|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.08 Advanced PL/SQL |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Bulk Operations in PL/SQL 3](#_Toc383496655)

[1.1. Getting Started with BULK Fetch 3](#_Toc383496656)

[1.1.1. Implicit Cursor 3](#_Toc383496657)

[1.1.2. Explicit Fetch Calls 3](#_Toc383496658)

[1.1.3. Implicit Fetch Calls 4](#_Toc383496659)

[1.1.4. Implicit Cursor BULK Mode 4](#_Toc383496660)

[1.1.5. Explicit Fetch Calls BULK Mode 4](#_Toc383496661)

[1.1.6. Implicit Fetch Calls BULK mode 5](#_Toc383496662)

[2. FORALL Statement 5](#_Toc383496663)

[3. Source Books and Articles 6](#_Toc383496664)

# Bulk Operations in PL/SQL

Bulk processing in Oracle pre-dates PL/SQL, just via a different name, namely array processing. For as a long as I have worked with Oracle (circa version 6 in the early 1990s), the concept of array processing has been available. Rather than fetch a single row of data from the database, a set of rows is fetched into a buffer, and that buffer is passed back to the client as an array. Similarly, rather than modify or create a single row of data in the database, a buffer with an array of rows is populated and passed to the database. Application programmers could do array processing natively via OCI, or many of the popular Oracle tools of the time (such as Forms) would transparently take care of the task. You will soon see the performance and scalability benefits of such a strategy.

Presumably the PL/SQL team at Oracle faced a nomenclature problem in version 8.1 when they introduced native array processing. By that stage, the term “array” was already entrenched in PL/SQL, referring to the INDEX BY array data structure, and perhaps from this conflict, the term “bulk” arose. In fact, array processing in PL/SQL pre-dates even version 8.1; it was available via the DBMS\_SQL package way back in Oracle 7. An example of PL/SQL array processing from version 7 is presented next (I will not step through the code in detail, since I’ll be covering simpler mechanisms shortly).

Sadly, even a decade after they were introduced, the bulk operations in PL/SQL are still an underused feature in modern PL/SQL-centric applications. Many a production PL/SQL program still processes data from the database in a row-by-row fashion. In particular, because most of my work is in the tuning arena, my primary motivation for the use of collections is that it encourages the developer to think more in terms of sets, rather than rows. While there is no functional reason that should prohibit developers from processing result sets one row at a time, from an efficiency and performance perspective, it is generally bad news.

Similarly, criticism is often aimed at PL/SQL in terms of performance, but the most common cause of this is row-at-a-time processing rather than anything inherent in the PL/SQL engine. PL/SQL is not alone in misplaced criticism of this kind. Developers not taking advantage of host-based arrays in Pro\*C led Oracle to add a PREFETCH compiler option, which converts the runtime Pro\*C code into array fetching even though the developer has coded it in a conventional row-at-a-time manner. Similarly, ODP.NET defaults to array processing for most queries via its Fetch Size parameter.

## Getting Started with BULK Fetch

One of the great things about transitioning your code to take advantage of bulk operations in PL/SQL, is that it’s easy to do and has a direct mapping to your existing code. Before bulk operations arrived in version 8.1, you could use one of the following three constructs to retrieve a row of data in PL/SQL

### Implicit Cursor

A standard SQL query (SELECT-INTO) is used to retrieve a single row of data, or columns from that single row from a table into target variables. If no rows are retrieved or more than a single row is retrieved, an exception is raised. Here’s an example:

SQL> declare

2 l\_descr hardware.descr%type;

3 begin

4 select descr

5 into l\_descr

6 from hardware

7 where aisle = 1

8 and item = 1;

9 end;

10 /

PL/SQL procedure successfully completed.

### Explicit Fetch Calls

A cursor is explicitly defined within the PL/SQL declaration section. The cursor is then opened and fetched from, typically within a loop until the available rows are exhausted, at which point the cursor is closed. Here’s an example:

SQL> declare

2 cursor c\_tool\_list is

3 select descr

4 from hardware

5 where aisle = 1

6 and item between 1 and 500;

7

8 l\_descr hardware.descr%type;

9 begin

10 open c\_tool\_list;

11 loop

12 fetch c\_tool\_list into l\_descr;

13 exit when c\_tool\_list%notfound;

14 end loop;

15 close c\_tool\_list;

16 end;

17 /

PL/SQL procedure successfully completed.

### Implicit Fetch Calls

The third type is a hybrid between the two approaches. A FOR loop takes care of the cursor management, the cursor being either explicitly defined in advance or the directly coded within the FOR loop itself. Here’s an example (bulk\_implicit\_fetch\_1.sql):

SQL> begin

2 for i in (

3 select descr

4 from hardware

5 where aisle = 1

6 and item between 1 and 500 )

7 loop

8 <processing code for each row>

9 end loop;

10 end;

11 /

PL/SQL procedure successfully completed.

Converting each of those constructs to a bulk collection model is easy and straightforward. The following are the three bulk processing constructs that are the analogs of the non-bulk constructs just shown.

### Implicit Cursor BULK Mode

A standard SQL query (SELECT-INTO) can now be used to retrieve multiple rows of data into a collection type simply by adding the BULK COLLECT keywords. Here’s an example:

SQL> declare

2 type t\_descr\_list is table of hardware.descr%type;

3 l\_descr\_list t\_descr\_list;

4 begin

5 select descr

6 bulk collect

7 into l\_descr\_list

8 from hardware

9 where aisle = 1

10 and item between 1 and 100;

11 end;

12 /

PL/SQL procedure successfully completed.

### Explicit Fetch Calls BULK Mode

The only changes required are to define a collection type to hold the results and to add the BULK COLLECT clause to the FETCH command. All of the rows in the cursor results will be fetched into the collection type variable in a single call. Here’s an example:

SQL> declare

2 cursor c\_tool\_list is

3 select descr

4 from hardware

5 where aisle = 1

6 and item between 1 and 500;

7

8 type t\_descr\_list is table of c\_tool\_list%rowtype;

9 l\_descr\_list t\_descr\_list;

10

11 begin

12 open c\_tool\_list;

13 fetch c\_tool\_list bulk collect into l\_descr\_list;

14 close c\_tool\_list;

15 end;

16 /

PL/SQL procedure successfully completed.

### Implicit Fetch Calls BULK mode

Things gets even easier when converting the hybrid approach to bulk collect because there are no code changes to make. One of the best features to arrive in Oracle 10g was the “automatic bulk collect” enhancement for FOR loops. Because a FOR loop on a cursor will, by definition, fetch all of the rows in the cursor (unless an explicit EXIT command is present within the loop), the PL/SQL compiler can safely employ a bulk collect optimization to retrieve those rows as efficiently as possible. You will automatically get this optimization if you are on at least version 10 of Oracle, and the database parameter plsql\_optimize\_level is set to at least 2 (the default). The following example shows the database optimization level followed by a FOR loop that is automatically implemented for you using the bulk processing features:

SQL> select banner from v$version where rownum = 1;

BANNER

----------------------------------------------------------------------------

Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - 64bit Production

SQL> select value from v$parameter where name = 'plsql\_optimize\_level';

VALUE

----------------------------------------------------------------------------

2

SQL> begin

2 for i in (

3 select descr

4 from hardware

5 where aisle = 1

6 and item between 1 and 500 )

7 loop

8 null;

9 end loop;

10 end;

11 /

PL/SQL procedure successfully completed.

# FORALL Statement

The FORALL statement, a feature of bulk SQL, sends DML statements from PL/SQL to SQL in batches rather than one at a time. To understand the FORALL statement, first consider the FOR LOOP statement. It sends these DML statements from PL/SQL to SQL one at a time:

DELETE FROM employees\_temp WHERE department\_id = depts(10);

DELETE FROM employees\_temp WHERE department\_id = depts(30);

DELETE FROM employees\_temp WHERE department\_id = depts(70);

DROP TABLE employees\_temp;

CREATE TABLE employees\_temp AS SELECT \* FROM employees;

DECLARE

TYPE NumList IS VARRAY(20) OF NUMBER;

depts NumList := NumList(10, 30, 70); -- department numbers

BEGIN

FOR i IN depts.FIRST..depts.LAST LOOP

DELETE FROM employees\_temp

WHERE department\_id = depts(i);

END LOOP;

END;

/

DROP TABLE employees\_temp;

CREATE TABLE employees\_temp AS SELECT \* FROM employees;

DECLARE

TYPE NumList IS VARRAY(20) OF NUMBER;

depts NumList := NumList(10, 30, 70); -- department numbers

BEGIN

FORALL i IN depts.FIRST..depts.LAST

DELETE FROM employees\_temp

WHERE department\_id = depts(i);

END;

/

A FORALL statement is usually much faster than an equivalent FOR LOOP statement. However, a FOR LOOP statement can contain multiple DML statements, while a FORALL statement can contain only one. The batch of DML statements that a FORALL statement sends to SQL differ only in their VALUES and WHERE clauses. The values in those clauses must come from existing, populated collections.

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Powell G. Oracle Data Warehouse Tuning for 10g. Oxford: Elsevier Digital Press, 2005.