Bases de Datos

Trabajo Práctico Nro. 2

Para este trabajo se partió de un motor de bases de datos en desarrollo, ***UBADB***, para cuyo Buffer Manager se implementaron dos estrategias nuevas de reemplazo de páginas: LRU (least recently used) y MRU (most recently used), además de la ya existente FIFO (first in, first out).

A continuación se presenta una descripción breve de lo desarrollado, y luego un breve análisis de la performance de las distintas estrategias para cuatro patrones de uso típicos: *file scan, index scan clustered, index scan unclustered* y *block nested loop join*.

IMPLEMENTACIÓN

El código relativo a las estrategias se encuentra en el paquete *ubadb.components.bu9erManager.bu9erPool.replacementStrategies.*

Se observó que las tres estrategias tenían lógica en común: en todas se buscaba encontrar el frame que maximizara cierta propiedad. Algorítimicamente, se debía iterar sobre todos los frame, tomando como candidato a víctima aquel que resultara de comparar contra el resto.

Con esto en cuenta, se decidió extraer este comportamiento en común y dejar para cada estrategia definir cómo se realizaba la comparación de frames. Esto se llevó a cabo utilizando lo que se conoce como *template method*, se definió el esqueleto del algoritmo en una clase abstracta, llamada *FrameComparisonReplacementStrategy*, que llama a un método abstracto que realiza la comparación. De esta forma, cada implementación sólo debía definir cuándo un frame reemplaza a otro como candidato a víctima:

- FIFO: un frame reemplaza a otro si tiene fecha de creación anterior
- LRU: un frame reemplaza a otro si tiene fecha de última referencia anterior
- MRU: un frame reemplaza a otro si tiene fecha de última referencia posterior

Cabe destacar que hasta los frames no guardan la información de cuál fue su fecha de última referencia, por lo cual las nuevas estrategias de reemplazo utilizan un tipo de frame distinto al genérico: *TrackedUsageBu9erFrame*. Al igual que *FIFOBu9erFrame*, éste agrega un campo de tipo *java.util.Date*, que en este caso representa el dato que nos faltaba.

A continuación se pueden ver extractos del código que muestran lo realizado:

```
public abstract class FrameComparisonReplacementStrategy implements PageReplacementStrategy {

    public BufferFrame findVictim(Collection<BufferFrame> bufferFrames) throws PageReplacementStrategyException {
        BufferFrame victim = null;

        for(BufferFrame bufferFrame : bufferFrames) {
            if(bufferFrame.canBeReplaced() && (victim == null || shoudReplace(victim, bufferFrame))) {
                victim = bufferFrame;
            }
        }

        if(victim == null)
            throw new PageReplacementStrategyException("No page can be removed from pool");
        else
            return victim;
    }

    protected abstract boolean shoudReplace(BufferFrame currentVictim, BufferFrame frame);

}
```

Extracto 1: FrameComparisonReplacementStrategy, con el esqueleto del algoritmo de comparación

```
public class FIFOReplacementStrategy extends FrameComparisonReplacementStrategy implements PageReplacementStrategy
{
    @Override
    protected boolean shoudReplace(BufferFrame currentVictim, BufferFrame frame) {
        FIFOBufferFrame current = (FIFOBufferFrame) currentVictim;
        FIFOBufferFrame other = (FIFOBufferFrame) frame;

        return other.getCreationDate().before(current.getCreationDate());
    }

        ...
}
```

Extracto 2: FIFOReplacementStrategy, que compara fechas de creación

```
public class LRUReplacementStrategy extends RecentUsageReplacementStrategy implements PageReplacementStrategy {

    @Override
    protected boolean shoudReplace(BufferFrame currentVictim, BufferFrame frame) {
        TrackedUsageBufferFrame current = (TrackedUsageBufferFrame) currentVictim;
        TrackedUsageBufferFrame other = (TrackedUsageBufferFrame) frame;
        return other.getLastReferenceDate().before(current.getLastReferenceDate());
    }

        ...
}
```

Extracto 3: LRUReplacementStrategy, que devuelve el frame con fecha de última referencia más antigua

```
public class MRUReplacementStrategy extends RecentUsageReplacementStrategy implements PageReplacementStrategy {

    @Override
    protected boolean shoudReplace(BufferFrame currentVictim, BufferFrame frame) {
        TrackedUsageBufferFrame current = (TrackedUsageBufferFrame) currentVictim;
        TrackedUsageBufferFrame other = (TrackedUsageBufferFrame) frame;
        return other.getLastReferenceDate().after(current.getLastReferenceDate());
    }

}
```

Extracto 4: MRUReplacementStrategy, que devuelve el frame con fecha de última referencia más reciente

EVALUACIÓN DE LAS ESTRATEGIAS

Como se dijo anteriormente, se deseaba evaluar el desempeño de las tres estrategias en casos de uso bastante comunes para un motor de bases de datos: *file scan, index scan clustered, index scan unclustered* y *block nested loop join.*

Se contaba con un generador, que según ciertos parámetros generaba los pedidos de páginas para trazas representativas de los casos anteriores.

Se modificó la clase *ubadb.apps.bu9erManagement.Bu9erManagementEvaluator* para generar los resultados presentados a continuación.

File Scan e Index Scan Clustered

Estos casos presentan la particularidad de que en ningún momento se requiere volver a pedir páginas previamente utilizadas. *File Scan* recorre secuencialmente las páginas de un archivo, mientras que *Index Scan Clustered* hace un recorrido por el índice para localizar el archivo, y luego leerlo (también secuencialmente). Se esperaba, entonces, que el hit-rate se mantuviera en 0 para cualquier medición que realizáramos. Las mediciones corroboraron lo esperado: no hubo *hits* en el Buffer Manager para ninguna de las estrategias en pedidos de tamaños variados.

Block Nested Loop Join

En este algoritmo de *join*, se carga una porción de la tabla *outer* a memoria, y se deja fija mientras se escanea secuencialmente toda la relación *inner.* Al finalizar, se realiza otra iteración, descargando de memoria la porción anterior y cargando una nueva de la tabla *outer.*

Se observó entonces que nunca iba a hacer falta pedir más de una vez ninguna página de *outer*, ya que mientras durara la iteración correspondiente a esa porción las páginas están marcadas como no reemplazables, por lo cuál nunca se bajan de memoria.

Se decidió realizar las pruebas con un tamaño de buffer fijo de 30 frames, dejando siempre porciones de tamaño 2 para *outer,* ya que se consideró que lo importante era variar el espacio disponible en buffer

Por lo tanto, la experimentación debía pasar por variar el espacio disponible en buffer para los bloques de la tabla *inner.* Se tomó un tamaño fijo de buffer (30 bloques) y se fue variando el tamaño asignado a las porciones de *ouHer*, para evaluar el desempeño según cuánto espacio quedaba disponible para la tabla inner. A su vez, se repitió el experimento para varios tamaños de *ouHer*, ya que a mañor tamaño se debían realizar más iteraciones.

Se presentan a continuación los resultados, en donde se grafica para distinta cantidad de iteraciones el hit-rate en

función de cuál es la diferencia entre el espacio libre en buffer y el tamaño de la tabla *inner.* Es decir,

$$diferencia = (\text{Tamaño}_{buffer} - \text{Tamaño}_{bloqueOuter}) - \text{Tamaño}_{inner}$$

Valores positivos de *diferencia* significan que la tabla inner no entra entera en el buffer, por lo cual se deberán liberar frames dentro de una misma iteración. Cuando *diferencia* es menor o igual a 0, significa que entra la tabla entera, por lo cual no hará falta liberar bloques de la tabla a lo largo del algoritmo.

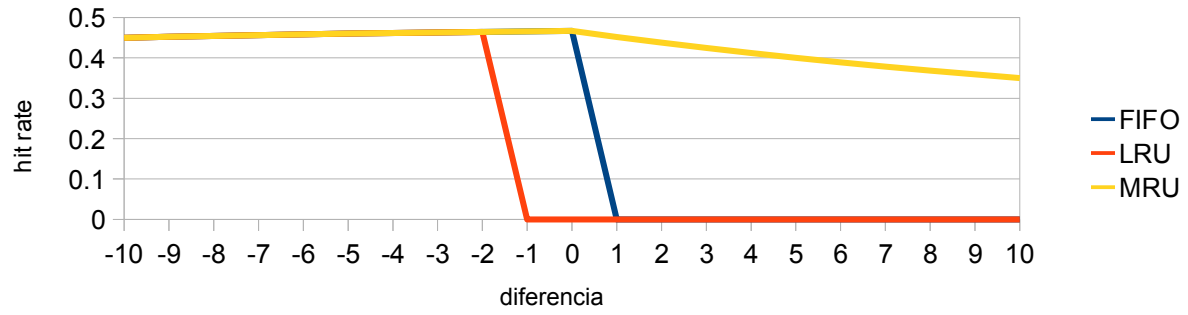

Gráfico 1: Evaluación de BNLJ con 2 iteraciones

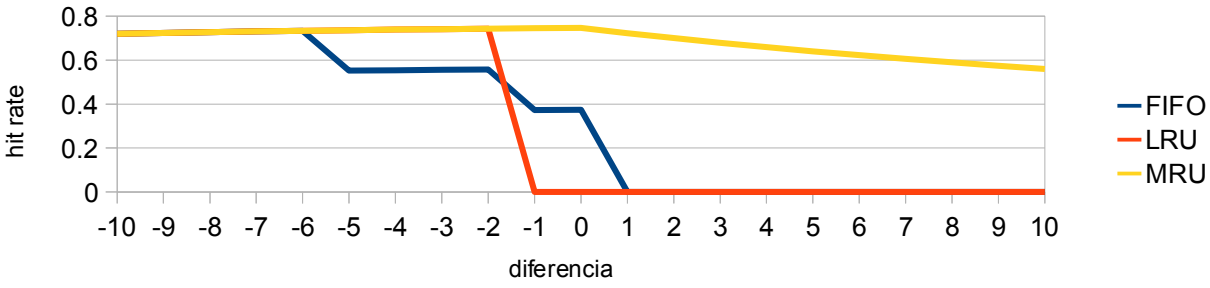

Gráfico 2: Evaluación de BNLJ con 5 iteraciones

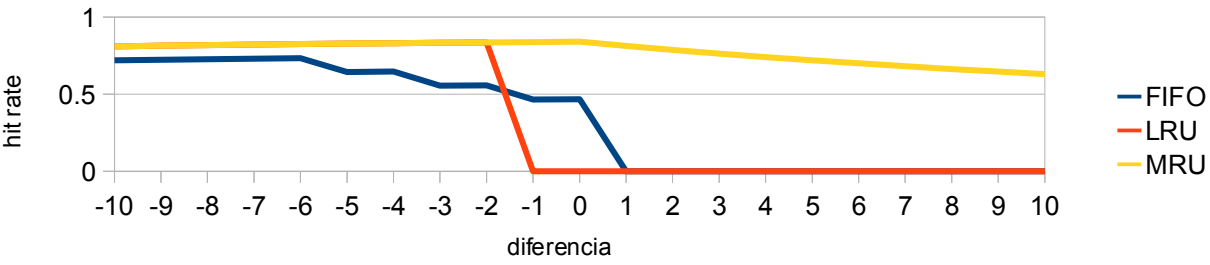

Gráfico 3: Evaluación de BNLJ con 10 iteraciones

Se ve un comportamiento similar a medida que aumenta la cantidad de iteraciones. En todos los casos el mayor hit-rate es alcanzado por la estrategia *MRU.* Para valores pequeños de *diferencia* (cuando la tabla *inner*) cable completamente en buffer las otras dos estrategias logran también valores similares. Sin embargo, en el momento que es necesario empezar a reemplazar se ve que tanto *LRU* como *FIFO* caen súbitamente a 0.

Esto se puede explicar sencillamente con un ejemplo. Supongamos que tenemos un buffer de 5 frames, con dos frames destinados a los bloques de *outer*. Asumamos también que *inner* ocupa un total de 4 bloques. Esto nos da un espacio de 3 frames que se utilizarán para ubicar bloques de *inner* a lo largo de cada iteración.

Evidentemente, los primeros tres bloques (IN_1, IN_2, IN_3) de *inner* serán ubicados en los estos frames libres. Al traer de disco el siguiente bloque (IN_4), se apela a la estrategia de reemplazo para evaluar cuál de los bloques se debe reemplazar. Tanto *FIFO* como *LRU* elegirán IN_1, ya que es tanto la más antigua como la que menos recientemente se usó, por lo cuál este bloque sale del buffer y se reemplaza con IN_4.

Como ya se terminó de recorrer *inner*, se reemplaza los bloques de *outer* por otros nuevos, y se comienza a recorrer *inner* nuevamente desde el principio. En este momento tenemos en buffer los bloques IN_2, IN_3, IN_4 , pero se desea traer de disco IN_1. Aquí ambas estrategias harán que se elimine del buffer IN_2. Se ve aquí cómo *FIFO* y *LRU* siempre optarán por bajar del buffer justo el próximo bloque a pedir, lo que causa el hit rate igual a 0.

Por su parte, *MRU* siempre bajará del buffer el último bloque usado, que será el último en ser pedido nuevamente. Para este algoritmo, *MRU* siempre tiende a conservar los bloques que se volverán a pedir más prontamente, lo que explica el hit elevado.

Index Scan Unclustered

Las trazas generadas en este patrón de uso siempre comienzan con un recorrido por el índice, tras lo cuál se sigue una serie de pedidos aleatorios sobre los bloques correspondientes a un archivo. Para evaluar las distintas estrategias se realizaron mediciones para distintos tamaños de buffer y cantidades de bloques a leer del archivo. Los resultados pueden verse en *Gráfico 4*.

A diferencia del caso anterior, no se ve aquí que ninguna estrategia sea claramente superior al resto. Esto se debe a la naturaleza aleatoria de las trazas: los *hits* se producirán con pedidos de bloques ya presentes en buffer, pero sin importar la estrategia siempre podría ocurrir que el próximo bloque pedido sea el que se acaba de borrar.

Se ve una tendencia a hit-rates mayores cuando se aumenta el tamaño del buffer, pero esto se debe sólamente a que al ser más grande el buffer aumenta la probabilidad de que los bloques pedidos estén en memoria.

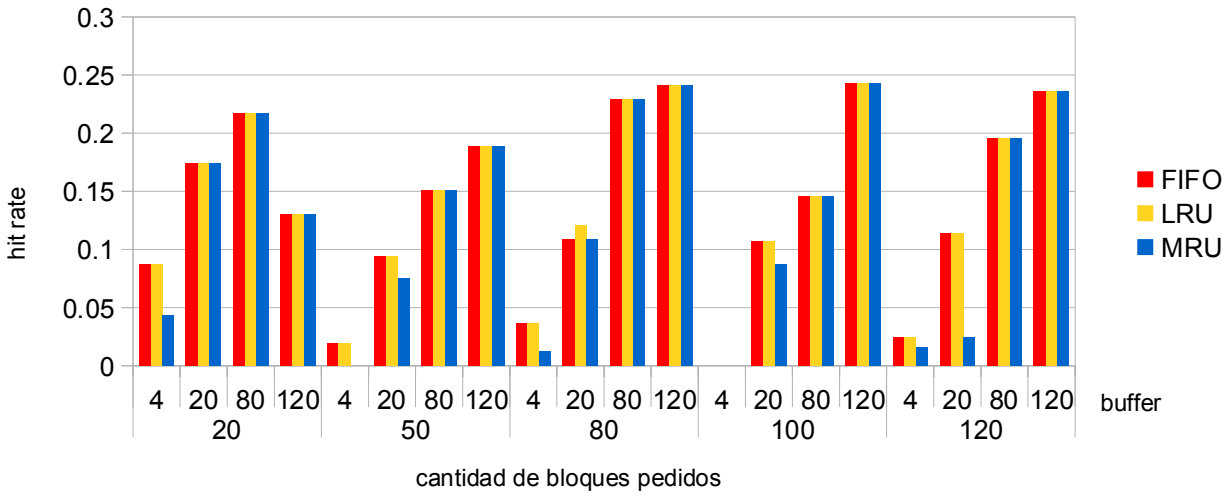

Gráfico 4: Evaluación de Index Scan Unclustered