
CAPÍTULO 21

Optimización del rendimiento de PL/SQL

Optimizar el rendimiento de una aplicación de Oracle es un proceso complejo: debe ajustar el SQL en su base de código, asegurarse de que el área global del sistema (SGA) esté configurada correctamente, optimizar la lógica algorítmica, etc. Ajustar programas PL/SQL individuales es un poco menos desalentador, pero sigue siendo un desafío más que suficiente. Antes de dedicar mucho tiempo a cambiar su código PL/SQL con la esperanza de mejorar el rendimiento de ese código, debe:

Sintonice el acceso a código y datos en el SGA

Antes de que su código pueda ejecutarse (y quizás demasiado lento), debe cargarse en el SGA de la instancia de Oracle. Este proceso puede beneficiarse de un esfuerzo de ajuste enfocado, generalmente realizado por un DBA. Encontrará más información sobre SGA y otros aspectos de la arquitectura PL/SQL en [capítulo 24](#).

Optimiza tu SQL

En prácticamente cualquier aplicación que escriba en la base de datos de Oracle, realizará la gran mayoría de los ajustes optimizando las instrucciones SQL ejecutadas en sus datos. Las ineficiencias potenciales de una combinación de 16 vías eclipsan los problemas habituales que se encuentran en un bloque de código de procedimiento. Para decirlo de otra manera, si tiene un programa que se ejecuta en 20 horas y necesita reducir su tiempo transcurrido a 30 minutos, prácticamente su única esperanza será concentrarse en el SQL dentro de su código. Hay muchas herramientas de terceros disponibles tanto para DBA como para desarrolladores que realizan análisis muy sofisticados de SQL dentro de las aplicaciones y recomiendan alternativas más eficientes.

Use el nivel de optimización del compilador más agresivo posible

Base de datos Oracle 10gramo introdujo un compilador de optimización para programas PL/SQL. El nivel de optimización predeterminado de 2 en esa versión tomó el enfoque más agresivo posible en términos de transformar su código para que se ejecutara más rápido. (Base de datos Oracle 11gramoy posteriores admiten un nivel de optimización aún mayor de 3. Sin embargo, el nivel de optimización predeterminado sigue siendo 2, y eso será suficiente para la gran mayoría de

su código.) Debe usar este nivel predeterminado a menos que el tiempo de compilación sea inaceptablemente lento y no esté viendo los beneficios de la optimización.

Una vez que esté seguro de que el *contexto* en el que se ejecuta su código PL/SQL no es obviamente ineficiente, debe centrar su atención en sus paquetes y otro código. Sugiero los siguientes pasos:

1. Escriba su aplicación teniendo en cuenta las mejores prácticas y estándares.

Si bien no debe adoptar enfoques claramente ineficientes para cumplir con los requisitos, tampoco debe obsesionarse con las implicaciones de rendimiento de cada línea de su código. Recuerde que la mayor parte del código que escriba nunca será un cuello de botella en el rendimiento de su aplicación, por lo que optimizarlo no resultará en ningún beneficio para el usuario. En su lugar, escriba la aplicación teniendo en mente la corrección y la mantenibilidad y luego...

2. Analiza el perfil de ejecución de tu aplicación.

¿Corre lo suficientemente rápido? Si es así, genial: no necesitas hacer ningún ajuste (por el momento). Si es demasiado lento, identifique qué elementos específicos de la aplicación están causando el problema y luego concéntrate directamente en esos programas (o partes de programas). Una vez identificado, puede...

3. Ajuste sus algoritmos.

Como lenguaje de procedimientos, PL/SQL se usa a menudo para implementar fórmulas y algoritmos complejos. Puede usar declaraciones condicionales, bucles, tal vez incluso GOTO y (espero) módulos reutilizables para hacer el trabajo. Estos algoritmos se pueden escribir de muchas maneras diferentes, algunas de las cuales funcionan muy mal. ¿Cómo sintonizas algoritmos mal escritos? Esta es una pregunta difícil sin respuestas fáciles. Ajustar algoritmos es mucho más complejo que ajustar SQL (que está "estructurado" y, por lo tanto, se presta más fácilmente al análisis automatizado).

4. Aproveche las funciones de rendimiento específicas de PL/SQL.

A lo largo de los años, Oracle ha agregado declaraciones y optimizaciones que pueden marcar una diferencia sustancial en la ejecución de su código. Considere el uso de construcciones que van desde la cláusula RETURNING hasta FORALL. Asegúrese de no vivir en el pasado y pagar el precio de las ineficiencias de las aplicaciones.

5. Equilibrar las mejoras de rendimiento con el consumo de memoria.

Varias de las técnicas que mejoran el rendimiento de su código también consumen más memoria, generalmente en el área global del programa (PGA), pero también a veces en el SGA. No le servirá de mucho hacer que su programa sea increíblemente rápido si el consumo de memoria resultante es inaceptable en el entorno de su aplicación.

Está fuera del alcance de este libro ofrecer consejos sustanciales sobre el ajuste de SQL y la configuración de la base de datos/SGA. ciertamente *poder*, por otro lado, contarte todo lo más im-

importantes funciones de optimización del rendimiento de PL/SQL y ofrece consejos sobre cómo aplicar esas funciones para lograr el código PL/SQL más rápido posible.

Finalmente, recuerde que la optimización general del rendimiento es un esfuerzo de equipo. Trabaje en estrecha colaboración con su DBA, especialmente cuando comience a aprovechar características clave como colecciones, funciones de tabla y la memoria caché de resultados de funciones.

Herramientas para ayudar en la optimización

En esta sección, presento las herramientas y técnicas que pueden ayudar a optimizar el rendimiento de su código. Estos se dividen en varias categorías: analizar el uso de la memoria, identificar cuellos de botella en el código PL/SQL, calcular el tiempo transcurrido, elegir el programa más rápido, evitar bucles infinitos y usar advertencias relacionadas con el rendimiento.

Análisis del uso de la memoria

Como mencioné, a medida que optimiza el rendimiento del código, también deberá tener en cuenta la cantidad de memoria que consume su programa. Los datos del programa consumen PGA, y cada sesión conectada a la base de datos Oracle tiene su propio PGA. Por lo tanto, la memoria total requerida para su aplicación suele ser mucho mayor que la memoria necesaria para una sola instancia del programa. El consumo de memoria es un factor especialmente crítico cuando se trabaja con colecciones (estructuras similares a matrices), así como tipos de objetos con una gran cantidad de atributos y registros que tienen una gran cantidad de campos.

Para una discusión en profundidad de este tema, consulte la sección "["PL/SQL y memoria de instancia de base de datos"](#) en la página 1076.

Identificación de cuellos de botella en código PL/SQL

Antes de que pueda ajustar su aplicación, debe averiguar qué se está ejecutando lentamente y dónde debe concentrar sus esfuerzos. Oracle y los proveedores externos ofrecen una variedad de productos para ayudarlo a hacer esto; generalmente se enfocan en analizar las declaraciones SQL en su código, ofreciendo implementaciones alternativas, etc. Estas herramientas son muy poderosas, pero también pueden ser muy frustrantes para los desarrolladores de PL/SQL. Tienden a ofrecer una cantidad abrumadora de datos de rendimiento sin decirle lo que realmente quiere saber: ¿dónde están los cuellos de botella en su código?

Para responder a estas preguntas, Oracle ofrece una serie de utilidades integradas. Aquí están los más útiles:

PERFIL_DBMS

Este paquete integrado le permite activar el perfilado de ejecución en una sesión. Luego, cuando ejecuta su código, la base de datos de Oracle usa tablas para realizar un seguimiento de la información detallada sobre cuánto tiempo tardó en ejecutarse cada línea de su código. entonces puedes

ejecute consultas en estas tablas o, preferiblemente, use pantallas en productos como Toad o SQL Navigator para presentar los datos de forma clara y gráfica.

DBMS_HPROF (perfilador jerárquico)

Base de datos Oracle 11g *gramo* introdujo un *perfilador jerárquico* que hace que sea más fácil subir los resultados de rendimiento a través de la pila de llamadas de ejecución.

DBMS_PROFILER proporciona datos "planos" sobre el rendimiento, lo que dificulta responder preguntas como "¿Cuánto tiempo en total se gasta en el procedimiento ADD_ITEM?" El perfilador jerárquico facilita la respuesta a tales preguntas.

PERFIL_DBMS

En caso de que no tenga acceso a una herramienta que ofrezca una interfaz para DBMS_PROFILER, aquí hay algunas instrucciones y ejemplos.

En primer lugar, es posible que Oracle no haya instalado DBMS_PROFILER automáticamente. Para ver si DBMS_PROFILER está instalado y disponible, conéctese a su esquema en SQL*Plus y emita este comando:

```
DESC DBMS_PROFILER
```

Si luego ve el mensaje:

```
ERROR:  
ORA-04043: el objeto dbms_profiler no existe
```

entonces usted (o su DBA) tendrá que instalar el programa. Para hacer esto, ejecute el *psORA_CLE_INICIO/rdbms/admin/profload.sql* archivo bajo una cuenta SYSDBA.

A continuación, debe ejecutar el *psORACLE_HOME/rdbms/admin/proftab.sql* archivo en su propio esquema para crear tres tablas pobladas por DBMS_PROFILER:

PLSQL_PROFILER_RUNS

Tabla principal de ejecuciones

PLSQL_PROFILER_UNITS

Unidades de programa ejecutadas en ejecución

PLSQL_PROFILER_DATA

Perfilar datos para cada línea en una unidad de programa

Una vez que se definen todos estos objetos, recopila información de perfiles para su aplicación escribiendo un código como este:

```
COMENZAR  
  DBMS_PROFILER.start_profiler (  
    'mi aplicación' || TO_CHAR (SYSDATE, 'AAAAMMDD HH24:MI:SS')  
  );  
  
  my_application_code;
```

```
DBMS_PROFILER.stop_profiler;
FIN;
```

Una vez que haya terminado de ejecutar el código de su aplicación, puede ejecutar consultas contra los datos en las tablas PLSQL_PROFILER_. Aquí hay un ejemplo de una consulta de este tipo que muestra las líneas de código que consumieron al menos el 1% del tiempo total de la ejecución:

```
/* Archivo en la web: slowest.sql */
SELECCIONE TO_CHAR (p1.total_time / 10000000, '99999999')
    || '-'
    || TO_CHAR (p1.total_ocurrencia
        como time_count,
        SUBSTR (p2.unidad_propietario, 1, 20)
    || ':'
    || DECODE (p2.unit_name,
        '<anónimo>',
        SUBSTR (p2.unit_name, 1, 20))
        como unidad,
        TO_CHAR (p1.line#) || '-' || p3.texto texto
DESDE plsql_profiler_data p1,
plsql_profiler_units p2,
all_source p3,
(SELECCIONE SUMA (total_time) COMO grand_total
    DESDE plsql_profiler_units) p4
DÓNDE      p2.unit_owner NO EN ('SYS', 'SYSTEM') Y
p1.runid = &&firstparm
Y (p1.total_time >= p4.grand_total / 100) Y
p1.runid = p2.runid
Y p2.número_unidad = p1.número_unidad Y
p3.TIPO = 'CUERPO DEL PAQUETE'
Y p3.propietario = p2.unidad_propietario Y
p3.línea = p1.línea#
AND p3.name = p2.unit_name
ORDENAR POR p1.total_time DESC
```

Como puede ver, estas consultas son bastante complejas (modifiqué una de las consultas enlazadas de Oracle para producir la combinación de cuatro vías anterior). Por eso es mucho mejor confiar en una interfaz gráfica en una herramienta de desarrollo PL/SQL.

El perfilador jerárquico

Base de datos Oracle 11g *gramo* introdujo un segundo mecanismo de creación de perfiles: DBMS_HPROF, conocido como perfilador jerárquico. Utilice este generador de perfiles para obtener el perfil de ejecución del código PL/SQL, organizado por las distintas llamadas a subprogramas en su aplicación. "Está bien", puedo oírte pensar, "pero ¿DBMS_PROFILER ya no hace eso por mí?" No precisamente. Los perfiladores no jerárquicos (planos) como DBMS_PROFILER registran el tiempo que su aplicación pasa dentro de cada subprograma, hasta el tiempo de ejecución de cada línea de código individual. Eso es útil, pero de una manera limitada. A menudo, también desea saber cuánto tiempo pasa la aplicación dentro de un subprograma en particular, es decir, necesita "resumir"

información del perfil a nivel de subprograma. Eso es lo que el nuevo perfilador jerárquico hace por usted.

El perfilador jerárquico de PL/SQL informa sobre el rendimiento de cada subprograma de su aplicación que se perfila, manteniendo distintos los tiempos de ejecución de SQL y PL/SQL. El generador de perfiles rastrea una amplia variedad de información, incluida la cantidad de llamadas al subprograma, la cantidad de tiempo pasado en ese subprograma, el tiempo pasado en el subárbol del subprograma (es decir, en sus subprogramas descendientes) e información detallada de padres e hijos .

El perfilador jerárquico tiene dos componentes:

Recolector de datos

Proporciona API que activan y desactivan la creación de perfiles jerárquicos. El motor de tiempo de ejecución PL/SQL escribe la salida del generador de perfiles "sin procesar" en el archivo especificado.

Analizador

Procesa la salida del generador de perfiles sin procesar y almacena los resultados en tablas jerárquicas del generador de perfiles, que luego se pueden consultar para mostrar la información del generador de perfiles.

Para utilizar el perfilador jerárquico, haga lo siguiente:

1. Asegúrese de que puede ejecutar el paquete DBMS_HPROF.
2. Asegúrese de tener privilegios de ESCRITURA en el directorio que especifique cuando llame a DBMS_HPROF.START_PROFILING.
3. Cree las tres tablas de perfiles (consulte los detalles de este paso a continuación).
4. Llame al procedimiento DBMS_HPROF.START_PROFILING para iniciar la recopilación de datos del perfilador jerárquico en su sesión.
5. Ejecute el código de su aplicación lo suficientemente largo y repetitivo para obtener suficiente cobertura de código para obtener resultados interesantes.
6. Llame al procedimiento DBMS_HPROF.STOP_PROFILING para finalizar la recopilación de datos de perfil.
7. Analice el contenido y luego ejecute consultas en las tablas del generador de perfiles para obtener resultados.



Para obtener las medidas más precisas del tiempo transcurrido para sus subprogramas, debe minimizar cualquier actividad no relacionada en el sistema en el que se ejecuta su aplicación.

Por supuesto, en un sistema de producción, otros procesos pueden ralentizar su programa. También puede realizar estas mediciones mientras utiliza *pruebas de aplicaciones reales*(RAT) en Oracle Database 11g *ramo* y posteriormente obtener tiempos de respuesta reales.

Para crear las tablas del generador de perfiles y otros objetos de base de datos necesarios, ejecute el *dbmshptab.sql* guión (ubicado en el *psORACLE_HOME/rdbms/admin* directorio). Este script creará estas tres tablas:

DBMSHP_RUNS

Información de nivel superior sobre cada ejecución de la utilidad ANALYZE de DBMS_HPROF.

DBMSHP_FUNCTION_INFO

Información detallada sobre la ejecución de cada subprograma perfilado en una ejecución particular de la utilidad ANALIZAR.

DBMSHP_PARENT_CHILD_INFO

Información padre-hijo para cada subprograma perfilado en DBMSHP_FUNCTION_INFO.

Aquí hay un ejemplo muy simple: quiero probar el rendimiento de mi procedimiento intab (que muestra el contenido de la tabla especificada usando DBMS_SQL). Entonces, primero empiezo a generar perfiles, especificando que quiero que los datos sin procesar del generador de perfiles se escriban en el archivo *tab_trace.txt* en el directorio TEMP_DIR. Este directorio debe haber sido definido previamente con la sentencia CREATE DIRECTORY:

```
EXEC DBMS_HPROF.start_profiling ('TEMP_DIR', 'intab_trace.txt')
```

Luego llamo a mi programa (ejecutar mi código de aplicación):

```
EXEC intab ('DEPARTAMENTOS')
```

Y luego termino mi sesión de creación de perfiles:

```
EXEC DBMS_HPROF.stop_perfilado;
```

Podría haber incluido las tres declaraciones en el mismo bloque de código; en cambio, los mantuve separados porque en la mayoría de las situaciones no incluirá comandos de creación de perfiles en o cerca del código de su aplicación.

Así que ahora ese archivo de rastreo está lleno de datos. *Ipodrá*ábralos y mire los datos, y tal vez tenga un poco de sentido de lo que encuentre allí. Sin embargo, un uso mucho mejor de mi tiempo y la tecnología de Oracle sería llamar a la utilidad ANALYZE de DBMS_HPROF. Esta función toma el contenido del archivo de seguimiento, transforma estos datos y los coloca en las tres tablas del generador de perfiles. Devuelve un número de ejecución, que luego debo usar al consultar el contenido de estas tablas. Llamo ANALIZAR de la siguiente manera:

```
COMENZAR
    DBMS_SALIDA.PUT_LINE (
        DBMS_HPROF.ANALYZE ('TEMP_DIR', 'intab_trace.txt'));
FIN;
/
```

¡Y eso es! Los datos se recopilaron y analizaron en las tablas, y ahora puedo elegir uno de los dos enfoques para obtener la información del perfil:

1. Ejecute el *plshprof* utilidad de línea de comandos (ubicada en el directorio *ORACLE_HOME/papelería*). Esta utilidad genera informes HTML simples a partir de uno o dos archivos de salida del perfilador sin procesar. Para ver un ejemplo de un archivo de salida de generador de perfiles sin procesar, consulte la sección titulada "Recopilación de datos de perfil" en el *Guía de desarrollo de la base de datos de Oracle*. Luego puedo examinar los informes HTML generados en el navegador de mi elección.
2. Ejecutar mis propias consultas "caseras". Supongamos, por ejemplo, que el bloque anterior devuelve 177 como número de ejecución. Primero, aquí hay una consulta que muestra todas las ejecuciones actuales:

```
SELECCIONE runid, run_timestamp, total_elapsed_time, run_comment
    DESDE dbmshp_runs
```

Aquí hay una consulta que me muestra todos los nombres de los subprogramas que se han perfilado, en todas las ejecuciones:

```
SELECCIONE símbolo, propietario, módulo, tipo, función, número de línea, espacio de nombres
    DESDE dbmshp_function_info
```

Aquí hay una consulta que me muestra información sobre la ejecución del subprograma para esta ejecución específica:

```
SELECCIONE FUNCIÓN, n.º de línea, espacio de nombres, subárbol_tiempo_transcurrido
    , function_elapsed_time, llamadas
    DESDE dbmshp_function_info
    DONDE runid = 177
```

Esta consulta recupera información padre-hijo para la ejecución actual, pero no de una manera muy interesante, ya que solo veo valores clave y no nombres de programas:

```
SELECT parentymid, childsymid, subtree_elapsed_time, function_elapsed_time
    , llamadas
    DESDE dbmshp_parent_child_info
    DONDE runid = 177
```

Aquí hay una consulta más útil, uniéndose a la tabla de información de funciones; ahora puedo ver los nombres de los programas principal y secundario, junto con el tiempo transcurrido y el número de llamadas.

```
SELECCIONAR      RPAD (' ', NIVEL * 2, '') || fi.propietario || '.' || fi.módulo COMO NOMBRE
    , fi.FUNCTION, pci.subtree_elapsed_time, pci.function_elapsed_time
    , pci.llamadas
    DESDE dbmshp_parent_child_info pci ÚNASE a dbmshp_function_info fi
        EN pci.runid = fi.runid Y pci.childsymid = fi.symbolid DONDE
        pci.runid = 177
    CONECTAR POR ANTERIOR childsymid = parentsymid
    COMENZAR CON pci.parentsymid = 1
```

El perfilador jerárquico es una utilidad muy poderosa y rica. Le sugiero que lea el Capítulo 13 del *Guía de desarrollo de la base de datos de Oracle* para una amplia cobertura de este generador de perfiles.

Cálculo del tiempo transcurrido

Así que ha encontrado el cuello de botella en su aplicación; es una función llamada CALC_TOTALS, y contiene un algoritmo complejo que claramente necesita algunos ajustes. Trabajas en la función durante un rato y ahora quieras saber si es más rápida. ciertamente podrías profile la ejecución de toda su aplicación de nuevo, pero sería mucho más fácil si simplemente pudiera ejecutar las versiones original y modificada "una al lado de la otra" y ver cuál es más rápida. Para hacer esto, necesita una utilidad que calcule el tiempo transcurrido de programas individuales, incluso líneas de código.*dentro* un programa.

El paquete DBMS_UTLILITY ofrece dos funciones para ayudarlo a obtener esta información: DBMS_UTLILITY.GET_TIME y DBMS_UTLILITY.GET_CPU_TIME. Ambos están disponibles para Oracle Database 10g *gramo* y después.

Puede usar fácilmente estas funciones para calcular el tiempo transcurrido (total y CPU, respectivamente) de su código hasta la centésima de segundo. Aquí está la idea básica:

1. Llame a DBMS_UTLILITY.GET_TIME (o GET_CPU_TIME) antes de ejecutar su código. Guarde esta "hora de inicio".
2. Ejecute el código cuyo rendimiento desea medir.
3. Llame a DBMS_UTLILITY.GET_TIME (o GET_CPU_TIME) para obtener la "hora de finalización". Reste el comienzo del final; esta diferencia es el número de centésimas de segundo que han transcurrido entre las horas de inicio y finalización.

Aquí hay un ejemplo de este flujo:

```
DECLARAR
    l_start_time PLS_INTEGER;
COMENZAR
    l_start_time := DBMS_UTLILITY.get_time;

    mi programa;

    DBMS_OUTPUT.put_line (
        'Transcurrido: ' || DBMS_UTLILITY.get_time - l_start_time);
FIN;
```

Ahora, aquí hay algo extraño: encuentro estas funciones extremadamente útiles, pero nunca (o rara vez) las llamo directamente en mis scripts de rendimiento. En su lugar, elijo *encapsular* ocultar el uso de estas funciones, y su fórmula relacionada "fin - inicio", dentro de un paquete o tipo de objeto. En otras palabras, cuando quiero probar my_program, escribiría lo siguiente:

```
COMENZAR
    sf_timer.start_timer ();

    mi programa;
```

```
sf_timer.show_elapsed_time ('Ejecutó mi_programa');
FIN;
```

Capturo la hora de inicio, ejecuto el código y muestro el tiempo transcurrido.

Evito las llamadas directas a DBMS_UTLILITY.GET_TIME y, en su lugar, uso el paquete de temporizador SFTK, sf_timer, por dos razones:

- Para mejorar la productividad. ¿Quién quiere declarar esas variables locales, escribir todo el código para llamar a esa función integrada y hacer los cálculos? Prefiero que mi utilidad lo haga por mí.
- Para obtener resultados consistentes. Si confía en la fórmula simple de "fin - comienzo", a veces puede terminar con un *negativo* tiempo transcurrido. Ahora, no me importa qué tan rápido sea tu código; ¡No es posible retroceder en el tiempo!

¿Cómo es posible obtener un tiempo transcurrido negativo? El número devuelto por DBMS_UTLILITY.GET_TIME representa el número total de segundos transcurridos desde un momento arbitrario. Cuando este número se vuelve muy grande (el límite depende de su sistema operativo), vuelve a 0 y comienza a contar nuevamente. Por lo tanto, si llama a GET_TIME justo antes del rollover, end – start será negativo.

Lo que realmente debe hacer para evitar el posible momento negativo es escribir un código como este:

```
DECLARAR
  c_big_number NÚMERO := POTENCIA (2, 32);
  l_start_time PLS_INTEGER;
COMENZAR
  l_start_time := DBMS_UTLILITY.get_time; mi
  programa;
  DBMS_OUTPUT.put_line (
    'Transcurrido: '
    || TO_CHAR (MOD (DBMS_UTLILITY.get_time - l_start_time + c_big_number
      , c_número_grande)));
FIN;
```

¿Quién en su sano juicio, y con los plazos a los que todos nos enfrentamos, querría escribir ese código cada vez que necesita calcular el tiempo transcurrido?

Entonces, en su lugar, creé el paquete sf_timer para ocultar estos detalles y facilitar el análisis y la comparación de los tiempos transcurridos.

Elegir el programa más rápido

Uno pensaría que elegir el programa más rápido sería claro e inequívoco. Ejecuta un script, ve cuál de sus diversas implementaciones se ejecuta más rápido y elige esa. Ah, pero ¿bajo qué escenario ejecutaste esas implementaciones? Solo porque verificó la velocidad máxima para la implementación C para un conjunto de circunstancias, eso

no significa que el programa siempre (o incluso en su mayoría) se ejecutará más rápido que las otras implementaciones.

Al probar el rendimiento, y especialmente cuando necesita elegir entre diferentes implementaciones de los mismos requisitos, debe considerar y probar todos los escenarios siguientes:

Resultados positivos

El programa recibió entradas válidas e hizo lo que se suponía que debía hacer.

resultados negativos

El programa recibió entradas no válidas (por ejemplo, una clave principal inexistente) y el programa no pudo realizar las tareas solicitadas.

La neutralidad de datos de sus algoritmos

Su programa funciona muy bien en una tabla de 10 filas, pero ¿qué pasa con 10 000 filas? Su programa escanea una colección en busca de datos coincidentes, pero ¿qué pasa si la fila coincidente está al principio, en el medio o al final de la colección?

Ejecución multiusuario del programa

El programa funciona bien para un solo usuario, pero debe probarlo para el acceso simultáneo de varios usuarios. No querrá enterarse de interbloqueos después de que el producto entre en producción, ¿verdad?

Prueba en todas las versiones compatibles de Oracle

Si su aplicación necesita funcionar bien en Oracle Database 10*gramo* y base de datos Oracle 11*gramo*, por ejemplo, debe ejecutar sus scripts de comparación en instancias de cada versión.

Los detalles de cada uno de sus escenarios dependen, por supuesto, del programa que esté probando. Sin embargo, sugiero que cree un procedimiento que ejecute cada una de sus implementaciones y calcule el tiempo transcurrido para cada una. La lista de parámetros de este procedimiento debe incluir el número de veces que desea ejecutar cada programa; muy rara vez podrá ejecutar cada programa una sola vez y obtener resultados útiles. Debe ejecutar su código suficientes veces para asegurarse de que la carga inicial de código y datos en la memoria no distorsione los resultados. Los otros parámetros del procedimiento están determinados por lo que necesita pasar a cada uno de sus programas para ejecutarlos.

Aquí hay una plantilla para dicho procedimiento, con llamadas a sf_timer en su lugar y listo para funcionar:

```
/* Archivo en la web: compare_performance_template.sql */
PROCEDIMIENTO compare_implementations (
    título_en      EN VARCHAR2
    , iteraciones_en   EN ENTERO
/*
Y ahora cualquier parámetro que necesite para pasar datos a los
programas que está comparando....
*/
)
```

ES

COMENZAR

```
DBMS_OUTPUT.put_line ('Comparar rendimiento de <CAMBIAR ESTO>:');
DBMS_OUTPUT.put_line (título_en);
DBMS_OUTPUT.put_line ('Cada programa ejecuta ' || iteraciones_en || ' veces.');//*
```

Para cada implementación, inicie el temporizador, ejecute el programa N veces y luego muestre el tiempo transcurrido.

```
/*
sf_timer.start_timer;
```

PARA indx EN 1 .. iteraciones_en

BUCLE

```
/* Llame a su programa aquí. */
NULO;
```

FIN DEL BUCLE;

```
sf_timer.show_elapsed_time ('<CAMBIAR ESTO>: Implementación 1');
```

--

```
sf_timer.start_timer;
```

PARA indx EN 1 .. iteraciones_en

BUCLE

```
/* Llame a su programa aquí. */
NULO;
```

FIN DEL BUCLE;

```
sf_timer.show_elapsed_time ('<CAMBIAR ESTO>: Implementación 2'); FIN
comparar_implementaciones;
```

Verá una serie de ejemplos del uso de sf_timer en este capítulo.

Evitar bucles infinitos

Si le preocupa el rendimiento, ¡sin duda querrá evitar los bucles infinitos! Los bucles infinitos son menos un problema para las aplicaciones de producción (¡suponiendo que su equipo haya hecho un trabajo de prueba decente!) y más un problema cuando está en el proceso de creación de sus programas. Es posible que deba escribir alguna lógica engañosa para terminar un bucle, y ciertamente no es productivo tener que cerrar y reiniciar su sesión mientras prueba su programa.

Me encontré con mi propia cantidad de bucles infinitos y finalmente decidí escribir una utilidad que me ayudara a evitar este molesto resultado: el paquete Loop Killer. La idea detrás de sf_loop_killer es que, si bien es posible que aún no esté seguro de cómo terminar el ciclo con éxito, sabe que si el cuerpo del ciclo ejecuta más de *n* veces (por ejemplo, 100 o 1000, dependiendo de su situación), tiene un problema.

Entonces, compila el paquete Loop Killer en su esquema de desarrollo y luego escribe una pequeña cantidad de código que conducirá a la terminación del ciclo cuando alcance una cantidad de iteraciones que considere un indicador inequívoco de un ciclo infinito.

Aquí están las especificaciones del paquete (el paquete completo está disponible en el sitio web del libro):

```
/* Archivo en la web: sf_loop_killer.pks/pkb */
PAQUETE sf_loop_killer
ES
    c_max_iteraciones      CONSTANTE PLS_INTEGER DEFAULT 1000;
    e_infinite_loop_detected      EXCEPCIÓN;
    c_infinite_loop_detected      PLS_INTEGER := -20999;
    PRAGMA EXCEPTION_INIT (e_infinite_loop_detected, -20999);

    PROCEDIMIENTO kill_after (max_iterations_in IN PLS_INTEGER);

    PROCEDIMIENTO increment_or_kill (by_in IN PLS_INTEGER DEFAULT 1);

    FUNCIÓN cuenta_actual RETURN PLS_INTEGER;
FIN sf_loop_killer;
```

Veamos un ejemplo del uso de esta utilidad: específico que quiero que el bucle se elimine después de 100 iteraciones. Luego llamo a increment_or_kill al final del cuerpo del ciclo. Cuando ejecuto este código (claramente un bucle infinito), veo la excepción no controlada que se muestra en [Figura 21-1](#).

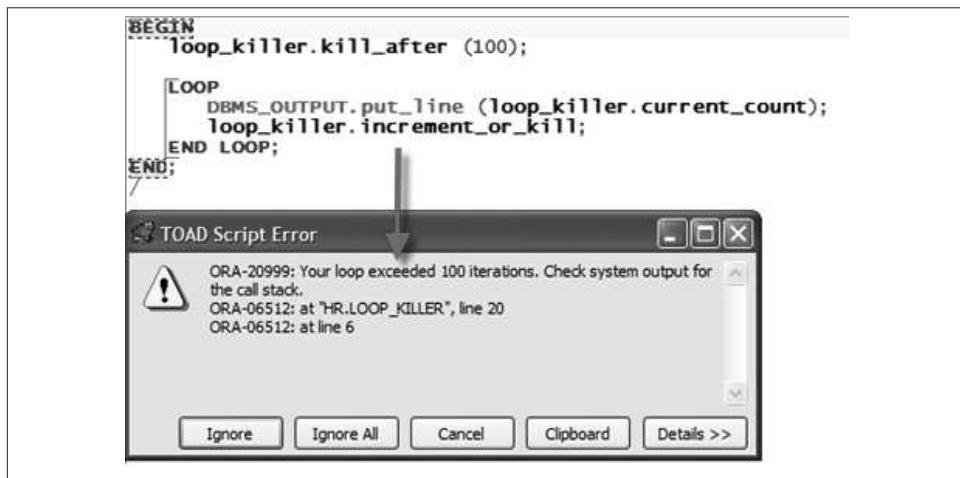


Figura 21-1. Usando el paquete Loop Killer

Advertencias relacionadas con el rendimiento

Oracle introdujo un marco de advertencias en tiempo de compilación en Oracle Database 10*gramoPL/SQL*. Cuando active las advertencias en su sesión, Oracle le dará retroalimentación sobre la calidad de su código y le ofrecerá consejos para mejorar la legibilidad y el rendimiento. Le recomiendo que use advertencias en tiempo de compilación para ayudar a identificar áreas de su código que podrían optimizarse.

Puede habilitar las advertencias para todo el conjunto de advertencias relacionadas con el rendimiento con la siguiente declaración:

```
ALTERAR SESIÓN ESTABLECER PLSQL_WARNINGS = 'HABILITAR: RENDIMIENTO'
```

Las advertencias de rendimiento incluyen lo siguiente:

- *PLW-06014: PLSQL_OPTIMIZE_LEVEL <= 1 desactiva la generación de código nativo*
- *PLW-07203: el parámetro "cadena" puede beneficiarse del uso de la sugerencia del compilador NOCOPY*
- *PLW-07204: la conversión fuera del tipo de columna puede resultar en un plan de consulta subóptimo*

Ver "[Advertencias de tiempo de compilación](#)" en la página 777 para obtener advertencias adicionales y más detalles sobre cómo trabajar con estas advertencias. Todas las advertencias están documentadas en el *Error de mensajes* libro del conjunto de documentación de Oracle.

El compilador optimizador

El compilador de optimización de PL/SQL puede mejorar drásticamente el rendimiento del tiempo de ejecución, con un costo relativamente pequeño en el momento de la compilación. Los beneficios de la optimización se aplican tanto a PL/SQL interpretado como compilado de forma nativa porque el compilador de optimización aplica optimizaciones mediante el análisis de patrones en el código fuente.

El compilador de optimización está habilitado de forma predeterminada. Sin embargo, es posible que desee modificar su comportamiento, ya sea reduciendo su agresividad o deshabilitándolo por completo. Por ejemplo, si, en el curso de las operaciones normales, su sistema debe realizar la recompilación de muchas líneas de código, o si una aplicación genera muchas líneas de PL/SQL ejecutado dinámicamente, la sobrecarga de optimización puede ser inaceptable. Sin embargo, tenga en cuenta que las pruebas de Oracle muestran que el optimizador duplica el rendimiento del tiempo de ejecución de PL/SQL computacionalmente intensivo.

En algunos casos, el optimizador puede incluso alterar el comportamiento del programa. Uno de esos casos podría ocurrir en el código escrito para Oracle9i/Base de datos que depende del tiempo relativo de las secciones de inicialización en varios paquetes. Si su prueba demuestra tal problema, pero desea disfrutar de los beneficios de rendimiento del optimizador, es posible que desee volver a escribir el código infractor o introducir una rutina de inicialización que asegure el orden de ejecución deseado.

La configuración del optimizador se define a través del parámetro de inicialización PLSQL_OPTIMIZE_LEVEL (y las sentencias ALTER DDL relacionadas), que se pueden establecer en 0, 1, 2 o 3 (3 está disponible solo en Oracle Database 11).*gramo*y después). Cuanto mayor sea el número, más agresiva será la optimización, lo que significa que el compilador hará un mayor esfuerzo y posiblemente reestructurará más código para optimizar el rendimiento.

Establezca su nivel de optimización de acuerdo con la mejor opción para su aplicación o programa, de la siguiente manera:

PLSQL_OPTIMIZE_LEVEL = 0

Zero esencialmente apaga la optimización. El compilador PL/SQL mantiene el orden de evaluación original del procesamiento de sentencias de Oracle9iBase de datos y versiones anteriores. Su código aún se ejecutará más rápido que en versiones anteriores, pero la diferencia no será tan dramática.

PLSQL_OPTIMIZE_LEVEL = 1

El compilador aplicará muchas optimizaciones a su código, como la eliminación de cálculos y excepciones innecesarios. En general, no cambiará el orden de su código fuente original.

PLSQL_OPTIMIZE_LEVEL = 2

Este es el valor predeterminado. También es la configuración más agresiva disponible antes de Oracle Database 11*gramo*. Aplicará muchas técnicas de optimización modernas más allá de las aplicadas en el nivel 1, y algunos de esos cambios pueden resultar en mover el código fuente relativamente lejos de su ubicación original. La optimización de nivel 2 ofrece el mayor impulso en el rendimiento. Sin embargo, puede causar que el tiempo de compilación en algunos de sus programas aumente sustancialmente. Si se encuentra con esta situación (o, alternativamente, si está desarrollando su código y desea minimizar el tiempo de compilación, sabiendo que cuando pase a producción aplicará un nivel de optimización más alto), intente reducir el nivel de optimización a 1.

PLSQL_OPTIMIZE_LEVEL = 3

Introducido en Oracle Database 11*gramo*, este nivel de optimización agrega la inserción de subprogramas anidados o locales. Puede ser beneficioso en casos extremos (gran cantidad de subprogramas locales o ejecución recursiva), pero para la mayoría de las aplicaciones PL/SQL, el nivel predeterminado de 2 debería ser suficiente.

Puede establecer el nivel de optimización para la instancia como un todo, pero luego anular el valor predeterminado para una sesión o para un programa en particular. Por ejemplo:

```
ALTERAR SESIÓN ESTABLECER PLSQL_OPTIMIZE_LEVEL = 0;
```

Oracle conserva la configuración del optimizador módulo por módulo. Cuando vuelva a compilar un módulo en particular con configuraciones no predeterminadas, las configuraciones se "mantendrán", lo que le permitirá volver a compilar más tarde usando REUTILIZAR CONFIGURACIONES. Por ejemplo:

```
ALTERAR PROCEDIMIENTO bigproc COMPILE PLSQL_OPTIMIZE_LEVEL = 0;
```

y luego:

```
ALTERAR PROCEDIMIENTO bigproc COMPILE CONFIGURACIÓN DE REUTILIZACIÓN;
```

Para ver todas las configuraciones del compilador para sus módulos, incluido el nivel del optimizador, interpretado versus nativo y los niveles de advertencia del compilador, consulte la vista USER_PLSQL_OBJECT_SETTINGS.

Información sobre cómo funciona el Optimizer

Además de hacer cosas que los meros programadores no pueden hacer, los optimizadores también pueden detectar y explotar patrones en su código que quizás no note. Uno de los principales métodos que emplean los optimizadores es *reordenando* el trabajo que hay que hacer, para mejorar la eficiencia del tiempo de ejecución. La definición del lenguaje de programación circunscribe la cantidad de reordenamiento que puede hacer un optimizador, pero la definición de PL/SQL deja mucho margen de maniobra, o "libertad", para el optimizador. El resto de esta sección analiza algunas de las libertades que ofrece PL/SQL y brinda ejemplos de cómo se puede mejorar el código a la luz de ellas.

Como primer ejemplo, considérese el caso de un *bucle invariante*, algo que está dentro de un bucle pero que permanece constante en cada iteración. Cualquier programador que se precie se fijará en esto:

```
FOR e IN (SELECT * FROM empleados WHERE DEPT = p_dept)
LOOP
    DBMS_OUTPUT.PUT_LINE('<DEPT>' || p_dept || '</DEPT>');
    DBMS_OUTPUT.PUT_LINE('<emp ID="" | | e.empno | | ">');
    etc.
FIN DEL BUCLE;
```

y decirle que probablemente se ejecutaría más rápido si sacara la pieza "invariable" del bucle, por lo que no se vuelve a ejecutar innecesariamente:

```
I_dept_str := '<DEPT>' || departamento_p || '</DEPT>'
FOR e IN (SELECT * FROM empleados WHERE DEPT = p_dept)
LOOP
    DBMS_SALIDA.PUT_LINE(I_dept_str);
    DBMS_OUTPUT.PUT_LINE('<emp ID="" | | e.empno | | ">');
    etc.
FIN DEL BUCLE;
```

Incluso un programador digno de sal podría decidir, sin embargo, que la claridad de la primera versión supera las ganancias de rendimiento que le daría la segunda. A partir de Oracle Database 10 gramo, PL/SQL ya no lo obliga a tomar esta decisión. Con la configuración predeterminada del optimizador, el compilador detectará el patrón en la primera versión y lo convertirá en código de bytes que implementa la segunda versión. La razón por la que esto puede suceder es que la definición del lenguaje no requiere que los bucles invariantes se ejecuten repetidamente; esta es una de las libertades que el optimizador puede explotar, y lo hace. Puede pensar que esta optimización es una pequeña cosa, y lo es, pero las pequeñas cosas pueden sumar. Nunca he visto una base de datos que se haya reducido con el tiempo. Muchos programas PL/SQL recorren todos los registros de una tabla en crecimiento, y una tabla de un millón de filas ya no se considera inusualmente grande. Personalmente, estaría muy feliz si Oracle eliminara automáticamente un millón de instrucciones innecesarias de mi código.

Como otro ejemplo, considere una serie de declaraciones como estas:

```
resultado1 := r * s *
t; ...
resultado2 := r * s * v;
```

Si no hay posibilidad de modificar *r* y *s* entre estas dos declaraciones, PL/SQL es libre de compilar el código de esta manera:

```
intermedio := r * s;
resultado1 := provisional *
t; ...
resultado2 := provisional * v;
```

El optimizador dará ese paso si piensa que almacenar el valor en una variable temporal será más rápido que repetir la multiplicación.

Oracle ha revelado estos y otros conocimientos sobre el optimizador PL/SQL en un documento técnico, "Freedom, Order, and PL/SQL Compilation", que está disponible en el [Red de tecnología Oracle](#) (ingrese el título del artículo en el cuadro de búsqueda). Para resumir algunos de los puntos principales del documento:

- A menos que su código requiera la ejecución de un fragmento de código en un orden particular según las reglas de las expresiones de cortocircuito o del orden de las sentencias, PL/SQL puede ejecutar el fragmento en un orden diferente al que se escribió originalmente. El reordenamiento tiene varias manifestaciones posibles. En particular, el optimizador puede cambiar el orden en que se ejecutan las secciones de inicialización del paquete, y si un programa que llama solo necesita acceso a una constante del paquete, el compilador puede simplemente almacenar esa constante con la persona que llama.
- PL/SQL trata la evaluación de índices de matrices y la identificación de campos en registros como operadores. Si tiene una colección anidada de registros y hace referencia a un elemento y campo en particular, como precio (producto) (tipo).liquidar, PL/SQL debe averiguar una dirección interna que esté asociada con la variable. Esta dirección se trata como una expresión; se puede almacenar y reutilizar más adelante en el programa para evitar el costo de volver a calcular.
- Como se mostró anteriormente, PL/SQL puede introducir valores intermedios para evitar cálculos.
- PL/SQL puede eliminar por completo operaciones como *x**0. Sin embargo, no se eliminará una llamada de función explícita; en la expresión *f()**0, la función *f()* siempre se llamará en caso de que haya efectos secundarios.
- PL/SQL no introduce nuevas excepciones.
- PL/SQL puede obviar la generación de excepciones. Por ejemplo, la excepción de dividir por 0 en este código se puede descartar porque no se puede alcanzar:

```
SI FALSO ENTONCES y := x/0; TERMINARA SI;
```

PL/SQL no tiene la libertad de cambiar qué manejador de excepciones manejará una excepción dada.

El punto 1 merece un poco de elaboración. En las aplicaciones que escribo, estoy acostumbrado a aprovechar las secciones de inicialización de paquetes, pero nunca me preocupé realmente por el orden de ejecución. Mis secciones de inicialización suelen ser pequeñas e implican la asignación de valores de búsqueda estáticos (normalmente recuperados de la base de datos), y estas operaciones parecen ser inmunes al orden de las operaciones. Si su aplicación debe garantizar el orden de ejecución, querrá sacar el código de la sección de inicialización y ponerlo en rutinas de inicialización separadas que invoque explícitamente. Por ejemplo, llamarías a:

```
pkgA.init();
pkgB.init();
```

justo donde necesita pkgA y luego pkgB inicializado. Este consejo es válido incluso si no está utilizando el compilador de optimización.

El punto 2 también merece algún comentario. El ejemplo es precio(producto)(tipo).liquidación. Si se hace referencia a este elemento varias veces donde el valor del tipo de variable cambia pero el valor de la variable producto no, entonces la optimización podría dividir el direccionamiento en dos partes: la primera para calcular el precio (producto) y la segunda (usada en varios lugares) para calcular el resto de la dirección. El código se ejecutará más rápido porque solo se vuelve a calcular la parte modificable de la dirección cada vez que se usa la referencia completa. Más importante aún, este es uno de esos cambios que el compilador puede hacer fácilmente, pero sería muy difícil para el programador hacerlo en el código fuente original debido a la semántica de PL/SQL. Muchos de los cambios de optimización son de este tipo;

PL/SQL incluye otras características para identificar y acelerar ciertos lenguajes de programación. En este código:

```
encimera:=encimera+1;
```

el compilador no genera código de máquina que hace la suma completa. En su lugar, PL/SQL detecta este idioma de programación y utiliza una instrucción especial de "incremento" de máquina virtual PL/SQL (PVM) que se ejecuta mucho más rápido que la suma convencional (esto se aplica a un subconjunto de tipos de datos numéricos: PLS_INTEGER y SIMPLE_INTEGER — pero no sucederá con NÚMERO).

También existe una instrucción especial para manejar código que concatena muchos términos:

```
cadena := 'valor1' || 'valor2' || 'valor3'...
```

En lugar de tratar esto como una serie de concatenaciones por pares, el compilador y PVM trabajan juntos y realizan la serie de concatenaciones en una sola instrucción.

La mayor parte de la reescritura que hace el optimizador será invisible para usted. Durante una actualización, puede encontrar un programa que no se comporte tan bien como pensaba, porque dependía de un orden de ejecución que el nuevo compilador ha cambiado. Parece probable que un común

El área problemática será el orden de inicialización del paquete, pero, por supuesto, su kilometraje puede variar.

Un comentario final: la forma en que el optimizador modifica el código es determinista, al menos para un valor dado de PLSQL_OPTIMIZE_LEVEL. En otras palabras, si escribe, compila y prueba su programa usando, digamos, el nivel de optimizador predeterminado de 2, su comportamiento no cambiará cuando mueva el programa a una computadora diferente o a una base de datos diferente, siempre que la base de datos de destino la versión y el nivel del optimizador son los mismos.

Optimización del tiempo de ejecución de Fetch Loops

Para versiones de bases de datos hasta Oracle9 inclusive/Versión 2 de la base de datos, un bucle FOR de cursor como el siguiente recuperaría exactamente una fila lógica por búsqueda.

```
PARA flecha EN (SELECCIONEalgoDEen algún lugar) BUCLE
```

```
...
```

```
FIN DEL BUCLE;
```

Entonces, si tuviera 500 filas para recuperar, habría 500 búsquedas y, por lo tanto, 500 "cambios de contexto" costosos entre PL/SQL y SQL.

Sin embargo, a partir de Oracle Database 10*gramo*, la base de datos realiza una "masificación" automática de esta construcción para que *cada búsqueda recupera (hasta) 100 filas*. El bucle FOR del cursor anterior usaría solo cinco recuperaciones para recuperar las 500 filas del motor SQL. Es como si la base de datos recodificara automáticamente su ciclo para usar la función BULKCOLLECT (descrita más adelante en este capítulo).

Esta característica aparentemente no documentada también funciona para el código de la forma:

```
PARA flecha ENnombre del cursor
```

```
BUCLE
```

```
...
```

```
FIN DEL BUCLE;
```

Sin embargo, lo hace *no* trabajar con código de la forma:

```
ABIERTOnombre del cursor;
```

```
BUCLE
```

```
    SALIR CUANDOnombre del cursor%EXTRAVIADO;
```

```
    BUSCARnombre del cursorEN ... FIN DEL BUCLE;
```

```
CERCAnombre del cursor;
```

Sin embargo, esta optimización interna debería ser una gran victoria para el caso del bucle FOR del cursor (que tiene el beneficio adicional de la concisión).

Técnicas de almacenamiento en caché de datos

Una técnica muy común para mejorar el rendimiento es crear cachés para los datos a los que se debe acceder repetidamente y que, al menos durante un período de tiempo, son estáticos (no cambian).

El SGA de la base de datos de Oracle es la "madre de todos los cachés", en lo que respecta a Oracle. Es un área de memoria (generalmente) muy grande y (siempre) muy compleja que sirve como intermediario entre la base de datos real (archivos en el disco) y los programas que manipulan esa base de datos.

Como se describe más detalladamente [encapítulo 20](#), el SGA almacena en caché la siguiente información (y mucho más, pero estos son los más relevantes para los programadores de PL/SQL):

- Cursors analizados
- Datos consultados por cursor de la base de datos
- Representaciones parcialmente compiladas de nuestros programas

Sin embargo, en su mayor parte, la base de datos no usa el SGA para almacenar en caché *datos del programa*. Cuando declara una variable en su programa, la memoria para esos datos se consume en el PGA (para servidor dedicado). Cada conexión a la base de datos tiene su propio PGA; la memoria requerida para almacenar los datos de su programa, por lo tanto, se copia en cada conexión que llama a ese programa.

Afortunadamente, hay un beneficio en el uso de la memoria PGA: su programa PL/SQL puede recuperar información más rápidamente de la PGA que de la SGA. Por lo tanto, el almacenamiento en caché basado en PGA ofrece algunas oportunidades interesantes para mejorar el rendimiento. Oracle también proporciona otros mecanismos de almacenamiento en caché específicos de PL/SQL para ayudar a mejorar el rendimiento de sus programas. En esta sección, aprenderá acerca de tres tipos de almacenamiento en caché PL/SQL (otra técnica que podría considerar utilizar contextos de aplicación):

Almacenamiento en caché basado en paquetes

Use el área de memoria UGA para almacenar datos estáticos que necesita recuperar muchas veces. Utilice programas PL/SQL para evitar acceder repetidamente a los datos a través de la capa SQL en el SGA. Esta es la técnica de almacenamiento en caché más rápida, pero también la más restrictiva en cuanto a las circunstancias en las que se puede utilizar de forma segura.

Almacenamiento en caché de funciones deterministas

Cuando declaras que una función es *determinística* y llame a esa función dentro de una instrucción SQL, Oracle almacenará en caché las entradas de la función y su valor de retorno. Si llama a la función con las mismas entradas, Oracle puede devolver el valor previamente almacenado sin llamar a la función.

Este último avance en el almacenamiento en caché de PL/SQL es el más emocionante y útil. Con una simple cláusula declarativa en el encabezado de su función, puede indicarle a la base de datos que almacene en caché los valores de entrada y retorno de la función. Sin embargo, a diferencia del enfoque determinista, la memoria caché de resultados de la función se utiliza cada vez que se llama a la función (no solo desde una instrucción SQL), y la memoria caché se invalida automáticamente cuando cambian los datos dependientes.



Cuando usa un caché basado en paquetes, almacena una copia de los datos. Debe estar muy seguro de que su copia es precisa y está actualizada. Es muy posible abusar de cada uno de estos enfoques de almacenamiento en caché y terminar con "datos sucios" que se entregan a los usuarios.

Almacenamiento en caché basado en paquetes

Una memoria caché basada en paquetes consta de una o más variables declaradas a nivel de paquete, en lugar de en cualquier subprograma del paquete. Los datos a nivel de paquete son candidatos para el almacenamiento en caché, porque este tipo de datos persiste durante una sesión, incluso si los programas en esa sesión no están usando los datos actualmente ni llamando a ninguno de los subprogramas en el paquete. En otras palabras, si declara una variable a nivel de paquete, una vez que asigna un valor a esa variable, mantiene ese valor hasta que se desconecta, vuelve a compilar el paquete o cambia el valor.

Exploraré el almacenamiento en caché basado en paquetes describiendo primero los escenarios en los que querrá usar esta técnica. Luego, veré un ejemplo simple de almacenamiento en caché de un solo valor. Finalmente, le mostraré cómo puede almacenar en caché toda o parte de una tabla relacional en un paquete y, por lo tanto, acelerar en gran medida el acceso a los datos de esa tabla.

Cuándo usar el almacenamiento en caché basado en paquetes

Consideré usar una caché basada en paquetes en las siguientes circunstancias:

- Aún no está utilizando Oracle Database 11g o más alto. Si está desarrollando aplicaciones para versiones recientes, casi siempre será mejor usar la caché de resultados de funciones, no una caché basada en paquetes.
- Los datos que desea almacenar en caché no cambian durante el tiempo que un usuario necesita los datos. Los ejemplos de datos estáticos incluyen pequeñas tablas de referencia ("O" es para "Abierto", "C" es para "Cerrado", etc.) que rara vez, si es que alguna vez, cambian, y secuencias de comandos por lotes que requieren una "instantánea" de datos consistentes. se toma en el momento en que comienza el guión y se usa hasta que finaliza.
- Su servidor de base de datos tiene suficiente memoria para admitir una copia de su caché para cada sesión conectada a la instancia (y usando su caché). Puedes usar la utilidad

descripto anteriormente en este capítulo para medir el tamaño del caché definido en su paquete.

Por el contrario, haz *noutilice* una memoria caché basada en paquetes si se cumple alguna de las siguientes condiciones:

- Los datos que está almacenando en caché posiblemente podrían cambiar durante el tiempo que el usuario accede al caché.
- El volumen de datos almacenados en caché requiere demasiada memoria por sesión, lo que provoca errores de memoria con una gran cantidad de usuarios.

Un ejemplo simple de almacenamiento en caché basado en paquetes

Considere la función USUARIO: devuelve el nombre de la sesión actualmente conectada. Oracle implementa esta función en el paquete STANDARD de la siguiente manera:

```
la función USUARIO devuelve varchar2 es
c varchar2 (255);
comenzar
    seleccione usuario en c desde sys.dual;
    volver c;
fin;
```

Por lo tanto, cada vez que llama a USER, ejecuta una consulta. Claro, es una consulta rápida, pero nunca debe ejecutarse más de una vez en una sesión, ya que el valor nunca cambia. Probablemente ahora te estés diciendo a ti mismo: ¿y qué? Un SELECT FROM dual no solo es muy eficiente, sino que la base de datos de Oracle también almacenará en caché la consulta analizada y el valor devuelto, por lo que ya está muy optimizado. ¿Haría alguna diferencia el almacenamiento en caché basado en paquetes? ¡Absolutamente!

Considere el siguiente paquete:

```
/* Archivo en la web: esteusuario.pkg */
PAQUETE esteusuario
ES
    cname CONSTANTE VARCHAR2(30) := USUARIO;
    FUNCIÓN nombre RETORNO VARCHAR2;
FIN;

CUERPO DEL PAQUETE este usuario
ES
    g_user VARCHAR2(30) := USUARIO;

    FUNCIÓN nombre RETORNO VARCHAR2 ES COMENZAR RETORNO g_user; FIN;
    FIN;
```

Guardo en caché el valor devuelto por el USUARIO de dos maneras diferentes:

- Una constante definida a nivel de paquete. El motor de tiempo de ejecución de PL/SQL llama al USUARIO para inicializar la constante cuando se inicializa el paquete (en el primer uso).

- Una función. La función devuelve el nombre de "este usuario": el valor devuelto por la función es una variable privada (cuerpo del paquete) que también tiene asignado el valor devuelto por USER cuando se inicializa el paquete.

Habiendo creado estos cachés, debería ver si valen la pena. ¿Alguna de las implementaciones es notablemente más rápida que simplemente llamar a la función USER altamente optimizada una y otra vez?

Entonces, construyo un script utilizando sf_timer para comparar el rendimiento:

```
/* Archivo en la web: esteusuario.tst */
PROCEDIMIENTO test_thisuser (count_in IN PLS_INTEGER) ES

    l_name all_users.username%TYPE;
    COMENZAR
        sf_timer.start_timer;
        FOR indx IN 1 .. count_in LOOP l_name := esteusuario.NOMBRE; FIN DEL
        BUCLE; sf_timer.show_elapsed_time ('Función empaquetada');
        --
        sf_timer.start_timer;
        FOR indx IN 1 .. count_in LOOP l_name := thisuser.cname; FIN DEL BUCLE;
        sf_timer.show_elapsed_time ('Constante empaquetada');
        --
        sf_timer.start_timer;
        FOR indx IN 1 .. count_in LOOP l_name := USER; FIN DEL BUCLE;
        sf_timer.show_elapsed_time ('Función USUARIO');
    FIN test_thisuser;
```

Y cuando ejecuto 100 y luego 1,000,000 de iteraciones, veo estos resultados:

```
Función empaquetada transcurrido: 0 segundos.
Transcurrido constante empaquetado: 0 segundos.
Función USUARIO Transcurrido: 0 segundos.
```

```
Transcurrida la función empaquetada: 0,48 segundos.
Transcurrido constante empaquetado: 0,06 segundos.
Transcurrido de la función USUARIO: 32,6 segundos.
```

Los resultados son claros: para un pequeño número de iteraciones, la ventaja del almacenamiento en caché no es evidente. Sin embargo, para un gran número de iteraciones, la memoria caché basada en paquetes es mucho más rápida que pasar por la capa SQL y SGA.

Además, acceder a la constante es más rápido que llamar a una función que devuelve el valor. Entonces, ¿por qué usar una función? La versión de función ofrece esta ventaja sobre la constante: *se esconde* el valor. Por lo tanto, si por alguna razón se debe cambiar el valor (no aplicable a este escenario), puede hacerlo sin volver a compilar la especificación del paquete, lo que obligaría a volver a compilar todos los programas que dependen de este paquete.

Si bien es poco probable que alguna vez se beneficie del almacenamiento en caché del valor devuelto por la función USUARIO, espero que pueda ver que el almacenamiento en caché basado en paquetes es claramente una forma muy eficiente de almacenar y recuperar datos. Ahora echemos un vistazo a un ejemplo menos trivial.

Almacenamiento en caché del contenido de la tabla en un paquete

Si su aplicación incluye una tabla que nunca cambia durante el horario normal de trabajo (es decir, es estática mientras un usuario accede a la tabla), puede crear fácilmente un paquete que almacene en caché el contenido completo de esa tabla, lo que aumenta el rendimiento de las consultas mediante un pedido de magnitud o más.

Supongamos que tengo una tabla de productos que es estática, definida de la siguiente manera:

```
/* Archivo en web: package_cache_demo.sql */
TABLE productos (
    número_producto CLAVE PRIMARIA ENTERA
    , descripción VARCHAR2(1000))
```

Aquí hay un cuerpo de paquete que ofrece dos formas de consultar datos de esta tabla: consultar cada vez o almacenar en caché los datos y recuperarlos del caché:

```
1 CUERPO DEL PAQUETE products_cache
2 ES
3 TIPO cache_t ES TABLA DE productos%ROWTYPE ÍNDICE POR PLS_INTEGER;
4 g_caché caché_t;
5
6 FUNCIÓN with_sql (product_number_in IN productos.product_number%TYPE)
7 DEVOLVER productos%ROWTYPE
8 ES
9     l_fila     productos%ROWTYPE;
10 COMENZAR
11     SELECCIONE * EN l_fila DESDE productos DONDE
12         número_producto = número_producto_en; RETORNO
13     l_fila;
14 FIN con_sql;
15
16 FUNCIÓN from_cache (product_number_in IN productos.product_number%TYPE)
17 DEVOLVER productos%ROWTYPE
18 ES
19 COMENZAR
20     RETORNO g_cache (producto_numero_en);
21 FIN de_caché;
22 COMENZAR
23     PARA product_rec IN (SELECCIONE * DE productos) BUCLE
24         g_cache (product_rec.product_number) := product_rec; FIN DEL
25     BUCLE;
26 FIN productos_cache;
```

La siguiente tabla explica las partes interesantes de este paquete.

Líneas)	Significado
3-4	Declare una caché de matriz asociativa, g_cache, que imite la estructura de la tabla de mis productos: cada elemento de la colección es un registro con la misma estructura que una fila de la tabla.
6-14	La función with_sql devuelve una fila de la tabla de productos para una clave principal dada, utilizando el método SELECT INTO "tradicional". En otras palabras, cada vez que llamas a esta función ejecutas una consulta.

Líneas)	Significado
16–21	La función from_cache también devuelve una fila de la tabla de productos para una clave principal dada, pero lo hace usando esa clave principal como valor de índice, ubicando así la fila en g_cache.
23–25	Cuando se inicialice el paquete, cargue el contenido de la tabla de productos en la colección g_cache. Tenga en cuenta que uso el valor de la clave principal como índice en la colección. Esta emulación de la clave principal es lo que hace posible (y tan simple) la implementación de from_cache.

Con este código en su lugar, la primera vez que un usuario llama a la función from_cache (o with_sql), la base de datos primero ejecutará este código.

A continuación, construyo y ejecuto un bloque de código para comparar el rendimiento de estos enfoques:

```

DECLARAR
    l_fila      productos%ROWTYPE;
COMENZAR
    sf_timer.start_timer;
    PARA indx EN 1 .. 100000
    BUCLE
        l_fila := productos_cache.from_cache (5000); FIN
    DEL BUCLE;
    sf_timer.show_elapsed_time ('Tabla de caché');
    --
    sf_timer.start_timer;
    PARA indx EN 1 .. 100000
    BUCLE
        l_fila := productos_cache.with_sql (5000); FIN DEL
    BUCLE;
    sf_timer.show_elapsed_time ('Ejecutar consulta cada vez'); FIN;

```

Y aquí están los resultados que veo:

```

Tabla de caché Transcurrido: 0,14 segundos.
Ejecutar consulta cada vez que transcurre: 4,7 segundos.

```

Una vez más, está muy claro que el almacenamiento en caché basado en paquetes es mucho, mucho más rápido que ejecutar una consulta repetidamente, incluso cuando esa consulta está completamente optimizada por todo el poder y la sofisticación de SGA.

Almacenamiento en caché justo a tiempo de los datos de la tabla

Supongamos que he identificado una tabla estática a la que quiero aplicar esta técnica de almacenamiento en caché. Sin embargo, hay un problema: la tabla tiene 100.000 filas de datos. Puedo crear un paquete como products_cache, que se muestra en la sección anterior, pero usa 5 MB de memoria en el PGA de cada sesión. Con 500 conexiones simultáneas, este caché consumirá 2,5 GB, lo cual es inaceptable. Afortunadamente, me doy cuenta de que aunque la tabla tiene muchas filas de datos, cada usuario generalmente consultará solo las mismas 50 o más filas de esos datos (hay

son, en otras palabras, focos de actividad). Por lo tanto, almacenar en caché la tabla completa en cada sesión es un desperdicio en términos de ciclos de CPU (la carga inicial de 100 000 filas) y memoria.

Cuando tu mesa es estática, pero no quieres ni necesitas *todos* los datos en esa tabla, debe considerar emplear un enfoque "justo a tiempo" para el almacenamiento en caché. Esto significa que lo haces *no* consulta el contenido completo de la tabla en tu caché de colección cuando el paquete se inicializa. En cambio, cada vez que el usuario solicita una fila, si está en el caché, se la devuelve de inmediato. De lo contrario, consulta esa única fila de la tabla, la agrega a la memoria caché y luego devuelve los datos.

La próxima vez que el usuario solicite esa misma fila, se recuperará de la memoria caché. El siguiente código demuestra este enfoque:

```
/* Archivo en la web: package_cache_demo.sql */
FUNCTION jit_from_cache (product_number_in IN productos.product_number%TYPE)
    DEVOLVER productos%ROWTYPE
ES
    l_fila      productos%ROWTYPE;
COMENZAR
    SI g_cache.EXISTE (product_number_in)
    ENTONCES
        /* Ya está en el caché, así que devuélvelo. */
        l_row := g_cache (product_number_in); DEMÁS
        /* Primera solicitud, así que consúltela desde la base de datos
           y luego agregarlo al caché. */ l_row :=
        with_sql (product_number_in); g_cache
        (producto_numero_en) := l_fila; TERMINARA
        SI;
RETORNO l_fila;
FIN jit_from_cache;
```

En general, el almacenamiento en caché justo a tiempo es un poco más lento que la carga única de todos los datos en el caché, pero sigue siendo mucho más rápido que las búsquedas repetidas en la base de datos.

Almacenamiento en caché de funciones deterministas

Se considera una función *determinista* si devuelve el mismo valor de resultado cada vez que se llama con los mismos valores para sus argumentos IN y 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. Ver [capítulo 17](#) para más detalles sobre las funciones deterministas.

Precisamente porque una función determinista se comporta de manera tan consistente, Oracle puede construir un caché a partir de las entradas y salidas de la función. Después de todo, si las mismas entradas siempre devuelven el mismo resultado, entonces no hay motivo para llamar a la función una segunda vez si las entradas coinciden con una invocación anterior de esa función.

Echemos un vistazo a un ejemplo de la naturaleza de almacenamiento en caché de las funciones deterministas. Supongamos que defino la siguiente encapsulación encima de SUBSTR (devuelve la cadena entre las ubicaciones inicial y final) como una función determinista:

```
/* Archivo en la web: deterministic_demo.sql */
FUNCTION betwnstr (
    string_in EN VARCHAR2, start_in EN PLS_INTEGER, end_in EN PLS_INTEGER)
DE VOLVER VARCHAR2 DETERMINISTA
ES
COMENZAR
    RETURN (SUBSTR (entrada_de_cadena, entrada_de_inicio, entrada_de_final - entrada_de_inicio + 1));
FIN entre calle;
```

Luego puedo llamar a esta función dentro de una consulta (no modifica ninguna tabla de la base de datos, lo que de otro modo impediría usarla de esta manera), como:

```
SELECCIONE betweenstr (apellido, 1, 5) first_five
DE empleados
```

Y cuando se llama a betwnstr de esta manera, la base de datos creará un caché de entradas y sus valores de retorno. Luego, si vuelvo a llamar a la función con las mismas entradas, la base de datos devolverá el valor sin llamar a la función. Para demostrar esta optimización, cambiaré betweenstr a lo siguiente:

```
FUNCTION entre str (
    string_in EN VARCHAR2, start_in EN PLS_INTEGER, end_in EN PLS_INTEGER)
DE VOLVER VARCHAR2 DETERMINISTA
ES
COMENZAR
    DBMS_LOCK.dormir (.01);
    RETURN (SUBSTR (entrada_de_cadena, entrada_de_inicio, entrada_de_final - entrada_de_inicio + 1));
FIN entre calle;
```

En otras palabras, usaré el subprograma de suspensión de DBMS_LOCK para pausar betweenstr por 1/100 de segundo.

Si llamo a esta función en un bloque de código PL/SQL (no desde dentro de una consulta), la base de datos *no* almacene en caché los valores de la función, por lo que cuando consulto las 107 filas de la tabla de empleados, tomará más de un segundo:

```
DECLARAR
    l_string empleados.apellido%TYPE;
COMENZAR
    sf_timer.start_timer;

    FOR rec IN (SELECCIONE * DE empleados)
    BUCLE
        cadena_l := cadena_entre('FEUERSTEIN', 1, 5); FIN DEL
        BUCLE;

    sf_timer.show_elapsed_time ('Función determinista en bloque');
```

```
FIN;  
/
```

La salida es:

Función determinista en bloque Transcurrido: 1,67 segundos.

Si ahora ejecuto la misma lógica, pero muevo la llamada a `btwnstr` dentro la consulta, el rendimiento es bastante diferente:

```
COMENZAR  
    sf_timer.start_timer;  
  
    PARA rec IN (SELECCIONE bewnstr ('FEUERSTEIN', 1, 5) DE empleados) BUCLE  
  
        NULO;  
        FIN DEL BUCLE;  
  
        sf_timer.show_elapsed_time ('Función determinista en consulta'); FIN;  
  
    /
```

La salida es:

Función determinista en consulta Transcurrido: 0,05 segundos.

Como puede ver, el almacenamiento en caché con una función determinista es un camino muy efectivo hacia la optimización. Solo asegúrese de lo siguiente:

- Cuando declara que una función es determinista, asegúrese de que realmente *es*. La base de datos de Oracle no analiza su programa para determinar si está diciendo la verdad. Si agrega la palabra clave DETERMINISTIC a una función que, por ejemplo, consulta datos de una tabla, la base de datos podría almacenar datos en caché de manera inapropiada, con la consecuencia de que un usuario ve "datos sucios".
- Debe llamar a esa función dentro de una instrucción SQL para obtener los efectos del almacenamiento en caché determinista; esa es una restricción significativa sobre la utilidad de este tipo de almacenamiento en caché.

Caché de resultados de la función (Oracle Database 11g)

Antes del lanzamiento de Oracle Database 11g, el almacenamiento en caché basado en paquetes ofrecía la mejor y más flexible opción para almacenar datos en caché para su uso en un programa PL/SQL. Lamentablemente, las circunstancias en las que se puede usar son bastante limitadas, ya que la fuente de datos debe ser estática y el consumo de memoria crece con cada sesión conectada a la base de datos de Oracle.

Reconociendo el beneficio de rendimiento de este tipo de almacenamiento en caché (así como el implementado para funciones deterministas), Oracle implementó el *caché de resultados de función* en base de datos Oracle 11g. Esta característica ofrece una solución de almacenamiento en caché que supera las debilidades del almacenamiento en caché basado en paquetes y ofrece un rendimiento casi igual de rápido.

Cuando activa la caché de resultados de función para una función, obtiene los siguientes beneficios:

- Oracle almacena tanto las entradas como sus valores de retorno en un caché separado para cada función. El caché se comparte entre todas las sesiones conectadas a esta instancia de la base de datos; es *no* duplicado para cada sesión. En base de datos Oracle 11*gramo*Versión 2 y posteriores, la memoria caché de resultados de funciones incluso se comparte entre instancias en un Real Application Cluster (RAC).
- Cada vez que se llama a la función, la base de datos comprueba si ya ha almacenado en caché los mismos valores de entrada. Si es así, entonces la función no se ejecuta. El valor almacenado en el caché simplemente se devuelve.
- Cada vez que se confirman cambios en las tablas que se identifican como dependencias para la memoria caché, la base de datos invalida automáticamente la memoria caché. Las llamadas posteriores a la función volverán a llenar el caché con datos consistentes.
- El almacenamiento en caché se produce cada vez que se llama a la función; no necesita invocarlo dentro de una declaración SQL.
- No hay necesidad de escribir código para declarar y llenar una colección; en su lugar, utiliza la sintaxis declarativa en el encabezado de la función para especificar la memoria caché.

Lo más probable es que utilice la función de caché de resultados con funciones que consultan datos de tablas. Los candidatos excelentes para el almacenamiento en caché de resultados son:

- Conjuntos de datos estáticos, como vistas materializadas. El contenido de estas vistas no cambia entre sus actualizaciones, entonces, ¿por qué obtener los datos varias veces?
- Tablas que se consultan con mucha más frecuencia de la que se modifican. Si una tabla se cambia en promedio cada cinco minutos, pero entre cambios se consultan las mismas filas cientos o miles de veces, la memoria caché de resultados se puede utilizar con buenos resultados.

Sin embargo, si su tabla cambia cada segundo, no desea almacenar en caché los resultados; en realidad podría *desacelerar* su aplicación, ya que Oracle pasará mucho tiempo completando y luego limpiando el caché. Elija cuidadosamente cómo y dónde aplicar esta función y trabaje en estrecha colaboración con su DBA para asegurarse de que el conjunto de SGA para la caché de resultados sea lo suficientemente grande como para contener todos los datos que espera que se almacenen en caché durante el uso típico de producción.

En las siguientes secciones, primero describiré la sintaxis de esta función. Luego, demostraré algunos ejemplos simples del uso de la caché de resultados, analizaré las circunstancias en las que debe usarla, cubriré los aspectos relacionados con DBA de la administración de la caché y revisaré las restricciones y los errores de esta función.

Habilitación de la caché de resultados de la función

Oracle ha hecho que sea muy fácil agregar el almacenamiento en caché de resultados de funciones a sus funciones. Simplemente necesita agregar la cláusula RESULT_CACHE al encabezado de su función, y Oracle lo toma desde allí.

La sintaxis de la cláusula RESULT_CACHE es:

```
RESULT_CACHE [ CONFIANZA_EN (tabla_o_vista[,tabla_o_vista2...tabla_o_vistaN] ]
```

La cláusula RELIES_ON le dice a Oracle en qué tablas o vistas se basa el contenido de la memoria caché. Esta cláusula solo se puede agregar a los encabezados de funciones de nivel de esquema y el *implementación*de una función empaquetada (es decir, en el cuerpo del paquete). A partir de Oracle Database 11*gramo*Versión 2, está en desuso. Aquí hay un ejemplo de una función empaquetada. Tenga en cuenta que la cláusula RESULT_CACHE debe aparecer tanto en la especificación como en el cuerpo:

```
CREAR O REEMPLAZAR EL PAQUETE get_data ES  
  
    FUNCIÓN session_constant RETURN VARCHAR2 RESULT_CACHE; FIN  
    obtener_datos;  
/  
  
CREAR O REEMPLAZAR EL CUERPO DEL PAQUETE get_data  
ES  
    FUNCIÓN session_constant RETORNO VARCHAR2  
        RESULTADO_CACHE  
    ES  
    COMENZAR  
        ...  
    FIN session_constant;  
    FIN obtener_datos;  
/
```

Una característica tan elegante; simplemente agregue una cláusula al encabezado de su función y vea una mejora significativa en el rendimiento.

La cláusula RELIES_ON (en desuso en 11.2)

Lo primero que debe comprender acerca de RELIES_ON es que ya no es necesario a partir de Oracle Database 11*gramo*Versión 2. A partir de esa versión, Oracle *automáticamente* determine de qué tablas dependen sus datos devueltos e invalide correctamente el caché cuando se cambie el contenido de esas tablas; incluir su propia cláusula RELIES_ON no hace nada. ejecutar el *11gR2_frc_no_relies_on.sql*script disponible en el sitio web del libro para verificar este comportamiento. Este análisis identifica tablas a las que se hace referencia a través de SQL estático (incrustado) o dinámico, así como tablas a las que solo se hace referencia *indirectamente*(a través de vistas).

Si está ejecutando Oracle Database 11*gramo*Versión 1 o anterior, sin embargo, depende de usted enumerar explícitamente todas las tablas y vistas desde las que se consultan los datos devueltos. Determinar qué tablas y vistas incluir en la lista suele ser bastante sencillo. Si su función contiene una declaración SELECT, asegúrese de que las tablas o vistas en cualquier cláusula FROM en esa consulta se agreguen a la lista.

Si selecciona desde una vista, debe enumerar solo esa vista, no todas las tablas que se consultan desde dentro de la vista. El guion llamado *11g_frc_views.sql*, también disponible en el sitio web,

demuestra cómo la base de datos determinará a partir de la propia definición de la vista todas las tablas cuyos cambios deben invalidar la memoria caché.

Estos son algunos ejemplos del uso de la cláusula RELIES_ON:

1. Como función de nivel de esquema con una cláusula RELIES_ON que indica que el caché se basa en la tabla de empleados:

```
CREAR O REEMPLAZAR FUNCIÓN name_for_id (id_in IN employee.employee_id%TYPE)
    RETURN empleados.apellido%TYPE
    RESULT_CACHE RELIES ON (empleados)
```

2. Una función empaquetada con una cláusula RELIES_ON (puede parecerse a *en el cuerpo*):

```
CREAR O REEMPLAZAR EL PAQUETE get_data ES

    FUNCIÓN name_for_id (id_in EN empleados.employee_id%TYPE)
        RETURN empleados.apellido%TYPE
        RESULT_CACHE
    FIN obtener_datos;
    /

    CREAR O REEMPLAZAR EL CUERPO DEL PAQUETE get_data
    ES
        FUNCIÓN name_for_id (id_in EN empleados.employee_id%TYPE)
            RETURN empleados.apellido%TYPE
            RESULT_CACHE RELIES ON (empleados)
    ES
    COMENZAR
    ...
    FIN nombre_para_id;
    FIN obtener_datos;
    /
```

3. Una cláusula RELIES_ON con varios objetos enumerados:

```
CREAR O REEMPLAZAR EL CUERPO DEL PAQUETE get_data
ES
    FUNCIÓN name_for_id (id_in EN empleados.employee_id%TYPE)
        VOLVER empleados.apellido%TIPO
        RESULT_CACHE CONFIA EN (empleados, departamentos, ubicaciones)
    ...
```

Ejemplo de caché de resultados de función: una función determinista

En una sección anterior hablé sobre el almacenamiento en caché asociado con las funciones deterministas. En particular, noté que este almacenamiento en caché solo entrará en juego cuando se llame a la función dentro de una consulta. Apliquemos ahora Oracle Database 11g *gramocaché* de resultados de la función a la función *btwnstr* y vea qué funciona cuando se llama de forma nativa en un bloque PL/SQL.

En la siguiente función, agrego la cláusula RESULT_CACHE al encabezado. También agrego una llamada a DBMS_OUTPUT.PUT_LINE para mostrar qué entradas se pasaron a la función:

```

/* Archivo en la web: 11g_frc_simple_demo.sql */
FUNCTION betwnstr (
    string_in EN VARCHAR2, start_in EN INTEGER, end_in          EN ENTERO)
    DEVOLVER VARCHAR2 RESULT_CACHE
ES
COMENZAR
    DBMS_OUTPUT.put_line (
        'entrecadena para' || cadena_en || ' ' || inicio_en || ' ' || fin_en); RETURN (SUBSTR
        (entrada_de_cadena, entrada_de_inicio, entrada_de_final - entrada_de_inicio + 1)); FIN;

```

Luego llamo a esta función para 10 filas en la tabla de empleados. Si la identificación del empleado es par, entonces aplico betweenstr al apellido del empleado. De lo contrario, le paso los mismos tres valores de entrada:

```

DECLARAR
    l_cadena    empleados.apellido%TYPE;
COMENZAR
    PARA rec IN (SELECCIONE * DE empleados DONDE ROWNUM < 11)
    BUCLE
        l_cadena :=  

            CASO MOD (rec.employee_id, 2)  

                CUANDO 0 ENTONCES betweenstr (rec.apellido, 1, 5) DE LO  

                CONTRARIO      entre calle ('FEUERSTEIN', 1, 5)  

            FIN;  

        FIN DEL BUCLE;  

    FIN;

```

Cuando ejecuto esta función, veo el siguiente resultado:

```

betwnstr para OConnell-1-5
betwnstr para FEUERSTEIN-1-5
betwnstr para Whalen-1-5
betwnstr para Fay-1-5
calle intermedia para Baer-1-5
entre calle para Gietz-1-5
betweenstr para King-1-5

```

Nótese que FEUERSTEIN aparece una sola vez, aunque fue llamado cinco veces. Eso demuestra la caché de resultados de la función en acción.

Ejemplo de caché de resultados de función: consulta de datos de una tabla

La mayoría de las veces querrá usar la caché de resultados de la función cuando esté consultando datos de una tabla cuyos contenidos se consultan con más frecuencia de lo que se modifican (entre cambios, los datos son estáticos). Supongamos, por ejemplo, que en mi aplicación de gestión inmobiliaria tengo una tabla que contiene las tasas de interés disponibles para diferentes tipos de préstamos. El contenido de esta tabla se actualiza a través de un trabajo programado que se ejecuta una vez por hora durante el día. Aquí está la estructura de la tabla y los datos que estoy usando en mi secuencia de comandos de demostración:

```

/* Archivo en la web: 11g_frc_demo_table.sql */
CREAR TABLA préstamo_info (
    NOMBRE VARCHAR2(100) CLAVE PRINCIPAL,
    longitud_de_préstamo INTEGER,
    tasa_de_interés_inicial NÚMERO,
    tasa_de_interés_regular NÚMERO,
    porcentaje_pago_inicial INTEGER)
/
COMENZAR
    INSERTAR EN PRÉSTAMO_info VALORES ('Fijo de cinco años', 5, 6, 6, 20); INSERTAR EN
    PRÉSTAMO_INFO VALORES ('Fijo por diez años', 10, 5.7, 5.7, 20); INSERTAR EN
    PRÉSTAMO_info VALORES ('Fijo por quince años', 15, 5.5, 5.5, 10); INSERTAR EN
    PRÉSTAMO_INFO VALORES ('Treinta años fijo', 30, 5, 5, 10); INSERTAR EN
    PRÉSTAMO_info VALORES ('Globo de dos años', 2, 3, 8, 0);
    INSERTAR EN PRÉSTAMO_info VALORES ('Globo de cinco años', 5, 4, 10, 5);
    COMPROMETERSE;
FIN;
/

```

Aquí hay una función para recuperar toda la información de una sola fila:

```

FUNCIÓN préstamo_info_para_nombre (NOMBRE_IN EN VARCHAR2)
    RETURN info_préstamo%ROWTYPE
    RESULT_CACHE RELIES_ON (info_préstamo)
ES
    l_fila    préstamo_info%ROWTYPE;
COMENZAR
    DBMS_OUTPUT.put_line ('> Buscando información de préstamo para ' || NAME_IN);

    SELECCIONE * EN l_fila DESDE préstamo_info DONDE NOMBRE = NOMBRE_EN;

    RETORNO l_fila;
FIN préstamo_info_para_nombre;

```

En este caso, la cláusula RESULT_CACHE incluye la subcláusula RELIES_ON para indicar que la memoria caché para esta función se basa en datos de ("depende de") la tabla de información_préstamo. Luego ejecuto el siguiente script, que llama a la función para dos nombres diferentes, luego cambia el contenido de la tabla y finalmente vuelve a llamar a la función para uno de los nombres originales:

```

DECLARAR
    l_fila    préstamo_info%ROWTYPE;
COMENZAR
    DBMS_OUTPUT.put_line ('Se corrigió la primera vez durante cinco años...');
    l_row:=préstamo_info_para_nombre ('Cinco años fijo');
    DBMS_OUTPUT.put_line ('Primera vez para globo de cinco años...'); l_row :=
    préstamo_info_para_nombre ('Globo de cinco años'); DBMS_OUTPUT.put_line
    ('Se corrigió la segunda vez durante cinco años...'); l_row :=
    préstamo_info_para_nombre ('Cinco años fijo');

    ACTUALIZAR préstamo_info SET porcentaje_pago inicial = 25
    WHERE NOMBRE = 'Treinta años fijo';

    COMPROMETERSE;

```

```

DBMS_OUTPUT.put_line ('Después de la confirmación, se corrigió la tercera vez durante cinco años...');
l_row := préstamo_info_para_nombre ('Cinco años fijo');
FIN;

```

Aquí está el resultado de ejecutar este script:

```

Primera vez por cinco años fijo...
> Buscando información de préstamo para cinco años
fijo Primera vez para cinco años globo...
> Buscando información de préstamo para el globo de cinco
años Segunda vez para el fijo de cinco años...
Después del compromiso, la tercera vez durante cinco años fijo...
> Buscando información de préstamo para cinco años fijos

```

Y aquí hay una explicación de lo que ves que sucede aquí:

- *El primera vez* que llamo a la función para "Cinco años fijos", el motor de tiempo de ejecución PL/SQL ejecuta la función, busca los datos, coloca los datos en el caché y devuelve los datos.
- La primera vez que llamo a la función para "Globo de cinco años", ejecuta la función, busca los datos, coloca los datos en el caché y devuelve los datos.
- *El segundo vez* que llamo a la función para "Cinco años fijos", no ejecuta la función (no hay "Buscando ..." para la segunda llamada). La caché de resultados de la función en el trabajo...
- Luego cambio un valor de columna para la fila con el nombre "Treinta años fijos" y confirme ese cambio.
- Finalmente, llamo a la función para el *tercer tiempo* para "Cinco años fijos". Esta vez, la función se vuelve a ejecutar para consultar los datos. Esto sucede porque le dije a Oracle que este RESULT_CACHE CONFIA_EN la tabla de información_de_préstamo, y el contenido de esa tabla ha cambiado.

Ejemplo de caché de resultados de función: almacenamiento en caché de una colección

Hasta ahora le he mostrado ejemplos de almacenamiento en caché de un valor individual y un registro completo. También puede almacenar en caché una colección completa de datos, incluso una colección de registros. En el siguiente código, cambié la función para devolver todos los nombres de los préstamos a una colección de cadenas (según el tipo de colección DBMS_SQL predefinido). Luego llamo a la función repetidamente, pero la colección se completa solo una vez (BULK COLLECT se describe más adelante en este capítulo):

```

/* Archivo en la web: 11g_frc_table_demo.sql */ FUNCIÓN
préstamo_nombres RETURN DBMS_SQL.VARCHAR2S
    RESULT_CACHE RELIES_ON (préstamo_info)
ES
    l_nombres      DBMS_SQL.VARCHAR2S;

```

```

COMENZAR
    DBMS_OUTPUT.put_line ('> Buscando nombres de préstamos....');

    SELECCIONE el nombre BULK COLLECT EN l_names DE préstamo_info;
    RETORNO l_nombres;
END préstamos_nombres;

```

Aquí hay una secuencia de comandos que demuestra que incluso al completar un tipo complejo como este, el caché de resultados de la función entrará en juego:

```

DECLARAR
    l_nombres      DBMS_SQL.VARCHAR2S;
COMENZAR
    DBMS_OUTPUT.put_line ('Primera vez recuperando todos los nombres...');

    l_nombres := prestamo_nombres ();
    DBMS_OUTPUT.put_line('Segunda vez recuperando todos los nombres...');

    l_nombres := prestamo_nombres ();

    ACTUALIZAR préstamo_info SET porcentaje_pago inicial = 25
        WHERE NOMBRE = 'Treinta años fijo';

    COMPROMETERSE;
    DBMS_OUTPUT.put_line ('Después de confirmar, por tercera vez recuperando todos los nombres...');

    l_nombres := prestamo_nombres ();
    FIN;
/

```

La salida es:

```

Primera vez recuperando todos los nombres...
> Buscando nombres de préstamos... Segunda
vez recuperando todos los nombres...
Después de confirmar, la tercera vez recuperando todos los nombres...
> Buscando nombres de préstamos....

```

Cuándo usar la caché de resultados de la función

El almacenamiento en caché debe realizarse siempre con el mayor cuidado. Si almacena en caché incorrectamente, su aplicación puede entregar datos incorrectos a los usuarios. La caché de resultados de la función es la más flexible y ampliamente útil de los diferentes tipos de cachés que puede usar en el código PL/SQL, pero aún puede meterse en problemas con ella.

Debería considerar agregar RESULT_CACHE al encabezado de su función en cualquiera de las siguientes circunstancias:

- Los datos se consultan desde una tabla con más frecuencia de lo que se actualiza. Supongamos, por ejemplo, que en mi aplicación de Recursos Humanos, los usuarios consultan el contenido de la tabla de empleados miles de veces por minuto, pero se actualiza en promedio una vez cada 10 minutos. Entre esos cambios, la tabla de empleados es estática, por lo que los datos se pueden almacenar en caché de forma segura y se reduce el tiempo de consulta.

- Una función que no consulta ningún dato se llama repetidamente (a menudo, en este escenario, recursivamente) con los mismos valores de entrada. Un ejemplo clásico de los textos de programación es el algoritmo de Fibonacci. Para calcular el valor de Fibonacci para el número *n* — es decir, $F(n)$: debe calcular $F(1)$ a través de $F(n-1)$ varias veces.
- Su aplicación (o cada usuario de la aplicación) se basa en un conjunto de valores de configuración que son estáticos durante el uso de la aplicación: ¡un complemento perfecto para la función de caché de resultados!

Cuándo no usar la caché de resultados de la función

No puede usar la cláusula RESULT_CACHE si se cumple alguna de las siguientes condiciones:

- La función se define dentro de la sección de declaración de un bloque anónimo. Para almacenar en caché los resultados, la función debe definirse en el nivel de esquema o dentro de un paquete.
- La función es una función de tabla segmentada.
- La función tiene cualquier parámetro OUT o IN OUT. En este caso, la función solo puede devolver datos a través de la cláusula RETURN.
- Cualquiera de los parámetros IN de la función es de cualquiera de estos tipos: BLOB, CLOB, NCLOB, REF CURSOR, colección, registro o tipo de objeto.
- El tipo de función RETURN es cualquiera de los siguientes: BLOB, CLOB, NCLOB, REF CURSOR, tipo de objeto o una colección o registro que contiene cualquiera de los tipos de datos enumerados anteriormente (por ejemplo, una colección de CLOB sería un no-go para el almacenamiento en caché de resultados de funciones).
- La función es una función de derechos de invocador y está utilizando Oracle Database 11g. En 11g, un intento de definir una función tanto en caché de resultados como usando derechos de invocador resultó en este error de compilación: *PLS-00999: restricción de implementación (puede ser temporal) RESULT_CACHE no está permitido en subprogramas en módulos Invoker-Rights*. Buenas noticias: esta restricción de implementación se eliminó en Oracle Database 12C, lo que permite el almacenamiento en caché de los resultados de las funciones definidas con la cláusula AUTHID CURRENT_USER. Conceptualmente, es como si Oracle pasara el nombre de usuario como un argumento invisible a la función.
- La función hace referencia a tablas de diccionario de datos, tablas temporales, secuencias o funciones SQL no deterministas.

No debe usar (o, como mínimo, evaluar con mucho cuidado su uso de) la cláusula RESULT_CACHE si se cumple alguna de las siguientes condiciones:

- Su función tiene efectos secundarios; por ejemplo, modifica el contenido de las tablas de la base de datos o modifica el estado externo de su aplicación (por ejemplo, enviando datos a sysout a través de DBMS_OUTPUT o enviando un correo electrónico). Ya que nunca puedes estar seguro

cuándo y si se ejecutará el cuerpo de la función, es probable que su aplicación no funcione correctamente en todas las circunstancias. Esta es una compensación inaceptable para mejorar el rendimiento.

- Su función (o la consulta dentro de ella) contiene dependencias específicas de la sesión, como una referencia a SYSDATE o USER, dependencias en la configuración de NLS (como una llamada a TO_CHAR que se basa en el modelo de formato predeterminado), etc.
- Su función ejecuta una consulta en una tabla en la que se aplica una política de seguridad de base de datos privada virtual (VPD). Exploro las ramificaciones del uso de VPD con almacenamiento en caché de resultados de funciones, en la sección “[Dependencias detalladas en 11.2 y versiones posteriores](#)” en la [página 863](#).

Detalles útiles del comportamiento de la caché de resultados de funciones

La siguiente información debería resultarle útil a medida que profundice en los detalles de la aplicación de la caché de resultados de la función a su aplicación:

- Al verificar si la función ha sido llamada previamente con las mismas entradas, Oracle considera que NULL es igual a NULL. En otras palabras, si mi función tiene un argumento de cadena y se llama con un valor de entrada NULO, la próxima vez que se llame con un valor NULO, Oracle decidirá que no necesita llamar a la función y, en cambio, puede devolver el resultado almacenado en caché.
- Los usuarios nunca ven datos sucios. Supongamos que una función de caché de resultados devuelve el apellido de un empleado para una identificación y que el apellido "Feuerstein" se almacena en caché para la identificación 400. Si un usuario cambia el contenido de la tabla de empleados, incluso si ese cambio aún no se ha realizado, comprometido, la base de datos omitirá el caché (y cualquier otro caché que dependa de los empleados) para la sesión de ese usuario. Todos los demás usuarios conectados a la instancia (o RAC, en Oracle Database 11g y versiones posteriores) seguirán aprovechando la memoria caché.
- Si la función propaga una excepción no controlada, la base de datos no almacenará en caché los valores de entrada para esa ejecución; es decir, el contenido de la caché de resultados para esta función no cambiará.

Administrar la memoria caché de resultados de la función

La caché de resultados de funciones es un área de memoria en el SGA. Oracle proporciona el elenco habitual de caracteres para que un administrador de base de datos pueda administrar ese caché:

Parámetro de inicialización RESULT_CACHE_MAX_SIZE

Especifica la cantidad máxima de memoria SGA que puede usar la caché de resultados de la función. Cuando la memoria caché se llena, Oracle utilizará el algoritmo utilizado menos recientemente para eliminar de la memoria caché los datos que han estado allí por más tiempo.

Paquete DBMS_RESULT_CACHE

Paquete suministrado que ofrece un conjunto de subprogramas para gestionar el contenido de la memoria caché. Este paquete será de interés principalmente para los administradores de bases de datos.

Vistas dinámicas de rendimiento

V\$RESULT_CACHE_STATISTICS

Muestra varias configuraciones de caché de resultados y estadísticas de uso, incluido el tamaño del bloque y la cantidad de resultados de caché creados con éxito.

V\$RESULT_CACHE_OBJECTS

Muestra todos los objetos para los que se han almacenado en caché los resultados

V\$RESULT_CACHE_MEMORY

Muestra todos los bloques de memoria y su estado, vinculándolos a la vista V%RESULT_CACHE_OBJECTS a través de la columna object_id

V\$RESULT_CACHE_DEPENDENCY

Muestra la relación de dependencia entre los resultados almacenados en caché y las dependencias.

Aquí hay un procedimiento que puede usar para mostrar dependencias:

```
/* Archivo en la web: show_frc_dependencies.sql */ CREAR O
REEMPLAZAR PROCEDIMIENTO show_frc_dependencies (
    nombre_como_en EN VARCHAR2)
ES
COMENZAR
    DBMS_OUTPUT.put_line ('Dependencias para "' || name_like_in || "'");
para grabación
    EN (SELECCIONE d.result_id
        /* Limpiar la visualización del nombre de la función */
        , TRADUCIR (SUBSTR (res.nombre, 1, INSTR (res.nombre, ':') - 1
            , 'A'', 'A')
        nombre de la función
        , dep.name depende_de
        DESDE v$result_cache_dependency d
        , v$result_cache_objects res
        , v$result_cache_objects dep
        DONDE res.id = d.result_id
        AND dep.id = d.depend_id AND
        res.name LIKE name_like_in)
    BUCLE
        /* No incluir la dependencia de uno mismo */
        IF rec.function_name <> rec.depends_on THEN
            DBMS_OUTPUT.put_line (
                rec.nombre_función || ' depende de ' || rec.depende_de); TERMINARA
            SI;
    FIN DEL BUCLE;
```

```
FIN;  
/
```

Dependencias detalladas en 11.2 y versiones posteriores

Una mejora significativa en 11.2 para la función de caché de resultados es el seguimiento detallado de dependencias. Oracle ahora recuerda específicamente de qué tablas depende cada conjunto de datos almacenados en caché (los valores del argumento IN y el resultado). En otras palabras, diferentes filas de datos almacenados en caché pueden tener diferentes conjuntos de tablas dependientes. Y cuando se confirman los cambios en una tabla, Oracle eliminará o eliminará solo los resultados de la memoria caché que dependen de esa tabla, no necesariamente la memoria caché completa (como habría sucedido en 11.1).

La mayoría de las funciones a las que aplica la función de caché de resultados tendrán las mismas dependencias de tabla, independientemente de los valores reales pasados para los parámetros formales. El siguiente es un ejemplo en el que las dependencias de la tabla pueden cambiar:

```
/* Archivo en la web: 11g_frc_dependencies.sql  
*/ CREATE TABLE tablea (col VARCHAR2 (2));
```

```
CREAR TABLA tableb (col VARCHAR2 (2));
```

```
COMENZAR
```

```
    INSERTAR EN LA TABLA DE VALORES ('a1');
```

```
    INSERTAR EN VALORES de tableb ('b1');
```

```
    COMPROMETERSE;
```

```
FIN;
```

```
/
```

```
CREAR O REEMPLAZAR FUNCIÓN dynamic_query (table_suffix_in VARCHAR2)
```

```
    DEVOLVER VARCHAR2
```

```
    RESULTADO_CACHE
```

```
ES
```

```
    l_return VARCHAR2 (2);
```

```
COMENZAR
```

```
    DBMS_OUTPUT.put_line ('SELECCIONAR DE la tabla' || table_suffix_in);
```

```
    EJECUTAR INMEDIATO 'seleccionar columna de la tabla' || table_suffix_in EN l_return;
```

```
    RETORNO l_retorno;
```

```
FIN;
```

```
/
```

Como puedes ver, si paso *a* para el parámetro de sufijo de tabla, la función obtendrá una fila de TABLEA, pero si paso *b*, entonces la función obtendrá una fila de TABLEB. Ahora supongamos que ejecuto el siguiente script, que usa el procedimiento definido en la sección anterior para mostrar los cambios en el caché de resultados y sus dependencias:

```
/* Archivo en la web: 11g_frc_dependencies.sql */  
DECLARAR  
    l_valor      VARCHAR2 (2);
```

```

COMENZAR
    _value := consulta_dinámica ('a'); show_frc_dependencies
    ('%DYNAMIC_QUERY%', 'Después de a(1)');

    _value := consulta_dinámica ('b'); show_frc_dependencies
    ('%DYNAMIC_QUERY%', 'Después de b(1)');

    ACTUALIZAR tablea SET col = 'a2';
    COMPROMETERSE;

    show_frc_dependencies ('%DYNAMIC_QUERY%', 'Después de cambiar a a2');

    _value := consulta_dinámica ('a'); show_frc_dependencies
    ('%DYNAMIC_QUERY%', 'Después de a(2)');

    _value := consulta_dinámica ('b'); show_frc_dependencies
    ('%DYNAMIC_QUERY%', 'Después de b(2)');

    ACTUALIZAR tableb SET col = 'b2';
    COMPROMETERSE;

    show_frc_dependencies ('%DYNAMIC_QUERY%', 'Después de cambiar a b2');
    FIN;
/

```

Aquí está el resultado de ejecutar el script anterior:

```

SELECCIONAR DE tablea
Después de a(1): Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEA

SELECCIONAR DE la tablab
Después de b(1): Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEB
HR.DYNAMIC_QUERY depende de HR.TABLEA

Después de cambiar a a2: Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEB

SELECCIONAR DE tablea
Después de a(2): Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEB
HR.DYNAMIC_QUERY depende de HR.TABLEA

Después de b(2): Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEB
HR.DYNAMIC_QUERY depende de HR.TABLEA

Después del cambio a b2: Dependencias para "%DYNAMIC_QUERY%"
HR.DYNAMIC_QUERY depende de HR.TABLEA

```

Como puede ver, incluso después de realizar (y confirmar) un cambio en una fila de una tabla, el resultado almacenado en caché para la función dynamic_query no se vacía por completo. En su lugar, solo se eliminan las filas que dependen de esa tabla específica.

La base de datos privada virtual y el almacenamiento en caché de resultados de funciones

Cuando usas el *base de datos privada virtual*, o VPD (también conocido como *seguridad a nivel de fila o control de acceso detallado*) en su aplicación, usted define *políticas de seguridad* para operaciones SQL en tablas. Luego, la base de datos de Oracle agrega automáticamente estas políticas en forma de predicados de la cláusula WHERE para restringir las filas que un usuario puede consultar o cambiar en una tabla en particular. Es imposible eludir estas políticas, ya que se aplican *adentro* la capa SQL, y son invisibles para el usuario. El resultado final: los usuarios conectados a dos esquemas diferentes pueden ejecutar lo que parece ser la misma consulta (como en SELECCIONAR apellido DE los empleados) y obtener resultados diferentes. Para obtener información detallada sobre la seguridad a nivel de fila, consulte [capítulo 23](#).

Por ahora, echemos un vistazo a un uso simplista del VPD y cómo puede generar datos incorrectos para los usuarios (todo el código de esta sección se puede encontrar en el *11g_frc_vpd.sql* archivo en el sitio web del libro). Supongamos que defino el siguiente paquete con dos funciones en mi esquema de aplicación de Recursos Humanos, una para devolver el apellido de un empleado para una identificación de empleado determinada y la otra para usar como política de seguridad de VPD:

```
/* Archivo en web: 11g_frc_vpd.sql */
PAQUETE emplu11g
ES
    FUNCIÓN last_name (employee_id_in EN empleados.employee_id%TYPE)
        RETURN empleados.apellido% TYPE
        result_cache;

    FUNCIÓN restringir_empleados (esquema_en VARCHAR2, NOMBRE_EN VARCHAR2)
        DEVOLVER VARCHAR2;
FIN emplu11g;

CUERPO DEL PAQUETE emplu11g
ES
    FUNCIÓN last_name (employee_id_in EN empleados.employee_id%TYPE)
        RETURN empleados.apellido% TYPE
        RESULT_CACHE RELIES_ON (empleados)
    ES
        onerow_rec     empleados%ROWTYPE;
        COMENZAR
            DBMS_OUTPUT.PUT_LINE ('Buscando apellido para ID de empleado'
                || empleado_id_en);
        SELECCIONE * EN onerow_rec
            DE empleados
            DONDE empleado_id = empleado_id_in;

        RETORNO onerow_rec.last_name;
FIN apellido;
```

```

FUNCIÓN restringir_empleados (esquema_en VARCHAR2, NOMBRE_EN VARCHAR2)
    DEVOLVER VARCHAR2
ES
COMENZAR
    DEVOLUCIÓN (USUARIO DEL CASO
        CUANDO 'HR' ENTONCES '1 = 1' DE LO
        CONTRARIO '1 = 2'
    FIN
);
FIN restringir_empleados;
FIN emplu11g;

```

La función restrict_employees establece de manera muy simple: si está conectado al esquema de recursos humanos, puede ver todas las filas en la tabla de empleados; de lo contrario, no puedes ver nada.

Luego asigno esta función como política de seguridad para todas las operaciones en la tabla de empleados:

```

COMENZAR
    DBMS_RLS.add_policy
        (objeto_esquema
            , nombre del objeto
            , Nombre de directiva
            , esquema_función
            , función_política
            , tipos_de_instrucciones
            , comprobación de actualización
        );
    FIN;

```

Luego doy al esquema SCOTT la capacidad de ejecutar este paquete y seleccionar de la tabla subyacente:

```

OTORGAR EJECUTAR EN emplu11g A scott /
OTORGAR SELECCIÓN DE empleados A scott /

```

Antes de ejecutar la función de caché de resultados, verifiquemos que la política de seguridad esté implementada y que afecte los datos que pueden ver HR y SCOTT.

Me conecto como HR y consulto desde la tabla de empleados con éxito:

```

SELECCIONA apellido
DE empleados
DONDE empleado_id = 198/
APELIDO
-----
OConnell

```

Ahora me conecto a SCOTT y ejecuto la misma consulta; nota la diferencia!

```
CONECTAR scott/ tiger@oracle11
SELECCIONAR apellido
DE hr.empleados
DONDE empleado_id = 198/ no
hay filas seleccionadas.
```

El VPD en el trabajo: cuando estoy conectado a SCOTT, no puedo ver las filas de datos que son visibles desde Recursos Humanos.

Ahora veamos qué sucede cuando ejecuto la misma consulta desde una función de caché de resultados propiedad de HR. Primero, me conecto como HR y ejecuto la función, luego muestro el nombre devuelto:

```
COMENZAR
DBMS_OUTPUT.put_line (emplu11g.last_name (198));FIN;/
Buscando el apellido del empleado ID 198 OConnell
```

Observe las dos líneas de salida:

1. Se muestra "Buscando apellido para el ID de empleado 198" porque se ejecutó la función.
2. Se muestra "OConnell" porque se encontró la fila de datos y se devolvió el apellido.

Ahora me conecto como SCOTT y ejecuto el mismo bloque de código. Dado que la función ejecuta un SELECT INTO que *debería* devolver filas, espero ver una excepción NO_DATA_FOUND no controlada. En cambio...

```
SQL>COMENZAR
2      DBMS_OUTPUT.put_line (hr.emplu11g.last_name (198));FIN;/
O'Connell
```

La función devuelve "OConnell" con éxito, pero observe que no se muestra el texto "Buscando...". Eso es porque el motor PL/SQL en realidad no ejecutó la función (y la llamada a DBMS_OUTPUT.PUT_LINE dentro de la función). Simplemente devolvió el apellido almacenado en caché.

Y este es precisamente el escenario que hace que el VPD sea una combinación tan peligrosa con la función de caché de resultados. Dado que la función se llamó por primera vez con el valor de entrada de 198 de HR, el apellido asociado con esa ID se almacenó en caché para su uso en todas las demás sesiones conectadas a esta misma instancia. Por lo tanto, un usuario conectado a SCOTT ve datos que no debería ver.

Para verificar que la función realmente debería devolver NO_DATA_FOUND si el almacenamiento en caché no estuviera en su lugar, ahora conectémonos a HR e invalidemos el caché al realizar un cambio en la tabla de empleados (cualquier cambio servirá):

```
COMENZAR
/* Todos nosotros, los empleados que no son directores ejecutivos, merecemos un aumento del 50 %, ¿no es así? */
```

```
ACTUALIZAR empleados SET salario = salario * 1.5;
COMPROMETER;FIN;/
```

Ahora, cuando me conecto a SCOTT y ejecuto la función, obtengo una excepción NO_DATA_FOUND no controlada:

```
SQL>COMENZAR
 2   DBMS_OUTPUT.put_line (hr.emplu11g.last_name (198));FIN;/
ORA-01403: no se encontraron datos ORA-06512: en "HR.EMPLU11G", línea 10
ORA-06512: en la línea 3
```

Por lo tanto, si está trabajando en una de esas aplicaciones relativamente raras que se basan en la base de datos privada virtual, tenga mucho cuidado al definir funciones que usen la caché de resultados de funciones.

Resumen de almacenamiento en caché

Si un valor no ha cambiado desde la última vez que se solicitó, debe buscar formas de minimizar el tiempo que lleva recuperar ese valor. Como ha demostrado durante años el SGA de la arquitectura de base de datos de Oracle, el almacenamiento en caché de datos es una tecnología crítica cuando se trata de optimizar el rendimiento. Podemos aprender del almacenamiento en caché transparente de SGA de cursores, bloques de datos y más, para crear nuestros propios cachés o aprovechar los cachés SGA no transparentes (lo que significa que necesitamos cambiar nuestro código de alguna manera para aprovecharlos).

Aquí resumo brevemente las recomendaciones que he hecho para el almacenamiento en caché de datos. Las opciones incluyen:

Almacenamiento en caché basado en paquetes

Cree un caché a nivel de paquete, probablemente una colección, que almacenará los datos recuperados previamente y los pondrá a disposición de la memoria de PGA mucho más rápido de lo que se puede recuperar de SGA. Hay dos inconvenientes principales en este caché: se copia para cada sesión conectada a la base de datos de Oracle y no puede actualizar el caché si una sesión realiza cambios en las tablas de las que se extraen los datos almacenados en caché.

Almacenamiento en caché de funciones deterministas

En su caso, define las funciones como DETERMINISTAS. Especificar esta palabra clave provocará el almacenamiento en caché de las entradas de la función y el valor de retorno dentro del alcance de la ejecución de una consulta SQL única.

Caché de resultados de funciones

Utilice la función de caché de resultados (Oracle Database 11g y posteriores) cada vez que solicita datos de una tabla que se consulta con mucha más frecuencia de lo que se cambia. Este enfoque declarativo del almacenamiento en caché basado en funciones es casi tan rápido como la caché a nivel de paquete. Se comparte en todas las sesiones conectadas a la instancia y se puede invalidar automáticamente cada vez que se realiza un cambio en la(s) tabla(s) de las que se extraen los datos almacenados en caché.

Procesamiento masivo para la ejecución repetida de sentencias SQL

Oracle introdujo una mejora significativa en las capacidades relacionadas con SQL de PL/SQL con la instrucción FORALL y la cláusula BULK COLLECT para consultas. Juntos, estos se conocen como *procesamiento a granel* para PL/SQL. ¿Por qué, se preguntarán, sería esto necesario? Todos sabemos que PL/SQL está estrechamente integrado con el motor SQL subyacente en la base de datos de Oracle. PL/SQL es el lenguaje de programación de base de datos elegido para Oracle, aunque ahora también puede usar Java dentro de la base de datos.

Pero esta estrecha integración no significa que no haya gastos generales asociados con la ejecución de SQL desde un programa PL/SQL. Cuando el motor de tiempo de ejecución PL/SQL procesa un bloque de código, ejecuta las declaraciones de procedimiento dentro de su propio motor, pero pasa las declaraciones SQL al motor SQL. La capa SQL ejecuta las sentencias SQL y luego devuelve información al motor PL/SQL, si es necesario.

Esta transferencia de control (mostrada en [Figura 21-2](#)) entre los motores PL/SQL y SQL se denomina *cambio de contexto*. Cada vez que ocurre un cambio, hay una sobrecarga adicional. Hay una serie de escenarios en los que se producen muchos cambios y se degrada el rendimiento. Como puede ver, PL/SQL y SQL pueden estar estrechamente integrados en el nivel sintáctico, pero "debajo de las cubiertas" la integración no es tan estrecha como podría ser.

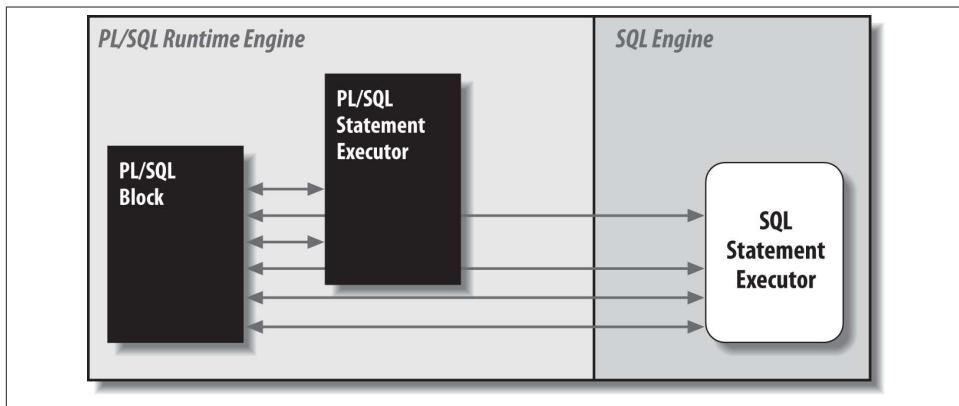


Figura 21-2. Cambio de contexto entre PL/SQL y SQL

Sin embargo, con FORALL y BULK COLLECT, puede ajustar la forma en que estos dos motores se comunican, indicando efectivamente al motor PL/SQL que comprima múltiples cambios de contexto en un solo interruptor, mejorando así el rendimiento de sus aplicaciones.

Considere la instrucción FORALL que se muestra en la figura. En lugar de usar un bucle FOR de cursor o un bucle numérico para recorrer las filas que se van a actualizar, utilizo un encabezado FORALL para especificar un número total de iteraciones para la ejecución. En tiempo de ejecución, el motor PL/SQL

toma la declaración de ACTUALIZACIÓN de "plantilla" y la colección de valores de variables de enlace y los genera en un conjunto de declaraciones, que luego se pasan al motor SQL con un solo cambio de contexto. En otras palabras, se ejecutan las mismas instrucciones SQL, pero todas se ejecutan en el mismo viaje de ida y vuelta a la capa SQL, lo que minimiza los cambios de contexto. Esto se muestra en [Figura 21-3](#).

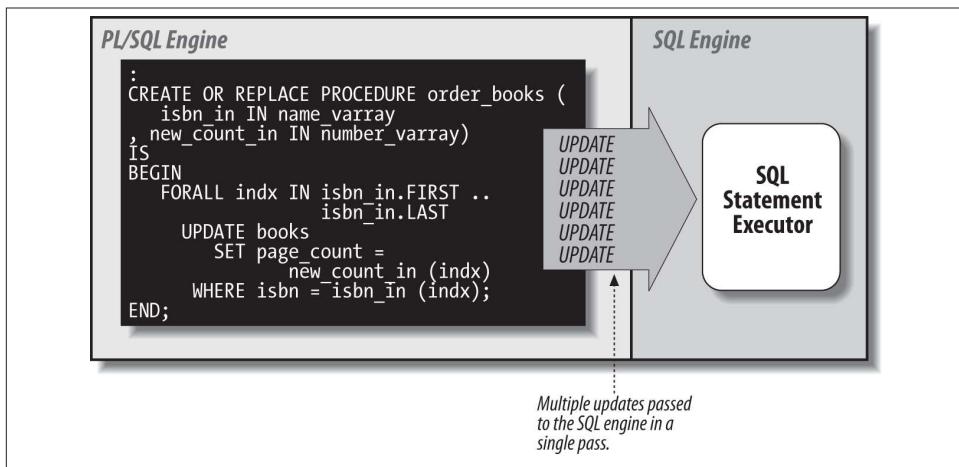


Figura 21-3. Un cambio de contexto con FORALL

Esta reducción en los cambios de contexto conduce a una reducción sorprendentemente pronunciada en el tiempo transcurrido para las sentencias SQL de varias filas ejecutadas en PL/SQL. Echemos un vistazo más de cerca a BULK COLLECT y FORALL.

Consultas de alta velocidad con BULK COLLECT

Con BULK COLLECT puede recuperar múltiples filas de datos a través de un cursor implícito o explícito con un solo viaje de ida y vuelta hacia y desde la base de datos. BULK COLLECT reduce la cantidad de cambios de contexto entre los motores PL/SQL y SQL y, por lo tanto, reduce la sobrecarga de recuperación de datos.

Eche un vistazo al siguiente fragmento de código. Necesito recuperar cientos de filas de datos sobre automóviles que tienen un historial ambiental deficiente. Coloco esos datos en un conjunto de colecciones para poder manipular fácil y rápidamente los datos tanto para el análisis como para los informes:

```

DECLARAR
  TYPE names_t ES LA TABLA DE transporte.name%TYPE; TIPO
  mileage_t ES TABLA DE transporte.kilometraje %TYPE; nombres
  nombres_t := nombres_t();
  kilometraje kilometraje_t := kilometraje_t();
  
```

```
CURSOR major_polluters_cur ES
```

```
    SELECCIONE el nombre, el kilometraje DESDE el transporte  
    WHERE transport_type = 'AUTOMOBILE' AND kilometraje < 20;
```

```
COMENZAR  
    PARA bad_car EN major_polluters_cur  
    BUCLE  
        nombres.EXTEND;  
        nombres (major_polluters_cur%ROWCOUNT) := bad_car.NAME;  
        kilometraje.EXTEND;  
        millas (major_polluters_cur%ROWCOUNT) := bad_car.mileage; FIN DEL  
        BUCLE;  
    -- Ahora trabaja con datos en las colecciones.  
FIN;
```

Esto ciertamente hace el trabajo, pero el trabajo puede tardar mucho tiempo en completarse. Considere esto: si la tabla de transporte contiene 2000 vehículos, entonces el motor PL/SQL emite 2000 recuperaciones individuales contra el cursor en el SGA.

Para ayudar en este escenario, use la cláusula BULK COLLECT en el elemento INTO de su consulta. Al usar esta cláusula en su cursor (explícito o implícito), le dice al motor SQL que vincule de forma masiva la salida de las múltiples filas obtenidas por la consulta en las colecciones especificadas antes de devolver el control al motor PL/SQL. La sintaxis de esta cláusula es:

```
... RECOGIDA A GRANEL EN nombre_de_la_colección[,nombre_de_la_colección] ...
```

dónde *nombre_de_la_colección* identifica una colección.

Aquí hay algunas reglas y restricciones a tener en cuenta al usar BULK COLLECT:

- Puede usar BULK COLLECT con SQL dinámico y estático.
- Puede utilizar palabras clave BULK COLLECT en cualquiera de las siguientes cláusulas: SELECT INTO, FETCH INTO y RETURNING INTO.
- El motor SQL inicializa y amplía automáticamente las colecciones a las que hace referencia en la cláusula BULK COLLECT. Comienza a llenar las colecciones en el índice 1 e inserta elementos consecutivamente (densos).
- SELECCIONAR... RECOGIDA A GRANEL no generar NO_DATA_FOUND si no se encuentran filas. En su lugar, debe verificar el contenido de la colección para ver si hay algún dato dentro.
- Si la consulta no devuelve filas, el método COUNT de la colección devolverá 0.

Exploraremos estas reglas y la utilidad de BULK COLLECT a través de una serie de ejemplos. Primero, aquí hay una reescritura del ejemplo de los principales contaminadores usando BULK COLLECT:

```
DECLARAR  
    TYPE names_t ES LA TABLA DE transporte.name%TYPE; TIPO  
    mileage_t ES TABLA DE transporte.kilometraje %TYPE;
```

```

nombres nombres_t;
kilometraje kilometraje_t;

COMENZAR
SELECCIONE nombre, millas RECOGIDA A GRANEL EN nombres, millas
DESDE el transporte
WHERE tipo_transporte = 'AUTOMÓVIL'
Y kilometraje < 20;

/* Ahora trabaje con datos en las colecciones */
END;

```

Ahora puedo eliminar el código de inicialización y extensión de la implementación de búsqueda fila por fila.

No tengo que depender de cursos implícitos para hacer este trabajo. Aquí hay otra reelaboración del ejemplo de los principales contaminadores, conservando el cursor explícito:

```

DECLARAR
TYPE names_t ES LA TABLA DE transporte.name%TYPE; TIPO
mileage_t ES TABLA DE transporte.kilometraje %TYPE; nombres
nombres_t;
kilometraje kilometraje_t;

CURSOR major_polluters_cur ES
SELECCIONE el nombre, el kilometraje DESDE el transporte
WHERE transport_type = 'AUTOMOBILE' AND kilometraje < 20;

COMENZAR
ABIERTO major_polluters_cur;
FETCH major_polluters_cur RECOGER A GRANEL EN nombres, millas;
CERRAR major_polluters_cur;
...
FIN;

```

También puedo simplificar mi vida y mi código buscando en una colección de registros, como puede ver aquí:

```

DECLARAR
TIPO transporte_aat ES TABLA DE transporte%ROWTYPE
ÍNDICE POR PLS_INTEGER;
I_transporte transporte_aat; COMENZAR

SELECCIONE * RECOGIDA A GRANEL EN I_transportation
DESDE el transporte
WHERE tipo_transporte = 'AUTOMÓVIL'
Y kilometraje < 20;

-- Ahora trabaja con datos en las colecciones.
FIN;

```



En base de datos Oracle 10g *gramo* y más adelante, el compilador PL/SQL optimizará automáticamente la mayoría de los bucles FOR del cursor para que se ejecuten con un rendimiento comparable al de BULK COLLECT. Tú haces *no* necesitas transformar explícitamente este código usted mismo, a menos que el cuerpo de su ciclo ejecute, directa o indirectamente, instrucciones DML. La base de datos no optimiza las declaraciones DML en FORALL, por lo que deberá convertir explícitamente el bucle FOR del cursor para usar BULK COLLECT. Luego puede usar las colecciones pobladas por BULK COLLECT para "impulsar" la declaración FORALL.

Limitación de filas recuperadas con BULK COLLECT

Oracle proporciona una cláusula LIMIT para BULKCOLLECT que le permite limitar la cantidad de filas obtenidas de la base de datos. La sintaxis es:

BUSCAR *cursor* RECOGIDA A GRANEL EN... [LÍMITE *filas*];

dónde *filas* puede ser cualquier literal, variable o expresión que se evalúe como un número entero (de lo contrario, la base de datos generará una excepción VALUE_ERROR).

LIMIT es muy útil con BULK COLLECT porque te ayuda a administrar la cantidad de memoria que usará tu programa para procesar datos. Suponga, por ejemplo, que necesita consultar y procesar 10.000 filas de datos. Tú *podrías* usar BULK COLLECT para recuperar todas esas filas y completar una colección bastante grande. Sin embargo, este enfoque consumirá mucha memoria en el PGA para esa sesión. Si este código es ejecutado por muchos esquemas de Oracle separados, el rendimiento de su aplicación puede degradarse debido al intercambio de PGA.

El siguiente bloque de código usa la cláusula LIMIT en un FETCH que está dentro de un ciclo simple:

```
DECLARAR
    CURSOR allrows_cur ES SELECCIONAR * DE empleados;
    TIPO employee_aat ES TABLA DE allrows_cur%ROWTYPE
        ÍNDICE POR BINARIO_INTEGER;
    l_empleados empleado_aat;
    COMENZAR
        ABRIR allrows_cur;
    BUCLE
        FETCH allrows_cur RECOGIDA A GRANEL EN l_employees LIMIT 100;

        /* Procesar los datos escaneando la colección. */ PARA l_row EN
        1 .. l_employees.COUNT
    BUCLE
        upgrade_employee_status (l_employees(l_row).employee_id); FIN DEL
    BUCLE;

    SALIR CUANDO allrows_cur%NO FOUND; FIN
DEL BUCLE;
```

```
CERRAR allrows_cur;
FIN;
```

Observe que termino el bucle comprobando el valor de allrows_cur%NOTFOUND en la parte inferior del bucle. Cuando consulto datos una fila a la vez, generalmente coloco este código inmediatamente después de la instrucción FETCH. Deberíanohágalo cuando use BULK COLLECT, porque cuando la búsqueda recupere el último conjunto de filas, el cursor se agotará (y %NOTFOUND devolverá TRUE), pero aún tendrá algunos elementos en la colección para procesar.

Por lo tanto, verifique el atributo %NOTFOUND en elabajode su ciclo, o verifique el contenido de la colección inmediatamente después de la obtención:

```
BUCLE
  FETCH allrows_cur RECOGIDA A GRANEL EN l_employees LIMIT 100; SALIR
  CUANDO l_employees.COUNT = 0;
```

La desventaja de este segundo enfoque es que realizará una obtención adicional que no devuelve filas, en comparación con la comprobación de %NOTFOUND en la parte inferior del cuerpo del ciclo.



A partir de Oracle Database 12C, también puede usar la cláusula FIRST ROWS para limitar el número de filas obtenidas con BULK COLLECT. El siguiente bloque de código usa la cláusula LIMIT en un FETCH que está dentro de un bucle simple. Este código recuperará solo las primeras 50 filas identificadas por la instrucción SELECT:

```
DECLARAR
  TIPO sueldos_t ES TABLA DE empleados.salario%TIPO;
  l_sueldos sueldos_t;
COMENZAR
  SELECCIONE el salario COLECTA A GRANEL EN las ventas DE los empleados
    OBTENER LAS PRIMERAS 50 FILAS SOLAMENTE;
  FIN;
  /
```

Obtención masiva de varias columnas

Como ha visto en ejemplos anteriores, ciertamente puede obtener de forma masiva el contenido de más de una columna. Sería más elegante si pudiera obtener esas columnas múltiples en una sola colección de registros. De hecho, Oracle puso a disposición esta función a partir de Oracle9/Versión 2 de la base de datos.

Suponga que me gustaría recuperar toda la información de mi tabla de transporte para cada vehículo cuyo kilometraje sea inferior a 20 millas por galón. Puedo hacerlo con un mínimo de codificación:

```
DECLARAR
  -- Declarar el tipo de colección
  TIPO VehTab ES TABLA DE transporte%ROWTYPE;
  -- Instanciar una colección particular del TIPO
```

```

gas_guzzlers VehTab;
COMENZAR
SELECCIONAR *
RECOGIDA A GRANEL EN
gas_guzzlers DESDE transporte
DONDE kilometraje <
20; ...

```

Antes de Oracle9/Versión 2 de la base de datos, el código anterior generaría esta excepción:

PLS-00597: la expresión 'GAS_GUZZLERS' en la lista INTO es de tipo incorrecto

Puede usar la cláusula LIMIT con BULK COLLECT en una colección de registros, tal como lo haría con cualquier otra instrucción BULK COLLECT.

Uso de la cláusula RETURNING con operaciones masivas

Ahora ha visto que BULK COLLECT se usa para cursos de consulta tanto implícitos como explícitos. También puede usar BULK COLLECT dentro de una declaración FORALL, para aprovechar la cláusula RETURNING.

La cláusula RETURNING le permite obtener información (como un valor actualizado recientemente para un salario) de una declaración DML. REGRESAR puede ayudarlo a evitar consultas adicionales a la base de datos para determinar los resultados de las operaciones DML que acaban de completarse.

Suponga que el Congreso ha aprobado una ley que requiere que una empresa pague a su empleado mejor pagado no más de 50 veces el salario de su empleado peor pagado. Trabajo en el departamento de TI de la empresa recién fusionada Northrop-Ford-Mattel-Yahoo-ATT, que emplea a un total de 250.000 trabajadores. La palabra ha llegado desde arriba: el CEO no aceptará un recorte salarial, por lo que necesito aumentar los salarios de todos los que ganen menos de 50 veces su paquete de compensación total de 2013 de \$145 millones y disminuir los salarios de toda la alta gerencia excepto para el CEO. Después de todo, ¡alguien tiene que compensar esta pérdida de beneficios!

¡Guau! Tengo muchas actualizaciones que hacer y quiero usar FORALL para hacer el trabajo lo más rápido posible. Sin embargo, también necesito realizar varios tipos de procesamiento en los datos de los empleados y luego imprimir un informe que muestre el cambio en el salario de cada empleado afectado. Esa cláusula de RETORNO sería muy útil aquí, así que intentémoslo. (Ver *elsolojusto.sql* archivo en el sitio web del libro para todos los pasos que se muestran aquí, además de la creación de tablas y declaraciones INSERT.)

Primero, crearé una función reutilizable para devolver la compensación de un ejecutivo:

```

/* Archivo en la web: onlyfair.sql */
FUNCTION salforexec (title_in EN VARCHAR2) EL NÚMERO DE DEVOLUCIÓN ES

CURSOR ceo_compensation ES
SELECCIONE salario + bonificación + stock_options +
mercedes_benz_allowance + yate_allowance

```

```

        DE compensación
        DONDE título = título_en;
        big_bucks NÚMERO;
COMENZAR
        ABIERTO ceo_compensation;
        FETCH ceo_compensation EN big_bucks;
        RETORNO big_bucks;
FIN;

```

En el bloque principal del programa de actualización, declaro una serie de variables locales y la siguiente consulta para identificar empleados mal pagados y empleados pagados en exceso que no tienen la suerte de ser el CEO:

```

DECLARAR
    big_bucks NÚMERO := salforexec ('CEO');
    min_sal NÚMERO := big_bucks / 50; nombres
    name_tab;
    old_salaries number_tab;
    new_salaries number_tab;

CURSOR empleados_afectados (director ejecutivo EN NÚMERO) ES

    SELECCIONE nombre, salario + bono old_salary
        DE compensación
        DONDE título != 'CEO'
            Y ((salario + bono < ceosal / 50)
            O (salario + bono > ceosal / 10));

```

Al comienzo de mi sección ejecutable, cargo todos estos datos en mis colecciones con una consulta BULK COLLECT:

```

ABIERTO afectados_empleados (big_bucks);
FETCH empleados_afectados
RECOGIDA A GRANEL EN nombres, old_salaries;

```

Entonces puedo usar la colección de nombres en mi actualización de FORALL:

```

FORALL indx IN nombres.PRIMERO .. nombres.L*
    ACTUALIZAR compensación
        salario fijo =
        MAYOR(
            DECODIFICAR (
                MAYOR (min_sal, salario),
                min_sal, min_sal,
                salario / 5),
                min_sal )
    DONDE nombre = nombres(indx)
    SALARIO DE DEVOLUCIÓN COLECCIÓN A GRANEL EN new_salaries;

```

Utilizo DECODE para darle a un empleado un gran impulso en los ingresos anuales o un recorte del 80 % en el salario para que el director general se sienta cómodo. Lo termino con una cláusula de RETORNO que se basa en BULK COLLECT para completar una tercera colección: los nuevos salarios.

Finalmente, debido a que usé RETURNING y no tengo que escribir otra consulta contra la tabla de compensación para obtener los nuevos salarios, puedo pasar inmediatamente a la generación de informes:

```
FOR indx IN nombres.PRIMERO .. nombres.ÚLTIMO
BUCLE
  DBMS_SALIDA.PUT_LINE (
    RPAD (nombres (índice), 20) ||
    RPAD (' Antiguo: ' || old_salaries(indx), 15) ||
    Nuevo: ' || nuevos_salarios(índx)
  );
FIN DEL BUCLE;
```

Aquí, entonces, está el informe generado a partir de *lasolojusto.sql*/guion:

John día y noche	Antiguo: 10500	Nuevo: 2900000
Cubículo de acebo	Antiguo: 52000	Nuevo: 2900000
Sandra Watchthebucks	Antiguo: 22000000	Nuevo: 4000000

Ahora todos pueden pagar una vivienda y atención médica de calidad. Y aumentarán los ingresos fiscales en todos los niveles, para que las escuelas públicas puedan obtener los fondos que necesitan.



Los valores de la columna RETURNING o las expresiones devueltas por cada ejecución en FORALL se agregan a la colección después de los valores devueltos anteriormente. Si usa RETURNING dentro de un bucle FOR no masivo, la última ejecución de DML sobrescribe los valores anteriores.

DML de alta velocidad con FORALL

BULK COLLECT acelera las consultas. FORALL hace lo mismo para inserciones, actualizaciones, eliminaciones y fusiones (FORALL con una fusión solo se admite en Oracle Database 11g *romo* y después); Me referiré a estas declaraciones colectivamente como "DML". FORALL le dice al motor de tiempo de ejecución de PL/SQL que vincule de forma masiva en la declaración SQL todos los elementos de una o más colecciones antes de enviar sus declaraciones al motor SQL.

Dada la centralidad de SQL para las aplicaciones basadas en Oracle y el fuerte impacto de las declaraciones DML en el rendimiento general, FORALL es probablemente la función de optimización más importante en el lenguaje PL/SQL.

Entonces, si aún no está usando FORALL, tengo malas y buenas noticias. La mala noticia es que el código base de su aplicación no se ha mejorado a lo largo de los años para aprovechar las funciones críticas de Oracle. La buena noticia es que cuando empiece a usarlo, sus usuarios experimentarán algunos aumentos de rendimiento muy agradables (y relativamente fáciles de lograr).

En las siguientes páginas encontrará explicaciones de todas las características y matices de FORALL, junto con muchos ejemplos.

Sintaxis de la instrucción FORALL

Aunque la declaración FORALL contiene un esquema de iteración (es decir, recorre todas las filas de una colección), no es un bucle FOR. En consecuencia, no tiene sentencia LOOP ni END LOOP. Su sintaxis es la siguiente:

```
PARA TODOSíndiceEN  
[límite_inferior ... límite_superior]  
ÍNDICES DEcolección_indexación|  
VALORES DEcolección_indexación  
]  
[GUARDAR EXCEPCIONES]  
declaración_sq;
```

dónde:

índice

Es un número entero, declarado implícitamente por Oracle, que es un valor de índice definido en la colección.

límite_inferior

Es el valor de índice inicial (fila o elemento de colección) para la operación

límite_superior

Es el valor de índice final (fila o elemento de colección) para la operación

declaración_sq;

¿Se debe realizar la instrucción SQL en cada elemento de la colección?

colección_indexación

¿Se utiliza la colección PL/SQL para seleccionar los índices en la matriz de vinculación a la que se hace referencia en el *declaración_sq*; las alternativas INDICES_OF y VALUES_OF están disponibles a partir de Oracle Database 10g

SALVAR EXCEPCIONES

Es una cláusula opcional que le dice a FORALL que procese todas las filas, guardando las excepciones que ocurran.

Debe seguir estas reglas al usar FORALL:

- El cuerpo de la declaración FORALL debe ser una sola declaración DML: INSERTAR, ACTUALIZAR, ELIMINAR o COMBINAR (en Oracle Database 11g *o* después).
- La declaración DML debe hacer referencia a los elementos de la colección, indexados por el *index_rowvariable* en la instrucción FORALL. El alcance de la *índice_fila* variable es solo la instrucción FORALL; no puede hacer referencia a él fuera de esa declaración. Tenga en cuenta, sin embargo, que los límites superior e inferior de estas colecciones no tienen que abarcar todo el contenido de la(s) colección(es).
- No declarar una variable para *índice_fila*. El motor PL/SQL lo declara implícitamente como PLS_INTEGER.

- Los límites inferior y superior deben especificar un rango válido de números de índice consecutivos para las colecciones a las que se hace referencia en la instrucción SQL. Las colecciones escasamente llenas generarán el siguiente error:

ORA-22160: el elemento en el índice [3] no existe

Ver el *falta_elemento.sql* archivo en el sitio web del libro para ver un ejemplo de este escenario.

A partir de Oracle Database 10*gramo*, puede usar la sintaxis INDICES OF y VALUES OF para permitir el uso de colecciones dispersas (donde hay elementos indefinidos entre FIRST y LAST). Estas cláusulas se tratan más adelante en este capítulo.

- Hasta la base de datos Oracle 11*gramo*, los campos dentro de las colecciones de registros no podían ser referenciados dentro de la declaración DML. En cambio, solo podría hacer referencia a la fila en la colección como un todo, ya sea que los campos fueran colecciones de escalares o colecciones de objetos más complejos. Por ejemplo, el siguiente código:

```
DECLARAR
    TIPO empleado_aat ES TABLA DE empleados%ROWTYPE
        ÍNDICE POR PLS_INTEGER;
    I_empleados empleado_aat;
COMENZAR
    PARA TODOS I_Index EN I_empleados.PRIMERO .. I_empleados.ÚLTIMO
        INSERTAR EN empleado (employee_id, last_name)
            VALORES (I_empleados (I_index).employee_id
                    , I_empleados (I_index).apellido
                );
FIN;
```

provocará el siguiente error de compilación en versiones anteriores a Oracle Database 11*gramo*:

PLS-00436: restricción de implementación: no se puede hacer referencia a los campos de la tabla de registros BULK In-BIND

Para usar FORALL en este caso, deberá cargar las identificaciones de los empleados y los apellidos en dos colecciones separadas. Afortunadamente, esta restricción se eliminó en Oracle Database 11*gramo*.

- El subíndice de colección al que se hace referencia en la instrucción DML no puede ser una expresión. Por ejemplo, el siguiente guión:

```
DECLARAR
    nombres nombre_varray := nombre_varray ();
    COMENZAR
        FORALL indx IN nombres.PRIMERO .. nombres.APELLIDO
            ELIMINAR DESDE emp DONDE ename = nombres (indx+10);
    FIN;
```

provocará el siguiente error:

PLS-00430: La variable de iteración FORALL INDEX no está permitida en este contexto

Ejemplos de FORALL

Estos son algunos ejemplos del uso de la instrucción FORALL:

- Cambiar el recuento de páginas de todos los libros cuyos ISBN aparecen en la colección isbns_in:

```
PROCEDIMIENTO order_books (
    isbns_in IN nombre_varray,
    new_counts_in IN number_varray)
ES
COMENZAR
FORALL indx IN isbns_in.FIRST .. isbns_in.LAST
    ACTUALIZAR libros
        SET page_count = new_counts_in (indx)
        WHERE isbn = isbns_in (indx);
FIN;
```

Tenga en cuenta que los únicos cambios de un bucle FOR típico en este ejemplo son cambiar FOR a FORALL y eliminar las palabras clave LOOP y END LOOP. Este uso de FORALL accede y pasa a SQL cada una de las filas definidas en las dos colecciones. Referirse de nuevo a [Figura 21-3](#) por el cambio en el comportamiento que resulta.

- Hacer referencia a más de una colección de una declaración DML. En este caso, tengo tres colecciones: negación, nombre_paciente y enfermedades. Solo los dos primeros están subíndices, lo que da como resultado que los elementos individuales de estas colecciones se pasen a cada INSERCIÓN. La tercera columna en cobertura_salud es una colección que enumera las condiciones previas. Debido a que el motor PL/SQL vincula de forma masiva solo las colecciones con subíndices, la colección completa de enfermedades se coloca en esa columna para cada fila insertada:

```
FORALL indx IN negación.PRIMERO .. negación.ÚLTIMO
    INSERTAR EN salud_cobertura
        VALORES (negación(indx), nombre_paciente(indx), enfermedades);
```

- Utilice la cláusula RETURNING en una sentencia FORALL para recuperar información sobre cada sentencia DELETE por separado. Tenga en cuenta que la cláusula RETURNING en FORALL debe usar BULK COLLECT INTO (la operación "masiva" correspondiente para consultas):

```
FUNCIÓN remove_emps_by_dept (deptlist EN dlist_t)
    VOLVER enolist_t
ES
enolista enolista_t;
COMENZAR
FORALL aDept IN listadept.FIRST..deptlist.LAST
    ELIMINAR DE empleados DONDE id_departamento EN deptlist(aDept)
        DEVOLVIENDO employee_id RECOGIDA A GRANEL EN enolist;
    RETORNO enolista;
FIN;
```

- Use los índices definidos en una colección para determinar qué filas en la matriz de enlace (la colección a la que se hace referencia dentro de la instrucción SQL) se usarán en el INSERT dinámico:

```

FORALL indx EN ÍNDICES DE l_top_employees
EJECUTAR INMEDIATO
  'INSERTAR EN' || l_tabla || ' VALORES (:emp_pk, :new_salary)'
  UTILIZANDO l_new_salaries(indx).employee_id,
  l_nuevos_salarios(indx).salario;

```

Atributos de cursor para FORALL

Puede usar atributos de cursor después de ejecutar una instrucción FORALL para obtener información sobre la ejecución de la operación DML dentro de FORALL. Oracle también ofrece un atributo adicional, %BULK_ROWCOUNT, para brindarle información más granular sobre los resultados de la instrucción DML masiva.

Tabla 21-1 describe el significado de los valores devueltos por estos atributos para FORALL.

Cuadro 21-1. Atributos de cursor SQL implícitos para sentencias FORALL

Nombre	Descripción
SQL%ENCONTRADO	Devuelve TRUE si la última ejecución de la instrucción SQL modificó una o más filas. Devuelve VERDADERO si la declaración DML no pudo cambiar ninguna fila.
SQL%ROWCOUNT	Devuelve el número total de filas procesadas por todas las ejecuciones de la instrucción SQL, no solo la última instrucción.
SQL%ISOPEN	Siempre devuelve FALSO y no debe utilizarse.
SQL%BULK_ROWCOUNT	Devuelve una pseudocolección que le indica el número de filas procesadas por cada instrucción SQL correspondiente ejecutada a través de FORALL. Tenga en cuenta que cuando %BULK_ROWCOUNT(i)es cero, %FOUND y %NOTFOUND son FALSO y VERDADERO, respectivamente.
SQL%BULK_EXCEPCIONES	Devuelve una pseudocolección que proporciona información sobre cada excepción generada en una instrucción FORALL que incluye la cláusula SAVE EXCEPTIONS.

Exploraremos ahora el atributo compuesto %BULK_ROWCOUNT. Este atributo, diseñado específicamente para su uso con FORALL, tiene la semántica de (actúa como) una matriz o colección asociativa. La base de datos se deposita en el *norte* elemento en esta colección el número de filas procesadas por el *norte* ejecución de INSERT, UPDATE, DELETE o MERGE de FORALL. Si no hay filas afectadas, el *norte*La fila contendrá un valor cero.

Este es un ejemplo del uso de %BULK_ROWCOUNT (y también el atributo general %ROWCOUNT):

```

DECLARAR
  TIPO isbn_list ES TABLA DE VARCHAR2 (13);

  mis_libros isbn_list
  := lista_isbn (
    '1-56592-375-8', '0-596-00121-5', '1-56592-849-0',
    '1-56592-335-9', '1-56592-674-9', '1-56592-675-7',
    '0-596-00180-0', '1-56592-457-6'

```

```

);
COMENZAR
FORALL book_index IN
    mis_libros.PRIMERO..mis_libros.ÚLTIMA
    ACTUALIZACIÓN libros
        SET page_count = page_count / 2
        WHERE isbn = my_books(book_index);

-- ¿Actualicé el número total de libros que esperaba? SI
SQL%ROWCOUNT != 8

ENTONCES
    DBMS_SALIDA.PUT_LINE (
        '¡Nos falta un libro!');
TERMINARA SI;

-- ¿La cuarta instrucción UPDATE afectó a alguna fila? SI
SQL%BULK_ROWCOUNT(4) = 0

ENTONCES
    DBMS_SALIDA.PUT_LINE (
        '¿Qué pasó con la programación Oracle PL/SQL?');
TERMINARA SI;
FIN;

```

Aquí hay algunos consejos sobre cómo funciona este atributo:

- La instrucción FORALL y %BULK_ROWCOUNT utilizan los mismos subíndices o números de fila en las colecciones. Por ejemplo, si la colección pasada a FORALL tiene datos en las filas 10 a 200, entonces la pseudocolección %BULK_ROWCOUNT también tendrá filas 10 a 200 definidas y llenas. Cualquier otra fila será indefinida.
- Cuando INSERT afecta solo a una sola fila (cuando especifica una lista de VALORES, por ejemplo), el valor de una fila en %BULK_ROWCOUNT será igual a 1. Sin embargo, para declaraciones INSERT...SELECT, %BULK_ROWCOUNT puede ser mayor que 1.
- El valor en una fila del pseudoarreglo %BULK_ROWCOUNT para eliminaciones, actualizaciones y selecciones de inserción puede ser cualquier número natural (0 o positivo); estas sentencias pueden modificar más de una fila, dependiendo de sus cláusulas WHERE.

Comportamiento ROLLBACK con FORALL

La declaración FORALL le permite pasar múltiples declaraciones SQL juntas (en masa) al motor SQL. Esto significa que tiene un solo cambio de contexto, pero cada declaración aún se ejecuta por separado en el motor SQL.

¿Qué sucede cuando falla una de esas declaraciones DML?

1. La instrucción DML que generó la excepción se retrotrae a un punto de guardado implícito marcado por el motor PL/SQL antes de la ejecución de la instrucción. Los cambios en todas las filas ya modificadas por esa declaración se revierten.
2. Cualquier operación DML anterior en esa instrucción FORALL que ya se haya completado sin errores se retrotraído.
3. Si no realiza una acción especial (agregando la cláusula SAVE EXCEPTIONS a FORALL, que se analiza a continuación), la instrucción FORALL completa se detiene y las instrucciones restantes no se ejecutan en absoluto.

Excepciones pasadas continuas con SAVE EXCEPTIONS

Al agregar la cláusula SAVE EXCEPTIONS a su encabezado FORALL, le indica a la base de datos de Oracle que continúe procesando incluso cuando se haya producido un error. La base de datos luego "guardará la excepción" (o múltiples excepciones, si ocurre más de un error). Cuando se complete la instrucción DML, generará la excepción ORA-24381. En la sección de excepciones, puede acceder a una pseudocolección llamada SQL%BULK_EXCEPTIONS para obtener información de error.

Aquí hay un ejemplo, seguido de una explicación de lo que está sucediendo:

```

/* Archivo en web: bulkexc.sql */
1  DECLAR
2    errores_masivos      EXCEPCIÓN;
3    PRAGMA EXCEPTION_INIT (bulk_errors, -24381);
4    TIPO namelist_t ES TABLA DE VARCHAR2 (32767);
5
6    enames_con_errores    lista_de_nombres
7    := lista_nombres_t ('ABC',
8        'DEF',
9        NULL, /* El apellido no puede ser NULL */
10       'LITTLE',
11       RPAD ('BIGBIGGERBIGGEST', 250, 'ABC'), /* Valor demasiado largo */
12       'SMITHIE'
13    );
14  COMENZAR
15    FORALL indx IN enames_with_errors.FIRST .. enames_with_errors.LAST
diecisés
16      SALVAR EXCEPCIONES
17      ACTUALIZAR EMPLEADOS
18      SET last_name = enames_with_errors (indx);
19
20  EXCEPCIÓN
21    CUANDO errores masivos
22    ENTONCES
23      DBMS_OUTPUT.put_line ('Actualizado ' || SQL%ROWCOUNT || ' filas.');
24
25    FOR indx IN 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP
26
27      DBMS_OUTPUT.PUT_LINE ('Error'

```

```

28      || índice
29      || ' ocurrió durante ' ||
30      'iteración'
31      || SQL%BULK_EXCEPTIONS (índice).ERROR_INDEX ||
32      ' actualizando el nombre a '
33      || enames_with_errors (SQL%BULK_EXCEPTIONS (indx).ERROR_INDEX);
34      DBMS_OUTPUT.PUT_LINE ('El error de Oracle es'
35          || SQLERRM ( -1 * SQL%BULK_EXCEPTIONS (indx).ERROR_CODE ) );
36
37      FIN DEL BUCLE;
38 FIN;

```

Cuando ejecuto este código con SERVEROUTPUT activado, veo estos resultados:

SQL>**Excepciones masivas de EXEC**

Se produjo el error 1 durante la iteración 3 al actualizar el nombre a BIGBIGGERBIGGEST El error de Oracle es ORA-01407: no se puede actualizar () a NULL

El error 2 ocurrió durante la iteración 5 al actualizar el nombre a Oracle. El error es ORA-01401: el valor insertado es demasiado grande para la columna.

En otras palabras, la base de datos encontró dos excepciones al procesar el DML para la colección de nombres. No se detuvo en la primera excepción, sino que continuó catalogando una segunda.

La siguiente tabla describe la funcionalidad de manejo de errores en este código.

Líneas)	Descripción
2-3	Declare una excepción con nombre para que la sección de excepción sea más legible.
4-13	Declare y complete una colección que impulsará la instrucción FORALL. He colocado intencionalmente datos en la colección que generarán dos errores.
15-18	Ejecute una instrucción UPDATE con FORALL usando la colección enames_with_errors.
25-37	Utilice un bucle FOR numérico para examinar el contenido de la pseudocolección SQL%BULK_EXCEPTIONS. Tenga en cuenta que puedo llamar al método COUNT para determinar la cantidad de filas definidas (errores generados), pero no puedo llamar a otros métodos, como FIRST y LAST.
31 y 33	El campo ERROR_INDEX de la fila de cada pseudocolección devuelve el número de fila en la colección de control de la instrucción FORALL para la que se generó una excepción.
35	El campo ERROR_CODE de la fila de cada pseudocolección devuelve el número de error de la excepción que se generó. Tenga en cuenta que este valor se almacena como un entero positivo; deberá multiplicarlo por -1 antes de pasarlo a SQLERRM o mostrar la información.

Conducir FORALL con arreglos no secuenciales

Antes de Oracle Database 10*gramo*, la colección a la que se hace referencia dentro de la declaración FORALL (la "matriz de enlace") tenía que llenarse de forma densa o consecutiva. En un código como el siguiente, si hubiera espacios entre los valores alto y bajo especificados en el rango del encabezado FORALL, Oracle generaría un error:

```

1 DECLARAR
2     TIPO employee_aat ES TABLA DE empleados.employee_id%TYPE
3         ÍNDICE POR PLS_INTEGER;
4     l_empleados empleado_aat;
5     COMENZAR
6     l_empleados (1) := 100;
7     l_empleados (100) := 1000;
8     PARA TODOS l_index EN l_empleados.PRIMERO .. l_empleados.ÚLTIMO
9         ACTUALIZAR empleados SET salario = 10000 DONDE
10            employee_id = l_employees (l_index);
11    FIN;
12 /

```

El mensaje de error se veía así:

```

DECLARAR
*
ERROR en la línea 1:
ORA-22160: el elemento en el índice [2] no existe

```

Además, no había manera de que usted omitiera filas en la matriz de enlace que no deseaba que fueran procesadas por la instrucción FORALL. Estas restricciones a menudo llevaron a la escritura de código adicional para comprimir las colecciones y ajustarlas a las limitaciones de FORALL. Para ayudar a los desarrolladores de PL/SQL a evitar esta codificación molesta, comience con Oracle Database 10 g r a m o PL/SQL ofrece las cláusulas INDICES OF y VALUES OF, las cuales le permiten especificar la parte de la matriz de enlace que FORALL procesará.

Primero, revisemos la diferencia entre estas dos cláusulas, y luego exploraré ejemplos para demostrar su utilidad:

ÍNDICES DE

Use esta cláusula cuando tenga una colección (llamémosla la *matriz de indexación*) cuyas filas definidas especifican qué filas en la matriz de vinculación (referenciadas dentro de la instrucción DML de FOR-ALL) desea que se procesen. En otras palabras, si el elemento en la posición *norte* (también conocido como el número de fila) no está definido en la matriz de indexación, desea que la declaración FORALL ignore el elemento en la posición *norte* en la matriz de enlace.

VALORES DE

Utilice esta cláusula cuando tenga una colección de enteros (nuevamente, la matriz de indexación) cuyo contenido (el valor del elemento en una posición específica) identifica la posición en la matriz de vinculación que desea que sea procesada por la instrucción FORALL.

ÍNDICES DE ejemplo. Me gustaría actualizar los salarios de algunos empleados a \$10,000.

Actualmente, nadie tiene tal salario:

```

SQL>SELECCIONE employee_id DE empleados DONDE salario = 10000; ninguna
fila seleccionada

```

Luego escribo el siguiente programa:

```

/* Archivo en la web: 10g_indices_of.sql */
1  DECLARAR
2      TIPO employee_aat ES TABLA DE empleados.employee_id%TYPE
3          ÍNDICE POR PLS_INTEGER;
4
5      l_empleados          empleado_aat;
6
7      TIPO boolean_aat ES TABLA DE BOOLEAN
8          ÍNDICE POR PLS_INTEGER;
9
10     l_employee_indices    booleano_aat;
11
12 COMENZAR
13     l_empleados (1) := 7839;
14     l_empleados (100) := 7654;
15     l_empleados (500) := 7950;
16     --
17     l_employee_indices (1) := VERDADERO;
18     l_employee_indices (500) := VERDADERO;
19     l_employee_indices (799) := VERDADERO;
20
21 FORALL l_index EN ÍNDICES DE l_employee_indices
22     ENTRE 1 Y 500
23     ACTUALIZAR empleados23           SET salario = 10000
24     DONDE empleado_id = l_empleados (l_index);
25 FIN;

```

La siguiente tabla describe la lógica del programa.

Líneas)	Descripción
2-5	Defina una colección de números de identificación de empleados.
7-10	Defina una colección de valores booleanos.
12-14	Rellene (escasamente) tres filas (1, 100 y 500) en la colección de identificaciones de
16-18	empleados. Defina solo dos filas en la colección, 1 y 500.
20-24	En la instrucción FORALL, en lugar de especificar un rango de valores del PRIMERO al ÚLTIMO, simplemente especifique ÍNDICES OF l_employee_indices. También incluyo una cláusula BETWEEN opcional para restringir cuál de esos valores de índice se usará.

Después de ejecutar este código, consulté la tabla para ver que, de hecho, solo se actualizaron dos filas de la tabla: se omitió el empleado con ID 7654 porque la colección de índices booleanos no tenía ningún elemento definido en la posición 100:

```
SQL>SELECCIONE employee_id DESDE empleado DONDE salario = 10000;
```

```
ID DE EMPLEADO
-----
7839
7950
```

Con ÍNDICES DE (línea 20), el *contenido* de la matriz de indexación se ignoran. Todo lo que importa son las posiciones o los números de fila que se definen en la colección.

VALORES DE ejemplo. Nuevamente, me gustaría actualizar los salarios de algunos empleados a \$10,000, esta vez usando la cláusula VALUES OF. Actualmente, nadie tiene tal salario:

```
SQL>SELECCIONE employee_id DESDE empleado DONDE salario = 10000;
ninguna fila seleccionada
```

Luego escribo el siguiente programa:

```
/* Archivo en la web: 10g_values_of.sql */
1  DECLAR
2      TIPO employee_aat ES TABLA DE empleados.employee_id%TYPE
3          ÍNDICE POR PLS_INTEGER;
4
5      l_empleados          empleado_aat;
6
7      TIPO indices_aat ES TABLA DE PLS_INTEGER
8          ÍNDICE POR PLS_INTEGER;
9
10     l_employee_indices    índices_aat;
11
12     COMENZAR
13         l_empleados (-77) := 7820;
14         l_empleados (13067) := 7799;
15         l_empleados (99999999) := 7369;
16         --
17         l_empleado_índices (100) := -77;
18         l_employee_indices (200) := 99999999;
19
20     PARA TODOS l_index EN VALORES DE l_employee_indices
21         ACTUALIZAR empleados
22             SET salario = 10000
23             DONDE empleado_id = l_empleados (l_index);
24     FIN;
```

La siguiente tabla describe la lógica del programa.

Líneas)	Descripción
2-6	Defina una colección de números de identificación de empleados.
7-10	Defina una colección de enteros.
12-14	Rellene (escasamente) tres filas (-77, 13067 y 99999999) en la colección de ID de empleados.
16-17	Quiero configurar la matriz de indexación para identificar cuál de esas filas usar en mi actualización. Debido a que estoy usando VALORES DE, los números de fila que uso no son importantes. En cambio, lo que importa es el valor que se encuentra en cada una de las filas de la matriz de indexación. Una vez más, quiero saltarme esa fila "central" de 13067, así que aquí defino solo dos filas en la matriz l_employee_indices y les asigno los valores -77 y 99999999, respectivamente.
19-22	En lugar de especificar un rango de valores del PRIMERO al ÚLTIMO, simplemente especifíco VALORES DE l_employee_indices. Observe que relleno las filas 100 y 200 en la colección de índices. VALORES DE hacen requieren una colección de indexación densamente llena.

Después de ejecutar este código, consulto la tabla para ver que nuevamente solo se actualizaron dos filas de la tabla: el empleado con ID 7799 se omitió porque la colección "valores de" no tenía ningún elemento cuyo valor fuera igual a 13067:

```
SQL>SELECCIONE employee_id DE empleados DONDE salario = 10000;
```

```
ID DE EMPLEADO
```

```
-----
```

```
7369
```

```
7820
```

Mejora del rendimiento con funciones de tabla canalizadas

Las funciones canalizadas son donde la elegancia y la simplicidad de PL/SQL convergen con el rendimiento de SQL. Las transformaciones de datos complejos son fáciles de desarrollar y respaldar con PL/SQL, sin embargo, para lograr un procesamiento de datos de alto rendimiento, a menudo recurrimos a soluciones SQL basadas en conjuntos. Las funciones canalizadas cierran la brecha entre los dos métodos sin esfuerzo, pero también tienen algunas características de rendimiento únicas propias, lo que las convierte en una excelente herramienta de optimización del rendimiento.

En las siguientes páginas, mostraré algunos ejemplos de requisitos típicos de procesamiento de datos y cómo puede ajustarlos con funciones canalizadas. Cubriré los siguientes temas:

- Cómo ajustar los requisitos típicos de carga de datos con funciones segmentadas. En cada caso, convertiré las soluciones heredadas basadas en filas en soluciones basadas en conjuntos que incluyen funciones canalizadas paralelas.
- Cómo explotar el contexto paralelo de las funciones segmentadas para mejorar el rendimiento de las descargas de datos.
- El rendimiento relativo de las opciones de partición y transmisión para funciones canalizadas paralelas.
- Cómo el optimizador basado en costos (CBO) trata las funciones de tabla estándar y canalizadas.
- Cómo se pueden resolver los requisitos complejos de carga de varias tablas con funciones canalizadas de varios tipos.

La sintaxis básica para las funciones de tabla canalizadas se cubrió en [capítulo 17](#). En resumen, se llama a una función canalizada en la cláusula FROM de una instrucción SQL y se consulta como si fuera una tabla relacional u otro origen de fila. A diferencia de las funciones de tabla estándar (que tienen que completar todo su procesamiento antes de pasar una colección de datos potencialmente grande al contexto de llamada), las funciones de tabla canalizadas transmiten sus resultados al cliente casi tan pronto como están preparados. En otras palabras, las funciones segmentadas no materializan todo su conjunto de resultados, y esta característica de optimización reduce drásticamente su huella de memoria PGA. Otra característica de rendimiento única de las funciones canalizadas es la capacidad de llamarlas en el contexto de una consulta paralela. Me he aprovechado de estas características de rendimiento únicas muchas veces,

Reemplazo de inserciones basadas en filas con cargas basadas en funciones canalizadas

Para demostrar el rendimiento de las funciones canalizadas, primero imaginemos un escenario de carga heredado típico que quiero llevar al siglo XXI. Usando el ejemplo de stockpivot, he codificado una carga simple fila por fila para obtener los datos de origen de stockpivot y pivotar cada registro en dos filas para la inserción. Está contenido en un paquete y es el siguiente:

```
/* Archivo en la web: stockpivot_setup.sql */
PROCEDIMIENTO load_stocks_legacy ES

CURSOR c_source_data ES
  SELECT ticker, open_price, close_price, trade_date FROM
    stocktable;

r_source_data stockpivot_pkg.stocktable_rt;
r_target_data stockpivot_pkg.tickertable_rt;

COMENZAR
  ABIERTO c_fuente_datos;
  BUCLE
    FETCH c_source_data EN r_source_data; SALIR
    CUANDO c_source_data%NOFOUND;

    /* Precio de apertura... */
    r_target_data.ticker      := r_fuente_datos.ticker; :=
    r_target_data.price_type  := 'O';
    r_target_data.price       := r_source_data.open_price; :=
    r_target_data.price_date := r_source_data.trade_date;
    INSERTAR EN LOS VALORES de la tabla de cotizaciones r_target_data;

    /* Precio de cierre... */
    r_target_data.price_type := 'C';
    r_target_data.precio     := r_source_data.close_price;
    INSERTAR EN LOS VALORES de la tabla de cotizaciones r_target_data;

  FIN DEL BUCLE;
  CERRAR c_source_data;
FIN load_stocks_legacy;
```

Regularmente veo código de este formato, y desde Oracle8/Base de datos Usualmente he usado BULK COLLECT y FORALL como mi principal herramienta de ajuste (cuando la lógica es demasiado compleja para una solución SQL basada en conjuntos). Sin embargo, una técnica alternativa (que vi por primera vez descrita por Tom Kyte¹) es usar una inserción basada en conjuntos de una función segmentada. En otras palabras, se usa una función canalizada para toda la lógica de preparación y transformación de datos heredados, pero la carga de la tabla de destino se maneja por separado como una inserción basada en conjuntos. Desde que leí sobre

1. Véase su discusión en *Arquitectura experta de base de datos Oracle*(Apress), págs. 640-643.

esta poderosa técnica, la he usado con éxito en mi propio trabajo de optimización del rendimiento, como se describe en las siguientes secciones.

Una implementación de función segmentada

Como se demuestra en [capítulo 17](#), lo primero que se debe tener en cuenta al crear una función canalizada son los datos que devolverá. Para esto, necesito crear un tipo de objeto para definir una sola fila de los datos de retorno de la función segmentada:

```
/* Archivo en la web: stockpivot_setup.sql */
CREAR TIPO stockpivot_ot AS OBJECT (
    ticker          VARCHAR2(10)
    , precio_tipo   VARCHAR2(1)
    , precio        NÚMERO
    , precio_fecha  FECHA
);
```

También necesito crear una colección de este objeto, ya que esto define el tipo de devolución de la función:

```
/* Archivo en la web: stockpivot_setup.sql */
CREAR TIPO stockpivot_ntt COMO TABLA DE stockpivot_ot;
```

Transformar el código heredado en una función canalizada es bastante simple. Primero debo definir la especificación de la función en el encabezado. También debo incluir un procedimiento de carga que describiré más adelante:

```
/* Archivo en la web: stockpivot_setup.sql */
CREAR PAQUETE stockpivot_pkg AS

TIPO stocktable_rct ES REF CURSOR
    RETURN tabla de existencias%ROWTYPE;

<recorte>

FUNCIÓN pipe_stocks(
    p_source_data IN stockpivot_pkg.stocktable_rct )
    RETURN stockpivot_ntt PIPELINED;

PROCEDIMIENTO load_stocks;

FIN stockpivot_pkg;
```

Mi función canalizada toma un REF CURSOR fuerte como parámetro de entrada (también podría usar un REF CURSOR débil en este caso). El parámetro del cursor en sí no es necesariamente necesario. Sería igual de válido para mí declarar el cursor en la función misma (como hice con el procedimiento heredado). Sin embargo, el parámetro del cursor será necesario para futuras iteraciones de esta función canalizada, por lo que lo introduce desde el principio.

La implementación de la función es la siguiente:

```
/* Archivo en la web: stockpivot_setup.sql */
1     FUNCIÓN pipe_stocks(
```

```

2      p_source_data IN stockpivot_pkg.stocktable_rct )
3      RETURN stockpivot_ntt PIPELINED IS
4
5      r_target_data stockpivot_ot := stockpivot_ot(NULL, NULL, NULL, NULL);
6      r_source_data stockpivot_pkg.stocktable_rt;
7
8      COMENZAR
9      BUCLE
10     FETCH p_source_data EN r_source_data; SALIR
11     CUANDO p_source_data%NOFOUND;
12
13     /* Primera fila... */
14     r_target_data.ticker      := r_fuente_datos.ticker;
15     r_target_data.price_type := 'O';
16     r_target_data.precio      := r_source_data.open_price;
17     r_target_data.price_date := r_source_data.trade_date; FILA
18     DE TUBO (r_target_data);
19
20     /* Segunda fila... */
21     r_target_data.price_type := 'C';
22     r_target_data.precio      := r_source_data.close_price;
23     FILA DE TUBO (r_target_data);
24
25     FIN DEL BUCLE;
26     CERRAR p_source_data;
27     DEVOLVER;
28 END pipe_stocks;

```

Además de la sintaxis general de la función canalizada (con la que ya debería estar familiarizado desde [capítulo 17](#)), la mayoría del código de la función canalizada es reconocible del ejemplo heredado. Las principales diferencias a considerar se resumen en la siguiente tabla.

Líneas)	Descripción
2	El cursor heredado se elimina del código y, en su lugar, se pasa como un parámetro REF CURSOR.
5	Mi variable de datos de destino ya no está definida como ROWTYPE de la tabla de destino. Ahora es del tipo de objeto STOCKPIVOT_OT que define los datos de retorno de la función canalizada.
18 y 23	En lugar de insertar registros en tickertable, yotuboregistros de la función. En esta etapa, la base de datos almacenará en el búfer una pequeña cantidad de mis filas de objetos canalizados en una colección correspondiente. Según el tamaño de la matriz del cliente, esta recopilación de datos almacenada en búfer estará disponible casi de inmediato.

Cargando desde una función segmentada

Como puede ver, con solo una pequeña cantidad de cambios en el programa de carga original, ahora tengo una función canalizada que prepara y canaliza todos los datos que necesito cargar en tickertable. Para completar la conversión de mi código heredado, solo necesito escribir un procedimiento adicional para insertar los datos canalizados en mi tabla de destino:

```

/* Archivo en la web: stockpivot_setup.sql */
PROCEDIMIENTO load_stocks ES

```

COMENZAR

```
INSERTAR EN LA tabla de cotizaciones (ticker, precio_tipo, precio, precio_fecha)
SELECCIONAR cotización, precio_tipo, precio, precio_fecha
DESDE LA TABLA (
    stockpivot_pkg.pipe_stocks(
        CURSOR(SELECCIONAR * DE la tabla de existencias)));
FIN cargar_stocks;
```

Eso completa la conversión básica del código heredado fila por fila a una solución de función segmentada. Entonces, ¿cómo se compara esto con el original? En mis pruebas, creé la tabla de existencias como una tabla externa con un archivo de 500 000 registros. El código fila por fila heredado se completó en 57 segundos (insertando 1 millón de filas en la tabla de cotizaciones), y la inserción basada en conjuntos usando la función canalizada se ejecutó en solo 16 segundos (los resultados de las pruebas para todos los ejemplos están disponibles en el sitio web del libro).

Teniendo en cuenta que esta es mi primera y más básica implementación de funciones canalizadas, la mejora en el rendimiento que se acaba de mostrar es bastante respetable. Sin embargo, no es exactamente el rendimiento que puedo obtener cuando uso una solución BULK COLLECT y FORALL simple (que se ejecuta en poco más de 5 segundos en mis pruebas), por lo que tendré que hacer algunas modificaciones en la carga de mi función segmentada.

Sin embargo, antes de hacer esto, tenga en cuenta que retuve las búsquedas de una sola fila del cursor principal y no hice nada para reducir el cambio de contexto "caro" (que requeriría una búsqueda BULK COLLECT). Entonces, ¿por qué es más rápido que el código fila por fila heredado?

Es más rápido principalmente debido al cambio a SQL basado en conjuntos. El DML basado en conjuntos (como INSERT...SELECT que usé en mi carga canalizada) casi siempre es considerablemente más rápido que una solución de procedimiento basada en filas. En este caso particular, me he beneficiado directamente de la optimización interna de la base de datos Oracle de las inserciones basadas en conjuntos. Específicamente, la base de datos escribe considerablemente menos información de rehacer para inserciones basadas en conjuntos (INSERTAR... SELECCIONAR) que para inserciones únicas (INSERTAR... VALORES). Es decir, si inserto 100 filas en una sola declaración, generará menos información de rehacer que si inserto 100 filas una por una.

Mi carga heredada original de 1 millón de filas de tabla de cotizaciones generó más de 270 MB de información de rehacer. Esto se redujo a poco más de 37 MB cuando usé la carga basada en funciones canalizadas, lo que contribuyó a una proporción significativa del ahorro de tiempo.



He omitido cualquier transformación de datos complicada de mis ejemplos en aras de la claridad. Debe asumir en todos los casos que las reglas de procesamiento de datos son lo suficientemente complejas para ordenar una solución de función canalizada PL/SQL en primer lugar. De lo contrario, probablemente solo usaría una solución SQL basada en conjuntos con funciones analíticas, factorización de subconsultas y expresiones CASE para transformar mis datos de gran volumen.

Ajuste de funciones canalizadas con capturas de matrices

A pesar de haber ajustado el código heredado con una implementación de función canalizada, aún no he terminado. Hay más posibilidades de optimización y necesito que mi procesamiento sea al menos tan rápido como una solución BULK COLLECT y FORALL. Tenga en cuenta que utilicé búsquedas de una sola fila desde el cursor de origen principal. La primera posibilidad de sintonización simple es, por lo tanto, utilizar búsquedas de matrices con BULK COLLECT.

Comienzo agregando un tamaño de matriz predeterminado a la especificación de mi paquete. El tamaño óptimo de recuperación de la matriz variará según sus requisitos específicos de procesamiento de datos, pero siempre prefiero comenzar mis pruebas con 100 y trabajar desde ahí. También agrego un tipo de matriz asociativa a la especificación del paquete (también podría declararse en el cuerpo); esto es para recuperaciones masivas desde el cursor de origen. Finalmente, agrego un segundo parámetro a la firma de la función canalizada para poder controlar el tamaño de recuperación de la matriz (esto no es necesario, por supuesto

- solo buena práctica). Mi especificación ahora es la siguiente:

```
/* Archivo en la web: stockpivot_setup.sql */
CREAR PAQUETE stockpivot_pkg AS
<recorte>
c_default_limit CONSTANTE PLS_INTEGER := 100;

TIPO stocktable_aat ES TABLA DE stocktable%ROWTYPE
ÍNDICE POR PLS_INTEGER;

FUNCIÓN pipe_stocks_array(
    p_source_data EN stockpivot_pkg.stocktable_rct,
    p_limit_size IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit )
RETURNS stockpivot_ntt PIPELINED;
<recorte>
FIN stockpivot_pkg;
```

La función en sí es muy similar a la versión original:

```
/* Archivo en la web: stockpivot_setup.sql */
FUNCIÓN pipe_stocks_array(
    p_source_data EN stockpivot_pkg.stocktable_rct,
    p_limit_size IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit )
RETURNS stockpivot_ntt PIPELINED IS

    r_target_data stockpivot_ot := stockpivot_ot(NULL, NULL, NULL, NULL);
    aa_source_data stockpivot_pkg.stocktable_aat;

    COMENZAR
        BUCLE
            FETCH p_source_data BULK COLLECT INTO aa_source_data LIMIT p_limit_size; SALIR
            CUANDO aa_source_data.COUNT = 0;

            /* Procesar el lote de registros (p_limit_size)... */ FOR i IN
            1 .. aa_source_data.COUNT LOOP

                /* Primera fila... */
```

```

r_target_data.ticker      := aa_source_data(i).ticker;
r_target_data.price_type := 'O';
r_target_data.precio      := aa_source_data(i).open_price;
r_target_data.price_date := aa_source_data(i).trade_date; FILA DE
TUBO (r_target_data);

/* Segunda fila... */
r_target_data.price_type := 'C';
r_target_data.precio      := aa_source_data(i).close_price;
FILA DE TUBO (r_target_data); FIN

DEL BUCLE;
FIN DEL BUCLE;
CERRAR p_source_data;
DEVOLVER;
FIN pipe_stocks_array;

```

La única diferencia con mi versión original es el uso de BULK COLLECT...LIMIT desde el cursor de origen. El procedimiento de carga es el mismo que antes, modificado para hacer referencia a la versión de matriz de la función canalizada. Esto redujo aún más mi tiempo de carga a solo 6 segundos, simplemente debido a la reducción en el cambio de contexto de PL/SQL basado en matriz. Mi solución de función segmentada ahora tiene un rendimiento comparable al de mi solución BULK COLLECT y FORALL.

Explotación de funciones canalizadas paralelas para obtener el máximo rendimiento

He logrado algunas buenas ganancias de rendimiento desde el cambio a una inserción basada en conjuntos desde una función segmentada. Sin embargo, tengo una opción de ajuste más para mi carga de stockpivot que me brindará un mejor rendimiento que cualquier otra solución: usar la capacidad paralela de las funciones canalizadas descritas en [capítulo 17](#). En esta próxima iteración, habilito en paralelo mi función stockpivot agregando otra cláusula a la firma de la función:

```

/* Archivo en la web: stockpivot_setup.sql */
CREAR PAQUETE stockpivot_pkg AS
<recorte>
    FUNCIÓN pipe_stocks_parallel(
        p_source_data EN stockpivot_pkg.stocktable_rct
        p_limit_size IN PLS_INTEGER DEFAULT stockpivot_pkg.c_default_limit )
    RETURN stockpivot_ntt
        TUBERÍA
        PARALLEL_ENABLE (PARTICIÓN p_source_data POR ANY);
<recorte>
FIN stockpivot_pkg;

```

Al usar el esquema de particionamiento ANY, he instruido a la base de datos de Oracle para que asigne aleatoriamente mis datos de origen a los procesos paralelos. Esto se debe a que el orden en que la función recibe y procesa los datos de origen no tiene efecto en la salida resultante (es decir, no hay dependencias entre filas). Eso no es siempre el caso, por supuesto.

Habilitación de la ejecución de funciones canalizadas en paralelo

A parte de la sintaxis de habilitación en paralelo en la especificación y el cuerpo, la implementación de la función es la misma que en el ejemplo de extracción de matriz (ver el `stockpivot_setup.sql` archivo en el sitio web para el paquete completo). Sin embargo, debo asegurarme de que la carga de mi tablero de cotizaciones sea ejecutado en paralelo. Primero, debo habilitar DML paralelo a nivel de sesión. Una vez hecho esto, invoco la consulta paralela de una de las siguientes maneras:

- Uso de la sugerencia PARALELO
- Uso de configuraciones paralelas de GRADO en los objetos subyacentes
- Forzar consulta paralela (ALTER SESSION FORCE PARALLEL (QUERY) PARALLEL norte)



La consulta paralela/DML es una función de Oracle Database Enterprise Edition. Si está usando Standard Edition o Standard Edition One, no tiene licencia para usar la función paralela de funciones canalizadas.

En mi carga, habilité DML paralelo a nivel de sesión y usé sugerencias para especificar un grado de paralelismo (DOP) de 4:

```
/* Archivo en la web: stockpivot_setup.sql */ EL
PROCEDIMIENTO load_stocks_parallel ES COMENZAR

EJECUTAR INMEDIATAMENTE 'ALTER SESSION HABILITAR DML PARALELO';

INSERTAR /*+ PARALELO(t, 4) */ EN la tabla de cotizaciones t
      (ticker, precio_tipo, precio, precio_fecha)
SELECT ticker, precio_tipo, precio, precio_fecha DE
LA TABLA(
      stockpivot_pkg.pipe_stocks_parallel(
      CURSOR(SELECT /*+ PARALLEL(s, 4) */ * FROM stocktable s)));
);

FIN load_stocks_parallel;
```

Esto reduce el tiempo de carga a poco más de 3 segundos, una mejora significativa en mi código heredado original y todas las demás versiones de mi carga de funciones canalizadas. Por supuesto, cuando estamos trabajando en pequeñas unidades de tiempo como esta, los costos de inicio de los procesos paralelos afectarán el tiempo de ejecución general, pero aun así logré una mejora de casi el 50 % en mi versión de matriz. El hecho de que las inserciones paralelas usen cargas de ruta directa en lugar de cargas de ruta convencionales también significa que la generación de rehacer se redujo aún más, ja solo 25 KB!

En los sistemas comerciales, es posible que esté ajustando procesos que se ejecutan durante una hora o más, por lo que las ganancias que puede lograr con cargas canalizadas en paralelo serán significativas tanto en términos proporcionales como reales.



Cuando está utilizando funciones canalizadas paralelas, su cursor de origen debe pasarse como un parámetro REF CURSOR. En las funciones canalizadas en serie, el cursor de origen se puede incrustar en la función misma (aunque he optado por no hacerlo en ninguno de mis ejemplos).

Además, REF CURSOR puede tener un tipo débil o fuerte para las funciones particionadas con el esquema ANY, pero para la partición basada en HASH o RANGE, debe estar fuertemente tipado. Ver [Capítulo 15](#) para obtener más detalles sobre REF CURSOR y variables de cursor.

Ajuste de operaciones de combinación con funciones canalizadas

Es posible que ahora esté considerando funciones canalizadas en serie o en paralelo como un mecanismo de ajuste para sus propias cargas de datos de gran volumen. Sin embargo, no todas las cargas involucran insertos como el ejemplo del pivote de stock. Muchas cargas de datos son incrementales y requieren fusiones periódicas de datos nuevos y modificados. La buena noticia es que el mismo principio de combinar transformaciones PL/SQL con SQL basado en conjuntos también se aplica a las fusiones (y actualizaciones).

Procesamiento de combinación PL/SQL basado en filas

Considere el siguiente procedimiento, tomado de mi ejemplo employee_pkg. Tengo una combinación de una gran cantidad de registros de empleados, pero mi código heredado usa una antigua técnica PL/SQL de intentar una actualización primero e insertar solo cuando la actualización coincide con cero registros en la tabla de destino:

```
/* Archivo en la web: employee_merge_setup.sql */
PROCEDIMIENTO upsert_employees ES
  n PLS_INTEGER := 0;
  COMENZAR
    PARA r_emp IN (SELECCIONE * DE empleados_staging) BUCLE
      ACTUALIZAR empleados
      COLOCAR    <recorte>
      DÓNDE    empleado_id = r_emp.employee_id;

      SI SQL%ROWCOUNT = 0 ENTONCES
        INSERTAR EN empleados (<recorte>)
        VALORES (<recorte>); TERMINARA SI;

    FIN DEL BUCLE;
FIN upsert_employees;
```

He quitado parte del código por razones de brevedad, pero puedes ver claramente la técnica de "upsert" en acción. Tenga en cuenta que he usado un bucle FOR de cursor implícito que se beneficiará de la optimización de obtención de matriz introducida en PL/SQL en Oracle Database 10*gramo*.

Para probar este procedimiento, creé una tabla de preparación de 500 000 registros de empleados (¡esta es una corporación enorme!) e inserté 250 000 de ellos en una tabla de empleados para crear una división equitativa entre actualizaciones e inserciones. Esta solución PL/SQL de “combinación de pobres” se completó en 46 segundos.

Uso de funciones canalizadas para MERGE basado en conjuntos

Convertir este ejemplo en un SQL MERGE basado en conjuntos desde una función canalizada es, una vez más, bastante simple. Primero, creo el objeto de soporte y los tipos de tablas anidadas (vea el *empleados_merge_setup.sql* para más detalles) y declare la función en el encabezado del paquete:

```
/* Archivo en la web: employee_merge_setup.sql */
CREAR PAQUETE employee_pkg AS

    c_default_limit CONSTANTE PLS_INTEGER := 100;

    TIPO empleado_rct ES REF CURSOR RETURN empleados_staging%ROWTYPE;
    TIPO employee_aat ES TABLA DE empleados_staging%ROWTYPE
        ÍNDICE POR PLS_INTEGER;
    <recorte>

    FUNCIÓN pipe_employees(
        p_source_data IN employee_pkg.employee_rct
        p_limit_size IN PLS_INTEGER DEFAULT employee_pkg.c_default_limit )
        RETURN employee_ntt
        TUBERÍA
        PARALLEL_ENABLE (PARTICIÓN p_source_data POR ANY);
    FIN empleado_paquete;
```

Habilite en paralelo la función canalizada y usé el esquema de partición CUALQUIER, como antes. La implementación de la función es la siguiente:

```
/* Archivo en la web: employee_merge_setup.sql */
FUNCIÓN pipe_employees(
    p_source_data IN employee_pkg.employee_rct,
    p_limit_size IN PLS_INTEGER DEFAULT employee_pkg.c_default_limit )
    RETURN employee_ntt
    TUBERÍA
    PARALLEL_ENABLE (PARTICIÓN p_source_data POR CUALQUIERA)
    ES aa_source_data employee_pkg.employee_aat;

COMENZAR
    BUCLE
        FETCH p_source_data BULK COLLECT INTO aa_source_data LIMIT p_limit_size; SALIR
        CUANDO aa_source_data.COUNT = 0;
        PARA i EN 1 .. aa_source_data.COUNT LOOP
            FILA DE TUBO (
                empleado_ot( aa_source_data(i).employee_id,
                <recorte>
                SYSDATE ));

FIN DEL BUCLE;
```

```

    FIN DEL BUCLE;
    CERRAR p_source_data;
    DEVOLVER;
END tubería_empleados;

```

Esta función simplemente obtiene los datos de origen y los canaliza en el formato correcto. Ahora puedo usar mi función en una declaración MERGE, que envuelvo en un procedimiento en employee_pkg, de la siguiente manera:

```

/* Archivo en la web: employee_merge_setup.sql */
PROCEDIMIENTO merge_employees ES
COMENZAR

    EJECUTAR INMEDIATAMENTE 'ALTER SESSION HABILITAR DML PARALELO';

    COMBINAR /*+ PARALELO(e, 4) */
        EN empleados e
        TABLA DE USO (
            empleado_pkg.pipe_employees(
                CURSOR(SELECCIONAR /*+ PARALELO(es, 4) */ *
                    FROM empleados_staging es))) s
        EN (e.employee_id = s.employee_id)

    CUANDO EMPAREJADO ENTONCES
        ACTUALIZAR
        COLOCAR <recorte>
    CUANDO NO COINCIDE ENTONCES
        INSERTAR (<recortar>)
        VALORES (<recortar>);

    FIN fusionar_empleados;

```

SQL MERGE de mi función canalizada paralela reduce el tiempo de carga en más del 50 %, a solo 21 segundos. El uso de funciones canalizadas paralelas como origen de filas para operaciones de SQL basadas en conjuntos es claramente una técnica de ajuste valiosa para cargas de datos de volumen.

Descarga de datos asincrónicos con funciones canalizadas en paralelo

Hasta ahora, he demostrado dos tipos de cargas de datos que se han beneficiado de la conversión a una función canalizada en paralelo. También es posible que desee aprovechar la característica paralela de las funciones canalizadas para aquellos momentos en los que necesita descargar datos (incluso en el siglo XXI, todavía tengo que ver un ODS/DSS/almacén corporativo interno que no extraiga datos para transferir a otros sistemas).

Un programa típico de extracción de datos

Imagina el siguiente escenario. Tengo un extracto diario de todos mis datos comerciales (mantenidos en tickertable) para transferirlos a un sistema de oficina intermedia, que espera un archivo plano delimitado. Para lograr esto, escribo una utilidad simple para descargar datos de un cursor:

```

/* Archivo en la web: parallel_unload_setup.sql */
PROCEDIMIENTO legacy_unload(

```

```

p_fuente      EN SYS_REFCURSOR,
p_nombre_de_archivo  EN VARCHAR2,
p_directorio EN VARCHAR2,
p_limit_size EN PLS_INTEGER DEFAULT unload_pkg.c_default_limit ) ES

EL TIPO row_aat ES LA TABLA DE VARCHAR2 (32767)
ÍNDICE POR PLS_INTEGER;
aa_filas fila_aat;
v_nombre    VARCHAR2(128) := p_nombre_archivo || '.TXT';
archivo_v   UTL_FILE.FILE_TYPE;

COMENZAR
  v_file := UTL_FILE.FOPEN( p_directory, v_name, 'w', c_maxline ); BUCLE

    FETCH p_source BULK COLLECT INTO aa_rows LIMIT p_limit_size; SALIR
    CUANDO aa_rows.COUNT = 0;
    PARA i EN 1 .. aa_rows.COUNT LOOP
      UTL_FILE.PUT_LINE(v_file, aa_rows(i)); FIN DEL
      BUCLE;
    FIN DEL BUCLE;
  CERRAR p_fuente;
  UTL_FILE.FCLOSE(v_file);
FIN legacy_unload;

```

Simplemente recorro el parámetro del cursor de origen usando un tamaño de búsqueda de matriz de 100 y escribo cada lote de filas en el archivo de destino usando UTL_FILE. El cursor de origen tiene solo una columna: el cursor se prepara con las columnas de origen ya concatenadas/delimitadas.

En las pruebas, 1 millón de filas delimitadas de la tabla de cotizaciones se descargaron en un archivo plano en solo 24 segundos (me aseguré de que la tabla de cotizaciones se escaneara por completo varias veces de antemano para reducir el impacto de la E/S física). Pero ticketable tiene una longitud de fila promedio de solo 25 bytes, por lo que se descarga muy rápido. Los sistemas comerciales escribirán significativamente más datos (tanto en longitud de fila como en recuento de filas) y potencialmente tardarán decenas de minutos.

Un descargador de funciones canalizadas habilitado en paralelo

Si reconoce este escenario de sus propios sistemas, debería considerar la optimización con funciones canalizadas paralelas. Si analiza el ejemplo heredado anterior, toda la manipulación de datos se puede colocar dentro de una función canalizada (específicamente, no hay operaciones DML). Entonces, ¿qué tal si tomo la lógica de búsqueda del cursor y la administración de UTL_FILE y la coloco dentro de una función canalizada paralela? Si hago esto, puedo aprovechar la función de consulta paralela de Oracle para descargar los datos en varios archivos mucho más rápido.

Por supuesto, las funciones canalizadas generalmente devuelven datos canalizados, pero en este caso mis filas de origen se escriben en un archivo y no necesito que se devuelvan al cliente. En cambio, devolveré una fila por proceso paralelo con algunos metadatos muy básicos para describir la información de la sesión y la cantidad de filas que extrajo. Mis tipos de apoyo son los siguientes:

```
/*
Archivo en la web: parallel_unload_setup.sql */
CREAR TIPO unload_ot COMO OBJETO
```

```

(nombre_archivo VARCHAR2(128),
no_records NÚMERO
, ID_sesión NÚMERO);

CREAR TIPO unload_ntt COMO TABLA DE unload_ot;

```

La implementación de mi función se basa en el procesamiento heredado, con alguna configuración adicional requerida para los metadatos que se devuelven:

```

/* Archivo en la web: parallel_unload_setup.sql */
1  FUNCIÓN parallel_unload(
2      p_fuente      EN SYS_REFCURSOR,
3      p_nombre_de_archivo  EN VARCHAR2,
4      directorio_p  EN VARCHAR2,
5      p_limit_size EN PLS_INTEGER DEFAULT unload_pkg.c_default_limit )
6
7      VOLVER descargar_ntt
8      PIPELINED PARALLEL_ENABLE (PARTICIÓN p_source POR CUALQUIERA) COMO
9      aa_rows row_aat;
10     v_sid    NÚMERO := SYS_CONTEXT('USERENV','SID'); VARCHAR2(128) :=
11     v_nombre   p_nombre_archivo || '_' || v_sid || '.TXT'; UTL_FILE.FILE_TYPE;
12     archivo_v
13     v_líneas PLS_INTEGER;
14
15     COMENZAR
16         v_file := UTL_FILE.FOPEN(p_directory, v_name, 'w', c_maxline); BUCLE
17             FETCH p_source BULK COLLECT INTO aa_rows LIMIT p_limit_size; SALIR
18             CUANDO aa_rows.COUNT = 0;
19             PARA i EN 1 .. aa_rows.COUNT LOOP
20                 UTL_FILE.PUT_LINE(v_file, aa_rows(i)); FIN DEL
21             BUCLE;
22             FIN DEL BUCLE;
23             v_lines := p_source%ROWCOUNT;
24             CERRAR p_fuente;
25             UTL_FILE.FCLOSE(v_file);
26             FILA DE TUBO (descargar_ot(v_nombre, v_líneas, v_sid));
27             DEVOLVER;
28     FIN paralelo_descarga;

```

Tenga en cuenta los puntos sobre esta función en la siguiente tabla.

Líneas)	Descripción
1 y 8	Mi función está habilitada en paralelo y dividirá los datos de origen por CUALQUIERA. Por lo tanto, puedo declarar mi cursor de origen en función del tipo SYS_REFCURSOR definido por el sistema.
10	Los metadatos de mi devolución incluirán el ID de sesión (SID). Está disponible en el contexto de la aplicación USERENV. Puede derivar el SID de vistas como V\$MYSTAT en versiones anteriores a Oracle Database 10gramo.
11	Quiero descargar en paralelo varios archivos, así que creo un nombre de archivo único para cada invocación paralela.
15-22 y 24-25	Reutilizo toda la lógica de procesamiento de la implementación heredada original.
26	Para cada invocación de la función, canalizo una sola fila que contiene el nombre del archivo, el número de filas extraídas y el identificador de la sesión.

Con un esfuerzo mínimo, he habilitado en paralelo mi descargador de datos, utilizando la función canalizada como un mecanismo de bifurcación asíncrono. Ahora veamos cómo invocar esta nueva versión:

```
/* Archivo en la web: parallel_unload_test.sql */
SELECCIONAR *
DE      MESA(
    descargar_pkg.parallel_unload(
        _fuente => CURSOR(SELECCIONAR /*+ PARALELO(t, 4)*/
            teletipo || ' ' ||
            precio_tipo || ' ' ||
            precio      || ' ' ||
            TO_CHAR(precio_fecha,'AAAAMMDDHH24MISS') tabla
        DE      de cotizaciones t),
        p_filename => 'tabla de
cotizaciones', p_directory => 'DIR' ));
```

Aquí está mi salida de prueba de SQL*Plus:

NOMBRE DEL ARCHIVO	NO_RECORDS	sesion_id
tickertable_144.txt	260788	144
tickertable_142.txt	252342	142
tickertable_127.txt	233765	127
tickertable_112.txt	253105	112

4 filas seleccionadas.

Transcurrido: 00:00:12.21

En mi sistema de prueba, con cuatro procesos paralelos, he reducido aproximadamente a la mitad mi tiempo de procesamiento. Recuerde que cuando trabaja con una pequeña cantidad de segundos, como en este ejemplo, el costo del inicio en paralelo puede tener un impacto en el tiempo de procesamiento. Para extracciones que tardan minutos o más en completarse, sus ahorros potenciales (tanto en términos reales como reales) pueden ser mucho mayores.



Es fácil mejorar aún más esta técnica "ajustando" las llamadas UTL_FILE, utilizando un mecanismo de almacenamiento en búfer. Consulte la función PARAL-LEL_UNLOAD_BUFFERED en el *parallel_unload_set up.sql* archivo en el sitio web del libro para la aplicación. En lugar de escribir cada línea en un archivo inmediatamente, agrego líneas a un gran búfer VARCHAR2 (alternativamente, podría usar una colección) y vaciar su contenido en un archivo periódicamente. Reducir las llamadas UTL_FILE de tal manera casi redujo a la mitad el tiempo de extracción de mi descargador paralelo, a poco menos de 7 segundos.

Implicaciones de rendimiento de las cláusulas de partición y transmisión en funciones canalizadas en paralelo

Todos mis ejemplos de funciones canalizadas en paralelo hasta ahora han usado el esquema de partición CUALQUIER, porque no ha habido dependencias entre las filas de datos de origen. Como se describe en [capítulo 17](#), hay varias opciones de partición y transmisión para controlar cómo se asignan y ordenan los datos de entrada de origen en procesos paralelos. En resumen, estos son:

- Opciones de partición (para asignar datos a procesos paralelos):
 - PARTICIÓN *p_cursor* POR CUALQUIERA
 - PARTICIÓN *p_cursor* POR RANGO(*cursor_columna(s)*)
 - PARTICIÓN *p_cursor* POR HASH(*cursor_columna(s)*)
- Opciones de transmisión (para ordenar datos dentro de un proceso paralelo):
 - GRUPO *p_cursor* POR (*cursor_columna(s)*)
 - ORDEN *p_cursor* POR (*cursor_columna(s)*)

El método particular que elija depende de sus requisitos específicos de procesamiento de datos. Por ejemplo, si necesita asegurarse de que todos los pedidos de un cliente específico se procesen juntos, pero en el orden de la fecha, puede usar la partición HASH con la transmisión ORDER. Si necesita asegurarse de que todos sus datos comerciales se procesen en el orden de los eventos, puede utilizar una combinación de RANGO/ORDEN.

Rendimiento relativo de las combinaciones de partición y transmisión

Estas opciones tienen sus propias características de rendimiento que resultan de la ordenación que implican. La siguiente tabla resume el tiempo necesario para canalizar 1 millón de filas de tabla de cotizaciones a través de una función de canalización paralela (con un DOP de 4) utilizando cada una de las opciones de partición y transmisión.²

Opción de partición	Opción de transmisión	Tiempo transcurrido (s)
CUALQUIER	—	5.37
CUALQUIER	ORDEN	8.06
CUALQUIER	GRUPO	9.58
PICADILLO	—	7.48
PICADILLO	ORDEN	7.84
PICADILLO	GRUPO	8.10
RANGO	—	9.84

2. Para probar usted mismo el rendimiento de estas opciones, utilice el *opciones_paralelas_*.sql* archivos disponibles en el sitio web de este libro.

Opción de partición	Opción de transmisión	Tiempo transcurrido (s)
RANGO	ORDEN	10.59
RANGO	GRUPO	10.90

Como era de esperar, las particiones ANY y HASH son comparables (aunque la opción ANY sin ordenar es cómodamente la más rápida), pero el mecanismo de partición RANGE es significativamente más lento. Probablemente esto también sea de esperar, porque los datos de origen deben ordenarse antes de que la base de datos pueda dividirlos entre los esclavos. Dentro de los propios procesos paralelos, la ordenación es más rápida que la agrupación en clústeres para todas las opciones de partición (este es quizás un resultado sorprendente, ya que la agrupación en clústeres no necesita ordenar todo el conjunto de datos). Su millaje puede variar, por supuesto.

Particionamiento con datos sesgados

Otra consideración con la partición es la división de la carga de trabajo entre los procesos paralelos. Las opciones ANY y HASH conducen a una distribución de datos razonablemente uniforme entre los procesos paralelos, independientemente del número de filas en la fuente. Sin embargo, dependiendo de las características de sus datos, la partición RANGE puede conducir a una asignación muy desigual, especialmente si los valores en las columnas de partición están sesgados. Si un proceso paralelo recibe una parte demasiado grande de los datos, esto puede negar cualquier beneficio de las funciones canalizadas paralelas. Para probar esto usted mismo, use los archivos llamados *paralelo/el_skew_*.sql* disponible en el sitio web del libro.



Todas mis llamadas a funciones segmentadas incluyen un parámetro REF CURSOR proporcionado a través de la función CURSOR(SELECT...). Como alternativa, es perfectamente legal preparar una variable REF CURSOR usando el OPEN cursor de referencia FOR... construye y pasa esta variable en lugar de la llamada CURSOR(SELECT...). Si elige hacer esto, ¡cuidado con el error 5349930! Cuando está utilizando funciones canalizadas habilitadas en paralelo, este error puede causar que un proceso paralelo muera inesperadamente con un ORA-01008: *no todas las variables enlazadas excepción*.

Funciones canalizadas y el optimizador basado en costos

Los ejemplos de este capítulo demuestran el uso de funciones segmentadas como fuentes de fila simples que generan datos para escenarios de carga y descarga. Sin embargo, en algún momento, es posible que deba unir una función canalizada a otro origen de fila (como una tabla, una vista o la salida intermedia de otras uniones dentro de un plan de ejecución de SQL). Las estadísticas de fuente de fila (como cardinalidad, distribución de datos, valores nulos, etc.) son fundamentales para lograr planes de ejecución eficientes, pero en el caso de las funciones canalizadas (o, de hecho, cualquier función de tabla), el optimizador basado en costos no lo hace. tener mucha información con la que trabajar.

Heurística de cardinalidad para funciones de tabla canalizadas

En base de datos Oracle 11g *gramo* En la versión 1 y anteriores, el CBO aplica una heurística de cardinalidad a las funciones de tabla y canalizadas en las declaraciones SQL, y esto a veces puede conducir a planes de ejecución inefficientes. La cardinalidad predeterminada parece depender del valor del parámetro de inicialización DB_BLOCK_SIZE, pero en una base de datos con un tamaño de bloque estándar de 8 KB, Oracle utiliza una heurística de 8168 filas. Puedo demostrar esto con bastante facilidad con una función canalizada que canaliza un subconjunto de columnas de la tabla de empleados. Usando Autotrace en SQL*Plus para generar un plan de ejecución, veo lo siguiente:

```
/* Archivos en la web: cbo_setup.sql y cbo_test.sql */ SQL>
SELECCIONAR *
2DESDE LA TABLA (pipe_employees) e;
```

Plan de ejecución

Valor hash del plan: 1802204150

identificación	Operación	Nombre	Filas
0	SELECCIONE DECLARACIÓN		
1	ITERADOR DE COLECCIÓN PICKLER FETCH	PIPE_EMPLEADOS	8168

Esta función canalizada en realidad devuelve 50 000 filas, por lo que si uno la función a la tabla de departamentos, corro el riesgo de obtener un plan subóptimo:

```
/* Archivo en la web: cbo_test.sql */ SQL>
SELECCIONAR *
2 DE departamentos d
3 , TABLE(pipe_employees) e
4 DÓNDE d.department_id = e.department_id;
```

Plan de ejecución

Valor hash del plan: 4098497386

identificación	Operación	Nombre	Filas
0	SELECCIONE DECLARACIÓN 1		8168
1	FUSIONAR UNIRSE		8168
2	ACCESO A LA TABLA POR ÍNDICE ROWID ÍNDICE	DEPARTAMENTOS	27
3	EXPLORACIÓN COMPLETA	DEPT_ID_PK	27
* 4	ORDENAR UNIRSE		8168
5	ITERADOR DE COLECCIÓN PICKLER FETCH	PIPE_EMPLEADOS	

Como se predijo, este parece ser un plan de ejecución subóptimo; es poco probable que una unión sortmerge sea más eficiente que una unión hash en este escenario. Entonces, ¿cómo influyo

la CBO? Para este ejemplo, podría usar sugerencias de acceso simples como LEADING y USE_HASH para anular de manera efectiva la decisión basada en costos de la CBO y asegurar una unión hash entre la tabla y la función canalizada. Sin embargo, para declaraciones SQL más complejas, es bastante difícil proporcionar todas las sugerencias necesarias para "bloquear" un plan de ejecución. A menudo es mucho mejor proporcionar a la CBO mejores estadísticas con las que tomar sus decisiones. Hay dos maneras de hacer esto:

Muestreo dinámico del optimizador

Esta función se mejoró en Oracle Database 11g *gramo*(11.1.0.7) para incluir muestreo para funciones de tabla y canalizadas.

Cardinalidad definida por el usuario

Hay varias formas de proporcionar al optimizador una estimación adecuada de la cardinalidad de una función segmentada.

A continuación, demostraré ambos métodos para mi función pipe_employees.

Uso del muestreo dinámico del optimizador para funciones canalizadas

El muestreo dinámico es una característica extremadamente útil que permite al optimizador tomar una pequeña muestra estadística de uno o más objetos en una consulta durante la fase de análisis. Puede usar el muestreo dinámico cuando no haya recopilado estadísticas en todas sus tablas en una consulta o cuando esté usando objetos transitorios como tablas temporales globales. A partir de la versión 11.1.0.7, la base de datos de Oracle puede usar muestreo dinámico para funciones de tabla o canalizadas.

Para ver qué diferencia puede hacer esta función, repetiré mi consulta anterior pero incluiré una sugerencia de DYNAMIC_SAMPLING para la función pipe_employees:

```
/* Archivo en la web: cbo_test.sql */ SQL>
SELECCIONAR /*+ MUESTREO_DINÁMICO(e 5) */
  2      *
  3  DE    departamentos      d
  4 ,    TABLE(pipe_employees) e
  5  DÓNDE d.department_id = e.department_id;
```

Plan de ejecución

Valor hash del plan: 815920909

identificación	Operación	Nombre	Filas
0	SELECCIONE DECLARACIÓN 1		50000
* 1	HASH ÚNETE		50000
2	ACCESO A MESA COMPLETO	DEPARTAMENTOS	27
3	ITERADOR DE COLECCIÓN PICKLER FETCH	PIPE_EMPLEADOS	

Esta vez, la CBO ha calculado correctamente las 50.000 filas que devuelve mi función y ha generado un plan más adecuado. Tenga en cuenta que usé la palabra *calculado* y no *estimado*. Esto se debe a que en la versión 11.1.0.7 y posteriores, el optimizador toma una muestra del 100 % de la tabla o función canalizada, independientemente del nivel de muestreo dinámico que se utilice. Usé el nivel 5, pero podría haber usado cualquier cosa entre el nivel 2 y el nivel 10 para obtener exactamente el mismo resultado. Esto significa, por supuesto, que el muestreo dinámico puede ser potencialmente costoso o llevar mucho tiempo si se utiliza para consultas que involucran funciones canalizadas de ejecución prolongada o de gran volumen.

Proporcionar estadísticas de cardinalidad al optimizador

La única información que puedo pasar explícitamente al CBO para mi función segmentada es su cardinalidad. Como suele ser el caso con Oracle, hay varias formas de hacerlo:

Sugerencia de CARDINALIDAD (sin documentar)

Le dice a la base de datos de Oracle la cardinalidad de un origen de fila en un plan de ejecución. Esta sugerencia es bastante limitada en uso y efectividad.

Sugerencia OPT_ESTIMATE (sin documentar)

Proporciona un factor de escala para corregir la cardinalidad estimada de un origen de fila, una unión o un índice en un plan de ejecución. Esta sugerencia se usa en los perfiles de SQL, una característica con licencia separada introducida en Oracle Database 10g. Edición de Empresa. Los perfiles de SQL se utilizan para almacenar factores de escala para las declaraciones de SQL existentes para mejorar y estabilizar sus planes de ejecución.

Interfaz del optimizador extensible

Asocia una función canalizada o de tabla con un tipo de objeto para calcular su cardinalidad y proporciona esta información directamente al CBO (disponible a partir de Oracle Database 10g).

Oracle Corporation no admite oficialmente las sugerencias CARDINALITY y OPT_ESTIMATE. Por esta razón, prefiero no usarlos en el código de producción. Además de los perfiles de SQL (o el muestreo dinámico, como se describió anteriormente), el único método admitido oficialmente para proporcionar estimaciones de cardinalidad de funciones canalizadas al CBO es usar las funciones de extensibilidad del optimizador introducidas en Oracle Database 10g.

Optimizador extensible y cardinalidad de función segmentada

La extensibilidad de Optimizer es parte de la implementación del cartucho de datos de Oracle: un conjunto de interfaces bien formadas que nos permiten ampliar la funcionalidad integrada de la base de datos con nuestro propio código y algoritmos (generalmente almacenados en tipos de objetos). Para funciones segmentadas y de tabla, la base de datos proporciona una interfaz dedicada específicamente para estimaciones de cardinalidad. En el siguiente ejemplo simple para mi función pipe_employees, voy a asociar mi función canalizada con un tipo de objeto especial que le informará al CBO sobre la cardinalidad de la función. La especificación de la función pipe_employees es la siguiente:

```
/* Archivo en la web: cbo_setup.sql */
FUNCTION pipe_employees(
    p_cardinality INTEGER DEFAULT 1 )
RETURN employee_ntt PIPELINED
```

Tenga en cuenta el parámetro `p_cardinality`. Mi cuerpo `pipe_employees` no usa este parámetro en absoluto; en cambio, voy a usar esto para decirle al CBO la cantidad de filas que espero que devuelva mi función. Como el optimizador extensible necesita que esto se haga a través de un tipo de interfaz, primero creo mi especificación de tipo de objeto de interfaz:

```
/* Archivo en la web: cbo_setup.sql */ CREAR TIPO
1 pipelined_stats_ot COMO OBJETO (
2
3     entero ficticio,
4
5     FUNCIÓN ESTÁTICA ODCIGetInterfaces (
6             p_interfaces OUT SYS.ODCIOBJECTLIST ) NÚMERO
7             DE DEVOLUCIÓN,
8
9     FUNCIÓN ESTÁTICA ODCIStatsTableFunction (
10            función_p      EN SYS.ODCIFUNCINFO,
11            p_stats       SALIDA SYS.ODCITabFuncStats,
12            p_args        ENTRADA SYS.ODCIArgDescList,
13            p_cardinalidad INTEGER )
14            NÚMERO DEVUELTO
15 );
```

Tenga en cuenta los puntos sobre esta especificación de tipo enumerados en la siguiente tabla.

Líneas)	Descripción
3	Todos los tipos de objetos deben tener al menos un atributo, por lo que he incluido uno llamado "ficticio" porque no es necesario para este ejemplo.
5 y 9	Estos métodos son parte de la interfaz bien formada para el Optimizador extensible. Hay varios otros métodos disponibles, pero los dos que he usado son los necesarios para implementar una interfaz de cardinalidad para mi función canalizada.
10-12	Estos parámetros <code>ODCIStatsTableFunction</code> son obligatorios. Los nombres de los parámetros son flexibles, pero sus posiciones y tipos de datos son fijos.
13	Todos los parámetros de una función canalizada o de tabla deben replicarse en su tipo de estadística asociado. En mi ejemplo, <code>pipe_employees</code> tiene un solo parámetro, <code>p_cardinality</code> , que también debo incluir en mi firma <code>ODCIStatsTableFunction</code> .

Mi algoritmo de cardinalidad se implementa en el tipo de cuerpo de la siguiente manera:

```
/* Archivo en la web: cbo_setup.sql */ CREAR
1 TIPO DE CUERPO pipelined_stats_ot AS
2
3     FUNCIÓN ESTÁTICA ODCIGetInterfaces (
4             p_interfaces OUT SYS.ODCIOBJECTLIST ) EL
5             NÚMERO DE DEVOLUCIÓN ES
6
7     COMENZAR
8         p_interfaces := SYS.ODCIOBJECTLIST(
9                         SYS.ODCIOBJECT ('SYS', 'ODCISTATS2'))
```

```

9      );
10     RETORNO ODCIConst.éxito;
11 FIN ODCIGetInterfaces;
12
13 FUNCIÓN ESTÁTICA ODCIStatsTableFunction (
14         función_p    EN SYS.ODCIFuncInfo,
15         p_stats      SALIDA SYS.ODCITabFuncStats,
16         p_args       ENTRADA SYS.ODCIArgDescList,
17         p_cardinality IN INTEGER ) EL
18         NÚMERO DEVUELTO ES
19
20 COMENZAR
21     p_stats := SYS.ODCITabFuncStats(NULL);
22     p_stats.num_rows := p_cardinalidad;
23     RETORNO ODCIConst.éxito;
24 FIN ODCIStatsTableFunction;
25 FIN;

```

Esta es una implementación de interfaz muy simple. Los puntos clave a tener en cuenta se enumeran en la siguiente tabla.

Líneas	Descripción
3–11	La base de datos de Oracle necesita esta asignación obligatoria. Aquí no se requiere una lógica definida por el usuario.
20–21	Este es mi algoritmo de cardinalidad. El parámetro p_stats OUT es cómo le digo al CBO la cardinalidad de mi función. Cualquier valor que pase al parámetro p_cardinality de mi pipe_employees será referenciado dentro de mi tipo de estadísticas. Durante la optimización de consultas (es decir, un "análisis duro"), el CBO invocará el método ODCIStatsTableFunction para recuperar el valor del parámetro p_stats y utilizarlo en sus cálculos.

En resumen, ahora tengo una función canalizada y un tipo de estadísticas. Todo lo que necesito hacer ahora es asociar los dos objetos usando el comando SQL ASSOCIATE STATISTICS. Esta asociación es lo que permite que suceda la "magia" que acabo de describir:

```

/* Archivo en la web: cbo_test.sql */
ASOCIA ESTADÍSTICAS CON FUNCIONES pipe_employees UTILIZANDO pipelined_stats_ot;

```

Ahora estoy listo para probar. Repetiré mi consulta anterior, pero incluiré la cantidad de filas que espero que devuelva mi función canalizada (esta función canaliza 50,000 filas):

```

/* Archivo en la web: cbo_test.sql */ SQL>
SELECCIONAR *
2 DE departamentos          d
3 , TABLE(pipe_employees(50000)) e
4 DÓNDE d.department_id = e.department_id;

```

Plan de ejecución

Valor hash del plan: 815920909

identificación	Operación	Nombre	Filas

0	SELECCIONE DECLARACIÓN 1		50000
*	HASH ÚNETE		50000
2	ACCESO A MESA COMPLETO	DEPARTAMENTOS	27
3	ITERADOR DE COLECCIÓN PICKLER FETCH	PIPE_EMPLEADOS	

Esta vez, mi cardinalidad esperada ha sido recogida y utilizada por la CBO, y tengo el plan de ejecución que esperaba. ¡Ni siquiera he tenido que usar pistas! En la mayoría de los casos, si la CBO recibe información precisa, tomará una buena decisión, como se demuestra en este ejemplo. Por supuesto, el ejemplo también destaca la "magia" del Optimizador Extensible. Proporcioné mi cardinalidad esperada como un parámetro para la función pipe_employees y, durante la fase de optimización, la base de datos accedió a este parámetro a través del tipo de estadística asociado y lo usó para establecer la cardinalidad de la fuente de la fila en consecuencia (usando mi algoritmo). Encuentro esto bastante impresionante.

Como pensamiento final, tenga en cuenta que tiene sentido encontrar una forma sistemática de derivar cardinalidades de funciones segmentadas. He demostrado un método; de hecho, debería agregar un parámetro p_cardinality a `domis` funciones canalizadas y asociarlas todas con el tipo de interfaz pipelined_statistics_ot. Los algoritmos que utiliza en sus tipos de interfaz pueden ser tan sofisticados como necesite. Pueden basarse en otros parámetros de función (por ejemplo, puede devolver cardinalidades diferentes basadas en valores de parámetros particulares). Tal vez podría almacenar las cardinalidades esperadas en una tabla de búsqueda y hacer que el tipo de interfaz consulte esto en su lugar. Hay muchas formas diferentes de usar esta función.

Ajuste de cargas de datos complejas con funciones canalizadas

Mi ejemplo de stockpivot transformó cada fila de entrada en dos filas de salida de la misma estructura de registro. Todos mis otros ejemplos canalizaron una sola fila de salida de una sola estructura de registro. Pero algunas transformaciones o cargas no son tan sencillas. Es bastante común cargar varias tablas desde una única tabla de preparación. ¿Las funciones canalizadas también pueden ser útiles en tales escenarios?

La buena noticia es que pueden; las cargas de varias tablas también se pueden ajustar con funciones segmentadas. La función en sí puede canalizar tantos tipos de registros diferentes como necesite, y las inserciones de tablas múltiples condicionales o incondicionales pueden cargar las tablas correspondientes con los atributos relevantes.

Una fuente, dos objetivos

Considere un ejemplo de carga de clientes y direcciones desde una sola entrega de archivos. Imaginemos que un solo registro de cliente tiene hasta tres direcciones almacenadas en su historial. Esto significa que se generan hasta cuatro registros para cada cliente. Por ejemplo:

CLIENTE_ID	APELLIDO	DIRECCIÓN_ID	CALLE_DIRECCIÓN	PRIMARIO
1060	Kelly	60455	7310 Calle Respiración	Y

1060 kelley	119885 7310 Breathing Street 65045 57	norte
103317 Anderson	Aguadilla Drive 65518 117 North Union	Y
103317 Anderson	Avenue 61112 27 South Las Vegas	norte
103317 Anderson	Boulevard	norte

He eliminado la mayor parte de los detalles, pero este ejemplo muestra que Kelley tiene dos direcciones en el sistema y Anderson tiene tres. Mi escenario de carga es que necesito agregar un solo registro por cliente a la tabla de clientes, y todos los registros de direcciones deben insertarse en la tabla de direcciones.

Canalización de múltiples tipos de registros desde funciones canalizadas

¿Cómo puede una función canalizada generar un registro de cliente y un registro de dirección al mismo tiempo? Sorprendentemente, hay dos formas relativamente simples de lograr esto:

- Usar tipos de objetos sustituibles (descritos [en capítulo 26](#)). Se pueden canalizar diferentes subtipos de una función en lugar del supertipo en el que se basa la función, lo que significa que cada registro canalizado se puede insertar en su tabla correspondiente en una instrucción condicional INSERT FIRST de varias tablas.
- Use registros anchos y desnormalizados con todos los atributos para cada tabla de destino almacenados en una sola fila canalizada. Cada registro que se canaliza se puede girar en varias filas de datos de destino e insertarse a través de una instrucción INSERT ALL de varias tablas.

Uso de características relacionales de objetos

Echemos un vistazo al primer método, ya que es la solución más elegante para este requisito. Primero necesito crear cuatro tipos para describir mis datos:

- Un objeto “supertipo” para encabezar la jerarquía de tipos. Este contendrá solo los atributos que los subtipos necesitan heredar. En mi caso, este será solo el id_cliente.
- Un tipo de colección de este supertipo. Usaré esto como el tipo de devolución para mi función canalizada.
- Un “subtipo” de objeto de cliente con los atributos restantes requeridos para la carga de la tabla de clientes.
- Un “subtipo” de objeto de dirección con los atributos restantes requeridos para la carga de la tabla de direcciones.

Elegí una pequeña cantidad de atributos con fines de demostración. Mis tipos se ven así:

```
/* Archivo en la web: multitype_setup.sql */
-- Supertipo...
CREAR TIPO cliente_ot COMO OBJETO
(NÚMERO id_cliente
```

```

) NO FINAL;

-- Colección de supertipo...
CREAR TIPO customer_ntt COMO TABLA DE customer_ot;

-- Subtipo de detalle de cliente...
CREAR TIPO customer_detail_ot BAJO customer_ot
(first_name VARCHAR2(20)
, apellido VARCHAR2(60)
, fecha_nacimiento FECHA
) FINALES;

-- Subtipo de detalle de dirección...
CREAR TIPO address_detail_ot BAJO customer_ot
( address_id      NÚMERO
, primario        VARCHAR2(1)
, dirección_calle VARCHAR2(40)
, código_postal  VARCHAR2(10)
) FINALES;

```

Si nunca ha trabajado con tipos de objetos, le sugiero que revise el contenido de [capítulo 26](#). Brevemente, sin embargo, el soporte de Oracle para la sustitución significa que puedo crear filas de customer_detail_ot o address_detail_ot, y usarlas donde se espera el supertipo customer_ot. Entonces, si creo una función canalizada para canalizar una colección del supertipo, esto significa que también puedo canalizar filas de cualquiera de los subtipos. Este es solo un ejemplo de cómo una jerarquía de tipos orientada a objetos puede ofrecer una solución simple y elegante.

Una función segmentada multitype

Echemos un vistazo al cuerpo de la función canalizada y luego explicaré los conceptos clave:

```

/* Archivo en la web: multitype_setup.sql */
1  FUNCIÓN customer_transform_multi(
2      p_fuente EN customer_staging_rct,
3      p_limit_size EN PLS_INTEGER POR DEFECTO customer_pkg.c_default_limit )
4
5      VOLVER cliente_ntt
6      TUBERÍA
7      PARALLEL_ENABLE (PARTICIÓN p_source BY HASH(customer_id))
8      ORDER p_source BY (customer_id, address_id) ES
9
10     fuente_aa      cliente_puesta en escena_aat;
11     v_customer_id customer_staging.customer_id%TYPE := -1; /* Necesita
12     un valor predeterminado no nulo */
13
14     COMENZAR
15         BUCLE
16             FETCH p_source BULK COLLECT INTO aa_source LIMIT p_limit_size; SALIR
17             CUANDO aa_source.COUNT = 0;
18
19     PARA i EN 1 .. aa_source.COUNT BUCLE

```

```

19          /* Canalizar solo la primera instancia de los detalles del cliente... */ IF
20          aa_source(i).customer_id != v_customer_id THEN
21              FILA DE TUBERÍA (customer_detail_ot( aa_source(i).customer_id,
22                                              aa_source(i).first_name,
23                                              aa_source(i).last_name,
24                                              aa_source(i).birth_date ));
25
26          TERMINARA SI;
27
28          FILA DE TUBERÍA( address_detail_ot( aa_source(i).customer_id,
29                                         aa_source(i).address_id,
30                                         aa_source(i).primary,
31                                         aa_source(i).street_address,
32                                         aa_source(i).postal_code ));
33
34          /* Guardar ID de cliente para la lógica de "interrupción de
35             control"... */ v_customer_id := aa_source(i).customer_id;
36
37          FIN DEL BUCLE;
38          FIN DEL BUCLE;
39          CERRAR p_fuente;
40          DEVOLVER;
41      FIN cliente_transform_multi;

```

Esta función está habilitada en paralelo y procesa los datos de origen en matrices para obtener el máximo rendimiento. Los principales conceptos específicos de multitipado se describen en la siguiente tabla.

Líneas)	Descripción
5	El retorno de mi función es una colección del supertipo de cliente. Esto me permite canalizar subtipos en su lugar.
7–8	Tengo dependencias de datos, por lo que he usado la partición hash con transmisión ordenada. Necesito procesar los registros de cada cliente juntos, porque tendrá que seleccionar los atributos del cliente solo del primer registro y luego permitir el paso de todas las direcciones.
21–26	Si este es el primer registro de origen para un cliente en particular, extraiga una fila de CUSTOMER_DETAIL_OT. Solo se canalizará un registro de detalles del cliente por cliente.
28–32	Para cada registro de origen, elija la información de la dirección y extraiga una fila de ADDRESS_DETAIL_OT.

Consulta de una función canalizada de varios tipos

Ahora tengo una sola función que genera filas de dos tipos y estructuras diferentes. Usando SQL*Plus, consultemos algunas filas de esta función:

```

/* Archivo en la web: multitype_query.sql */ SQL>
SELECCIONAR *
2  DE    MESA(
3      paquete_cliente.transform_cliente_multi(
4          CURSOR( SELECT * FROM customer_staging ) ) nt
5  DÓNDE  ROWNUM <= 5;

```

IDENTIFICACIÓN DEL CLIENTE

```
-----  
1  
1  
1  
1  
2
```

Qué sorpresa, ¿dónde están mis datos? Aunque utilicé SELECT *, solo tengo la columna CUSTOMER_ID en mis resultados. La razón de esto es simple: mi función está definida para devolver una colección del supertipo customer_ot, que tiene solo un atributo. Entonces, a menos que codifique explícitamente el rango de subtipos que devuelve mi función, la base de datos no expondrá ninguno de sus atributos. De hecho, si hago referencia a cualquiera de los atributos de los subtipos utilizando el formato de consulta anterior, la base de datos generará un *ORA-00904: identificador no válido* excepción.

Afortunadamente, Oracle proporciona dos formas de acceder a instancias de tipos de objetos: la función VALOR y la pseudocolumna OBJECT_VALUE. Veamos qué hacen (son intercambiables):

```
/* Archivo en la web: multitype_query.sql */  
SQL>SELECCIONE VALOR(nt) COMO object_instance -- podría usar nt.OBJECT_VALUE en su lugar  
2 DE      MESA(  
3          paquete_cliente.transform_cliente_multi(  
4              CURSOR( SELECT * FROM customer_staging ) ) ) nt  
5 DÓNDE  ROWNUM <= 5;  
  
OBJETO_INSTANCIA(CLIENTE_ID)  
-----  
-- CUSTOMER_DETAIL_OT(1, 'Abigail', 'Kessel', '31/03/1949')  
ADDRESS_DETAIL_OT(1, 12135, 'N', '37 North Coshocton Street', '78247')  
ADDRESS_DETAIL_OT(1, 12136, 'N', '47 East Sagadahoc Road', '90285')  
ADDRESS_DETAIL_OT(1, 12156, 'Y', '7 Sur 3er círculo', '30828')  
CUSTOMER_DETAIL_OT(2, 'Anne', 'KOCH', '23/09/1949')
```

Esto es más prometedor. Ahora tengo los datos tal como los devuelve la función canalizada, así que voy a hacer dos cosas con ellos. Primero determinaré el tipo de cada registro usando la condición IS OF; esto me será útil más adelante. En segundo lugar, usará la función TREAT para reducir cada registro a su subtipo subyacente (hasta que haga esto, la base de datos cree que mis datos son del supertipo y, por lo tanto, no me permitirá acceder a ninguno de los atributos). La consulta ahora se parece a esto:

```
/* Archivo en la web: multitype_query.sql */ SQL>  
SELECCIONE EL CASO  
2      CUANDO VALOR(nt) ES DE TIPO (customer_detail_ot)  
3      ENTONCES 'C'  
4      ELSE 'A'  
5      FIN  
6 ,     TREAT(VALUE(nt) COMO customer_detail_ot) COMO cust_rec  
7 ,     TRATAR(VALOR(nt) COMO address_detail_ot) COMO addr_rec  
8 DE      TABLE(
```

```

9          paquete_cliente.transform_cliente_multi(
10         CURSOR( SELECT * FROM customer_staging ) ) nt
11  DÓNDE  ROWNUM <= 5;

RECORD_TYPE CUST_REC                               DIRECCIÓN_REC
-----C
CLIENTE_DETALLE_OT(1, 'Abigail',
'Kessel', '31/03/1949')

A          DIRECCIÓN_DETALLE_OT(1,
12135, 'N', '37 North Coshocton
Street', '78247')

A          DIRECCIÓN_DETALLE_OT(1,
12136, 'N', '47 East Sagadahoc
Road', '90285')

A          DIRECCIÓN_DETALLE_OT(1,
12156, 'Y', '7 Sur 3er Círculo', '3082
8')

C          CLIENTE_DETALLE_OT(2, 'Anne',
'KOCH', '23/09/1949')

```

Ahora tengo mis datos en el formato de subtipo correcto, lo que significa que puedo acceder a los atributos subyacentes. Hago esto ajustando la consulta anterior en una vista en línea y accediendo a los atributos usando la notación de puntos, de la siguiente manera:

```

/* Archivo en la web: multitype_query.sql */
SELECT ilv.record_type
,      NVL(ilv.cliente_rec.cliente_id,
       ilv.addr_rec.customer_id) COMO customer_id
,      ilv.cliente_rec.primer_nombre      como nombre
,      ilv.cust_rec.apellido            como apellido
,      <recorte>
,      ilv.addr_rec.postal_code        AS código_postal
DE   (
      SELECCIONE EL CASO...
      <recorte>
DE   MESA(
      paquete_cliente.transform_cliente_multi(
      CURSOR( SELECCIONE * DESDE cliente_estadificación ) ) nt
) I LV;

```

Cargar varias tablas desde una función canalizada de varios tipos

He quitado algunas líneas del ejemplo anterior, pero deberías reconocer el patrón. Ahora tengo todos los elementos necesarios para una inserción multitable en mis tablas de clientes y direcciones. Aquí está el código de carga:

```

/* Archivo en la web: multitype_setup.sql */
INSERTAR PRIMERO

```

```

CUANDO tipo_registro = 'C'
ENTONCES
    EN clientes
        VALUES (customer_id, first_name, last_name, birth_date)
WHEN record_type = 'A'
ENTONCES
    EN direcciones
        VALORES (address_id, customer_id, principal, street_address,
            Código Postal)
SELECCIONE ilv.record_type
,      NVL(ilv.cliente_rec.cliente_id) COMO customer_id
,      ilv.addr_rec.customer_id COMO customer_id
,      ilv.cliente_rec.primer_nombre      como nombre
,      ilv.cust_rec.apellido            como apellido
,      ilv.cust_rec.birth_date         COMO fecha_nacimiento
,      ilv.addr_rec.address_id        COMO dirección_id
,      ilv.addr_rec.primario          como primario
,      ilv.addr_rec.street_address   AS street_address
,      ilv.addr_rec.postal_code       AS código_postal
DE (
SELECCIONE EL CASO
    CUANDO VALOR(nt) ES DE TIPO (customer_detail_ot)
    ENTONCES 'C'
    ELSE 'A'
    FIN                                como tipo_registro
    TREAT(VALUE(nt) COMO customer_detail_ot) COMO cust_rec
    ,      TRATAR(VALOR(nt) COMO address_detail_ot) COMO addr_rec
DE TABLE(
    paquete_cliente.transform_cliente_multi(
        CURSOR( SELECCIONE * DESDE cliente_estadificación ))) nt
) I LV;

```

Con esta instrucción INSERT FIRST, tengo una carga compleja que utiliza una variedad de características relacionales de objetos de una manera que me permite conservar los principios basados en conjuntos. Este enfoque también podría funcionar para usted.

Un método multtipo alternativo

La alternativa a este método es crear un solo registro de objeto "ancho" y canalizar una sola fila para cada conjunto de direcciones de clientes. Te mostraré la definición de tipo para aclarar lo que quiero decir con esto, pero mira el [configuración_multitipo.sql](#)archivos en el sitio web del libro para el ejemplo completo:

```

/* Archivo en la web: multitype_setup.sql */
CREATE TYPE customer_address_ot AS OBJECT
(customer_id          NÚMERO
, nombre_de_pila     VARCHAR2(20)
, apellido           VARCHAR2(60)
, fecha_de_nacimiento FECHA
, dirección1_dirección_id NÚMERO
, dir1_primario      VARCHAR2(1)
, addr1_street_address VARCHAR2(40)

```

```

, dirección1_código_postal      VARCHAR2(10)
, addr2_address_id             NÚMERO
, dir2_primario                VARCHAR2(1)
, addr2_street_address VARCHAR2(40)
, addr2_postal_code            VARCHAR2(10)
, addr3_address_id             NÚMERO
, dir3_primario                VARCHAR2(1)
, addr3_street_address VARCHAR2(40)
, addr3_postal_code            VARCHAR2(10)
, FUNCIÓN DE CONSTRUCTOR customer_address_ot DEVOLVER
    AUTO COMO RESULTADO
);

```

Puede ver que cada una de las tres instancias de dirección por cliente está "desnormalizada" en sus respectivos atributos. Cada fila canalizada desde la función se pivota en cuatro filas con una instrucción INSERT ALL condicional. La sintaxis INSERT es más simple y, para este ejemplo particular, más rápida que el método de tipo sustituible. La técnica que elijas dependerá de tus circunstancias particulares; tenga en cuenta, sin embargo, que puede encontrar que a medida que aumenta el número de atributos, el rendimiento del método desnormalizado puede degradarse. Habiendo dicho eso, he usado este método con éxito para ajustar una carga que inserta hasta nueve registros en cuatro tablas para cada transacción financiera distinta en una aplicación de ventas.



Puede esperar experimentar una degradación en el rendimiento de la implementación de una función canalizada cuando se utilizan filas anchas o filas con muchas columnas (pertinente al ejemplo de registro múltiple desnormalizado descrito anteriormente). Por ejemplo, probé una carga masiva canalizada en serie de 50 000 filas contra inserciones fila por fila usando varias columnas de 10 bytes cada una. En Oracle9iDatabase, la solución basada en filas se volvió más rápida que la solución segmentada con solo 50 columnas. Afortunadamente, esto aumenta a entre 100 y 150 columnas en todas las versiones principales de Oracle Database 10.gramay base de datos Oracle 11 gramo.

Una palabra final sobre las funciones canalizadas

En esta discusión de las funciones segmentadas, he mostrado varios escenarios en los que dichas funciones (en serie o en paralelo) pueden ayudarlo a mejorar el rendimiento de sus cargas y extracciones de datos. Como herramienta de ajuste, algunas de estas técnicas deberían resultar útiles. Sin embargo, lo hagono ¡Recomiendo que convierta toda su base de código en funciones segmentadas! Son una herramienta específica que probablemente se aplique solo a un subconjunto de sus tareas de procesamiento de datos. Si necesita implementar transformaciones complejas que son demasiado difíciles de manejar cuando se representan en SQL (generalmente como funciones analíticas, expresiones CASE, subconsultas o incluso usando la cláusula MODEL aterradora), entonces encapsúlelas en funciones canalizadas, como he mostrado en esta sección , puede proporcionar beneficios de rendimiento sustanciales.

Técnicas de optimización especializadas

Deberías *siempre* usar FORALL y BULK COLLECT de manera proactiva para todas las operaciones de SQL de filas múltiples no triviales (es decir, aquellas que involucran más de unas pocas docenas de filas). Deberías *siempre* buscar oportunidades para almacenar datos en caché. Y para muchas tareas de procesamiento de datos, debe considerar seriamente el uso de funciones segmentadas. En otras palabras, algunas técnicas son tan ampliamente efectivas que deben usarse en cada oportunidad posible.

Sin embargo, otras técnicas de optimización del rendimiento realmente solo lo ayudarán en circunstancias relativamente especializadas. Por ejemplo: la recomendación de usar el tipo de datos PLS_INTEGER en lugar de INTEGER probablemente no le sirva de mucho a menos que esté ejecutando un programa con una gran cantidad de operaciones con enteros.

Y eso es lo que cubro en esta sección: características relacionadas con el rendimiento de PL/SQL que pueden marcar una diferencia notable, pero solo en circunstancias más especializadas. En general, sugiero que no se preocupe demasiado por aplicar todos y cada uno de estos de forma proactiva. En su lugar, concéntrese en crear código legible y mantenible, y luego, si identifica cuellos de botella en programas específicos, vea si alguna de estas técnicas podría ofrecer algún alivio.

Uso de la sugerencia del modo de parámetro NOCOPY

La sugerencia del parámetro NOCOPY solicita que el motor de tiempo de ejecución de PL/SQL pase un argumento IN OUT por referencia en lugar de por valor. Esto puede acelerar el rendimiento de sus programas, porque los argumentos por referencia no se copian dentro de la unidad de programa. Cuando pasa estructuras grandes y complejas como colecciones, registros u objetos, este paso de copia puede ser costoso.

Para comprender NOCOPY y su impacto potencial, le resultará útil revisar cómo PL/SQL maneja los parámetros. Hay dos formas de pasar valores de parámetros:

Por referencia

Cuando un parámetro actual se pasa por referencia, significa que se pasa un puntero al parámetro actual al parámetro formal correspondiente. Tanto los parámetros reales como los formales luego hacen referencia o apuntan a la misma ubicación en la memoria que contiene el valor del parámetro.

Por valor

Cuando un parámetro actual se pasa por valor, el valor del parámetro actual se copia al parámetro formal correspondiente. Si el programa finaliza sin excepción, el valor del parámetro formal se vuelve a copiar en el parámetro actual. Si se produce un error, los valores modificados no se vuelven a copiar en el parámetro real.

El paso de parámetros en PL/SQL sin el uso de NOCOPY sigue las reglas descritas en la siguiente tabla.

modo de parámetro	Pasado por valor o referencia (comportamiento predeterminado)
EN	Por referencia
AFUERA	Por valor
EN FUERA	Por valor

Puede deducir de estas definiciones y reglas que cuando una estructura de datos grande (como una colección, un registro o una instancia de un tipo de objeto) se pasa como un parámetro OUT o IN OUT, esa estructura se pasará por valor y su aplicación podría experimentar una degradación del rendimiento y de la memoria como resultado de todas estas copias. La sugerencia NOCOPY es una forma de intentar evitar esto. Esta característica encaja en una declaración de parámetros de la siguiente manera:

```
nombre_parámetro
[ EN | DENTRO FUERA | FUERA | ENTRADA FUERA NOCOPY | SIN COPIA ]tipo_datos_parámetro
```

Puede especificar NOCOPY solo junto con el modo OUT o IN OUT. Aquí hay una lista de parámetros que usa la sugerencia NOCOPY para sus dos argumentos IN OUT:

```
PROCEDIMIENTO analizar_resultados (
    date_in EN FECHA,
    valores IN OUT NOCOPY numeros_varray,
    validad_flags IN OUT NOCOPY valid_rectype);
```

Hay dos cosas que debe tener en cuenta acerca de NOCOPY:

- El parámetro real correspondiente para un parámetro OUT bajo la sugerencia NOCOPY se establece en NULL cada vez que se llama al subprograma que contiene el parámetro OUT.
- NOCOPY es un *pista*, no un comando. Esto significa que el compilador podría decidir silenciosamente que no puede cumplir con su solicitud de tratamiento de parámetros NOCOPY. La siguiente sección enumera las restricciones de NOCOPY que pueden causar que esto suceda.

Restricciones en NOCOPY

Varias situaciones harán que el compilador PL/SQL ignore la sugerencia NOCOPY y, en su lugar, use el método por valor predeterminado para pasar el parámetro OUT o IN OUT. Estas situaciones son las siguientes:

El parámetro real es un elemento de una matriz asociativa

Puede solicitar NOCOPY para una colección completa (cada fila de la cual podría ser un registro completo), pero no para un elemento individual de la tabla. Una solución sugerida es copiar la estructura a una variable independiente, ya sea escalar o de registro, y luego pasarla como el parámetro NOCOPY. De esa manera, al menos no estás copiando toda la estructura.

Ciertas restricciones se aplican a los parámetros reales

Algunas restricciones harán que se ignore la sugerencia NOCOPY; estos incluyen una especificación de escala para una variable numérica y la restricción NOT NULL. Sin embargo, puede pasar una variable de cadena que ha sido restringida por tamaño.

Los parámetros reales y formales son estructuras de registro.

Uno o ambos registros se declararon usando %ROWTYPE o %TYPE, y las restricciones en los campos correspondientes en estos dos registros son diferentes.

Al pasar el parámetro real, el motor PL/SQL debe realizar una conversión de tipo de datos implícita

Una solución alternativa sugerida es la siguiente: dado que siempre es mejor realizar conversiones explícitas de todos modos, hágalo y luego pase el valor convertido como el parámetro NO-COPY.

El subprograma que solicita la sugerencia NOCOPY se utiliza en una llamada a procedimiento externo o remoto

En estos casos, PL/SQL siempre pasará el parámetro real por valor.

Beneficios de rendimiento de NOCOPY

Entonces, ¿cuánto puede ayudarte NOCOPY? Para responder a esta pregunta, construí un paquete con dos procedimientos de la siguiente manera:

```
/* Archivo en la web: nocopy_performance.tst */
PAQUETE nocopy_test
ES
    TIPO numeros_t ES TABLA DE NUMERO;

    PROCEDIMIENTO pass_by_value (numbers_inout IN OUT number_t);

    PROCEDIMIENTO pass_by_ref (numbers_inout IN OUT NOCOPY numeros_t); FIN
nocopy_test;
```

Cada uno de ellos duplica el valor en cada elemento de la tabla anidada, como en:

```
PROCEDIMIENTO pass_by_value (numbers_inout IN OUT number_t) IS

COMENZAR
    PARA indx EN 1 .. números_entrada.CUENTO
    BUCLE
        números_entrada(indx) := números_entrada(indx) * 2; FIN DEL
        BUCLE;
    FIN;
```

Luego hice lo siguiente para cada procedimiento:

- Cargó la tabla anidada con 100 000 filas de datos
- Llamó al procedimiento 1000 veces

En base de datos Oracle 10*gramo*, vi estos resultados:

Por valor (sin NOCOPY) - CPU transcurrida: 20,49 segundos. Por referencia (con NOCOPY) - CPU transcurrida: 12,32 segundos.

En base de datos Oracle 11*gramo*, sin embargo, vi estos resultados:

Por valor (sin NOCOPY) - CPU transcurrida: 13,12 segundos. Por referencia (con NOCOPY) - CPU transcurrida: 12,82 segundos.

Realicé pruebas similares en colecciones de cadenas, con resultados similares.

Después de ejecutar pruebas repetidas, concluyo que antes de Oracle Database 11*gramo* puede ver una mejora sustancial en el rendimiento, pero en Oracle Database 11*gramo* esa ventaja se reduce mucho, supongo por el ajuste general del motor PL/SQL en esta nueva versión.

La desventaja de NOCOPY

Dependiendo de su aplicación, NOCOPY puede mejorar el rendimiento de los programas con parámetros IN OUT o OUT. Sin embargo, estas posibles ganancias vienen con una compensación: si un programa termina con una excepción no controlada, no puede confiar en los valores en un parámetro real NOCOPY.

¿Qué quiero decir con *confianza*? Revisemos cómo se comporta PL/SQL con respecto a sus parámetros cuando una excepción no controlada finaliza un programa. Supongamos que paso un registro IN OUT a mi procedimiento de cálculo de totales. El motor de tiempo de ejecución de PL/SQL primero hace una copia de ese registro y luego, durante la ejecución del programa, realiza cualquier cambio en esa copia. El parámetro real en sí no se modifica hasta que `compute_totals` finaliza correctamente (sin propagar una excepción). En ese momento, la copia local se vuelve a copiar en el parámetro real y el programa que llamó a `calcular_totales` puede acceder a los datos modificados. Sin embargo, si `compute_totals` termina con una excepción no controlada, el programa que realiza la llamada puede estar seguro de que el valor real del parámetro no ha cambiado.

Esa certeza desaparece con la sugerencia NOCOPY. Cuando un parámetro se pasa por referencia (el efecto de NOCOPY), cualquier cambio realizado en el parámetro formal también se realiza inmediatamente en el parámetro real. Supongamos que mi programa `calcular_totales` lee una colección de 10 000 filas y realiza cambios en cada fila. Si se genera un error en la fila 5000 y se propaga fuera de `calcule_totals` sin controlar, mi colección de parámetros reales solo cambiará a la mitad.

El `nocopy.tst` archivo en el sitio web del libro demuestra los desafíos de trabajar con NOCOPY. Debe ejecutar este script y asegurarse de comprender las complejidades de esta función antes de usarla en su aplicación.

Más allá de eso, y en general, debe ser juicioso en el uso de la sugerencia NOCOPY. Úselo solo cuando sepa que tiene un problema de rendimiento relacionado con su

paso de parámetros y prepárese para las posibles consecuencias cuando se generen excepciones.



El Product Manager de PL/SQL, Bryn Llewellyn, discrepa conmigo con respecto a NOCOPY. Está mucho más inclinado a recomendar un uso amplio de esta característica. Él argumenta que el efecto secundario de las estructuras de datos parcialmente modificadas no debería ser una gran preocupación, porque esta situación surge solo cuando ha ocurrido un error inesperado. Cuando esto sucede, casi siempre detendrá el procesamiento de la aplicación, registrará el error y propagará la excepción al bloque adjunto. El hecho de que una colección se encuentre en un estado incierto es probable que tenga poca importancia en este momento.

Usar el tipo de datos correcto

Cuando está realizando una pequeña cantidad de operaciones, puede que realmente no importe si el motor PL/SQL necesita realizar conversiones implícitas o si utiliza una implementación relativamente lenta. Por otro lado, si sus algoritmos requieren grandes cantidades de cálculos intensivos, el siguiente consejo podría marcar una diferencia notable.

Evite las conversiones implícitas

PL/SQL, al igual que SQL, realizará conversiones implícitas en muchas circunstancias. En el siguiente bloque, por ejemplo, PL/SQL debe convertir el entero 1 en un número (1.0) antes de agregarlo a otro número y asignar el resultado a un número:

```
DECLARAR
    l_number NÚMERO := 2.0;
    COMENZAR
        l_numero := l_numero + 1;
    FIN;
```

La mayoría de los desarrolladores son conscientes de que las conversiones implícitas realizadas dentro de una instrucción SQL pueden causar una degradación del rendimiento al desactivar el uso de índices. La conversión implícita en PL/SQL también puede afectar el rendimiento, aunque por lo general no tan dramáticamente como en SQL.

ejecutar el *test_implicit_conversion.sql* para ver si puede verificar una mejora en el rendimiento de su entorno.

Use PLS_INTEGER para cálculos enteros intensivos

Cuando declara una variable entera como PLS_INTEGER, usará menos memoria que INTEGER y se basará en la aritmética de la máquina para realizar el trabajo de manera más eficiente. En un programa que requiere cálculos intensivos de enteros, simplemente cambiar la forma en que declara sus variables podría tener un impacto notable en el rendimiento. Ver "El

["Tipo PLSQL_INTEGER" en la página 247](#) para una discusión más detallada de los diferentes tipos de números enteros.

Utilice BINARY_FLOAT o BINARY_DOUBLE para la aritmética de punto flotante

Base de datos Oracle 10*gramo* introdujo dos nuevos tipos de punto flotante: BINARY_FLOAT y BINARY_DOUBLE. Estos tipos se ajustan al estándar de punto flotante IEEE 754 y usan aritmética de máquina nativa, lo que los hace más eficientes que las variables NÚMERO o ENTERO. Ver "[Los tipos BINARY_FLOAT y BINARY_DOUBLE](#)" en la [página 251](#) para detalles.

Optimización del rendimiento de funciones en SQL (12.1 y superior)

Base de datos Oracle 12*C*ofrece dos mejoras significativas para mejorar el rendimiento de las funciones PL/SQL ejecutadas desde dentro de una instrucción SQL:

- La cláusula CON FUNCIÓN
- El pragma UDF

La cláusula WITH FUNCTION se explora en detalle en [capítulo 17](#).

El pragma UDF ofrece un método mucho más simple que CON FUNCIÓN para mejorar el rendimiento de las funciones ejecutadas desde dentro de SQL.

Para aprovechar esta característica, agregue esta declaración a la sección de declaración de su función:

```
PRAGMA UDF;
```

Esta declaración le dice a Oracle: "Planeo llamar a esta función principalmente desde SQL, a diferencia de los bloques PL/SQL". Oracle puede utilizar esta información para reducir el costo de un cambio de contexto de SQL a PL/SQL para ejecutar la función.

El resultado es que la función se ejecutará significativamente más rápido desde SQL (el administrador de productos PL/SQL sugiere que podría ver una mejora de 4X en el rendimiento), pero se ejecutará un poco *Más lento* cuando se ejecuta desde un bloque PL/SQL (!).

El *12c_udf.sql* archivo demuestra el uso de esta función. Este archivo compara el rendimiento de funciones no habilitadas para UDF y habilitadas para UDF. ¡Pruébelo y vea qué tipo de beneficios puede experimentar con esta mejora tan simple!

En pocas palabras: intente usar el pragma UDF primero. Si eso no hace una diferencia lo suficientemente grande en el rendimiento, intente CON FUNCIÓN.

Retrocediendo para ver el panorama general del rendimiento

Este capítulo ofrece numerosas formas de mejorar el rendimiento de sus programas PL/SQL. Casi todos ellos vienen con una compensación: mejor rendimiento por más memoria, mejor rendimiento por una mayor complejidad del código y costos de mantenimiento, y así sucesivamente. Ofrezco estas recomendaciones para garantizar que optimice el código de manera que ofrezca el mayor beneficio tanto para sus usuarios como para su equipo de desarrollo:

- Asegúrese de que sus sentencias SQL estén correctamente optimizadas. El ajuste del código PL/SQL simplemente no puede compensar el arrastre de los escaneos de tablas completos innecesarios. Si su SQL se ejecuta lentamente, no puede solucionar el problema en PL/SQL.
- Asegúrese de que el nivel de optimización de PL/SQL esté establecido en al menos 2. Ese es el valor predeterminado, pero los desarrolladores pueden "meterse" con esta configuración y terminar con un código que el compilador no optimizó por completo. Puede aplicar este nivel de optimización con la directiva \$ERROR de la compilación condicional (cubierta en [capítulo 20](#)).
- Utilice BULK COLLECT y FORALL en cada oportunidad posible. Esto significa que si está ejecutando consultas fila por fila o declaraciones DML, es hora de escribir un montón de código más para introducir y procesar su SQL a través de colecciones. Reescribir los bucles FOR del cursor es menos crítico, pero las construcciones OPEN...LOOP...CLOSE siempre obtendrán una fila a la vez y realmente deberían reemplazarse.
- Esté atento a los conjuntos de datos estáticos y, cuando los encuentre, determine el mejor método de almacenamiento en caché para evitar recuperaciones de datos costosas y repetitivas. Incluso si aún no está utilizando Oracle Database 11*gramo* posterior, comience a encapsular sus consultas detrás de las interfaces de funciones. De esa manera, puede aplicar rápida y fácilmente la caché de resultados de la función cuando actualice su versión de Oracle Database.
- Su código no tiene que ser lo más rápido posible, simplemente tiene que ser lo suficientemente rápido. No se obsesione con la optimización de cada línea de código. En su lugar, priorice la legibilidad y la mantenibilidad sobre el rendimiento espectacular. Consigue que tu código funcione correctamente (cumple con los requisitos del usuario). Luego, haga una prueba de esfuerzo del código para identificar los cuellos de botella. Finalmente, elimine los cuellos de botella aplicando algunas de las técnicas de ajuste más especializadas.
- Asegúrese de que su DBA conozca las opciones de compilación nativas, especialmente en Oracle Database 11*gramo* más alto. Con estas opciones, Oracle compilará de forma transparente el código PL/SQL hasta los comandos de código de máquina.

