The WITH FUNCTION feature is a very useful enhancement to the SQL language. You should, however, ask yourself this question each time you contemplate using it: "Do I need this same functionality in multiple places in my application?"

If the answer is "yes," then you should decide if the performance improvement of using WITH FUNCTION outweighs the potential downside of needing to copy and paste this logic into multiple SQL statements.

Note that as of 12.1 you cannot execute a static SELECT statement that contains a WITH FUNCTION clause *inside a PL/SQL block*. I know that seems very strange, and I am sure it will be possible in 12.2, but for now, the following code will raise an error as shown:

```
SQL> BEGIN
  2     WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
  3             RETURN VARCHAR2
  4          IS
  5          BEGIN
  6             RETURN fname_in || ' ' || lname_in;
  7          END;
  8
  9     SELECT LENGTH (full_name (first_name, last_name))
 10       INTO c
 11       FROM employees;
 12
 13     DBMS_OUTPUT.put_line ('count = ' || c);
 14  END;
 15  /
WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
               *
ERROR at line 2:
ORA-06550: line 2, column 18:
PL/SQL: ORA-00905: missing keyword
```

In addition to the WITH FUNCTION clause, 12.1 also offers the UDF pragma, which can improve the performance of PL/SQL functions executed from SQL. See Chapter 21 for details.

Table Functions

A *table function* is a function that can be called from within the FROM clause of a query, as if it were a relational table. Table functions return collections, which can then be transformed with the TABLE operator into a structure that can be queried using the SQL language. Table functions come in very handy when you need to:

- Perform very complex transformations of data, requiring the use of PL/SQL, but you need to access that data from within a SQL statement.
- Pass complex result sets back to the host (that is, non-PL/SQL) environment. You can open a cursor variable for a query based on a table function, and let the host environment fetch through the cursor variable.

Table functions open up all sorts of possibilities for PL/SQL developers, and to demonstrate some of those possibilities we will explore both streaming table functions and pipelined table functions in more detail in this chapter:

Streaming table functions
: *Data streaming* enables you to pass from one process or stage to another without having to rely on intermediate structures. Table functions, in conjunction with the CURSOR expression, enable you to stream data through multiple transformations, all within a single SQL statement.

Pipelined table functions
: These functions return a result set in *pipelined* fashion, meaning that data is returned while the function is still executing. Add the PARALLEL_ENABLE clause to a pipelined function's header, and you have a function that will execute in parallel within a parallel query.

Prior to Oracle Database 12*c*, table functions could return only nested tables and VARRAYs. From 12.1, you can also define table functions that return an integer-indexed associative array whose type is defined in a package specification.

Let's explore how to define table functions and put them to use in an application.

Calling a function in a FROM clause

To call a function from within a FROM clause, you must do the following:

- Define the RETURN datatype of the function to be a collection (either a nested table or a VARRAY).
- Make sure that all of the other parameters to the function are of mode IN and have SQL datatypes. (You cannot, for example, call a function with a Boolean or record type argument inside a query.)
- Embed the call to the function inside the TABLE operator (if you are running Oracle8*i* Database, you will also need to use the CAST operator).

Here is a simple example of a table function. First, I will create a nested table type based on an object type of pets:

```
/* File on web: pet_family.sql */
CREATE TYPE pet_t IS OBJECT (
   name   VARCHAR2 (60),
   breed  VARCHAR2 (100),
   dob    DATE);

CREATE TYPE pet_nt IS TABLE OF pet_t;
```

Now I will create a function named pet_family. It accepts two pet objects as arguments: the mother and the father. Then, based on the breed, it returns a nested table with the entire family defined in the collection:

```
FUNCTION pet_family (dad_in IN pet_t, mom_in IN pet_t)
   RETURN pet_nt
IS
   l_count PLS_INTEGER;
   retval   pet_nt := pet_nt ();

   PROCEDURE extend_assign (pet_in IN pet_t) IS
   BEGIN
      retval.EXTEND;
      retval (retval.LAST) := pet_in;
   END;
BEGIN
   extend_assign (dad_in);
   extend_assign (mom_in);

   IF    mom_in.breed = 'RABBIT'   THEN l_count := 12;
   ELSIF mom_in.breed = 'DOG'      THEN l_count := 4;
   ELSIF mom_in.breed = 'KANGAROO' THEN l_count := 1;
   END IF;

   FOR indx IN 1 .. l_count
   LOOP
      extend_assign (pet_t ('BABY' || indx, mom_in.breed, SYSDATE));
   END LOOP;

   RETURN retval;
END;
```

The pet_family function is silly and trivial; the point to understand here is that your PL/SQL function may contain extremely complex logic—whatever is required within your application and can be accomplished with PL/SQL—that exceeds the expressive capabilities of SQL.

Now I can call this function in the FROM clause of a query, as follows:

```
SELECT pets.NAME, pets.dob
 FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                       , pet_t ('Hippy', 'RABBIT', SYSDATE)
```

```
                        )
            ) pets;
```

And here is a portion of the output:

```
NAME       DOB
---------- ---------
Hoppy      27-FEB-02
Hippy      27-FEB-02
BABY1      27-FEB-02
BABY2      27-FEB-02
...
BABY11     27-FEB-02
BABY12     27-FEB-02
```

Passing table function results with a cursor variable

Table functions help overcome a problem that developers have encountered in the past—namely, how do I pass data that I have produced through PL/SQL-based programming (i.e., data that is not intact inside one or more tables in the database) back to a non-PL/SQL host environment? Cursor variables allow me to easily pass back SQL-based result sets to, say, Java programs, because cursor variables are supported in JDBC. Yet if I first need to perform complex transformations in PL/SQL, how then do I offer that data to the calling program?

Now, we can combine the power and flexibility of table functions with the wide support for cursor variables in non-PL/SQL environments (explained in detail in Chapter 15) to solve this problem.

Suppose, for example, that I need to generate a pet family (bred through a call to the pet_family function, as shown in the previous section) and pass those rows of data to a frontend application written in Java. I can do this very easily as follows:

```
/* File on web: pet_family.sql */
FUNCTION pet_family_cv
   RETURN SYS_REFCURSOR
IS
   retval SYS_REFCURSOR;
BEGIN
   OPEN retval FOR
      SELECT *
        FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                              , pet_t ('Hippy', 'RABBIT', SYSDATE)
                               )
                   );

   RETURN retval;
END pet_family_cv;
```

In this program, I am taking advantage of the predefined weak REF CURSOR type, SYS_REFCURSOR (introduced in Oracle9*i* Database), to declare a cursor variable. I

OPEN FOR this cursor variable, associating with it the query that is built around the pet_family table function.

I can then pass this cursor variable back to the Java frontend. Because JDBC recognizes cursor variables, the Java code can then easily fetch the rows of data and integrate them into the application.

Creating a streaming function

A *streaming function* accepts as a parameter a result set (via a CURSOR expression) and returns a result set in the form of a collection. Because you can apply the TABLE operator to this collection and then query from it in a SELECT statement, these functions can perform one or more transformations of data within a single SQL statement.

Streaming functions, support for which was added in Oracle9*i* Database, can be used to hide algorithmic complexity behind a function interface and thus simplify the SQL in your application. I will walk through an example to explain the kinds of steps you will need to go through yourself to take advantage of table functions in this way.

Consider the following scenario. I have a table of stock ticker information that contains a single row for the open and close prices of a stock:

```
/* File on web: tabfunc_streaming.sql */
TABLE stocktable (
  ticker VARCHAR2(10),
  trade_date DATE,
  open_price NUMBER,
  close_price NUMBER)
```

I need to transform (or *pivot*) that information into another table:

```
TABLE tickertable (
  ticker VARCHAR2(10),
  pricedate DATE,
  pricetype VARCHAR2(1),
  price NUMBER)
```

In other words, a single row in stocktable becomes two rows in tickertable. There are many ways to achieve this goal. A very traditional and straightforward approach in PL/SQL might look like this:

```
FOR rec IN  (SELECT * FROM stocktable)
LOOP
   INSERT INTO tickertable
               (ticker, pricetype, price)
        VALUES (rec.ticker, 'O', rec.open_price);

   INSERT INTO tickertable
               (ticker, pricetype, price)
        VALUES (rec.ticker, 'C', rec.close_price);
END LOOP;
```

There are also 100% SQL solutions, such as:

```
INSERT ALL
      INTO tickertable
           (ticker, pricedate, pricetype, price
           )
    VALUES (ticker, trade_date, 'O', open_price
           )
      INTO tickertable
           (ticker, pricedate, pricetype, price
           )
    VALUES (ticker, trade_date, 'C', close_price
           )
   SELECT ticker, trade_date, open_price, close_price
     FROM stocktable;
```

Let's assume, however, that the transformation that I must perform to move data from stocktable to tickertable is very complex and requires the use of PL/SQL. In this situation, a table function used to stream the data as it is transformed would offer a much more efficient solution.

First of all, if I am going to use a table function, I will need to return a nested table or VARRAY of data. I will use a nested table because VARRAYs require the specification of a maximum size, and I don't want to have that restriction in my implementation. This nested table type must be defined as a schema-level type or within the specification of a package, so that the SQL engine can resolve a reference to a collection of this type.

I would like to return a nested table based on the table definition itself—that is, I would like it to be defined as follows:

```
TYPE tickertype_nt IS TABLE of tickertable%ROWTYPE;
```

Unfortunately, this statement will fail because %ROWTYPE is not a SQL-recognized type. That attribute is available only inside a PL/SQL declaration section. So, I must instead create an object type that mimics the structure of my relational table, and then define a nested table TYPE against that object type:

```
TYPE TickerType AS OBJECT (
  ticker VARCHAR2(10),
  pricedate DATE
  pricetype VARCHAR2(1),
  price NUMBER);

TYPE TickerTypeSet AS TABLE OF TickerType;
```

For my table function to stream data from one stage of transformation to the next, it will have to accept as its argument a set of data—in essence, a query. The only way to do that is to pass in a cursor variable, so I will need a REF CURSOR type to use in the parameter list of my function.

I create a package to hold the REF CURSOR type based on my new nested table type:

```
PACKAGE refcur_pkg
IS
   TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
```

Finally, I can write my stock pivot function:

```
     /* File on web: tabfunc_streaming.sql */
1    FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
2       RETURN tickertypeset
3    IS
4       l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
5       l_row_from_query  dataset%ROWTYPE;
6       retval tickertypeset := tickertypeset ();
7    BEGIN
8       LOOP
9          FETCH dataset
10           INTO l_row_from_query;
11
12         EXIT WHEN dataset%NOTFOUND;
13         -- Create the OPEN object type instance
14         l_row_as_object.ticker := l_row_from_query.ticker;
15         l_row_as_object.pricetype := 'O';
16         l_row_as_object.price := l_row_from_query.open_price;
17         l_row_as_object.pricedate := l_row_from_query.trade_date;
18         retval.EXTEND;
19         retval (retval.LAST) := l_row_as_object;
20         -- Create the CLOSED object type instance
21         l_row_as_object.pricetype := 'C';
22         l_row_as_object.price := l_row_from_query.close_price;
23         retval.EXTEND;
24         retval (retval.LAST) := l_row_as_object;
25      END LOOP;
26
27      CLOSE dataset;
28
29      RETURN retval;
30   END stockpivot;
```

As with the pet_family function, the specifics of this program are not important, and your own transformation logic will be qualitatively more complex. The basic steps performed here, however, will likely be repeated in your own code, so I will review them in the following table.

Line(s)	Description
1–2	The function header: pass in a result set as a cursor variable, and return a nested table based on the object type.
4	Declare a local object, which will be used to populate the nested table.
5	Declare a local record based on the result set. This will be populated by the FETCH from the cursor variable.
6	The local nested table that will be returned by the function.

Line(s)	Description
8–12	Start up a simple loop to fetch each row separately from the cursor variable, terminating the loop when no more data is in the cursor.
14–19	Use the "open" data in the record to populate the local object, and then place it in the nested table, after EXTENDing to define the new row.
21–25	Use the "close" data in the record to populate the local object, and then place it in the nested table, after EXTENDing to define the new row.
27–30	Close the cursor and return the nested table. Mission completed. Really.

And now that I have this function in place to do all the fancy but necessary footwork, I can use it inside my query to stream data from one table to another:

```
BEGIN
   INSERT INTO tickertable
      SELECT *
        FROM TABLE (stockpivot (CURSOR (SELECT *
                                          FROM stocktable)));
END;
```

My inner SELECT retrieves all rows in the stocktable. The CURSOR expression around that query transforms the result set into a cursor variable, which is passed to stockpivot. That function returns a nested table, and the TABLE operator then translates it into a relational table format that can be queried.

It may not be magic, but it *is* a bit magical, wouldn't you say? Well, if you think a streaming function is special, check out pipelined functions!

Creating a pipelined function

A *pipelined function* is a table function that returns a result set as a collection but does so asynchronously to the termination of the function. In other words, the database no longer waits for the function to run to completion, storing all the rows it computes in the PL/SQL collection, before it delivers the first rows. Instead, as each row is ready to be assigned into the collection, it is piped out of the function. This section describes the basics of pipelined table functions. The performance implications of these functions are explored in detail in Chapter 21.

Let's take a look at a rewrite of the stockpivot function and see more clearly what is needed to build pipelined functions:

```
      /* File on web: tabfunc_pipelined.sql */
1     FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
2     RETURN tickertypeset PIPELINED
3     IS
4        l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
5        l_row_from_query  dataset%ROWTYPE;
6     BEGIN
7        LOOP
```

```
 8          FETCH dataset INTO l_row_from_query;
 9          EXIT WHEN dataset%NOTFOUND;
10
11          -- first row
12          l_row_as_object.ticker := l_row_from_query.ticker;
13          l_row_as_object.pricetype := 'O';
14          l_row_as_object.price := l_row_from_query.open_price;
15          l_row_as_object.pricedate := l_row_from_query.trade_date;
16          PIPE ROW (l_row_as_object);
17
18          -- second row
19          l_row_as_object.pricetype := 'C';
20          l_row_as_object.price := l_row_from_query.close_price;
21          PIPE ROW (l_row_as_object);
22       END LOOP;
23
24       CLOSE dataset;
25       RETURN;
26    END;
```

The following table notes several changes to our original functionality.

Line(s)	Description
2	The only change from the original stockpivot function is the addition of the PIPELINED keyword.
4–5	Declare a local object and local record, as with the first stockpivot. What's striking about these lines is what I *don't* declare—namely, the nested table that will be returned by the function. A hint of what is to come...
7–9	Start up a simple loop to fetch each row separately from the cursor variable, terminating the loop when no more data is in the cursor.
12–15 and 19–21	Populate the local object for the open and close tickertable rows to be placed in the nested table.
16 and 21	Use the PIPE ROW statement (valid only in pipelined functions) to "pipe" the objects immediately out from the function.
25	At the bottom of the executable section, the function doesn't return anything! Instead, it calls the unqualified RETURN (formerly allowed only in procedures) to return control to the calling block. The function already returned all of its data with the PIPE ROW statements.

You can call the pipelined function as you would the nonpipelined version. You won't see any difference in behavior, unless you set up the pipelined function to be executed in parallel as part of a parallel query (covered in the next section) or include logic that takes advantage of the asynchronous return of data.

Consider, for example, a query that uses the ROWNUM pseudocolumn to restrict the rows of interest:

```
BEGIN
   INSERT INTO tickertable
      SELECT *
        FROM TABLE (stockpivot (CURSOR (SELECT *
                                          FROM stocktable)))
```

```
        WHERE ROWNUM < 10;
END;
```

My tests show that on Oracle Database 10*g* and Oracle Database 11*g*, if I pivot 100,000 rows into 200,000 and then return only the first 9 rows, the pipelined version completes its work in 0.2 seconds, while the nonpipelined version takes 4.6 seconds.

Clearly, piping rows back does work and does make a noticeable difference!

Enabling a function for parallel execution

One enormous step forward for PL/SQL, introduced first in Oracle9*i* Database, is the ability to execute functions within a parallel query context. Prior to Oracle9*i* Database, a call to a PL/SQL function inside SQL caused serialization of that query—a major problem for data warehousing applications. You can now add information to the header of a pipelined function in order to instruct the runtime engine how the data set being passed into the function should be partitioned for parallel execution.

In general, if you would like your function to execute in parallel, it must have a single, strongly typed REF CURSOR input parameter.[1]

Here are some examples:

- Specify that the function can run in parallel and that the data passed to that function can be partitioned arbitrarily:

  ```
  FUNCTION my_transform_fn (
       p_input_rows in employee_info.recur_t )
    RETURN employee_info.transformed_t
    PIPELINED
    PARALLEL_ENABLE ( PARTITION p_input_rows BY ANY )
  ```

  In this example, the keyword ANY expresses the programmer's assertion that the results are independent of the order in which the function gets the input rows. When this keyword is used, the runtime system randomly partitions the data among the various query processes. This keyword is appropriate for use with functions that take in one row, manipulate its columns, and generate output rows based on the columns of this row only. If your program has other dependencies, the outcome will be unpredictable.

- Specify that the function can run in parallel, that all the rows for a given department go to the same process, and that all of these rows are delivered consecutively:

  ```
  FUNCTION my_transform_fn (
     p_input_rows in employee_info.recur_t )
  RETURN employee_info.transformed_t
  PIPELINED
  ```

1. The input REF CURSOR need *not* be strongly typed to be partitioned by ANY.

```
CLUSTER P_INPUT_ROWS BY (department)
PARALLEL_ENABLE
  ( PARTITION P_INPUT_ROWS BY HASH (department) )
```

Oracle uses the term *clustered* to signify this type of delivery, and *cluster key* for the column (in this case, department) on which the aggregation is done. But significantly, the algorithm does *not* care in what order of cluster key it receives each successive cluster, and Oracle doesn't guarantee any particular order here. This allows for a quicker algorithm than if rows were required to be clustered and delivered in the order of the cluster key. It scales as *order N* rather than *order N.log(N)*, where *N* is the number of rows.

In this example, I can choose between HASH (department) and RANGE (department), depending on what I know about the distribution of the values. HASH is quicker than RANGE and is the natural choice to be used with CLUSTER...BY.

- Specify that the function can run in parallel and that the rows that are delivered to a particular process, as directed by PARTITION...BY (for that specified partition), will be locally sorted by that process. The effect will be to parallelize the sort:

```
FUNCTION my_transform_fn (
   p_input_rows in employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
ORDER P_INPUT_ROWS BY (C1)
PARALLEL_ENABLE
  ( PARTITION P_INPUT_ROWS BY RANGE (C1) )
```

Because the sort is parallelized, there should be no ORDER...BY in the SELECT used to invoke the table function. (In fact, an ORDER...BY clause in the SELECT statement would subvert the attempt to parallelize the sort.) Thus, it's natural to use the RANGE option together with the ORDER...BY option. This will be slower than CLUSTER...BY, and so should be used only when the algorithm depends on it.

The CLUSTER...BY construct can't be used together with the ORDER...BY in the declaration of a table function. This means that an algorithm that depends on clustering on one key, c1, and then on ordering within the set row for a given value of c1 by, say, c2, would have to be parallelized by using the ORDER...BY in the declaration in the table function.

Deterministic Functions

A function is considered to be *deterministic* if it returns the same result value whenever it is called with the same values for its IN and IN OUT arguments. Another way to think about deterministic programs is that they have no side effects. Everything the program changes is reflected in the parameter list.