Table of Contents

[CHAPTER 14: ADVANCED FUNCTIONS 2](#_Toc413949575)

[Theory 2](#_Toc413949576)

[AIM 4](#_Toc413949577)

[Lab Exercise 14: ADVANCED FUNCTIONS 5](#_Toc413949578)

[1. Deterministic Function 6](#_Toc413949579)

[2. Table Function 14](#_Toc413949580)

[3. Pipelined Table Function 19](#_Toc413949581)

[SUMMARY 31](#_Toc413949582)

[REFERENCES 32](#_Toc413949583)

[INDEX 33](#_Toc413949584)

# CHAPTER 14: ADVANCED FUNCTIONS

## Theory

There are a set of advanced functions (Deterministic, Pipelined, User-Defined Aggregate Function) which may satisfy some special requirements or improve the performance.

Deterministic Function is the function that returns the same result whenever it is given the same parameters. Thus, it should not depend on the content of other database objects or on time. The basic structure of the Deterministic Function is as the following:

**CREATE FUNCTION** func\_name

(par1 PLS\_INTEGER,

par2 VARCHAR2)

RETURN NUMBER **DETERMINISTIC**

IS

.......

Deterministic Function tries to use previously calculated results when possible rather than re-executing the function. After creating the function, you must create a so-called "Function-based index". This index stores the map between inputs and output. In this tutorial, you will examine the performance benefit of using DETERMINSTIC clause.

A Table Function is a user-defined PL/SQL function that returns a collection of rows (a nested table or varray). You can select from this collection as if it were a database table by invoking the table function using TABLE clause in a SELECT statement.

SELECT \* FROM **TABLE(***collection\_name*|*cursor\_name***)**;

Thus, a Table Function can take either a cursor or a collection (VARRAY or Nested Table) as input and returns a collection or rows (table) that can be used in a SELECT statement.

A Table Function can cause a performance issue because collect all rows of the collection or a cursor and then start executing the SELECT statement. This means, if the collection is very large (say 10,000,000 rows or more), a Table Function must load these rows in memory before start fetching rows.

One of the suggested optimizing solutions of a regular Table Function performance is to use a Pipelined Table Function. A Pipelined Table Function returns a row to its invoker immediately after processing that row and continues to process rows. Thus, it returns one row after another. Memory consumption and response time improve because the entire collection need not be constructed and returned to the server before the query can return a single result row.

First, You have to create your own Pipelined Table Function and it must be either a standalone function or a package function.

**CREATE FUNCTION** pipelined\_func

(par1 PLS\_INTEGER,

par2 VARCHAR2)

RETURN collection\_name **PIPELINED**

IS

BEGIN

....

**PIPE ROW(**collection\_name(i)**)**;

...  
END;

You must use "PIPE ROW" call to put/return new row to the invoker. If the invoker no longer needs more rows, the exception "NO\_DATA\_NEEDED" raises and terminates the Pipelined Table Function.

After defining the Pipelined Table Function, you can use it in SELECT statement as shown below:

SELECT \* FROM **TABLE(***pipelined\_func(par1, par2)***)**;

Oracle provides several pre-defined aggregate functions such as MAX, MIN, and SUM for performing operations on a set of rows. These pre-defined aggregate functions can be used only with scalar data, not with complex data types such as multimedia data stored using object types, opaque types, and LOBs. You can, however, define custom implementations of these functions for complex data types. You can also define entirely new aggregate functions to use with complex data. User-defined aggregate functions can be used in SQL DML statements just like Oracle's built-in aggregates. When functions are registered with the server, Oracle simply invokes the user-defined aggregation routines supplied by you instead of the native routines.

User-defined aggregate will not be covered in this chapter. It will be covered in the Object Type chapters.

## AIM

The AIM of the following exercise is to demonstrate some enhanced functions supported by in Oracle PL/SQL.

The steps involved will include:

* Deterministic Functions
* Table Function
* Pipelined Table Function

In general, lab exercises are done in sequential order. Thus, it is assumed that you successfully completed the previous labs. However, not all previous labs are required. Please be sure to run the following lab before proceeding:

* Installing Oracle Database 12c.
* Stored Procedure
* Stored Function

Estimated Completion Time:

25 minutes

# Lab Exercise 14: ADVANCED FUNCTIONS

|  |
| --- |
|  |

## Deterministic Function

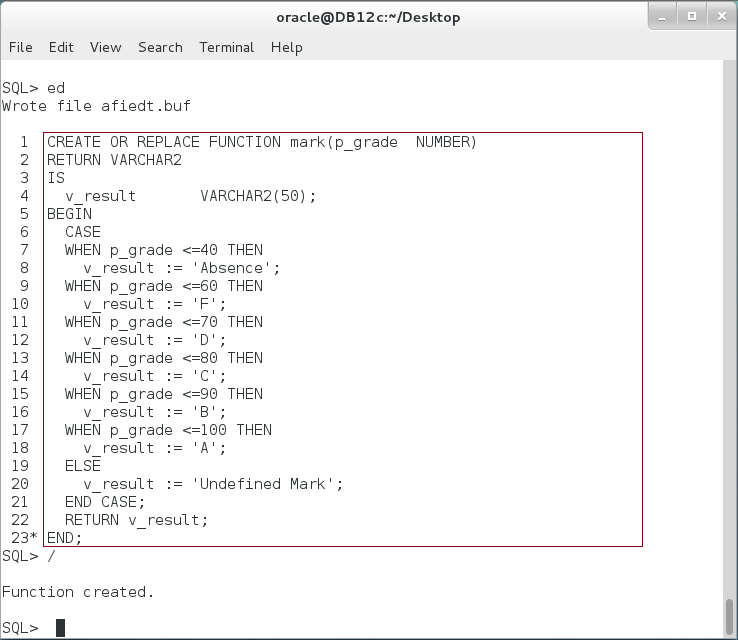
**Step 1:** Open the Terminal, open SQL\*Plus console and connect to hr schema.

|  |  |
| --- | --- |
| Command | Description |
| sqlplus | Open SQL\*Plus console. |
| hr/oracle | connect to **hr** schema. |

****

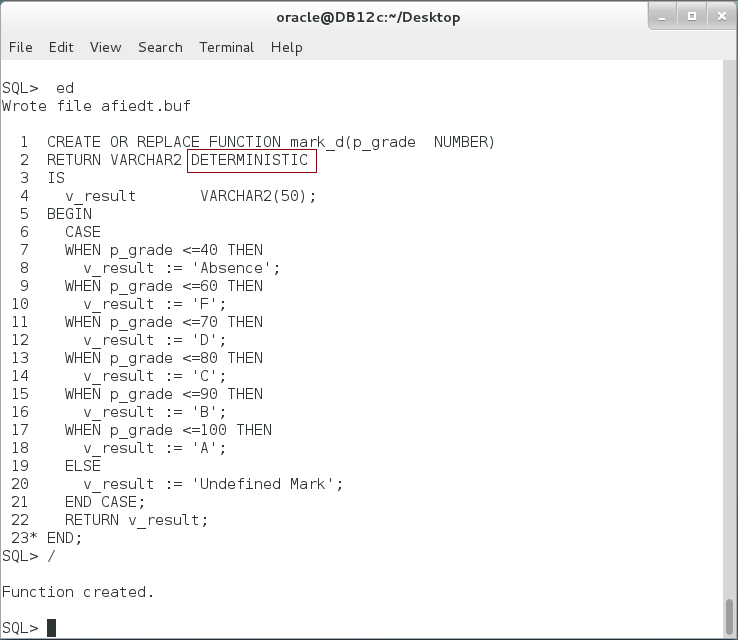
**Step 2:** Create a "**mark**" function that get a "**grade**" number and find its equivalent mark. Execute the following block:

|  |  |
| --- | --- |
| Command | Description |
| **CREATE OR REPLACE FUNCTION mark**(p\_grade NUMBER) |  |
| RETURN VARCHAR2 |  |
| IS |  |
| v\_result VARCHAR2(50); |  |
| BEGIN |  |
| **CASE** |  |
| **WHEN** p\_grade <=40 THEN |
| v\_result := 'Absence'; |
| **WHEN** p\_grade <=60 THEN |
| v\_result := 'F'; |
| **WHEN** p\_grade <=70 THEN |
| v\_result := 'D'; |
| **WHEN** p\_grade <=80 THEN |
| v\_result := 'C'; |
| **WHEN** p\_grade <=90 THEN |
| v\_result := 'B'; |
| **WHEN** p\_grade <=100 THEN |
| v\_result := 'A'; |
| **ELSE** |
| v\_result := 'Undefined Mark'; |
| **END** **CASE**; |
| **RETURN** v\_result; |
| END; |  |
| / |

****

**Step 3:** In the previous, you may notice that the output of the function **does not** depend on other database objects or on time. So, it returns the same output for the same inputs. In this case, you may think of DETERMINISTIC function. Create "**mark\_d**" deterministic function as shown below:

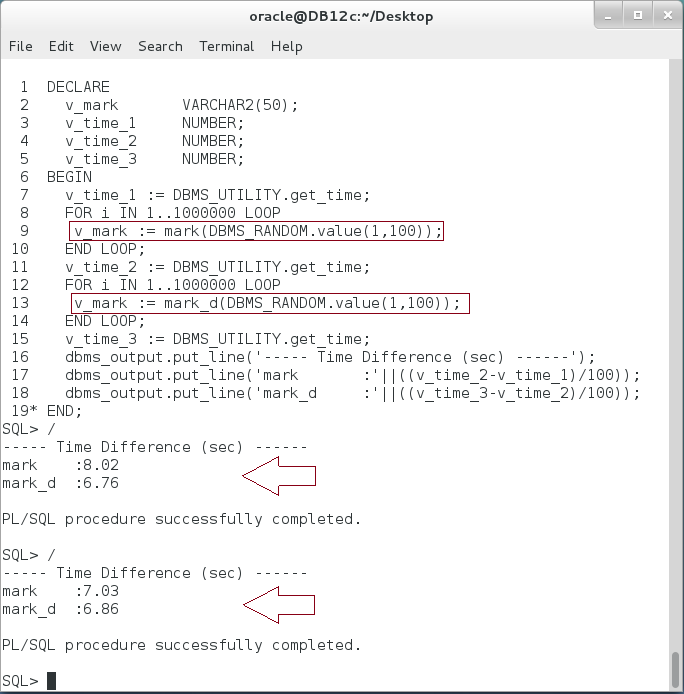
|  |  |
| --- | --- |
| Command | Description |
| CREATE OR REPLACE FUNCTION **mark\_d**(p\_grade NUMBER) |  |
| RETURN VARCHAR2 **DETERMINISTIC** |  |
| IS |  |
| v\_result VARCHAR2(50); |  |
| BEGIN |  |
| CASE |  |
| WHEN p\_grade <=40 THEN |
| v\_result := 'Absence'; |
| WHEN p\_grade <=60 THEN |
| v\_result := 'F'; |
| WHEN p\_grade <=70 THEN |
| v\_result := 'D'; |
| WHEN p\_grade <=80 THEN |
| v\_result := 'C'; |
| WHEN p\_grade <=90 THEN |
| v\_result := 'B'; |
| WHEN p\_grade <=100 THEN |
| v\_result := 'A'; |
| ELSE |
| v\_result := 'Undefined Mark'; |
| END CASE; |
| RETURN v\_result; |
| END; |  |
| / |



**Please note:** "**mark\_d**" is exactly as "**mark**" function but we add the clause "DETERMINISTIC" after RETURN expression.

**Step 4:** We will try to note if there is any performance difference between "**mark**" and "**mark\_d**" function. Execute the following block:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE | Create bubble\_sort Function. |
| v\_mark VARCHAR2(50); |
| v\_time\_1 NUMBER; |
| v\_time\_2 NUMBER; |
| v\_time\_3 NUMBER; |
| BEGIN |
| v\_time\_1 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN 1..1000000 **LOOP** |  |
| v\_mark := **mark**(DBMS\_RANDOM.value(1,100)**)**; |  |
| **END LOOP**; |  |
| v\_time\_2 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN 1..1000000 **LOOP** |  |
| v\_mark := **mark\_d(**DBMS\_RANDOM.value(1,100)**)**; |  |
| **END LOOP;** |  |
| v\_time\_3 := DBMS\_UTILITY.get\_time; |  |
| dbms\_output.put\_line('----- Time Difference (sec) ------'); |  |
| dbms\_output.put\_line('mark:'||((v\_time\_2-v\_time\_1)/100)); |  |
| dbms\_output.put\_line('mark\_d:'||((v\_time\_3-v\_time\_2)/100)); |  |
| END; |  |
| / |  |



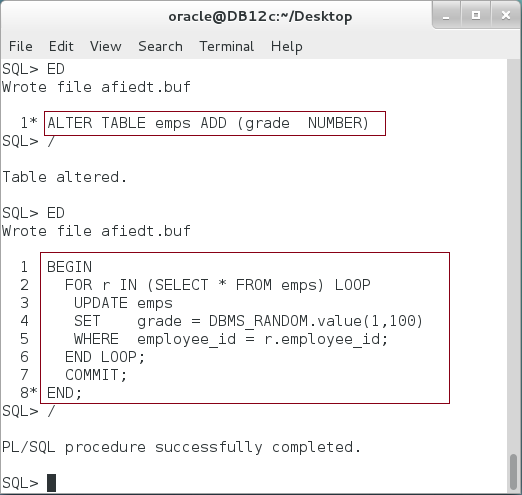
**Opps**: The results show no significant difference between a regular function "mark" and deterministic function "mark\_d". **Can you explain why?**

**Step 5:** Deterministic clause does not take effect unless it appears on two situations:

* "Function-Based index"
* Materialized View with write enable.

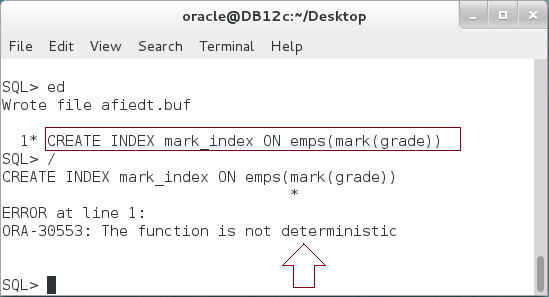
Now, we will think to create "function-Based index". To view the error, execute the following command:

|  |  |
| --- | --- |
| Command | Description |
| ALTER TABLE emps ADD (grade NUMBER) | Add new column "grade" on "emps" table. |
| / |
| BEGIN | Populate the new column with random grades. |
| FOR r IN (SELECT \* FROM emps) LOOP |
| UPDATE emps |
| SET grade = DBMS\_RANDOM.value(1,100) |
| WHERE employee\_id = r.employee\_id; |
| END LOOP; |
| COMMIT; |
| END; |
| / |  |



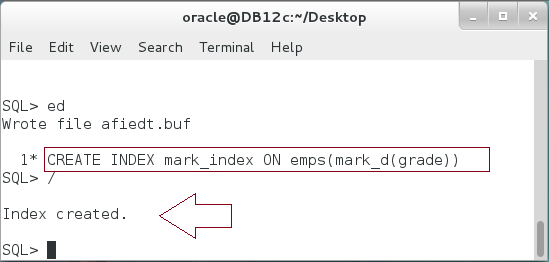
**Step 6:** Try to create a function-based index on "grade" column using "mark" function.

|  |  |
| --- | --- |
| Command | Description |
| CREATE INDEX mark\_index ON emps(**mark(grade)**) |  |

****

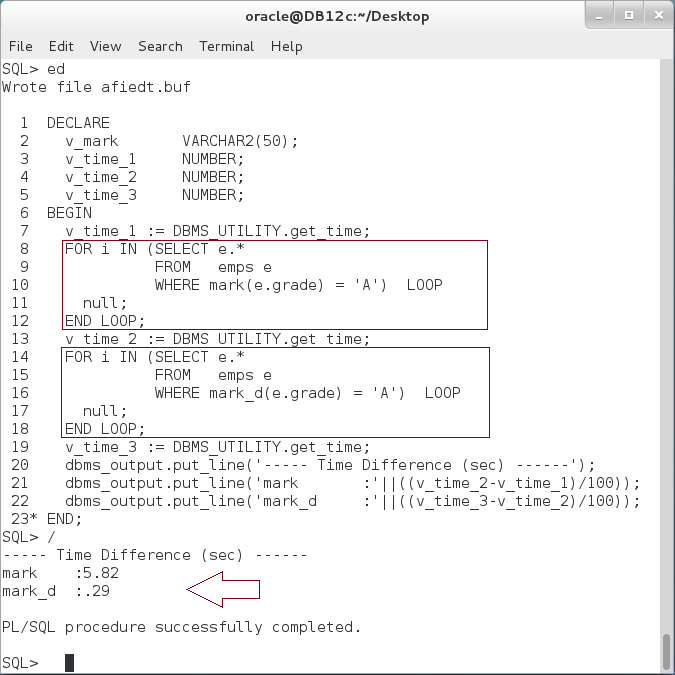
**Step 7:** Create a function-based index on "grade" column using "mark" function:

|  |  |
| --- | --- |
| Command | Description |
| CREATE INDEX mark\_index ON emps(**mark\_d(grade)**); |  |

****

**Step 8:** Compare between regular function "**mark**" and deterministic function "**mark\_d**" using the function-based index as shown below:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| v\_mark VARCHAR2(50); |  |
| v\_time\_1 NUMBER; |  |
| v\_time\_2 NUMBER; |  |
| v\_time\_3 NUMBER; |  |
| BEGIN |  |
| v\_time\_1 := DBMS\_UTILITY.get\_time; |  |
| **FOR i IN** (SELECT e.\* | Use "mark" function. |
| FROM emps e |
| WHERE **mark(e.grade) = 'A'**) LOOP |
| null; |
| **END LOOP;** |
| v\_time\_2 := DBMS\_UTILITY.get\_time; |  |
| **FOR i IN** (SELECT e.\* | Use "mark\_d" function. |
| FROM emps e |
| WHERE **mark\_d(e.grade) = 'A'**) LOOP |
| null; |
| **END LOOP;** |
| v\_time\_3 := DBMS\_UTILITY.get\_time; |  |
| dbms\_output.put\_line('----- Time Difference (sec) ------'); |  |
| dbms\_output.put\_line('mark:'||((v\_time\_2-v\_time\_1)/100)); |  |
| dbms\_output.put\_line('mark\_d:'||((v\_time\_3-v\_time\_2)/100)); |  |
| END; |  |

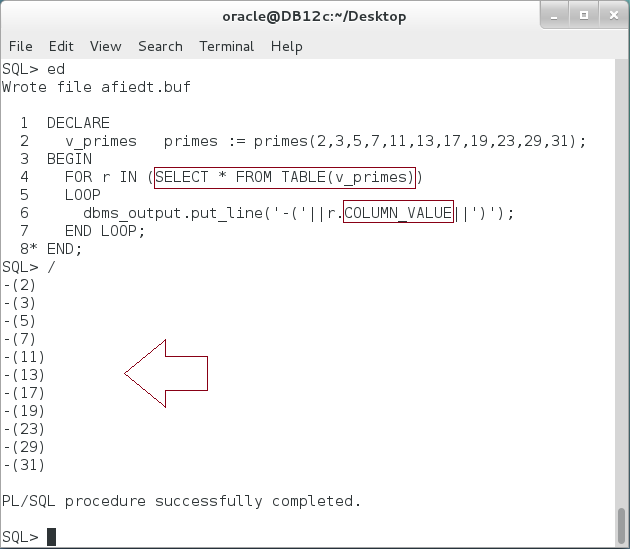
****

**Please note:** This is a significance difference between Deterministic and non-Deterministic function.

## Table Function

**Step 1:** Execute the following PL/SQL block:

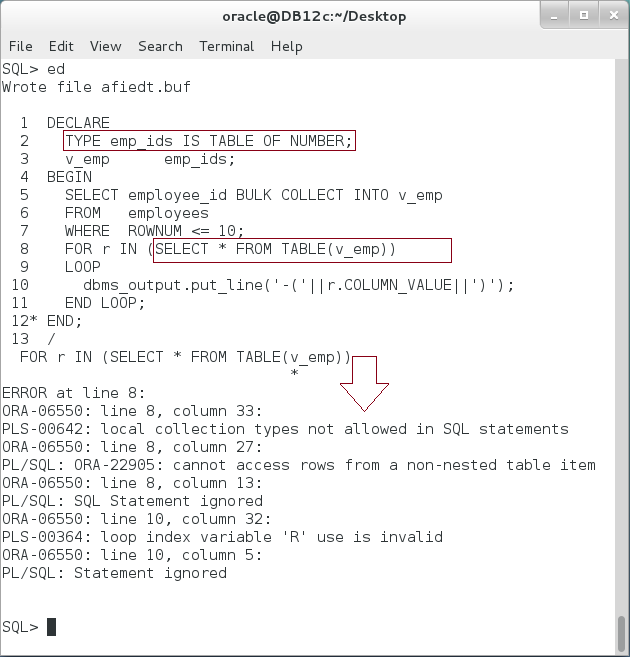
|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| v\_primes primes := primes(2,3,5,7,11,13,17,19,23,29,31); | TABLE function takes "collection" of rows and returns a |
| BEGIN |
| FOR r IN (SELECT \* FROM **TABLE(**v\_primes**)**) |
| LOOP |
| dbms\_output.put\_line('-('||r.**COLUMN\_VALUE**||')'); |
| END LOOP; |
| END; |



**Please note:** You can now treats a collection (VARRAY or Nested Table) as a table. You may also notice the name of the column name "COLUMN\_NAME".

**Step 2:** Execute the following block:

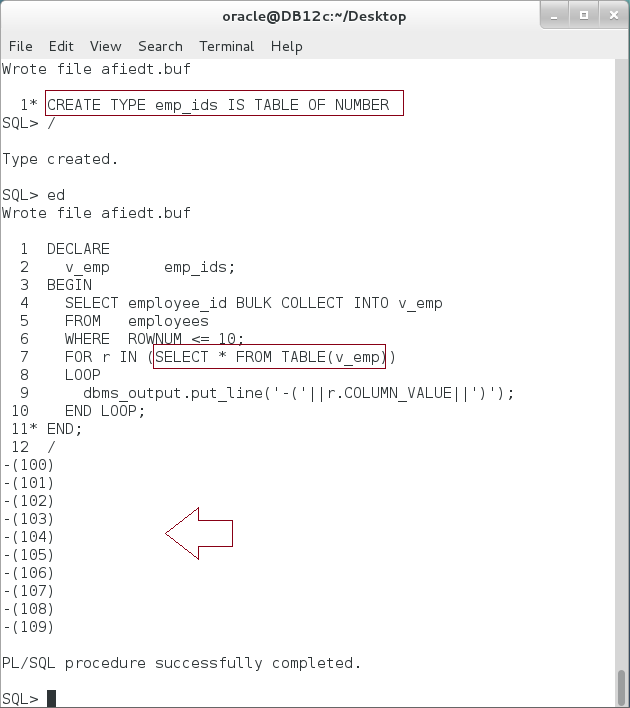
|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| **TYPE** emp\_ids **IS TABLE** **OF** NUMBER; |  |
| v\_emp emp\_ids; |  |
| BEGIN |  |
| SELECT employee\_id BULK COLLECT INTO v\_emp | Populate values in "**v\_emp**" nested table. |
| FROM employees |
| WHERE ROWNUM <= 10; |
| FOR r IN (SELECT \* FROM **TABLE(v\_emp)**) | Try to use the nested table in TABLE function. |
| LOOP |
| dbms\_output.put\_line('-('||r.COLUMN\_VALUE||')'); |
| END LOOP; |
| END; |  |
| / |  |

****

**Please note:** Using local collection is not allowed. To use a collection, you must define it as a standalone type.

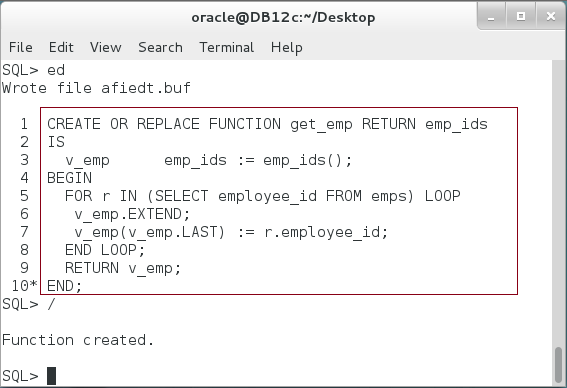
**Step 3:** Create a standalone type and re-execute the previous block:

|  |  |
| --- | --- |
| Line | Description |
| **CREATE TYPE** emp\_ids IS TABLE OF NUMBER; | Create a standalone type. |
| / |
| DECLARE |  |
| v\_emp emp\_ids; |  |
| BEGIN |
| SELECT employee\_id BULK COLLECT INTO v\_emp |  |
| FROM employees |  |
| WHERE ROWNUM <= 10; |  |
|  |  |
| FOR r IN (SELECT \* FROM **TABLE(v\_emp)**) | Use a standalone collection in TABLE function. |
| LOOP |
| dbms\_output.put\_line('-('||r.COLUMN\_VALUE||')'); |
| END LOOP; |  |
| END; |  |
| / |  |



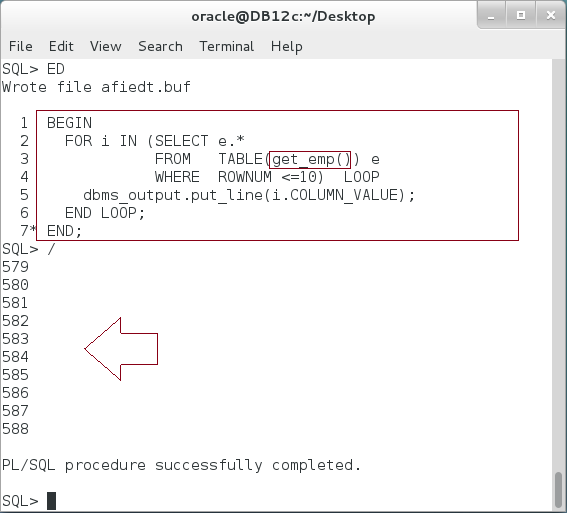
**Step 4:** You can also create use a function that return a collection as an input for TABLE function:

|  |  |
| --- | --- |
| Line | Description |
| **CREATE OR REPLACE FUNCTION** get\_emp **RETURN emp\_ids** | Create a function that returns nested table. |
| IS |
| v\_emp emp\_ids := emp\_ids(); |  |
| BEGIN |  |
| FOR r IN (SELECT employee\_id FROM emps) LOOP |
| v\_emp.EXTEND; |  |
| v\_emp(v\_emp.LAST) := r.employee\_id; |  |
| END LOOP; |  |
| RETURN v\_emp; |  |
| END; |  |
| / |



**Step 4:** Use the previous function as shown below:

|  |  |
| --- | --- |
| Line | Description |
| BEGIN |  |
| FOR i IN (SELECT e.\* |
| FROM **TABLE(get\_emp())** e | Use a regular function that returns collection. |
| WHERE ROWNUM <=10) LOOP |
| dbms\_output.put\_line(i.COLUMN\_VALUE); |
| END LOOP; |  |
| END; |  |
| / |  |

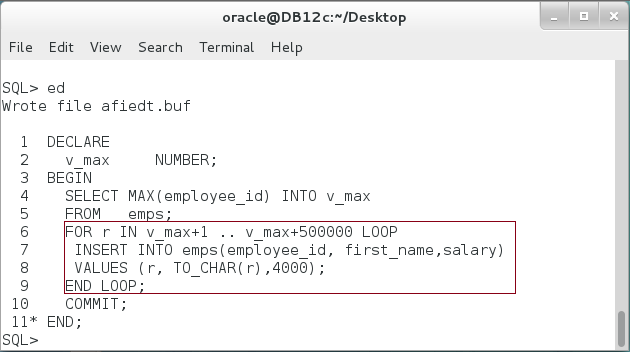


**Please note:** A regular function that returns a collection works well but when the data becomes very large, its performance and memory usage affected dramatically. In the previous block, only 10 rows were needed, however, the function must load all the collection elements before returning to TABLE function!

## Pipelined Table Function

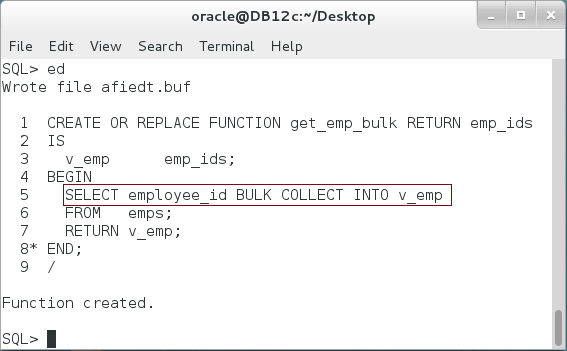
**Step 1:** We will start by populating "emps" table with more rows as shown below:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| v\_max NUMBER; |  |
| BEGIN |  |
| SELECT MAX(employee\_id) INTO v\_max | Find the last employee id |
| FROM emps; |
| **FOR r IN** v\_max+1 .. v\_max+500000 LOOP | Add 500,000 more rows. |
| **INSERT INTO** emps(employee\_id, first\_name,salary) |
| VALUES (r, TO\_CHAR(r),4000); |
| **END LOOP;** |
| COMMIT; |  |
| END; |  |
| / |  |

****

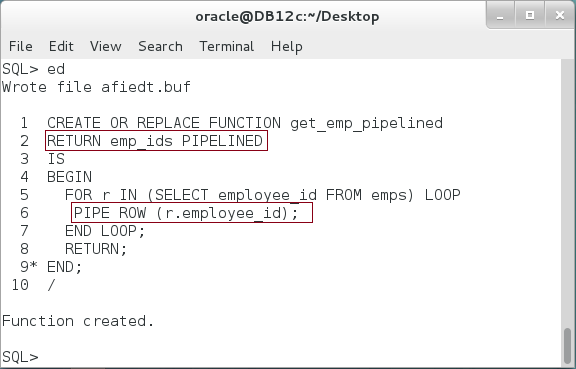
**Step 2:** For comparison reason, we will create a similar to the regular function but with BULK COLLECT INTO clause:

|  |  |
| --- | --- |
| Command | Description |
| CREATE OR REPLACE FUNCTION **get\_emp\_bulk** RETURN emp\_ids | Create "get\_emp\_bulk" function. |
| IS |
| v\_emp emp\_ids; |
| BEGIN |
| **SELECT** employee\_id **BULK COLLECT INTO** v\_emp |
| FROM emps; |
| RETURN v\_emp; |
| END; |

****

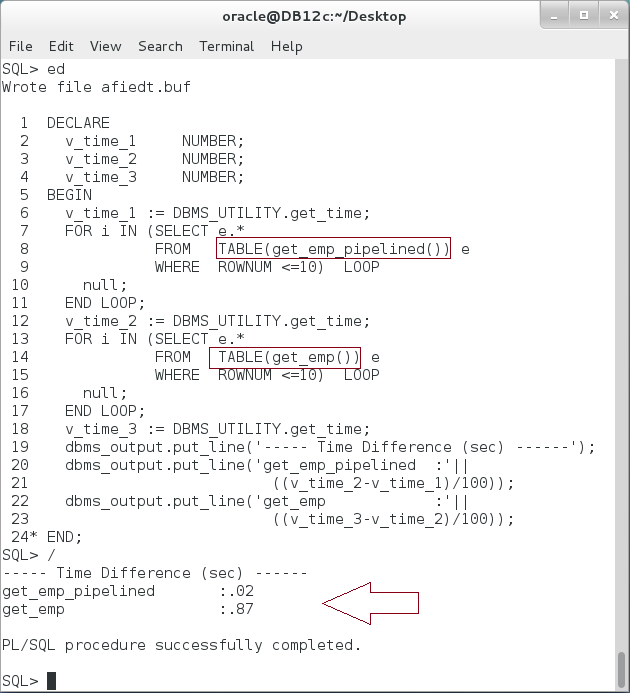
**Step 3:** Create a **Pipelined Function** as shown below:

|  |  |
| --- | --- |
| Command | Description |
| CREATE OR REPLACE FUNCTION **get\_emp\_pipelined** | Return a collection but pipelined. |
| **RETURN emp\_ids PIPELINED** |
| IS |  |
| BEGIN |  |
| FOR r IN (SELECT employee\_id FROM emps) LOOP |  |
| **PIPE ROW (r.employee\_id);** | Push the row to the invoker |
| END LOOP; |
| **RETURN;** | **RETURN** keyword is not as regular function. |
| END; |
| / |

****

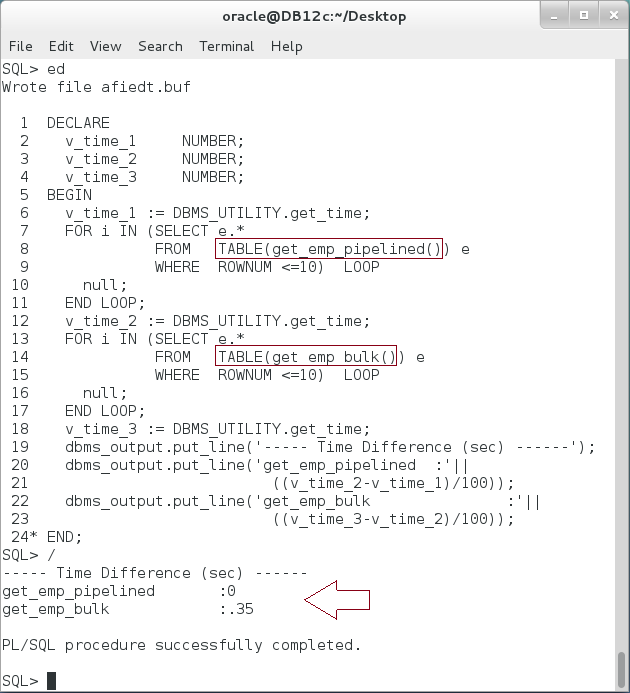
**Step 4:** First, compare between pipelined function and regular function in time:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| v\_time\_1 NUMBER; |  |
| v\_time\_2 NUMBER; |  |
| v\_time\_3 NUMBER; |  |
| BEGIN |  |
| v\_time\_1 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN (SELECT e.\* | Pipelined Function |
| FROM **TABLE(get\_emp\_pipelined())** e |
| WHERE ROWNUM <=10) LOOP |
| null; |
| **END** **LOOP**; |
| v\_time\_2 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN (SELECT e.\* | Regular function |
| FROM **TABLE(get\_emp())** e |
| WHERE ROWNUM <=10) LOOP |
| null; |
| **END** **LOOP**; |
| v\_time\_3 := DBMS\_UTILITY.get\_time; |  |
| dbms\_output.put\_line('----- Time Difference (sec) ------'); |  |
| dbms\_output.put\_line('get\_emp\_pipelined:'||((v\_time\_2-v\_time\_1)/100)); |  |
| dbms\_output.put\_line('get\_emp:'||((v\_time\_3-v\_time\_2)/100)); |  |
| END; |  |
| / |  |

****

**Step 5:** Second, you may compare pipelined function with bulk function:

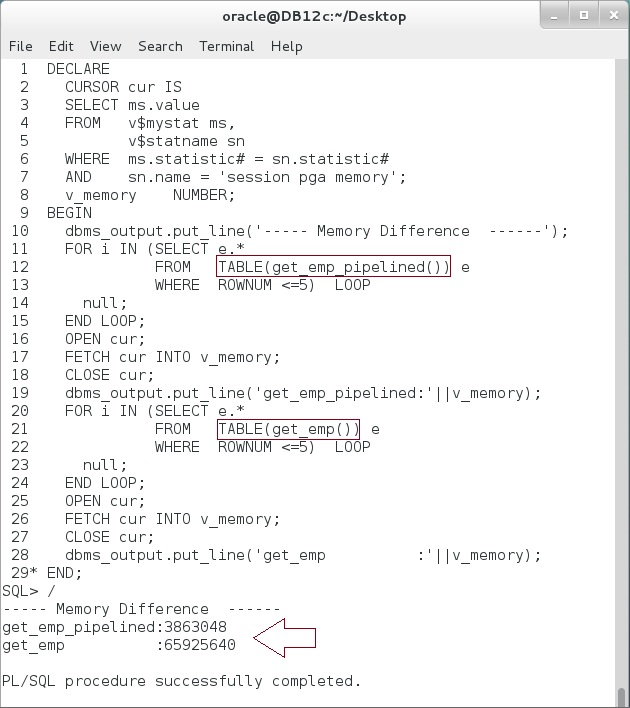
|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| v\_time\_1 NUMBER; |  |
| v\_time\_2 NUMBER; |  |
| v\_time\_3 NUMBER; |  |
| BEGIN |  |
| v\_time\_1 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN (SELECT e.\* | Pipelined Function |
| FROM **TABLE(get\_emp\_pipelined())** e |
| WHERE ROWNUM <=10) LOOP |
| null; |
| **END** **LOOP**; |
| v\_time\_2 := DBMS\_UTILITY.get\_time; |  |
| **FOR** i IN (SELECT e.\* | Regular **bulk** function |
| FROM **TABLE(get\_emp\_bluk())** e |
| WHERE ROWNUM <=10) LOOP |
| null; |
| **END** **LOOP**; |
| v\_time\_3 := DBMS\_UTILITY.get\_time; |  |
| dbms\_output.put\_line('----- Time Difference (sec) ------'); |  |
| dbms\_output.put\_line('get\_emp\_pipelined:'||((v\_time\_2-v\_time\_1)/100)); |  |
| dbms\_output.put\_line('get\_emp\_bulk:'||((v\_time\_3-v\_time\_2)/100)); |  |
| END; |  |
| / |  |

****

**Please note:**  The time difference may increase if the regular/pipelined function do complex works before generating the row.

**Step 6:** Third, you will compare the memory usage between the pipelined function and regular function:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| CURSOR cur IS | Query the user memory usage. |
| SELECT ms.value |
| FROM v$mystat ms, |
| v$statname sn |
| WHERE ms.statistic# = sn.statistic# |
| AND sn.name = 'session pga memory'; |
| v\_memory NUMBER; |
| BEGIN |  |
| dbms\_output.put\_line('----- Memory Difference ------'); |  |
| FOR i IN (SELECT e.\* |  |
| FROM **TABLE(get\_emp\_pipelined()) e** | Use Pipelined Function |
| WHERE ROWNUM <=5) LOOP |
| null; |
| END LOOP; |  |
| OPEN cur; |  |
| FETCH cur INTO v\_memory; |  |
| CLOSE cur; |  |
| dbms\_output.put\_line('get\_emp\_pipelined:'||v\_memory); |  |
| FOR i IN (SELECT e.\* | Use regular function |
| FROM **TABLE(get\_emp()) e** |
| WHERE ROWNUM <=5) LOOP |
| null; |  |
| END LOOP; |  |
| OPEN cur; |  |
| FETCH cur INTO v\_memory; |  |
| CLOSE cur; |  |
| dbms\_output.put\_line('get\_emp :'||v\_memory); |  |
| END; |  |
| / |  |

****

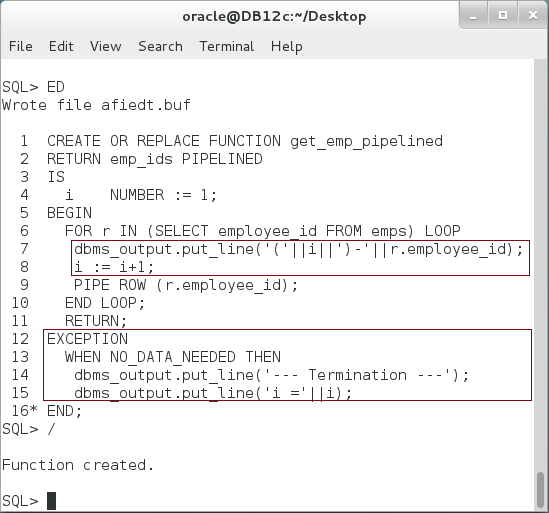
**Please note:**

* The memory used by pipelined function is less than the memory used by the regular function.
* You need "SELECT ANY DICTIONARY" role to access the query.

**Step 7:** Modify the previous block to compare pipelined function with bulk function. **Is there any difference? Explain the result**.

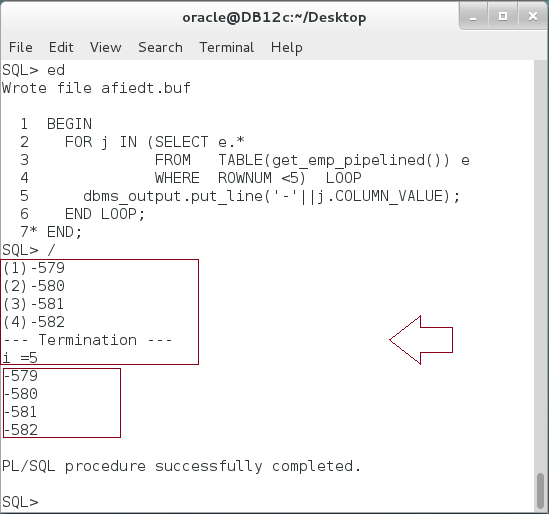
**Step 8:** To understand how a pipelined function work, modify the pipelined function as shown below:

|  |  |
| --- | --- |
| Command | Description |
| CREATE OR REPLACE FUNCTION get\_emp\_pipelined |  |
| RETURN emp\_ids PIPELINED |  |
| IS |  |
| **i NUMBER := 1;** |  |
| BEGIN |  |
| FOR r IN (SELECT employee\_id FROM emps) LOOP |  |
| **dbms\_output.put\_line('('||i||')-'||r.employee\_id);** | Display some text after each iteration. |
| **i := i+1;** |
| PIPE ROW (r.employee\_id); |  |
| END LOOP; |  |
| RETURN; |  |
| **EXCEPTION** | Handle **NO\_DATA\_NEEDED** exception. |
| **WHEN NO\_DATA\_NEEDED THEN** |
| **dbms\_output.put\_line('--- Termination ---');** |
| **dbms\_output.put\_line('i ='||i);** |
| END; |
| / |  |



**Step 9:** To understand how a pipelined function work, modify the pipelined function as shown below:

|  |  |
| --- | --- |
| Command | Description |
| BEGIN |  |
| FOR j IN (SELECT e.\* | Select only 4 rows from the table. |
| FROM TABLE(get\_emp\_pipelined()) e |
| **WHERE ROWNUM <5**) LOOP |
| dbms\_output.put\_line('-'||j.COLUMN\_VALUE); |
| END LOOP; |
| END; |
| / |  |

****

**Please note:** PIPE ROW command returns the result row by row, when no other rows is needed, the NO\_DATA\_NEEDED is raised to tell the pipelined function stop fetching new rows. This enhances the memory usage because the function does need to fetch all rows before the invoker can access just few rows.

# SUMMARY

Deterministic function always returns the same output of the same input(s). It should not depend on time or other database objects. Be careful when using the deterministic function, if you build a function that its output does not depend only on the input(s), the deterministic function will return undetermined output. Remember that, Oracle neglects a DETERMINSTIC clause if the function appears on situations other than (1) function-based index and (2) materialized view with write enable. Table function enables you to access a collection as table. Its input may be any standalone collection or a function that return a collection (VARRAY or Nested Table). Table function may suffer of performance issue when accessing a very large data. One of the common solutions to overcome this issue is to use Pipelined Table Function. A Pipelined Table Function pushes one row after another until no other data is needed. Using a Pipelined Table Function decreases the memory consumption since not all data should be loaded in the memory before returning a single row to the invoker. It may also enhance the performance if the invoker needs a small number of rows.

After completing this lab exercise, you should be able to use Deterministic Function, Table Function, and Pipelined Table Function.

# REFERENCES

* http://docs.oracle.com/database/121/LNPLS/function.htm#CJAFICJJ
* http://rwijk.blogspot.fi/2008/04/deterministic-clause.html
* http://oracle-base.com/articles/misc/pipelined-table-functions.php
* http://docs.oracle.com/database/121/LNPLS/tuning.htm#LNPLS916
* http://docs.oracle.com/database/121/LNPLS/create\_function.htm#LNPLS01370
* http://docs.oracle.com/cd/B19306\_01/server.102/b14200/operators006.htm
* https://docs.oracle.com/database/121/ADDCI/aggr\_functions.htm#ADDCI4623

# INDEX

Deterministic Function 2, 7, 32

Pipelined Table Function 3, 32

Table Function 2, 3, 5, 15, 20, 32