Introducing Oracle Pipelined Table Functions

Brent A. House
UTOUG Training Days 2014
Brent House

BI Developer – Intermountain Healthcare
Using Oracle since 1997

uintafun@gmail.com
SQL Query – get data from a table

Two required clauses: SELECT & FROM

SELECT first_name, last_name
FROM employees;
Query the dictionary for the built-in functions

SELECT DISTINCT object_name
FROM all_arguments
WHERE package_name = 'STANDARD'
ORDER BY object_name;
Manipulating the column data by invoking a function

Built-in functions UPPER & TO_DATE

SELECT first_name, UPPER(last_name) FROM employees;

SELECT TO_DATE('20140228','YYYYMMDD') FROM dual;
User-defined functions

Let’s move to PL/SQL, and use it to encapsulate code with our own function.
DECLARE FUNCTION tomorrowdate
    RETURN DATE IS
    BEGIN
    RETURN trunc(sysdate+1);
    END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(tomorrowdate);
END;
/

Anonymously: no stored object created
Creating a function in PL/SQL

CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
< function_body >
RETURN
END [function_name];
http://www.tutorialspoint.com/plsql/plsql_functions.htm
Basic function – returns text string

CREATE OR REPLACE FUNCTION basicfunc
RETURN VARCHAR2 IS
BEGIN
  RETURN 'a basic user-defined function';
END basicfunc;

SELECT 'This is ' || basicfunc FROM dual;
Output: “This is a basic user-defined function”
Basic Function – returns tomorrow’s date

CREATE OR REPLACE FUNCTION tomorrowdatefunc
  (todaydate IN date)
RETURN date IS
BEGIN
  RETURN trunc(todaydate + 1);
END tomorrowdatefunc;

SELECT 'Tomorrow is ' || tomorrowdatefunc(sysdate) FROM dual;
Output: “Tomorrow is 25-FEB-14”
Recap

• SQL SELECT queries can be manipulated
  • Functions
    • Built-in
    • User defined functions

• Stored Functions are invoked by a calling process
• Functions as anonymous blocks, stand-alone or inside packages.
SQL Standards

**SQL1999** is when user-defined functions were added to the ANSI standards for SQL.

**SQL2003** New in SQL were SQL invoked functions that return a "table". Table functions give increased functionality by allowing sets of tuples from any external data sources to be invoked (as if they were a table).

Dr. Yingshi Li
What is a table function?

• PL/SQL stored object
• Returns a collection type instance
  • VARRAY
  • Nested Table
• Query a table function in the FROM clause
SQL and PL/SQL tables – what?

PL/SQL can contain execution statements which directly SELECT from SQL tables.

SQL cannot select from PL/SQL tables without TABLE operator.
Table functions

• We invoke table functions in SQL
  • Needs a collection type or REF cursor as input argument
  • The table function returns a collection
  • Used to generate a list that can be used by Java or other languages

SELECT * FROM TABLE( CAST(employee_nt AS employee_nt_t));
Nested tables

Because nested tables are shared by both SQL & PL/SQL, they enable us to use table functions in SQL

```
SELECT * FROM
  TABLE( CAST(employee_nt AS employee_nt_t) )
```

**TABLE Operator**

Tells Oracle to act as though the variable is an SQL Table

**CAST Operator**

Explicitly converts the data type
Table functions benefit from pipelining

With a regular table function, the data is accumulated in the variable during processing. A cursor returns all at one time and can result in a large result set. Add in the pipelined option to push each processed row as soon as it’s ready.

**Pipelined Table Functions**

- Return the rows right after generated
- Produce smaller sized batches of rows
- Need the PIPELINED keyword added to code
- Need the PIPE ROW operator added to execute statements
Oracle bringing in pipelined table functions

8i: Able to use "SELECT FROM TABLE(CAST(plsql_function AS collection_type))" for binding user-generated lists of data.
Problem: large memory footprint.

9i Release 1 (9.0), Oracle has introduced pipelined table functions (pipelined functions). Can pipe (or stream) data in small arrays of prepared data, rather than fully materialized.

Easily combine the complex procedures of PL/SQL with SQL for bulk operations. For example transforming data through a pipeline or chained pipeline functions.

http://www.oracle-developer.net/display.php?id=207
How are pipelined table functions being used?

ETL Tools: External Tables, Table Functions and Merge
http://www.akadie.com/services/ora_etl.html

Generating a range of dates to use as though it were a table.
http://www.akadia.com/services/ora_pipe_functions.html

Data transformations: improved time between table function and pipelined table function.

Transformation Pipelines

In traditional Extract Transform Load (ETL) processes you may be required to load data into a staging area, then make several passes over it to transform it into a state where it can be loaded into your destination schema. Passing the data through staging tables can represent a significant amount of disk I/O for both the data and the redo generated. An alternative is to perform the transformations in pipelined table functions so data can be read from an external table and inserted directly into the destination table, removing much of the disk I/O.

Table function – can extract, transform and load in a single step. Add pipelining to speed up.

Back in Oracle8i
SQL*Loader reads external file, writes to staging table
Read from staging table with PL/SQL Procedure, write to target table

Since Oracle 9i
Insert into target table using pipelined table function as source
Recap

• ANSI standard
• SELECT (invoke) table functions
• Nested tables
• Benefits of pipelined functions
• How are they being used?
External table data to transform

CREATE TABLE stock_ext
( ticker VARCHAR2(13),
  open_price VARCHAR2(6),
  close_price VARCHAR2(6))
ORGANIZATION EXTERNAL
...

adapted from
http://examples.oreilly.com/oraclep3/individual_files/tabfunc.sql
Pivot the data

<table>
<thead>
<tr>
<th>Ticker</th>
<th>PriceType</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMB</td>
<td>O</td>
<td>39.94</td>
</tr>
<tr>
<td>WMB</td>
<td>C</td>
<td>39.85</td>
</tr>
<tr>
<td>SPY</td>
<td>O</td>
<td>174.78</td>
</tr>
<tr>
<td>SPY</td>
<td>C</td>
<td>175.17</td>
</tr>
<tr>
<td>GLD</td>
<td>O</td>
<td>121.76</td>
</tr>
<tr>
<td>GLD</td>
<td>C</td>
<td>121.29</td>
</tr>
</tbody>
</table>

Each row in `stock_ext` becomes 2 rows in Nested Table

<table>
<thead>
<tr>
<th>Ticker</th>
<th>open_price</th>
<th>close_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMB</td>
<td>39.94</td>
<td>39.85</td>
</tr>
<tr>
<td>SPY</td>
<td>174.78</td>
<td>175.17</td>
</tr>
<tr>
<td>GLD</td>
<td>121.76</td>
<td>121.29</td>
</tr>
</tbody>
</table>
Create the TYPEs in the schema

CREATE TYPE TickerType AS OBJECT
( ticker VARCHAR2(20),
 PriceType VARCHAR2(1),
 price NUMBER);

CREATE TYPE TickerTypeSet AS TABLE OF TickerType;
Verify the collection type definitions

<table>
<thead>
<tr>
<th>TYPE_NAME</th>
<th>TYPECODE</th>
<th>INCOMPLETE</th>
<th>FINAL</th>
<th>INSTANTIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKERTYPE</td>
<td>OBJECT</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>TICKERTYPESET</td>
<td>COLLECTION</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

SELECT type_name, typecode, incomplete, final, instantiable
FROM user_types;
Details of the collection types

<table>
<thead>
<tr>
<th>TYPE_NAME</th>
<th>COLL_TYPE</th>
<th>ELEM_TYPE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKERTYPESET</td>
<td>TABLE</td>
<td>TICKERTYPE</td>
</tr>
</tbody>
</table>

SELECT type_name, coll_type, elem_type_name
FROM user_coll_types;
You know you’re geeky when you hear care package and immediately think of Call of Duty.
Create package, define the ref cursor type

CREATE OR REPLACE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN stock_ext%ROWTYPE;
END refcur_pkg;
/

Pipelined Table Function

CREATE OR REPLACE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
in_rec p%ROWTYPE;
BEGIN and END BLOCK

BEGIN
  LOOP
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
    -- first row
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.open_price;
    PIPE ROW(out_rec);
    -- second row
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.close_price;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
  RETURN;
END;
Transform rows from external table

SELECT *
FROM TABLE(StockPivot(CURSOR(SELECT * FROM STOCK_EXT))) x;
A view, so I don’t forget the syntax ;)

CREATE VIEW stock_close AS
SELECT x.Ticker, x.Price
FROM TABLE(StockPivot(CURSOR(SELECT * FROM STOCK_EXT))) x
WHERE x.PriceType='C';
Query to list all pipelined table functions

```
SELECT OWNER, OBJECT_NAME, OBJECT_TYPE, PIPELINED
FROM ALL_PROCEDURES
WHERE PIPELINED='YES' AND OBJECT_TYPE='FUNCTION';
```
Key parts of Pipelined Table Functions

PIPELINED option
RETURN Data Type
PIPE ROW Statement
RETURN Statement
ReCap

• Transform external table rows from 1 to 2.
• Packaging
• Create a view
• Vital parts for pipelined functions
Build and compare regular and pipeline table functions

http://www.dba-oracle.com/plsql/t_plsql_pipelining.htm

Dr. Tim Hall has a chapter in his book that does this nicely.
Create the two TYPES: record and collection

CREATE OR REPLACE TYPE t_square_root_row AS OBJECT ( 
    start_number NUMBER,
    square_root NUMBER,
    description VARCHAR2(50)
);
/

CREATE OR REPLACE TYPE t_square_root_tab AS TABLE OF t_square_root_row;
/
Both type objects are created

```
SELECT type_name, typecode, incomplete, final, instantiable
FROM user_types;
```

<table>
<thead>
<tr>
<th>TYPE_NAME</th>
<th>TYPECODE</th>
<th>INCOMPLETE</th>
<th>FINAL</th>
<th>INSTANTIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_SQUARE_ROOT_ROW</td>
<td>OBJECT</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>T_SQUARE_ROOT_TAB</td>
<td>COLLECTION</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Details of the collection types

SELECT type_name, coll_type, elem_type_name
FROM user_coll_types;

<table>
<thead>
<tr>
<th>TYPE_NAME</th>
<th>COLL_TYPE</th>
<th>ELEM_TYPE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_SQUARE_ROOT_TAB</td>
<td>TABLE</td>
<td>T_SQUARE_ROOT_ROW</td>
</tr>
</tbody>
</table>
Create package specification to be used by both functions

CREATE OR REPLACE PACKAGE tf_pk AS

  FUNCTION get_square_roots_tf
    (p_start_range IN NUMBER, p_end_range IN NUMBER)
  RETURN t_square_root_tab;

  FUNCTION get_square_roots_ptf (p_start_range IN NUMBER, p_end_range IN NUMBER)
    RETURN t_square_root_tab PIPELINED;

END tf_api;
/

FUNCTION get_square_roots_tf (p_start_range IN NUMBER, p_end_range IN NUMBER)

    RETURN t_square_root_tab
AS
    l_row t_square_root_row := t_square_root_row(NULL, NULL);
    l_tab t_square_root_tab := t_square_root_tab();
BEGIN
    FOR i IN p_start_range .. p_end_range LOOP
        -- Build up a new row.
        l_row.start_number := i;
        l_row.square_root := ROUND(SQRT(i), 2);
        l_row.description := 'The square root of ' || i || ' is ' || l_row.square_root;
        -- Extend the collection (populate the nested table and add the row.
        l_tab.extend;
        l_tab(l_tab.last) := l_row;
    END LOOP;
    -- Return the collection.
    RETURN l_tab;
END get_square_roots_tf;

Regular Table Function
FUNCTION get_square_roots_ptf (p_start_range IN NUMBER, p_end_range IN NUMBER)
  RETURN t_square_root_tab PIPELINED
AS
  l_row t_square_root_row := t_square_root_row(NULL, NULL);
BEGIN
  FOR i IN p_start_range .. p_end_range LOOP

    -- Build up a new row.
    l_row.start_number := i;
    l_row.square_root := ROUND(SQRT(i), 2);
    l_row.description := 'The square root of ' || i || ' is ' || l_row.square_root;

    -- Pipe the row out.
    PIPE ROW (l_row);
  END LOOP;

  -- Perform return.
  RETURN;
END get_square_roots_ptf;
END tf_api;
/
Both table functions generate same output

-- Query the regular table function.
SELECT *
FROM   TABLE(tf_api.get_square_roots_tf(1, 100)) a;

-- Query the pipelined table function.
SELECT *
FROM   TABLE(tf_api.get_square_roots_ptf(1, 100)) a;
Create a function to retrieve current PGA usage

CREATE OR REPLACE FUNCTION get_used_memory RETURN NUMBER AS
  l_used_memory NUMBER;
BEGIN
  SELECT ms.value INTO l_used_memory
  FROM v$mystat ms,
      v$statname sn
  WHERE ms.statistic# = sn.statistic#
    AND sn.name = 'session pga memory';
  RETURN l_used_memory;
END get_used_memory;
/

CREATE OR REPLACE FUNCTION get_used_memory RETURN NUMBER AS
  l_used_memory NUMBER;
BEGIN
  SELECT ms.value INTO l_used_memory
  FROM v$mystat ms,
      v$statname sn
  WHERE ms.statistic# = sn.statistic#
    AND sn.name = 'session pga memory';
  RETURN l_used_memory;
END get_used_memory;
/
Run the test from SQLPLUS command line

conn test/test
-- Test regular table function.
SET SERVEROUTPUT ON
DECLARE
  l_start NUMBER;
BEGIN
  l_start := get_used_memory;
  FOR cur_rec IN (SELECT *
                  FROM   TABLE(tf_api.get_square_roots_tf(1, 100000, 'FALSE')))
  LOOP
    NULL;
  END LOOP;
  DBMS_OUTPUT.put_line('Regular table function : ' || (get_used_memory - l_start));
END;
/
Run the test from SQLPLUS command line

conn test/test
-- Test pipelined table function.
SET SERVEROUTPUT ON
DECLARE
  l_start NUMBER;
BEGIN
  l_start := get_used_memory;
  FOR cur_rec IN (SELECT *
      FROM TABLE(tf_api.get_square_roots_ptf(1, 100000, 'FALSE')))
    LOOP
      NULL;
    END LOOP;
  DBMS_OUTPUT.put_line('Pipelined table function : ' || (get_used_memory - l_start));
END;
/
Regular Table Function PGA memory used

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2   l_start  NUMBER;
3   BEGIN
4     l_start := get_used_memory;
5     FOR cur_rec IN (SELECT *
6                       FROM   TABLE(tf_api.get_square_roots_tf<1, 100000, 'FALS
7                         E'>))
8       LOOP
9         NULL;
10       END LOOP;
11
12   DBMS_OUTPUT.put_line('Regular table function : ' ||
13     (get_used_memory - l_start));
14 END;
/
```

Regular table function : **19791872**

PL/SQL procedure successfully completed.
Pipelined table function PGA memory used

```sql
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2    l_start NUMBER;
3    BEGIN
4    l_start := get_used_memory;
5    
6    FOR cur_rec IN (SELECT *
7              FROM   TABLE(tf_api.get_square_roots_ptf<1, 100000, 'FALSE'))
8    LOOP
9    NULL;
10   END LOOP;
11  
12  DBMS_OUTPUT.put_line('Pipelined table function : ' ||
13          (get_used_memory - l_start));
14  END;
15  /

Pipelined table function : **131072**

PL/SQL procedure successfully completed.
```
The pipeline used less PGA memory

The regular table function uses over 150 times the memory used by the pipelined table function.

Regular Table Function used 19,791,872
Pipelined Table Function used 131,072
Pipelined table functions

• Used when must loop through each row of data
• Types
  • Records
  • Collections
• Compare original code to pipelined
Useful resources

Oracle Documentation
www.oracle.com/pls/db112/homepage

Tom Kyte
http://asktom.oracle.com

Mike McLaughlin
http://blog.mclaughlinsoftware.com

Burleson Consulting with Tim Hall book excerpt
http://www.dba-oracle.com/plsql/t_plsql_pipelining.htm
Thank you