**pipelined functions in oracle 9i**

Since Oracle 8.0, it has been possible to select from a collection of data (usually loaded via a function) as a "virtual table". This technique became popular in the 8i timeframe thanks to Tom Kyte and numerous web forums where "SELECT FROM TABLE(CAST(plsql\_function AS collection\_type))" became a common technique for binding user-generated lists of data. However, as a technique for processing large volumes of data, "table functions" of this sort are limited due to their potentially large memory footprints.

In 9i Release 1 (9.0), Oracle has introduced pipelined table functions (known simply as pipelined functions). These build on the table function principle but with some critical differences, three of which stand out in particular:

* first, data is piped (or streamed) to the caller in small arrays of prepared data, rather than fully materialised as with original table functions;
* second, pipelined functions can be parallelised by Oracle which means that PL/SQL can be executed by multiple slaves for the first time; and
* third, pipelined functions make it easy to convert PL/SQL procedures into rowsources for bulk SQL operations, combining complex transformation logic with the benefits of SQL.

This article introduces the fundamentals of pipelined functions, starting with a simple example.

**a simple pipelined function**

The best way to visualise a pipelined function is to see a simple example. For any pipelined function we require the following elements:

* a collection type; and
* the pipelined PL/SQL function (usually contained inside a package).

In the following example, we will create a simple row-generator that pipes out a given number of records. To begin, we require a collection type that is structured according to the data that we wish to pipe. In our row-generator function, we will pipe a single number attribute, so we can simply create a collection type of Oracle's NUMBER datatype.

SQL> CREATE OR REPLACE TYPE number\_ntt AS TABLE OF NUMBER;

2 /

Type created.

Oracle will use this collection type to "buffer" small numbers of records as it pipes data to the function caller. We can now create our first pipelined function. We will create a standalone function for simplicity (we will normally use packaged functions) and some syntax will be unfamiliar. Some comments follow the code.

SQL> CREATE FUNCTION row\_generator (

2 rows\_in IN PLS\_INTEGER

3 ) RETURN number\_ntt PIPELINED IS

4 BEGIN

5 FOR i IN 1 .. rows\_in LOOP

6 PIPE ROW (i);

7 END LOOP;

8 RETURN;

9 END;

10 /

Function created.

Note in particular the following:

* the PIPELINED keyword in the function definition. There are several extensions to this that we will see later in this article;
* the return type must be a collection type (in our example we are using the simple NUMBER\_NTT collection type based on NUMBER);
* the new PIPE ROW syntax to add a single record into the return collection stream. The record we pipe must be based on the same underlying type as the collection type. In our row-generator, we pipe out records of a single number attribute;
* the "empty" RETURN statement. In a pipelined function, the PIPE ROW statement is responsible for "returning" data and not the RETURN statement.

Now we have created a simple pipelined function, we can use it to generate 10 rows of data, as follows.

SQL> SELECT \*

2 FROM TABLE( row\_generator(10) );

COLUMN\_VALUE

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

1

2

3

4

5

6

7

8

9

10

10 rows selected.

Note how we wrap the function call in the TABLE operator. This "virtualises" the collection and allows us to select from it as a datasource such as a table or view. The TABLE operator has been available since Oracle 8.0 (when it was named "THE" - it was renamed to "TABLE" in 8i).

Note also the name of the column generated by our function. Remember we based our collection type on the built-in datatype NUMBER. In situations such as this (i.e. when the collection type is not based on any explicitly-named attributes), Oracle defaults to COLUMN\_VALUE. In practice, we are more likely to base our collection types on more complex structures than a single scalar built-in, as we will see later in this article.

**streaming vs materialising: a visual example**

It is quite difficult to visualise how a pipelined function differs from a table function from a static example in an article. In the following example, we will attempt to demonstrate the "streaming" effect of pipelined functions and compare this to a standard table function. First we will create a simple table function that will return a small collection of our previous collection type.

SQL> CREATE FUNCTION table\_function RETURN number\_ntt AS

2 nt number\_ntt := number\_ntt();

3 BEGIN

4 FOR i IN 1 .. 5 LOOP

5 DBMS\_LOCK.SLEEP(1);

6 nt.EXTEND;

7 nt(nt.LAST) := i;

8 END LOOP;

9 RETURN nt; --<-- return whole collection

10 END table\_function;

11 /

Function created.

Note how the table function loads a collection variable in full before returning it to the "consumer". Next we create a pipelined function to generate the same number of rows as the table function.

SQL> CREATE FUNCTION pipelined\_function RETURN number\_ntt PIPELINED AS

2 BEGIN

3 FOR i IN 1 .. 5 LOOP

4 DBMS\_LOCK.SLEEP(1);

5 PIPE ROW (i); --<-- send row to consumer

6 END LOOP;

7 RETURN;

8 END pipelined\_function;

9 /

Function created.

We can see that both functions include a 1 second sleep inside their respective loops and that the overall duration will be approximately 5 seconds. We will use this to demonstrate the difference between the execution of the functions and the delivery of the first rows. To help us with this, we will create a small function to return a TIMESTAMP against each generated row (using the built-in LOCALTIMESTAMP function directly will not work as it returns a constant value for the entire recordset).

SQL> CREATE FUNCTION get\_time RETURN TIMESTAMP IS

2 BEGIN

3 RETURN LOCALTIMESTAMP;

4 END get\_time;

5 /

Function created.

Finally, we setup our session's timestamp format and set the sqlplus arraysize to 1 to ensure that the pipelined function streams a single record at a time.

SQL> ALTER SESSION SET NLS\_TIMESTAMP\_FORMAT = 'HH24:MI:SS.FF3';

Session altered.

SQL> set arraysize 1

We are now ready to demonstrate the effect of the pipelined function. We will begin by selecting from the simple table function.

SQL> SELECT get\_time() AS ts FROM DUAL;

TS

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

07:02:25.000

1 row selected.

SQL> SELECT column\_value

2 , get\_time() AS ts

3 FROM TABLE(table\_function);

COLUMN\_VALUE TS

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

1 07:02:30.062 --<-- first row takes 5 seconds

2 07:02:30.062

3 07:02:30.062

4 07:02:30.062

5 07:02:30.062

5 rows selected.

We can see quite clearly that the table function did not deliver any data until the function completed. The elapsed time between the call to the function and the delivery of the data was a little over 5 seconds. We can compare this now to the pipelined function, as follows.

SQL> SELECT get\_time() AS ts FROM dual;

TS

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

07:02:35.359

1 row selected.

SQL> SELECT column\_value

2 , get\_time() AS ts

3 FROM TABLE(pipelined\_function);

COLUMN\_VALUE TS

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

1 07:02:36.390 --<-- first row after 1 second

2 07:02:37.390

3 07:02:38.390

4 07:02:39.390

5 07:02:40.390 --<-- last row after 5 seconds

5 rows selected.

The pipelined function starts to return data as soon as it is prepared (in this case after a 1 second sleep). This means that the consumer is provided with data almost as soon as it is ready and not when the entire rowsource is materialised (in Oracle literature, pipelined functions often feed other pipelined functions). In actual fact, no rowsource is ever materialised with pipelined functions and this leads to memory savings and performance improvements. We will now see the impact on memory.

**streaming vs materialising: memory usage**

To continue with our comparison of pipelined functions with table functions, we will now examine the memory usage and its effect on timings. In the following example, we will create two functions; one table and one pipelined. Both functions will generate 1,000,000 records of 10 bytes. We start by creating a general VARCHAR2 collection type and the two functions.

SQL> CREATE OR REPLACE TYPE varchar2\_ntt AS TABLE OF VARCHAR2(4000);

2 /

Type created.

SQL> CREATE OR REPLACE FUNCTION table\_function RETURN varchar2\_ntt AS

2 nt varchar2\_ntt := varchar2\_ntt();

3 BEGIN

4 FOR i IN 1 .. 1000000 LOOP

5 nt.EXTEND;

6 nt(nt.LAST) := RPAD('x',10);

7 END LOOP;

8 RETURN nt; --<-- return whole collection

9 END table\_function;

10 /

Function created.

SQL> CREATE OR REPLACE FUNCTION pipelined\_function RETURN varchar2\_ntt PIPELINED AS

2 BEGIN

3 FOR i IN 1 .. 1000000 LOOP

4 PIPE ROW (RPAD('x',10)); --<-- send row to consumer

5 END LOOP;

6 RETURN;

7 END pipelined\_function;

8 /

Function created.

Next we create a small function to give us the PGA memory usage in our session.

SQL> CREATE FUNCTION get\_memory RETURN NUMBER IS

2 n NUMBER;

3 BEGIN

4 SELECT value INTO n

5 FROM v$mystat

6 WHERE statistic# = 20; --<-- 'pga memory max'

7 RETURN n;

8 END get\_memory;

9 /

Function created.

We can now compare the memory "footprint" of the functions. For each function we will capture the memory usage, run a full select from the function using autotrace (to limit screen output) and output the memory delta. We will begin with the table function.

SQL> var v\_memory NUMBER;

SQL> exec :v\_memory := get\_memory();

PL/SQL procedure successfully completed.

SQL> set timing on

SQL> set autotrace traceonly statistics

SQL> SELECT \*

2 FROM TABLE(table\_function);

1000000 rows selected.

Elapsed: 00:00:14.03

Statistics

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

21 recursive calls

0 db block gets

23 consistent gets

0 physical reads

0 redo size

13067026 bytes sent via SQL\*Net to client

733825 bytes received via SQL\*Net from client

66668 SQL\*Net roundtrips to/from client

5 sorts (memory)

0 sorts (disk)

1000000 rows processed

SQL> BEGIN

2 DBMS\_OUTPUT.PUT\_LINE(

3 TO\_CHAR(get\_memory() - :v\_memory) || ' bytes of PGA used.'

4 );

5 END;

6 /

**38736696 bytes of PGA used.**

PL/SQL procedure successfully completed.

We can see that the table function used approximately 35Mb of PGA memory to materialise the entire collection. For this reason, table functions are neither scalable nor suitable for concurrency. This somewhat restricts their suitability to smaller array "helper" functions, such as in-list binding of user-inputs (this is a common technique that has become an FAQ on some online forums).

Pipelined functions, on the other hand, are targetted directly at high-volume processing of complex data transformations, which means they are designed to be scalable and efficient. To be scalable, they must be intelligent with memory. To test whether this is the case, we can repeat the simple memory test with our single-attribute pipelined function as follows.

SQL> exec :v\_memory := get\_memory();

PL/SQL procedure successfully completed.

SQL> set autotrace traceonly statistics

SQL> SELECT \*

2 FROM TABLE(pipelined\_function);

1000000 rows selected.

Elapsed: 00:00:12.01

Statistics

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

21 recursive calls

0 db block gets

26 consistent gets

0 physical reads

0 redo size

13067026 bytes sent via SQL\*Net to client

733825 bytes received via SQL\*Net from client

66668 SQL\*Net roundtrips to/from client

5 sorts (memory)

0 sorts (disk)

1000000 rows processed

SQL> BEGIN

2 DBMS\_OUTPUT.PUT\_LINE(

3 TO\_CHAR(get\_memory() - :v\_memory) || ' bytes of PGA used.'

4 );

5 END;

6 /

**196608 bytes of PGA used.**

PL/SQL procedure successfully completed.

We can see a huge saving in the memory footprint. In fact, it doesn't really matter if we process 1 row or 1 million rows from the pipelined function. The memory usage will be low (assuming we don't pipe massive LOBs of course).

**complex pipelined functions**

So far we have seen a simple, single-attribute pipelined function based on a built-in datatype (we have also seen a lot of detail on the mechanics of pipelined functions). Of course, in "real" systems, we are more likely to be processing more complex data structures (i.e. records with multiple attributes). To process complex records, a pipelined function requires three elements as follows:

* a defining record structure. This can be an object type or a PL/SQL record type defined in a package specification;
* a collection type of the underlying record structure. This can be a collection type created via the CREATE TYPE... command or a collection type declared in a PL/SQL package specification; and
* the pipelined function itself, which returns the collection type (as we saw with our simple row-generator example).

Therefore, we have two ways to define the underlying types; either by creating them or by declaring them in a package. We will examine both methods below.

**pipelined functions with user-defined types**

This method requires us to create an object type to define the record structure that is returned by our pipelined function. The object types we create can have methods if required, but in most cases we will use them purely as persistent record definitions.

In the following example, we will create a pipelined function based on the ubiquitous EMP table. We will define our record and collection structures using an object and collection type, respectively. The pipelined function will simply fetch and pipe EMP data. Obviously this is not the intended use for pipelined functions, but the focus here is to demonstrate the mechanics as clearly as possible.

We will begin by creating our EMP "record" as an object type.

SQL> CREATE TYPE emp\_ot AS OBJECT

2 ( empno NUMBER(4)

3 , ename VARCHAR2(10)

4 , job VARCHAR2(9)

5 , mgr NUMBER(4)

6 , hiredate DATE

7 , sal NUMBER(7,2)

8 , comm NUMBER(7,2)

9 , deptno NUMBER(2)

10 );

11 /

Type created.

Next we create a collection type based on this structure. This will be the return type of our pipelined function.

SQL> CREATE TYPE emp\_ntt AS TABLE OF emp\_ot;

2 /

Type created.

Using these types, we will now create our pipelined function to pipe out EMP data as follows. Note the use of the object type constructor to prepare a single row in the correct format for piping.

SQL> CREATE FUNCTION pipelined\_emp RETURN emp\_ntt PIPELINED AS

2 BEGIN

3 FOR r\_emp IN (SELECT \* FROM emp) LOOP

4 PIPE ROW ( emp\_ot( r\_emp.empno,

5 r\_emp.ename,

6 r\_emp.job,

7 r\_emp.mgr,

8 r\_emp.hiredate,

9 r\_emp.sal,

10 r\_emp.comm,

11 r\_emp.deptno ) );

12 END LOOP;

13 RETURN;

14 END pipelined\_emp;

15 /

Function created.

We can now select from this function as follows. The columns names are derived from the underlying object type that we used to define a single record (EMP\_OT).

SQL> SELECT pe.empno

2 , pe.ename

3 , pe.job

4 , pe.sal

5 FROM TABLE(pipelined\_emp) pe;

EMPNO ENAME JOB SAL

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

7369 SMITH CLERK 800

7499 ALLEN SALESMAN 1600

7521 WARD SALESMAN 1250

7566 JONES MANAGER 2975

7654 MARTIN SALESMAN 1250

7698 BLAKE MANAGER 2850

7782 CLARK MANAGER 2450

7788 SCOTT ANALYST 3000

7839 KING PRESIDENT 5000

7844 TURNER SALESMAN 1500

7876 ADAMS CLERK 1100

7900 JAMES CLERK 950

7902 FORD ANALYST 3000

7934 MILLER CLERK 1300

14 rows selected.

**pipelined functions with pl/sql types**

In addition to creating our own types to support a pipelined function, we can also let Oracle do it for us. This alternative method enables us to define our underlying record and collection types inside a PL/SQL package specification. Oracle use these types as the basis for creating a system-generated set of object/collection types to support the pipelined function. We can see a small example of this below.

The following package specification contains an EMP record type and associated collection type. We declare our pipelined function to use these types.

SQL> CREATE PACKAGE pipelined\_pkg AS

2

3 TYPE emp\_rt IS RECORD

4 ( empno NUMBER(4)

5 , ename VARCHAR2(10)

6 , job VARCHAR2(9)

7 , mgr NUMBER(4)

8 , hiredate DATE

9 , sal NUMBER(7,2)

10 , comm NUMBER(7,2)

11 , deptno NUMBER(2)

12 );

13

14 TYPE emp\_ntt IS TABLE OF pipelined\_pkg.emp\_rt;

15

16 FUNCTION pipelined\_emp

17 RETURN pipelined\_pkg.emp\_ntt PIPELINED;

18

19 END pipelined\_pkg;

20 /

Package created.

Remember that Oracle uses these PL/SQL types as the basis for creating SQL types. We can see this in the dictionary as follows. The types are system-named according to the object ID of the package specification.

SQL> SELECT type\_name

2 , typecode

3 , attributes

4 FROM user\_types

5 WHERE INSTR( type\_name, (SELECT object\_id

6 FROM user\_objects

7 WHERE object\_name = 'PIPELINED\_PKG'

8 AND object\_type = 'PACKAGE') ) > 0;

TYPE\_NAME TYPECODE ATTRIBUTES

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

SYS\_PLSQL\_33433\_71\_1 COLLECTION 0

SYS\_PLSQL\_33433\_9\_1 OBJECT 8

SYS\_PLSQL\_33433\_DUMMY\_1 COLLECTION 0

3 rows selected.

SQL> SELECT type\_name

2 , attr\_no

3 , attr\_name

4 , attr\_type\_name

5 FROM user\_type\_attrs

6 WHERE INSTR( type\_name, (SELECT object\_id

7 FROM user\_objects

8 WHERE object\_name = 'PIPELINED\_PKG'

9 AND object\_type = 'PACKAGE') ) > 0

10 ORDER BY

11 attr\_no;

TYPE\_NAME ATTR\_NO ATTR\_NAME ATTR\_TYPE\_NAME

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

SYS\_PLSQL\_33433\_9\_1 1 EMPNO NUMBER

SYS\_PLSQL\_33433\_9\_1 2 ENAME VARCHAR2

SYS\_PLSQL\_33433\_9\_1 3 JOB VARCHAR2

SYS\_PLSQL\_33433\_9\_1 4 MGR NUMBER

SYS\_PLSQL\_33433\_9\_1 5 HIREDATE DATE

SYS\_PLSQL\_33433\_9\_1 6 SAL NUMBER

SYS\_PLSQL\_33433\_9\_1 7 COMM NUMBER

SYS\_PLSQL\_33433\_9\_1 8 DEPTNO NUMBER

8 rows selected.

SQL> SELECT type\_name

2 , coll\_type

3 , elem\_type\_name

4 FROM user\_coll\_types

5 WHERE INSTR( type\_name, (SELECT object\_id

6 FROM user\_objects

7 WHERE object\_name = 'PIPELINED\_PKG'

8 AND object\_type = 'PACKAGE') ) > 0;

TYPE\_NAME COLL\_TYPE ELEM\_TYPE\_NAME

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

SYS\_PLSQL\_33433\_71\_1 TABLE SYS\_PLSQL\_33433\_9\_1

SYS\_PLSQL\_33433\_DUMMY\_1 TABLE NUMBER

2 rows selected.

Note that the record type in this example actually matches the EMP table, so we could instead do the following, replacing emp%ROWTYPE for emp\_rt wherever needed and dispensing with the explicit record type for brevity.

SQL> CREATE OR REPLACE PACKAGE pipelined\_pkg AS

2

3 TYPE emp\_ntt IS TABLE OF emp%ROWTYPE;

4

5 FUNCTION pipelined\_emp

6 RETURN pipelined\_pkg.emp\_ntt PIPELINED;

7

8 END pipelined\_pkg;

9 /

Package created.

The main difference between this and an explicit record structure is that Oracle uses the EMP table as the basis for creating the underlying object type (i.e. we can search the dictionary for the system-generated types with the EMP table's OBJECT\_ID in the name).

Either way, we now have a defining record type (emp%ROWTYPE or emp\_rt) and a collection type (emp\_ntt). We can now implement our pipelined function in the package body. Note that this package body matches our original specification that included the explicit emp\_rt record type.

SQL> CREATE PACKAGE BODY pipelined\_pkg AS

2

3 FUNCTION pipelined\_emp RETURN pipelined\_pkg.emp\_ntt PIPELINED IS

4

5 r\_emp pipelined\_pkg.emp\_rt;

6

7 BEGIN

8

9 FOR r\_tmp IN (SELECT \* FROM emp WHERE ROWNUM <= 5) LOOP

10

11 /\*

12 || Our implicit cursor-for-loop record matches the target

13 || record type so we can actually pipe it out...

14 \*/

15 PIPE ROW (r\_tmp);

16

17 /\*

18 || We are more likely to use an explicit record variable.

19 || Let's pipe out another record...

20 \*/

21 r\_emp := r\_tmp;

22 r\_emp.ename := LOWER(r\_emp.ename); --<-- dummy transformation

23 r\_emp.job := LOWER(r\_emp.job); --<-- dummy transformation

24 PIPE ROW (r\_emp);

25

26 END LOOP;

27

28 RETURN;

29

30 END pipelined\_emp;

31

32 END pipelined\_pkg;

33 /

Package body created.

The key difference to note is that we do not need to "understand" the underlying object type. We work only with our PL/SQL record type, which is possibly more familiar to many developers. Oracle is responsible for casting the records to the underlying type that the SQL engine understands.

For demonstration purposes only, we are piping two records for every source record in this example. This enables us to see that as long as the record variable matches the type that underpins the returning collection, we can pipe it. In our package body we pipe out the implicit record from our cursor-for-loop in addition to the explicit record variable in our function declaration. These happen to be of the same structure. The following SQL statement shows clearly that we have piped two output records for every input record (remember we transformed some of the attributes of the explicit record variable).

SQL> SELECT pe.empno

2 , pe.ename

3 , pe.job

4 , pe.sal

5 FROM TABLE(pipelined\_pkg.pipelined\_emp) pe;

EMPNO ENAME JOB SAL

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

7369 SMITH CLERK 800

7369 smith clerk 800

7499 ALLEN SALESMAN 1600

7499 allen salesman 1600

7521 WARD SALESMAN 1250

7521 ward salesman 1250

7566 JONES MANAGER 2975

7566 jones manager 2975

7654 MARTIN SALESMAN 1250

7654 martin salesman 1250

10 rows selected.

**recap: complex pipelined functions**

To recap therefore, we can use one of two methods to define the underlying record structures for use in our pipelined functions. We can create object and collection types explicitly (CREATE TYPE...) or we can use standard PL/SQL declarations in a package specification (record and collection types). If we choose the former, we create additional dependencies and more source objects, but we have the full implementation of the application under our control. If we choose the latter, we can use familiar PL/SQL record syntax but need to be aware that Oracle will create additional database objects on our behalf.

The examples we have seen so far have been contrived and extremely basic. We have not seen anything in these functions that actually warrants their use! For the remainder of this article we will examine pipelined functions from a more real-life perspective, including options for performance and their practical application.

**parallel pipelined functions**

Pipelined functions have a unique capability over any other form of PL/SQL processing - they are able to be parallelised by Oracle. There are certain conditions required for this to happen, but it basically means that Oracle can divide a unit of serialised PL/SQL processing among a set of parallel slaves.

To create a parallel pipelined function we require two additional elements to our code:

* additional parallel-enabling syntax in our function definition; and
* source data supplied via a cursor variable (i.e. not statically compiled within the function itself as in our previous examples).

We will see a simple example of a parallel pipelined function as follows. We will use explicit types rather than PL/SQL types and, to keep the code simple, we will create a standalone function. We will use a larger dataset based on a multiple of ALL\_SOURCE (we will generate approximately 1 million records) to compare the impact of parallel processing.

First, we create our types as follows.

SQL> CREATE TYPE all\_source\_ot AS OBJECT

2 ( owner VARCHAR2(30)

3 , name VARCHAR2(30)

4 , type VARCHAR2(12)

5 , line NUMBER

6 , text VARCHAR2(4000)

7 );

8 /

Type created.

SQL> CREATE TYPE all\_source\_ntt

2 AS TABLE OF all\_source\_ot;

3 /

Type created.

We create our parallel-enabled pipelined function as follows. Note how we include the session SID in the output. This will enable us to see the effect of parallelism (i.e. each slave will have its own session).

SQL> CREATE FUNCTION parallel\_pipelined\_function(

2 cursor\_in IN SYS\_REFCURSOR

3 ) RETURN all\_source\_ntt

4 PIPELINED

5 PARALLEL\_ENABLE (PARTITION cursor\_in BY ANY) IS

6

7 TYPE incoming\_data\_ntt IS TABLE OF all\_source%ROWTYPE;

8 v\_incoming incoming\_data\_ntt;

9

10 v\_outgoing all\_source\_ot;

11

12 v\_sid INTEGER;

13

14 BEGIN

15

16 /\*

17 || This will help us to see parallelism in action...

18 \*/

19 SELECT sid INTO v\_sid

20 FROM v$mystat

21 WHERE ROWNUM = 1;

22

23 /\*

24 || Process the incoming datasource...

25 \*/

26 LOOP

27

28 FETCH cursor\_in BULK COLLECT INTO v\_incoming LIMIT 500;

29

30 FOR i IN 1 .. v\_incoming.COUNT LOOP

31

32 v\_outgoing := all\_source\_ot( v\_sid,

33 v\_incoming(i).name,

34 v\_incoming(i).type,

35 v\_incoming(i).line,

36 v\_incoming(i).text );

37

38 PIPE ROW (v\_outgoing);

39

40 END LOOP;

41

42 EXIT WHEN cursor\_in%NOTFOUND;

43

44 END LOOP;

45 CLOSE cursor\_in;

46

47 RETURN;

48

49 END parallel\_pipelined\_function;

50 /

Function created.

We can see some new syntax in our parallel pipelined function. In particular, note the following:

* **Line 2:** the function accepts a cursor parameter. This is a prerequisite of parallel pipelined functions, i.e. we cannot parallelise cursors that are embedded within the function itself (such as those in our earlier examples). In our example we have used the built-in weak refcursor type SYS\_REFCURSOR, but we can alternatively use our own weak or strong refcursor types (defined in a package specification);
* **Line 5:** we declare the function parallel-enabled (non-pipelined functions can also be declared with PARALLEL\_ENABLE) and the syntax in parentheses is specific to pipelined functions. The PARTITION BY clause tells Oracle how to divide the source dataset (pointed to by the cursor parameter) among the parallel slaves. For weak refcursors such as SYS\_REFCURSOR, we can only use the ANY keyword (i.e. Oracle will decide how to divide the source data). If we use a strong refcursor type, however, we can partition our source dataset by HASH or RANGE on a named cursor attribute or list of attributes (the attribute names are derived from the strong refcursor's return record type). This is useful when the incoming data needs to be processed in related groups and maybe in a certain order; using RANGE or HASH partitioning ensures that inter-dependent data is sent to the same parallel slave for processing. We will not see any examples of RANGE or HASH parallel pipelined functions in this article, but note that performance will degrade slightly with the PARTITION BY RANGE option as Oracle will need to sort the source data first. The PARTITION BY HASH option displays similar performance to PARTITION BY ANY;
* **Lines 7-8 and 28:** this is purely optional but seeing as we are coding for performance, we will fetch data from the source cursor in arrays using BULK COLLECT;
* **Lines 19-21 and 32:** we have included the session SID in the output. This will enable us to see the effect of parallelism when we use this function later (i.e. each slave will have its own session ID). This is purely for demonstration purposes and has nothing to do with parallel pipelined function syntax; and
* **Line 45:** the cursor parameter is already open when it is passed into the pipelined function. As our function is the only user of this cursor, we need to close it.

Moving on, we now have a parallel pipelined function ready for testing. To test it, we will create a large input dataset based on ALL\_OBJECTS. We will create a single table of this dataset for simplicity as follows.

SQL> CREATE TABLE source\_table

2 NOLOGGING

3 AS

4 SELECT a.\*

5 FROM all\_source a

6 , TABLE(row\_generator(20));

Table created.

SQL> SELECT COUNT(\*) FROM source\_table;

COUNT(\*)

----------

998960

1 row selected.

To test our parallel pipelined function, we will select from it twice; once in parallel and once in serial. We will time each query and verify that Oracle parallelised the query in two ways: first using the returned data itself (remember that we piped the session ID from our function) and second using V$PQ\_SESSTAT. First we will run the parallel version.

SQL> SELECT nt.owner

2 , COUNT(\*)

3 FROM TABLE(

4 parallel\_pipelined\_function(

5 CURSOR(SELECT \* FROM source\_table))) nt

6 GROUP BY

7 nt.owner;

OWNER COUNT(\*)

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

10 496553

13 502407

2 rows selected.

Elapsed: 00:00:38.07

SQL> SELECT \*

2 FROM v$pq\_sesstat

3 WHERE statistic = 'Queries Parallelized';

STATISTIC LAST\_QUERY SESSION\_TOTAL

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

Queries Parallelized 1 1

1 row selected.

We can see that Oracle used two parallel slaves and divided the work relatively evenly between them. Note how we passed in the cursor parameter using the CURSOR expression. This opens the cursor for the embedded SQL statement and passes it through to the pipelined function as a cursor variable.

Finally we can test the parallel pipelined function in serial as follows.

SQL> ALTER SESSION DISABLE PARALLEL QUERY;

Session altered.

SQL> set timing on

SQL> SELECT nt.owner

2 , COUNT(\*)

3 FROM TABLE(

4 parallel\_pipelined\_function(

5 CURSOR(SELECT \* FROM source\_table))) nt

6 GROUP BY

7 nt.owner;

OWNER COUNT(\*)

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

11 998960

1 row selected.

Elapsed: 00:00:49.01

SQL> SELECT \*

2 FROM v$pq\_sesstat

3 WHERE statistic = 'Queries Parallelized';

STATISTIC LAST\_QUERY SESSION\_TOTAL

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

Queries Parallelized 0 2

1 row selected.

This time we can see that the function has executed in serial. It is interesting to note that we haven't saved much time with the parallel execution on the demonstration database. Note that on larger database servers, this approach will yield much better gains where the number of CPUs/slaves will be higher and the degree of parallelism can be controlled by appropriate hints.

**pipelined functions and the optimizer**

We have seen that pipelined functions are alternative rowsources to tables. Our examples in this article have been simple, but usually pipelined functions will be full of complex transformation logic that turns input row A into output row B (and maybe B2, B3 etc). It is usually the case that they will generate significant volumes of data (as they are primarily an ETL tool).

Because pipelined functions generate data, the CBO needs to know how much, especially if a function is one of several rowsources in a SQL statement (i.e. it is joined to tables/views/other pipelined functions). The following demonstrates the execution plan for a simple select from our parallel pipelined function using Autotrace.

SQL> set autotrace traceonly explain

SQL> SELECT \*

2 FROM TABLE(

3 parallel\_pipelined\_function(

4 CURSOR(SELECT \* FROM source\_table))) nt;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=11 Card=8168 Bytes=898480)

1 0 VIEW\* (Cost=11 Card=8168 Bytes=898480) :Q84001

2 1 COLLECTION ITERATOR\* (PICKLER FETCH) OF 'PARALLEL\_PIPELINED\_FUNCTION' :Q84001

3 2 TABLE ACCESS\* (FULL) OF 'SOURCE\_TABLE' (Cost=360 Card=998960 Bytes=87908480) :Q84000

1 PARALLEL\_TO\_SERIAL SELECT C0 FROM (SELECT VALUE(A2) C0 FROM

TABLE("SCOTT"."PARALLEL\_PIPELINED\_FUNCT

2 PARALLEL\_COMBINED\_WITH\_PARENT

3 PARALLEL\_TO\_PARALLEL SELECT /\*+ NO\_EXPAND ROWID(A1) \*/ A1."OWNER"

C0,A1."NAME" C1,A1."TYPE" C2,A1."LI

Two things stand out from this plan. First, the COLLECTION ITERATOR PICKLER FETCH. This is Oracle's mechanism for materialising the collection data from memory. Second, and more important, is the CBO's estimated rowcount. This defaults to 8,168 which is clearly incorrect. There is no way for Oracle to correctly identify the number of rows that will be generated from the execution of a pipelined function, even though it is clear that it is accessing all the source table's data (Step 3). The CBO cannot see inside the function's logic and even if it could, the function might generate multiple rows from one input or might discard most rows due to certain conditions; the possibilities are endless.

Note that Step 3 in the plan is related to parallel execution only. This is the step used by Oracle to assign the incoming data to parallel slaves for processing. It disappears from the plan with parallel query disabled (we will see this below).

There is an undocumented CARDINALITY hint that enables us to tell the CBO how many rows will be generated by a pipelined function. The problem with this hint though is that its behaviour is seemingly erratic (and is not explicitly supported by Oracle). In the following Autotrace explain plans, we can see the effect (or lack of effect) of this hint. First we can see the effect on our simple pipelined\_emp function.

SQL> set autotrace traceonly explain

SQL> SELECT \*

2 FROM TABLE(pipelined\_emp);

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 COLLECTION ITERATOR (PICKLER FETCH) OF 'PIPELINED\_EMP'

SQL> SELECT --+ CARDINALITY(e,1000)

2 \*

3 FROM TABLE(pipelined\_emp) e;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=11 Card=1000 Bytes=2000)

1 0 COLLECTION ITERATOR (PICKLER FETCH) OF 'PIPELINED\_EMP'

In the hinted version, we have told the CBO that our function will generate 1,000 records and we can see this in the plan. In the unhinted version, Oracle does not even make an estimate (this appears to be the case with selects from pipelined functions without either a join or an input cursor). Next we can see the effect of this hint with a join.

SQL> SELECT \*

2 FROM dept d

3 , TABLE(pipelined\_emp) e

4 WHERE d.deptno = e.deptno;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=14 Card=8168 Bytes=163360)

1 0 HASH JOIN (Cost=14 Card=8168 Bytes=163360)

2 1 TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)

3 1 COLLECTION ITERATOR (PICKLER FETCH) OF 'PIPELINED\_EMP'

SQL> SELECT --+ CARDINALITY(e,100000)

2 \*

3 FROM dept d

4 , TABLE(pipelined\_emp) e

5 WHERE d.deptno = e.deptno;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=14 Card=100000 Bytes=2000000)

1 0 HASH JOIN (Cost=14 Card=100000 Bytes=2000000)

2 1 TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)

3 1 COLLECTION ITERATOR (PICKLER FETCH) OF 'PIPELINED\_EMP'

We can see the 8,168 figure appearing again in our first example. It is interesting that this time the cardinality is applied to the join and not the pipelined function fetch. We can see this more clearly with the second example where we told the CBO that the function would generate 100,000 rows. Oracle has assumed a join for every row and applied this cardinality to Step 1 (hash join between the two rowsources). If we return to our parallel pipelined function, we can see that this hint has no such effect; in fact, it is not recognised at all (note that this is a serial plan, rather than parallel, which explains why the CURSOR SQL does not appear).

SQL> SELECT \*

2 FROM TABLE(

3 parallel\_pipelined\_function(

4 CURSOR(SELECT \* FROM source\_table))) nt;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=11 Card=8168 Bytes=898480)

1 0 VIEW (Cost=11 Card=8168 Bytes=898480)

2 1 COLLECTION ITERATOR (PICKLER FETCH) OF 'PARALLEL\_PIPELINED\_FUNCTION'

SQL> SELECT --+ CARDINALITY(nt, 100000)

2 \*

3 FROM TABLE(

4 parallel\_pipelined\_function(

5 CURSOR(SELECT \* FROM source\_table))) nt;

Execution Plan

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

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=11 Card=8168 Bytes=898480)

1 0 VIEW (Cost=11 Card=8168 Bytes=898480)

2 1 COLLECTION ITERATOR (PICKLER FETCH) OF 'PARALLEL\_PIPELINED\_FUNCTION'

As a strategy, therefore, the undocumented CARDINALITY hint does not seem very suitable! Its behaviour is inconsistent and applies to very limited scenarios. For "sensitive" queries involving joins to pipelined functions, we would be far better using some other form of plan stability, whether it be by hints or stored outlines.

**practical uses for pipelined functions**

As has been stated, the examples in this article are deliberately simple for the purposes of demonstration. In "real" systems and applications, pipelined functions have far more flexibility and can solve a number of problems, including performance issues. To complete this article, I will briefly describe some examples of my pipelined function implementations.

**usage 1: pl/sql-based etl**

This is by far the most important and exciting application for pipelined functions and is how Oracle markets the technology. In many cases, PL/SQL ETL routines seem to follow a standard "cursor-for-loop" process that can be expressed by the following pseudo-code:

PROCEDURE load\_business\_data IS

CURSOR cursor\_name IS

SELECT some\_columns

FROM some\_staging\_tables

WHERE they\_join\_and\_match\_certain\_criteria;

BEGIN

FOR record IN cursor LOOP

--[do transformations, lookups, validations]--

INSERT INTO parent\_table VALUES (record.attributes, variables, etc...);

INSERT INTO child\_table VALUES (record.attributes, variables, etc...);

--[and so on]--

END LOOP;

COMMIT;

END;

Sometimes, the "do transformations, lookups, validations" stage is just too complex to be expressed as a bulk SQL statement (i.e. the "best practice" for ETL in Oracle). Pipelined functions can therefore transform these ETL processes to provide a middle-ground between the slow PL/SQL-only implementation and the fast SQL-only implementation. The pseudo-code for a pipelined function-based ETL process is as follows:

PIPELINED FUNCTION

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

FUNCTION pipelined\_transformation(

cursor\_in IN refcursor\_type

) RETURN collection\_type PIPELINED PARALLEL\_ENABLE (PARTITION cursor\_in BY ANY) IS

fetch\_array array\_type;

pipe\_record record\_type;

BEGIN

LOOP

FETCH cursor\_in BULK COLLECT INTO fetch\_array LIMIT arraysize;

EXIT WHEN fetch\_array.COUNT = 0;

FOR i in 1 .. fetch\_array.COUNT LOOP

--[do transformations, lookups, validations]--

--[prepare pipe\_record]--

PIPE ROW (pipe\_record);

END LOOP;

END LOOP;

CLOSE cursor\_in;

RETURN;

END;

LOAD PROCEDURE

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

PROCEDURE load\_business\_data IS

BEGIN

INSERT ALL

INTO parent\_table VALUES (...)

INTO child\_table VALUES (...)

--[and so on]--

SELECT \*

FROM TABLE(

pipelined\_transformation(

CURSOR( SELECT some\_columns

FROM some\_staging\_tables

WHERE they\_join\_and\_match\_certain\_criteria )));

COMMIT;

END;

I have used this technique several times and it is very easy to convert existing row-by-row PL/SQL routines to use this method. In general, the main body of the existing code (the looping PL/SQL parts) become the pipelined function logic. Insert statements are replaced with PIPE ROW statements and the embedded SQL cursor is removed and a cursor parameter added (if the aim is to use parallel SQL and DML). The existing load procedure itself becomes an INSERT..SELECT from the new pipelined function as seen in the pseduo-code above. The performance gains from this re-factoring are two-fold: first, the performance benefits of a bulk SQL INSERT statement; second the potentially massive performance gains of parallel SQL and DML. With the former in serial mode I have achieved up to 30% gain in performance, but with the latter I have achieved up to 80% reduction in load time.

Note that it is also possible to have the pipelined function return different record types for loads involving multiple tables. This will be the subject of a future article.

**usage 2: wrappers over dictionary views with long columns**

Several views in the data dictionary have LONG columns. These are problematic as we cannot search inside or copy these columns easily. The common views where this causes problems are XXX\_VIEWS, XXX\_TRIGGERS, XXX\_TAB\_COLUMNS, XXX\_CONSTRAINTS, XXX\_TAB\_PARTITIONS and XXX\_SUBPARTITIONS (where XXX = USER/ALL/DBA as appropriate). Using pipelined functions, we can create a simple wrapper over each view to overcome these problems, taking advantage of PL/SQL's ability to implicitly convert LONG columns to VARCHAR2 (up to 32,767 bytes).

For the underlying object type to each pipelined function, we take a copy of the respective dictionary view structure but with the LONG changed to a CLOB for the relevant attribute. During processing, any LONG values that exceed 32,767 bytes (some large view-texts for example) are handled as special cases using DBMS\_SQL. As these are exceptional cases, they do not generally slow down the processing of the majority of records that have LONG values under 32,767 bytes in length. We can then add a view over the pipelined function and read and query the CLOB column as required. In 9i, CLOB columns can be interrogated using Oracle's standard string functions such as INSTR, SUBSTR and LIKE. For additional functionality, we can also add filters to the views using application contexts. This prevents us from returning every record from the underlying views with every query we issue.

**usage 3: querying objects with stale/empty statistics**

The DBMS\_STATS.GATHER\_SCHEMA\_STATS procedure contains an option to list tables and indexes with stale or empty statistics. As this is a PL/SQL procedure that returns an index-by array of a PL/SQL record, its use is restricted to PL/SQL. However, we can easily add a pipelined function (or even a simple table function) wrapper to this procedure to be able to present a SQL view of the objects. We can also create a view to encapsulate the pipelined function call.

**further reading**

For an alternative overview of table and pipelined functions, see the online [PL/SQL User's Guide and Reference](http://docs.oracle.com/cd/B10501_01/appdev.920/a96624/08_subs.htm#19677).

**acknowledgements**

My original parallel\_pipelined\_function example relied on the V$PQ\_SESSTAT view to demonstrate that parallelism was happening. I have since updated the example to include each slave's SID from V$MYSTAT in the output. This method is found in Tom Kyte's [Expert Oracle Database Architecture](http://www.amazon.co.uk/Expert-Oracle-Database-Architecture-Programming/dp/1590595300/ref=pd_bbs_1/203-3812606-2581523?ie=UTF8&s=books&qid=1179865574&sr=8-1).

**source code**

The source code for the examples in this article can be downloaded from [here](http://www.oracle-developer.net/content/code/207.zip).

Adrian Billington, May 2002 (updated May 2007)