Bases de datos

Primer Cuatrimestre de 2013

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Primera parte

Integrante	LU	Correo electrónico
Ángel Abregú	082/09	angelj_a@hotmail.com
Esteban Capillo	484/04	estebancapillo@gmail.com
Martín Rados	185/93	radosm@gmail.com
Mauricio Alfonso	065/09	mauricio.alfonso.88@gmail.com

1. Introducción

En éste trabajo práctico tomamos un motor de bases de datos incompleto llamado UBADB e implementamos el `Buffer Manager` y el uso de múltiples `Buffer Pools`. El `Buffer Manager` es un componente de las bases de datos que se encarga de traer páginas del disco y guardarlas en un buffer que funciona como una caché. El `Buffer Manager` tiene la responsabilidad de borrar páginas de los Buffer si no hay espacio suficiente, esto lo hace con alguna estrategia de reemplazo que puede ser FIFO, LRU, etc.

La implementación de múltiples `Buffer Pools` permite a los usuarios tener más de un buffer, de distintos tamaños y con diferentes estrategias de reemplazo de páginas, e indicar a en qué buffer debe ir cada tabla. De esta manera puede, por ejemplo, mantener siempre en memoria una determinada tabla que se accede constantemente así disminuyendo los accesos a disco, que son el gran cuello de botella de los motores de bases de datos. Para esto el usuario indica con un archivo `XML` el catálogo de la base de datos, indicando para cada tabla que buffer se desea usar.

Además testeamos las clases implementadas usando tests de unidad `JUnit`, y probamos la eficacia de el uso de múltiples buffer pools.

2. Manejo de buffers en Oracle

Oracle dispone de 3 buffer caché diferentes, DEFAULT, KEEP y RECYCLE. Todos ellos funcionan con el algoritmo LRU, determinando el aging de las páginas con el algoritmo Touch Count.

El buffer caché DEFAULT siempre existe, y la mayoría de los sistemas funcionan adecuadamente utilizando sólo ese. Sin embargo, hay dos situaciones que pueden aparecer en algunos sistemas y hacen necesaria la configuración de KEEP y/o RECYCLE buffer caché para tratar de reducir la cantidad de accesos a disco, que es el objetivo principal del buffer caché.

En primer lugar, muchas veces es necesario, por cuestiones de performance mantener un conjunto de objetos siempre en memoria ya que son muy frecuentemente accedidos, en este caso se puede recurrir al KEEP caché, dimensionándolo de manera tal de obtener un alto hit ratio, no necesariamente 100 %, ya que pretender esto podría significar estar desperdiciando memoria que sería útil para los otros pools. Un primer dimensionamiento puede realizarse observando cuántos bloques utiliza el objeto en disco (dato calculado con las estadísticas) y/o analizando en momento de ejecución cuántos bloques de esos objetos hay en memoria. Luego de dimensionado el KEEP caché e indicado cuáles objetos deben ir a él, hay que ir analizando el hit ratio y aumentar o disminuir el tamaño hasta conseguir el hit ratio deseado.

Otra problemática es la de segmentos grandes (>10 % del tamaño del caché DEFAULT) que son accedidos en forma aleatoria o que se escanea en forma completa raramente (por ejemplo un proceso batch que se ejecuta una vez por semana). Esta forma de acceso hace improbable que los bloques leídos en memoria (y que posiblemente hicieron que otros bloques de otros objetos hayan sido removidos del caché) vuelvan a ser utilizados. Para minimizar el impacto, para estos objetos se puede utilizar el RECYCLE caché, ya que no interesa que sean mantenidos en memoria los bloques de los mismos. Generalmente el tamaño del RECYCLE caché es menor al del DEFAULT caché, pero teniendo cuidado de que no sea tan chico que los bloques leidos sean eliminados antes de ser utilizados.

Para indicar a qué buffer caché va un objeto en particular se usa el comando ALTER (ALTER TABLE, ALTER INDEX, etc). Si se cambia el caché de un objeto que ya tiene bloques en memoria, esos bloques quedan en el cache anterior, sólo pasan al nuevo buffer caché cuando son leídos nuevamente de disco.

Los tamaños de los caché están dados por los parámetros de inicialización DB_BUFFER_CACHE, DB_KEEP_CACHE_SIZE y DB_RECYCLE_CACHE_SIZE.

3. Catalog Manager

La clase `CatalogManagerImpl`, que implementa la interfaz `CatalogManager`, se encarga del manejo del catálogo de la base de datos, que se encuentra almacenado físicamente en el disco como archivo XML. El contructor toma dos parámetros, ambos `String`: el primero es la ruta relativa del catálogo y el segundo, la ruta absoluta de un directorio (*prefijo*). Si el prefijo es una cadena vacía, se usará el directorio de trabajo del programa que está corriendo.

El método `loadCatalog` se encarga de cargar el catálogo desde la ruta especificada. Por medio del método `XstreamXmlUtil.fromXml` se lee el archivo XML, se interpreta el contenido y se instancia un objeto de clase `Catalog`.

El segundo método, accede al catálogo cargado y permite obtener el descriptor de la tabla con el `TableId` indicado. Recordemos que un descriptor de tabla (representado por la clase `TableDescriptor`) contiene el `TableId`, el nombre de la tabla visible al usuario, la ubicación física de la tabla en el disco y el nombre el buffer pool asociado a dicha tabla.

El test de unidad correspondiente es `CatalogManagerImplTest`. Se verifica que el catálogo cargado sea el mismo que un catálogo guardado. Primero se crea una instancia de prueba de `Catalog`, y se guarda en disco por medio del método `XstreamXmlUtil.toXml`. Luego, se crea una instancia de `CatalogManagerImpl`, y se procede a cargar el catálogo desde la ruta previamente establecida. Los tests chequean la igualdad entre el primer catálogo y el que fue cargado; para eso se debe testear que dado un `TableId`, se devuelva un descriptor de tabla válido (i.e., no `null`).

4. Múltiples Buffer Pools

Implementamos la clase `MultipleBufferPool` para que el usuario pueda elegir entre varios buffers en vez de tener acceso a uno solo. Para esto fue necesario modificar el catálogo y el descriptor de tabla para que se indique por cada tabla a que pool pertenece. `MultipleBufferPool` recibe en su constructor 3 parámetros: el primero es un mapa (diccionario) que por cada pool (representado con un `String`) tiene un entero indicando su tamaño máximo; el segundo es también un mapa que por cada pool tiene una estrategia de reemplazo de páginas (`PageReplacementStrategy`); por último necesita una referencia al `CatalogManager`.

Los métodos para acceder a las páginas en el `MultipleBufferPool` son los mismos de la interfaz `BufferPool`. Para obtener el buffer correspondiente a partir de un `Page` o `PageId`, el `MultipleBufferPool` le pregunta al `CatalogManager` el descriptor de tabla de dicha página, y con ése descriptor obtiene el `String` que representa al pool correspondiente.

Además fueron implementados tests de unidad en la clase `MultipleBufferPoolTest` para verificar su correcto funcionamiento. Para dichos tests usamos un catálogo con dos pools llamados `POOL_1` y `POOL_2` y 4 páginas para cada uno. Se prueban cada uno de los métodos públicos, en 14 tests de diferentes situaciones.

5. Evaluación de Múltiples Buffer Pools

Para evaluar la eficacia de los múltiples Buffer Pools generamos trazas que usan tablas para diferentes pools y comparamos el hit rate del acceso a buffer usando múltiples Buffer Pools con el de usar un Buffer Pool simple. Para eso primero generamos trazas individuales sobre un catálogo con tres tablas llamadas `Materias`, `Cursadas` y `Estudiantes`. `Materias` es una tabla chica que se va a acceder mucho, mientras que `Cursadas` es una tabla más grande a la que se accede menos y `Estudiantes` es una tabla mediana. El catálogo de dichas tablas lo generamos con la clase `CatalogGenerator`.

En la clase `SeqMainTraceGenerator` generamos una traza que accede primero a todas las páginas de `Materias`, luego a todas las de `Cursada`, luego a la de `Estudiantes` y por último accedemos nuevamente a todas las páginas de `Materias` y `Estudiantes`. Dicha traza se genera en el archivo `generated_seq/prueba.trace`. Con la clase `MainTraceGenerator`, generamos otra traza similar, pero con accesos aleatorios a ambas tablas, y lo guarda en el archivo `generated/prueba.trace`.

Luego probamos en `MainEvaluator` el hit rate de ambas trazas usando `SingleBufferPool` y `MultipleBufferPool`. Para `MultipleBufferPool` usamos 3 buffers: uno grande llamado `DEFAULT`, y dos pequeños llamados `KEEP` (donde guardamos las páginas de Materias) y `RECYCLE` (donde guardamos las páginas de `Estudiantes`). Para `SingleBufferPool` usamos un único Buffer Pool, de tamaño igual a las suma de los 3 buffers anteriores, para asegurarnos de tener una comparación justa.

Notamos que se puede mejorar el hit rate al usar `MultipleBufferPools`, sólo si los buffers tienen tamaño adecuado.

Los hit rates obtenidos pueden no representar una mejora considerable, pero es importante mencionar que corresponden a todos los accesos, es decir a todos los pools. Seguramente el hit rate correspondiente al pool `KEEP` tiene un porcentaje de hits muy alto, lo cuál mejora levemente el hitrate total.

6. Conclusiones

Pudimos corroborar que en algunos casos se puede mejorar el hit rate de los Buffer Pools, lo cuál significa una reducción en los accesos a disco y una mejora en los tiempos de ejecución del motor de bases de datos. Sin embargo el uso correcto de múltiples buffer pools requiere que el usuario conozca qué páginas se leen con más o menos frecuencia, y si es buena idea o no gastar memoria para páginas que sólo usan ciertas páginas. Es decir, el usuario debe tener información adicional que no tienen las estrategias de reemplazo de páginas.

Es importante mencionar que si no se usan adecuadamente, los Buffer Pools Múltiples pueden empeorar la eficacia del motor, en vez de mejorarlo. Por ejemplo notamos en nuestra prueba con dos tablas que si asignamos al buffer que llamamos `KEEP` un tamaño menor al de la tabla que queremos guardar, obtenemos un hit rate menor que usando un sólo Buffer Pool grande.