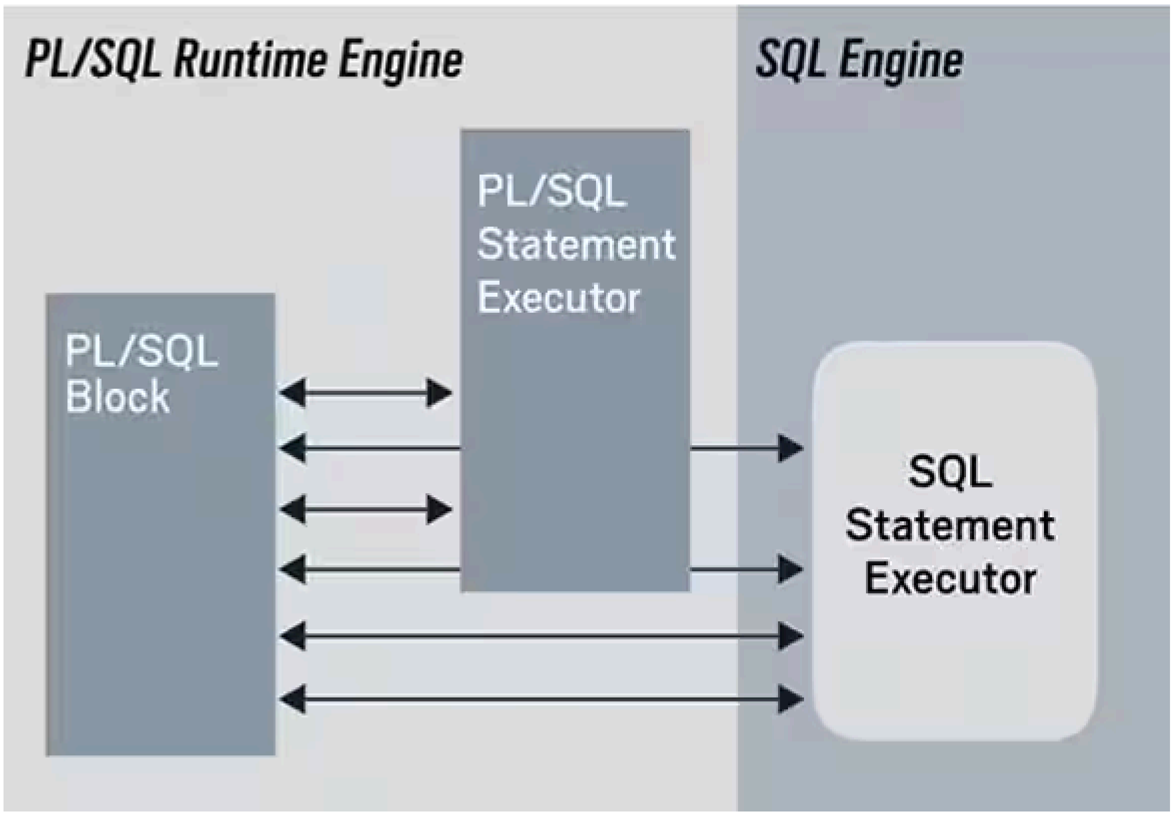

Figure 1: Switching between the PL/SQL and SQL runtime engines

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

Copy code snippet

```
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,

Copy code snippet

```
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. We might refer 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

Copy code snippet

```
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
```

```
              + 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 email notification. My procedure might then look like the version in **Listing 3**.

Code listing 3: increase_salary procedure with eligibility checking added

Copy code snippet

```
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 data 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** and the table that follows it 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

Copy code snippet

```
   CREATE OR REPLACE PROCEDURE increase_salary (
   department_id_in   IN employees.department_id%TYPE,
   increase_pct_in    IN NUMBER)
IS
   TYPE employee_ids_t IS TABLE OF employees.employee_id%TYPE
           INDEX BY PLS_INTEGER;
   l_employee_ids   employee_ids_t;
   l_eligible_ids   employee_ids_t;

   l_eligible       BOOLEAN;
BEGIN
   SELECT employee_id
     BULK COLLECT INTO l_employee_ids
     FROM employees
    WHERE department_id = increase_salary.department_id_in;

   FOR indx IN 1 .. l_employee_ids.COUNT
   LOOP
      check_eligibility (l_employee_ids (indx),
                         increase_pct_in,
                         l_eligible);

      IF l_eligible
      THEN
         l_eligible_ids (l_eligible_ids.COUNT + 1) :=
            l_employee_ids (indx);
      END IF;
   END LOOP;

   FORALL indx IN 1 .. l_eligible_ids.COUNT
      UPDATE employees emp
         SET emp.salary =
                  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 email 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.

The table above 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 I'll cover FORALL.

About BULK COLLECT

To take advantage of bulk processing for queries, 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

Copy code snippet

```
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

Copy code snippet

```
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.

Handling 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 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

Copy code snippet

```
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;
```

Context switching 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 a 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:

Copy code snippet

```
FUNCTION betwnstr (
   string_in      IN   VARCHAR2
 , start_in       IN   INTEGER
 , end_in         IN   INTEGER
)
   RETURN VARCHAR2
```

I can then call this function as follows:

Copy code snippet

```
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, pay 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:

Copy code snippet

```
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.

Using 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 that 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:

Copy code snippet