La función CON FUNCIÓN es una mejora muy útil para el lenguaje SQL. Sin embargo, debe hacerse esta pregunta cada vez que considere usarla: "¿Necesito esta misma funcionalidad en varios lugares de mi aplicación?"

Si la respuesta es "sí", debe decidir si la mejora del rendimiento de usar WITH FUNCTION supera la posible desventaja de tener que copiar y pegar esta lógica en varias instrucciones SQL.

Tenga en cuenta que a partir de 12.1 no puede ejecutar una instrucción SELECT estática que contenga una cláusula WITH FUNCTION*dentro de un bloque PL/SQL*. Sé que parece muy extraño, y estoy seguro de que será posible en 12.2, pero por ahora, el siguiente código generará un error como se muestra:

```
SQL>COMENZAR
  2      CON FUNCIÓN full_name (fname_in EN VARCHAR2, lname_in EN VARCHAR2)
  3              DEVOLVER VARCHAR2
  4          ES
  5              COMENZAR
  6              VOLVER fname_in || ' ' || nombre_en;
  7              FIN;
  8
  9      SELECCIONE LONGITUD (nombre completo (nombre, apellido))
 10        EN c
 11        DE empleados;
 12
 13      DBMS_OUTPUT.put_line ('cuenta = ' || c);
 14      FIN;
 15  /
CON FUNCIÓN full_name (fname_in EN VARCHAR2, lname_in EN VARCHAR2)
                      *
ERROR en la línea 2: ORA-06550: línea 2,
columna 18: PL/SQL: ORA-00905: palabra
clave faltante
```

Además de la cláusula WITH FUNCTION, 12.1 también ofrece el pragma UDF, que puede mejorar el rendimiento de las funciones PL/SQL ejecutadas desde SQL. Vercapitulo 21para detalles.

Funciones de tabla

A*función de tabla*es una función que se puede llamar desde dentro de la cláusula FROM de una consulta, como si fuera una tabla relacional. Las funciones de tabla devuelven colecciones, que luego se pueden transformar con el operador TABLE en una estructura que se puede consultar utilizando el lenguaje SQL. Las funciones de tabla son muy útiles cuando necesita:

- Realizar transformaciones de datos muy complejas, que requieren el uso de PL/SQL, pero necesita acceder a esos datos desde dentro de una instrucción SQL.
- Pasar conjuntos de resultados complejos al entorno host (es decir, no PL/SQL). Puede abrir una variable de cursor para una consulta basada en una función de tabla y dejar que el entorno del host obtenga la variable de cursor.

Las funciones de tabla abren todo tipo de posibilidades para los desarrolladores de PL/SQL y, para demostrar algunas de esas posibilidades, exploraremos las funciones de tabla de transmisión y las funciones de tabla segmentadas con más detalle en este capítulo:

Funciones de la tabla de transmisión

transmisión de datos le permite pasar de un proceso o etapa a otra sin tener que depender de estructuras intermedias. Las funciones de tabla, junto con la expresión CURSOR, le permiten transmitir datos a través de múltiples transformaciones, todo dentro de una sola instrucción SQL.

Funciones de tabla canalizadas

Estas funciones devuelven un conjunto de resultados en *canalizado* moda, lo que significa que los datos se devuelven mientras la función aún se está ejecutando. Agregue la cláusula PARALLEL_ENABLE al encabezado de una función canalizada y tendrá una función que se ejecutará en paralelo dentro de una consulta paralela.

Antes de Oracle Database 12*C*, las funciones de tabla solo pueden devolver tablas anidadas y VARRAY. Desde 12.1, también puede definir funciones de tabla que devuelvan una matriz asociativa indexada por enteros cuyo tipo se define en una especificación de paquete.

Exploremos cómo definir funciones de tabla y ponerlas en uso en una aplicación.

Llamar a una función en una cláusula FROM

Para llamar a una función desde dentro de una cláusula FROM, debe hacer lo siguiente:

- Defina el tipo de datos RETURN de la función para que sea una colección (ya sea una tabla anidada o un VARRAY).
- Asegúrese de que todos los demás parámetros de la función sean del modo IN y tengan tipos de datos SQL. (Por ejemplo, no puede llamar a una función con un argumento booleano o de tipo de registro dentro de una consulta).
- Incruste la llamada a la función dentro del operador TABLE (si está ejecutando Oracle8). *i*base de datos, también deberá utilizar el operador CAST).

Aquí hay un ejemplo simple de una función de tabla. Primero, crearé un tipo de tabla anidada basada en un tipo de objeto de mascotas:

```
/* Archivo en la web: pet_family.sql */
CREATE TYPE pet_t IS OBJECT (
    nombre          VARCHAR2 (60),
    criar           VARCHAR2 (100),
    fecha de nacimiento FECHA);

CREAR TIPO pet_nt ES TABLA DE pet_t;
```

Ahora crearé una función llamada pet_family. Acepta dos objetos favoritos como argumentos: la madre y el padre. Luego, según la raza, devuelve una tabla anidada con toda la familia definida en la colección:

```
FUNCIÓN pet_family (dad_in IN pet_t, mom_in IN pet_t)
    VOLVER pet_nt
ES
    l_cuenta PLS_INTEGER;
    retval pet_nt := pet_nt ();

    COMIENZA EL PROCEDIMIENTO extend_assign (pet_in IN
    pet_t)
        retval.EXTEND;
        retval (retval.LAST) := pet_in; FIN;

COMENZAR
    extender_asignar (papá_en);
    extender_asignar (mamá_en);

    SI      mom_in.breed = 'CONEJO'      ENTONCES l_count := 12;
    ELSIF mom_in.breed = 'PERRO'         ENTONCES l_count := 4;
    ELSIF mom_in.breed = 'CANGURO' THEN l_count := 1;
    TERMINARA SI;

    PARA indx EN 1 .. l_count
    BUCLE
        extend_assign (pet_t ('BEBE' || indx, mom_in.breed, SYSDATE)); FIN DEL
        BUCLE;

    RETORNO recuperación;
FIN;
```

La función pet_family es tonta y trivial; El punto a entender aquí es que su función PL/SQL puede contener una lógica extremadamente compleja, lo que sea que se requiera dentro de su aplicación y se pueda lograr con PL/SQL, que excede las capacidades expresivas de SQL.

Ahora puedo llamar a esta función en la cláusula FROM de una consulta, de la siguiente manera:

```
SELECCIONA mascotas.NOMBRE, mascotas.dob
  DESDE LA TABLA (mascota_familia (mascota_t ('Hoppy', 'RABBIT', SYSDATE)
                                 , mascota_t ('Hippy', 'CONEJO', SYSDATE)
```

```
                                    )
                ) mascotas;
```

Y aquí hay una parte de la salida:

```
NOMBRE          fecha de nacimiento
- - - - - - - - - - - - - - - - - - -
Lúpulo          27-FEB-02
hippy           27-FEB-02
BEBÉ1           27-FEB-02
BEBÉ2           27-FEB-02
. . .
BEBÉ11          27-FEB-02
BEBÉ12          27-FEB-02
```

Pasar los resultados de la función de tabla con una variable de cursor

Las funciones de tabla ayudan a superar un problema que los desarrolladores han encontrado en el pasado — a saber, ¿cómo paso los datos que he producido a través de la programación basada en PL/SQL (es decir, datos que no están intactos dentro de una o más tablas en la base de datos) de vuelta a un entorno de host que no es PL/SQL? Las variables de cursor me permiten pasar fácilmente conjuntos de resultados basados en SQL a, por ejemplo, programas Java, porque las variables de cursor son compatibles con JDBC. Sin embargo, si primero necesito realizar transformaciones complejas en PL/SQL, ¿cómo ofrezco esos datos al programa que llama?

Ahora, podemos combinar el poder y la flexibilidad de las funciones de tabla con el amplio soporte para variables de cursor en entornos que no son PL/SQL (explicado en detalle enCapítulo 15) para resolver este problema.

Supongamos, por ejemplo, que necesito generar una familia de mascotas (creada a través de una llamada a la función pet_family, como se muestra en la sección anterior) y pasar esas filas de datos a una aplicación frontend escrita en Java. Puedo hacer esto muy fácilmente de la siguiente manera:

```
/* Archivo en web: pet_family.sql */
FUNCION pet_family_cv
    RETORNO SYS_REFCURSOR
ES
    retval SYS_REFCURSOR;
COMENZAR
    OPEN retval FOR
        SELECCIONAR *
            DESDE LA TABLA (mascota_familia (mascota_t ('Hoppy', 'RABBIT', SYSDATE)
                                            , mascota_t ('Hippy', 'CONEJO', SYSDATE)
                                            )
                           ) ;

    RETORNO recuperación;
FIN mascota_familia_cv;
```

En este programa, aprovecho el tipo de REF CURSOR débil predefinido, SYS_REFCURSOR (introducido en Oracle9*i*Database), para declarar una variable de cursor. I

OPEN FOR esta variable de cursor, asociándola a la consulta que se construye alrededor de la función de tabla pet_family.

Luego puedo pasar esta variable de cursor de vuelta a la interfaz de Java. Debido a que JDBC reconoce las variables del cursor, el código Java puede obtener fácilmente las filas de datos e integrarlas en la aplicación.

Creación de una función de transmisión

A*función de transmisión*acepta como parámetro un conjunto de resultados (a través de una expresión CURSOR) y devuelve un conjunto de resultados en forma de colección. Debido a que puede aplicar el operador TABLE a esta colección y luego consultarla en una declaración SELECT, estas funciones pueden realizar una o más transformaciones de datos dentro de una sola declaración SQL.

Funciones de transmisión, cuyo soporte se agregó en Oracle9*i*,La base de datos se puede utilizar para ocultar la complejidad algorítmica detrás de una interfaz de función y, por lo tanto, simplificar el SQL en su aplicación. Voy a mostrar un ejemplo para explicar los tipos de pasos que deberá realizar usted mismo para aprovechar las funciones de la tabla de esta manera.

Considere el siguiente escenario. Tengo una tabla de información bursátil que contiene una sola fila para los precios de apertura y cierre de una acción:

```
/* Archivo en la web: tabfunc_streaming.sql */
TABLE stocktable (
    marcador VARCHAR2(10),
    trade_date FECHA,
    precio abierto NÚMERO,
    close_price NÚMERO)
```

Necesito transformar (o*pivote*) esa información en otra tabla:

```
tabla de cotizaciones TABLE (
    marcador VARCHAR2(10),
    preciofecha FECHA,
    tipoprecio VARCHAR2(1),
    precio NÚMERO)
```

En otras palabras, una sola fila en la tabla de existencias se convierte en dos filas en la tabla de cotizaciones. Hay muchas maneras de lograr este objetivo. Un enfoque muy tradicional y sencillo en PL/ SQL podría verse así:

```
PARA REC IN (SELECCIONE * DESDE la tabla de existencias)
BUCLE
    INSERTAR EN la tabla de cotizaciones
                (ticker, tipoprecio, precio) VALORES
        (rec.ticker, 'O', rec.open_price);

    INSERTAR EN la tabla de cotizaciones
                (ticker, tipoprecio, precio) VALORES
        (rec.ticker, 'C', rec.close_price); FIN DEL BUCLE;
```

También existen soluciones 100% SQL, como:

```
INSERTAR TODO
        en la tabla de cotizaciones
                (ticker, fecha de precio, tipo de precio, precio)

    VALORES (ticker, trade_date, 'O', open_price
                )
        en la tabla de cotizaciones
                (ticker, fecha de precio, tipo de precio, precio)

    VALORES (ticker, trade_date, 'C', close_price
                )
    SELECCIONE ticker, trade_date, open_price, close_price
        DESDE la mesa de almacenamiento;
```

Supongamos, sin embargo, que la transformación que debo realizar para mover datos de stocktable a tickertable es muy compleja y requiere el uso de PL/SQL. En esta situación, una función de tabla utilizada para transmitir los datos a medida que se transforman ofrecería una solución mucho más eficiente.

En primer lugar, si voy a utilizar una función de tabla, tendré que devolver una tabla anidada o VARRAY de datos. Usaré una tabla anidada porque los VARRAY requieren la especificación de un tamaño máximo y no quiero tener esa restricción en mi implementación. Este tipo de tabla anidada debe definirse como un tipo de nivel de esquema o dentro de la especificación de un paquete, para que el motor SQL pueda resolver una referencia a una colección de este tipo.

Me gustaría devolver una tabla anidada basada en la propia definición de la tabla, es decir, me gustaría que se defina de la siguiente manera:

```
TIPO tickertype_nt ES TABLA de tickertable%ROWTYPE;
```

Desafortunadamente, esta instrucción fallará porque %ROWTYPE no es un tipo reconocido por SQL. Ese atributo está disponible solo dentro de una sección de declaración PL/SQL. Entonces, en su lugar, debo crear un tipo de objeto que imite la estructura de mi tabla relacional y luego definir un TIPO de tabla anidada contra ese tipo de objeto:

```
TIPO TickerType COMO OBJETO (
   marcador VARCHAR2(10),
   preciofecha FECHA
   tipoprecio VARCHAR2(1),
   NÚMERO de precio);

TIPO TickerTypeSet COMO TABLA DE TickerType;
```

Para que mi función de tabla transmita datos de una etapa de transformación a la siguiente, tendrá que aceptar como argumento un conjunto de datos, en esencia, una consulta. La única forma de hacerlo es pasar una variable de cursor, por lo que necesitaré un tipo REF CURSOR para usar en la lista de parámetros de mi función.

Creo un paquete para contener el tipo REF CURSOR basado en mi nuevo tipo de tabla anidada:

```
PAQUETE refcur_pkg
ES
    TIPO refcur_t ES REF CURSOR RETURN StockTable%ROWTYPE; FIN
    refcur_pkg;
```

Finalmente, puedo escribir mi función de pivote de acciones:

```
      /* Archivo en la web: tabfunc_streaming.sql */ FUNCIÓN
1     stockpivot (conjunto de datos refcur_pkg.refcur_t)
2         RETURN tickertypeset
3     ES
4         l_row_as_object tipo de teletipo := tipo de teletipo (NULL, NULL, NULL, NULL);
5         l_row_from_query conjunto de datos%ROWTYPE;
6         retval tickertypeset := tickertypeset ();
7         COMENZAR
8         BUCLE
9             FETCH conjunto de datos
10              EN l_row_from_query;
11
12            SALIR CUANDO el conjunto de datos% NO SE ENCUENTRA;
13            - - Crear la instancia de tipo de objeto OPEN
14            l_row_as_object.ticker := l_row_from_query.ticker;
15            l_row_as_object.pricetype := 'O';
dieciséis     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            - - Crear la instancia de tipo de objeto CERRADO
21            l_row_as_object.pricetype := 'C'; l_row_as_object.price :=
22            l_row_from_query.close_price; retval.EXTEND;
23
24            retval (retval.LAST) := l_row_as_object; FIN
25        DEL BUCLE;
26
27        CERRAR conjunto de datos;
28
29        RETORNO recuperación;
30    FIN pivote de stock;
```

Al igual que con la función pet_family, los detalles de este programa no son importantes y su propia lógica de transformación será cualitativamente más compleja. Los pasos básicos realizados aquí, sin embargo, probablemente se repetirán en su propio código, por lo que los revisaré en la siguiente tabla.

Líneas)	Descripción
1–2	El encabezado de la función: pasa un conjunto de resultados como una variable de cursor y devuelve una tabla anidada según el tipo
4	de objeto. Declare un objeto local, que se usará para completar la tabla anidada.
5	Declarar un registro local basado en el conjunto de resultados. Esto se completará con FETCH de la variable del
6	cursor. La tabla anidada local que devolverá la función.

Líneas)	Descripción
8–12	Inicie un ciclo simple para obtener cada fila por separado de la variable del cursor, finalizando el ciclo cuando no haya más datos en el cursor.
14–19	Use los datos "abiertos" en el registro para completar el objeto local y luego colóquelo en la tabla anidada, después de EXTENDIR para definir la nueva fila.
21–25	Use los datos de "cierre" en el registro para completar el objeto local y luego colóquelo en la tabla anidada, después de EXTENDIR para definir la nueva fila.
27–30	Cierra el cursor y devuelve la tabla anidada. Misión completada. En realidad.

Y ahora que tengo esta función para hacer todo el trabajo de base elegante pero necesario, puedo usarla dentro de mi consulta para transmitir datos de una tabla a otra:

```
COMENZAR
    INSERTAR EN la tabla de cotizaciones
        SELECCIONAR *
          DESDE LA TABLA (stockpivot (CURSOR (SELECCIONAR *
                                              DESDE la tabla de existencias)));
FIN;
```

Mi SELECCIÓN interna recupera todas las filas en la tabla de existencias. La expresión CURSOR alrededor de esa consulta transforma el conjunto de resultados en una variable de cursor, que se pasa a stockpivot. Esa función devuelve una tabla anidada y el operador TABLE luego la traduce a un formato de tabla relacional que se puede consultar.

Puede que no sea magia, pero*es*un poco mágico, ¿no crees? Bueno, si cree que una función de transmisión es especial, ¡consulte las funciones canalizadas!

Creación de una función segmentada

A*función segmentada*es una función de tabla que devuelve un conjunto de resultados como una colección, pero lo hace de forma asíncrona a la terminación de la función. En otras palabras, la base de datos ya no espera a que la función se ejecute hasta el final, almacenando todas las filas que calcula en la colección PL/SQL, antes de entregar las primeras filas. En cambio, como cada fila está lista para ser asignada a la colección, se canaliza fuera de la función. Esta sección describe los conceptos básicos de las funciones de tabla canalizadas. Las implicaciones de rendimiento de estas funciones se exploran en detalle encapitulo 21.

Echemos un vistazo a una reescritura de la función stockpivot y veamos más claramente lo que se necesita para construir funciones segmentadas:

```
    /* Archivo en la web: tabfunc_pipelined.sql */ FUNCIÓN
1   stockpivot (conjunto de datos refcur_pkg.refcur_t) RETURN
2   tickertypeset PIPELINED
3   ES
4       l_row_as_object tipo de teletipo := tipo de teletipo (NULL, NULL, NULL, NULL);
5       l_row_from_query conjunto de datos%ROWTYPE;
6   COMENZAR
7       BUCLE
```

```
 8              FETCH conjunto de datos EN l_row_from_query; SALIR CUANDO el
 9              conjunto de datos% NO SE ENCUENTRA;
10
11              - - primera fila
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; FILA
dieciséis       DE TUBO (l_row_as_object);
17
18              - - segunda fila
19              l_row_as_object.pricetype := 'C'; l_row_as_object.price :=
20              l_row_from_query.close_price; FILA DE TUBO
21              (l_row_as_object);
22          FIN DEL BUCLE;
23
24          CERRAR conjunto de datos;
25          DEVOLVER;
26      FIN;
```

La siguiente tabla muestra varios cambios en nuestra funcionalidad original.

Líneas)	Descripción
2	El único cambio con respecto a la función stockpivot original es la adición de la palabra clave PIPELINED.
4–5	Declare un objeto local y un registro local, como con el primer stockpivot. Lo llamativo de estas líneas es lo que nodeclare, es decir, la tabla anidada que devolverá la función. Una pista de lo que está por venir...
7–9	Inicie un ciclo simple para obtener cada fila por separado de la variable del cursor, finalizando el ciclo cuando no haya más datos en el cursor.
12–15 y 19–21	Rellene el objeto local para que las filas abiertas y cerradas del cuadro de cotizaciones se coloquen en la tabla anidada.
16 y 21	Utilice la declaración PIPE ROW (válida solo en funciones canalizadas) para "canalizar" los objetos inmediatamente fuera de la función.
25	En la parte inferior de la sección ejecutable, ¡la función no devuelve nada! En su lugar, llama a RETURN no calificado (anteriormente permitido solo en procedimientos) para devolver el control al bloque de llamada. La función ya devolvió todos sus datos con las declaraciones PIPE ROW.

Puede llamar a la función canalizada como lo haría con la versión no canalizada. No verá ninguna diferencia en el comportamiento, a menos que configure la función canalizada para que se ejecute en paralelo como parte de una consulta en paralelo (que se trata en la siguiente sección) o incluya una lógica que aproveche la devolución asincrónica de datos.

Considere, por ejemplo, una consulta que usa la pseudocolumna ROWNUM para restringir las filas de interés:

```
COMENZAR
    INSERTAR EN la tabla de cotizaciones
        SELECCIONAR *
            DESDE LA TABLA (stockpivot (CURSOR (SELECCIONAR *
                                                DESDE la tabla de existencias)))
```

```
            DONDE ROWNUM < 10;
    FIN;
```

Mis pruebas muestran que en Oracle Database 10*gramo*y base de datos Oracle 11*gramo*, si giro 100 000 filas en 200 000 y luego devuelvo solo las primeras 9 filas, la versión canalizada completa su trabajo en 0,2 segundos, mientras que la versión no canalizada tarda 4,6 segundos.

¡Claramente, la tubería de las filas hacia atrás funciona y hace una diferencia notable!

Habilitación de una función para la ejecución en paralelo

Un enorme paso adelante para PL/SQL, introducido por primera vez en Oracle9*i* Base de datos, es la capacidad de ejecutar funciones dentro de un contexto de consulta paralela. Antes de Oracle9*i* Database, una llamada a una función PL/SQL dentro de SQL provocó la serialización de esa consulta, un problema importante para las aplicaciones de almacenamiento de datos. Ahora puede agregar información al encabezado de una función segmentada para indicarle al motor de tiempo de ejecución cómo debe particionarse el conjunto de datos que se pasa a la función para una ejecución paralela.

En general, si desea que su función se ejecute en paralelo, debe tener un solo parámetro de entrada REF CURSOR fuertemente tipado.[1]

Aquí hay unos ejemplos:

- Especifique que la función se puede ejecutar en paralelo y que los datos pasados a esa función se pueden particionar arbitrariamente:

```
FUNCIÓN mi_transform_fn (
      p_input_rows en employee_info.recur_t )
  RETURN employee_info.transformed_t
  PIPELINED
  PARALLEL_ENABLE (PARTICIÓN p_input_rows POR CUALQUIERA)
```

  En este ejemplo, la palabra clave ANY expresa la afirmación del programador de que los resultados son independientes del orden en que la función obtiene las filas de entrada. Cuando se utiliza esta palabra clave, el sistema de tiempo de ejecución divide aleatoriamente los datos entre los distintos procesos de consulta. Esta palabra clave es adecuada para usar con funciones que toman una fila, manipulan sus columnas y generan filas de salida basadas únicamente en las columnas de esta fila. Si su programa tiene otras dependencias, el resultado será impredecible.

- Especifique que la función se puede ejecutar en paralelo, que todas las filas de un departamento determinado vayan al mismo proceso y que todas estas filas se entreguen consecutivamente:

```
FUNCIÓN mi_transform_fn (
    p_input_rows en employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
```

1. La entrada REF CURSOR necesita*no*estar fuertemente tipado para ser particionado por ANY.

```
CLUSTER P_INPUT_ROWS POR (departamento)
PARALLEL_ENABLE
   ( PARTICIÓN P_INPUT_ROWS POR HASH (departamento) )
```

Oracle usa el término *agrupado* para significar este tipo de entrega, y *clave de clúster* para la columna (en este caso, departamento) en la que se realiza la agregación. Pero significativamente, el algoritmo no *no* importa en qué orden de clave de clúster recibe cada clúster sucesivo, y Oracle no garantiza ningún orden en particular aquí. Esto permite un algoritmo más rápido que si las filas fueran requeridas para ser agrupadas y entregadas en el orden de la clave de agrupación. Se escala como *pedido norte* en vez de *orden N.log(N)*, dónde *norte* es el número de filas.

En este ejemplo, puedo elegir entre HASH (departamento) y RANGO (departamento), dependiendo de lo que sepa sobre la distribución de los valores. HASH es más rápido que RANGE y es la opción natural para usar con CLUSTER...BY.

- Especificar que la función puede ejecutarse en paralelo y que las filas que se entregan a un proceso en particular, según lo indique PARTICIÓN... POR (para esa partición especificada), serán ordenadas localmente por ese proceso. El efecto será paralelizar el género:

```
FUNCIÓN mi_transform_fn (
    p_input_rows en employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
ORDENAR P_INPUT_ROWS POR (C1)
PARALLEL_ENABLE
   ( PARTICIÓN P_INPUT_ROWS POR RANGO (C1) )
```

Debido a que la ordenación está paralelizada, no debería haber ORDER...BY en el SELECT utilizado para invocar la función de tabla. (De hecho, una cláusula ORDER...BY en la sentencia SELECT subvertiría el intento de paralelizar la ordenación). Por lo tanto, es natural usar la opción RANGE junto con la opción ORDER...BY. Esto será más lento que CLUSTER...BY, por lo que debe usarse solo cuando el algoritmo dependa de él.

La construcción CLUSTER...BY no se puede usar junto con ORDER...BY en la declaración de una función de tabla. Esto significa que un algoritmo que depende de la agrupación en una clave, c1, y luego de ordenar dentro de la fila establecida para un valor dado de c1 por, digamos, c2, tendría que ser paralelizado usando ORDER...BY en el declaración en la función de tabla.

Funciones deterministas

Se considera que una función es *determinista* si devuelve el mismo valor de resultado cada vez que se llama con los mismos valores para sus argumentos IN e IN OUT. Otra forma de pensar en los programas deterministas es que no tienen efectos secundarios. Todo lo que cambia el programa se refleja en la lista de parámetros.