DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

15 / 11 / 2013 Bases De Datos

Grupo 4

Integrante	LU	Correo electrónico
Carreiro, Martin	45/10	martin301290@gmail.com
Kujawski, Kevin	459/10	kevinkuja@gmail.com
Ortiz De Zarate, Juan Manuel	403/10	jmanuoz@gmail.com
Teren, Leonardo	332/09	lteren@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Introducción

El Buffer Manager es uno de los componentes más importantes dentro de un motor de BD. Su principal función es administrar un espacio de memoria de la BD, utilizado como una especie de memoria caché. El objetivo es que las diferentes aplicaciones que usan la BD y requieren páginas de disco, puedan recuperar la página de este espacio de memoria y accedan lo menos posible al disco. El espacio de memoria administrado por el Buffer Manager puede ser organizado de diferentes formas y la estrategia para decidir cuál página reemplazar cuando ya no queda más espacio también puede variar.

Este trabajo se encarga de implementar los algoritmos Touch Count, LRU y MRU, y compararemos su comportamiento a partir de posibles situaciones.

2. LRU

2.1. Descripción

El algoritmo Least Recently Used (LRU) descarta primero los elementos menos usados recientemente. El algoritmo lleva el seguimiento de lo que se va usando, lo que resulta caro si se quiere hacer con precisión. La implementación de esta técnica requiere llevar la cuenta de la edad de cada elemento de caché y buscar el menos usado en base a ella. En una implementación como esa, cada vez que se usa un elemento, la edad de todos las demás elementos cambia.

2.2. Implementación

2.2.1. LRU Buffer Frame

```
package ubadb.core.components.bufferManager.bufferPool.replacementStrategies.lru;

import java.util.Date;

import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import ubadb.core.exceptions.BufferFrameException;

public class LRUBufferFrame extends BufferFrame
{
                private Date referencedDate;

        public LRUBufferFrame(Page page) {
                super(page);
                referencedDate = new Date();
        }

        public Date getReferencedDate()
        {
                return referencedDate;
        }

        public void pin(){
                super.pin();
                referencedDate = new Date();
        }

        public void unpin() throws BufferFrameException{
                super.unpin();
                referencedDate = new Date();
        }


}
```

2.2.2. LRU Buffer Frame Strategy

```
package ubadb.core.components.bufferManager.bufferPool.replacementStrategies.lru;
```

```java
import java.util.Collection;
import java.util.Date;

import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import ubadb.core.components.bufferManager.bufferPool.
        replacementStrategies.PageReplacementStrategy;
import ubadb.core.exceptions.PageReplacementStrategyException;

public class LRUReplacementStrategy implements PageReplacementStrategy
{
    public BufferFrame findVictim(Collection<BufferFrame> bufferFrames) throws
       PageReplacementStrategyException
    {
        LRUBufferFrame victim = null;
                Date oldestReplaceablePageDate = null;

                for(BufferFrame bufferFrame : bufferFrames)
                {
                        LRUBufferFrame lruBufferFrame = (LRUBufferFrame)
                           bufferFrame; //safe cast as we know all frames are
                           of this type
                        if(lruBufferFrame.canBeReplaced() &&
                           (oldestReplaceablePageDate==null ||
                                lruBufferFrame.getReferencedDate().
                                        before(oldestReplaceablePageDate)))
                        {
                                victim = lruBufferFrame;
                                oldestReplaceablePageDate =
                                   lruBufferFrame.getReferencedDate();
                        }
                }

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

    public BufferFrame createNewFrame(Page page)
    {
        return new LRUBufferFrame(page);
    }

    public String toString() {
        return "LRU Replacement Strategy";
    }
}
```

3. MRU

3.1. Descripción

MRU descarta primero -al contrario de LRU- los elementos más usados recientemente. Los algoritmos MRU son los más útiles en situaciones en las que cuanto más antiguo es un elemento, más probable es que se acceda a él.

3.2. Implementación

3.2.1. MRU Buffer Frame

```
package ubadb.core.components.bufferManager.bufferPool.replacementStrategies.mru;
import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import ubadb.core.exceptions.BufferFrameException;

import java.util.Date;

public class MRUBufferFrame extends BufferFrame
{
                private Date referencedDate;

        public MRUBufferFrame(Page page) {
                super(page);
                referencedDate = new Date();
        }

        public Date getReferencedDate()
        {
                return referencedDate;
        }

        public void pin(){
                super.pin();
                referencedDate = new Date();
        }

        public void unpin() throws BufferFrameException{
                super.unpin();
                referencedDate = new Date();
        }

}
```

3.2.2. MRU Buffer Frame Strategy

```
package ubadb.core.components.bufferManager.bufferPool.replacementStrategies.mru;

import java.util.Collection;
import java.util.Date;

import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import ubadb.core.components.bufferManager.bufferPool.
        replacementStrategies.PageReplacementStrategy;
import ubadb.core.exceptions.PageReplacementStrategyException;

public class MRUReplacementStrategy implements PageReplacementStrategy
{
    public BufferFrame findVictim(Collection<BufferFrame> bufferFrames) throws
       PageReplacementStrategyException
    {
        MRUBufferFrame victim = null;
                Date newestReplaceablePageDate = null;
```

```
                for(BufferFrame bufferFrame : bufferFrames)
                {
                        MRUBufferFrame mruBufferFrame = (MRUBufferFrame)
                           bufferFrame; //safe cast as we know all frames are
                           of this type
                        if(mruBufferFrame.canBeReplaced() &&
                           (newestReplaceablePageDate==null
                                || mruBufferFrame.getReferencedDate().
                                        after(newestReplaceablePageDate)))
                        {
                                victim = mruBufferFrame;
                                newestReplaceablePageDate =
                                   mruBufferFrame.getReferencedDate();
                        }
                }

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

    public BufferFrame createNewFrame(Page page)
    {
        return new MRUBufferFrame(page);
    }

    public String toString() {
            return "MRU Replacement Strategy";
    }
}
```

4. Touch Count

4.1. Motivación

Tanto la estrategia MRU como la LRU aplican un mal uso de la caché en los casos en que se realizan full scans de las tablas debido a que se cargan todos los bloques de éstas en el buffer. Por ejemplo, si el buffer es de 500 bloques y la tabla contiene 600, todos los bloques populares serán removidos y reemplazados con los bloques de esta tabla. Esto es sumamente contraproducente para la consistencia de la performance de la base de datos ya que fuerza un excesivo uso del sistema y destruye una cache bien conformada.
Para solucionar este problema existe la famosa estrategia MLRU (modified least recently used). Esta lo que hace es ubicar los bloques leídos por un full-scan al final de la cola limitando, adicionalmente, la cantidad de bloques para ello.
Sin embargo esta estrategia no funciona para index-scan y si debemos realizar dicho sobre un amplio rango volvemos al mismo problema. Para esto se pensó la estrategia Touch-Count, que será desarrollada a lo largo de esta sección.

4.2. Descripción

El algoritmo que utiliza Oracle para manejar las páginas del Buffer Pool es conocido como “Touch Count” y es una variante del popular LRU.

4.2.1. Hot N Cold

Este algoritmo contiene una lista de buffers dividida en dos secciones, una denominada fría (cold) y otra caliente (hot), donde ambas tienen la misma cantidad de bloques (en caso de ser impar ésta, la zona fría tiene ese bloque de más)
La idea es que esta lista se mantenga balanceada en base a cuántos accesos tuvo, manteniendo en la sección caliente los más ”populares”.
Cada vez que se agrega un nuevo bloque a la lista, este se introduce en el medio, siempre en la parte fría. En caso de quedar esta con una diferencia mayor a 1 con respecto a la caliente se debe balancear nuevamente la lista.

4.2.2. Increase Touch Count

Los accesos a los bloques se contabilizan a través de un valor denominado ”Touch Count”que en la práctica no es estrictamente la cantidad de acccesos. Por ejemplo este valor sólo puede ser actualizado una vez cada 3 segundos para no sobrecargar los accesos a memoria y también por practicidad el valor sólo es incrementando en el unpin (cuando se ”suelta.[el] bloque), porque sino estarían duplicados innecesariamente. Además estos valores son “reseteados“ cuando un bloque se mueve de zona.

4.2.3. Hot N Cold Movement

Un bloque pasa de la zona fría a la caliente y su Touch Count se reinicia a 0 cuando éste es mayor a 2 (o durante un balanceo como se mencionó anteriormente). Además el último bloque hot también es movido y su Touch Count pasa a 1.

4.3. Implementación

4.3.1. Touch Count Buffer Frame

```
package
    ubadb.core.components.bufferManager.bufferPool.replacementStrategies.touchcount;

import java.util.Date;

import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import ubadb.core.exceptions.BufferFrameException;

public class TouchBufferFrame extends BufferFrame implements
    Comparable<TouchBufferFrame>{

                public Integer count;
                public Date lastTouch;
```

```
        public TouchBufferFrame(Page page) {
                super(page);
                count = 0;
                lastTouch = new Date();
        }

        public void pin(){
                super.pin();
                increaseCount();
        }

        public void unpin() throws BufferFrameException{
                super.unpin();
                increaseCount();
        }

        public void increaseCount(){
                Date now = new Date();

                @SuppressWarnings("unused")
                        long difference = (long) ((now.getTime() -
                           lastTouch.getTime())/1000);

                if((now.getTime() - lastTouch.getTime())/1000 >= 3){
                        count++;
                        lastTouch = new Date();
                }
        }

                @Override
                public int compareTo(TouchBufferFrame arg0) {
                        return count.compareTo(((TouchBufferFrame)arg0).count);
                }

}
```

4.3.2. Touch Count Buffer Frame Strategy

```
package
   ubadb.core.components.bufferManager.bufferPool.replacementStrategies.touchcount;

import java.util.Collection;
import java.util.LinkedList;

import ubadb.core.common.Page;
import ubadb.core.components.bufferManager.bufferPool.BufferFrame;
import
   ubadb.core.components.bufferManager.bufferPool.replacementStrategies.PageReplacementS
import ubadb.core.exceptions.PageReplacementStrategyException;

public class TouchCountReplacementStrategy implements PageReplacementStrategy {

                private LinkedList<TouchBufferFrame> cold;
                private LinkedList<TouchBufferFrame> hot;

                public TouchCountReplacementStrategy(){
                        cold = new LinkedList<TouchBufferFrame>();
                        hot = new LinkedList<TouchBufferFrame>();
                }

        public BufferFrame findVictim(Collection<BufferFrame> bufferFrames)
           throws PageReplacementStrategyException {
                        hotNColdMovement(); // Mover segun parametro del paper.
```

```
                TouchBufferFrame victim = firstColdFrame();
        return victim;
}

//Devuelvo el primer COLD Frame reemplazable
private TouchBufferFrame firstColdFrame() throws
    PageReplacementStrategyException{
        for (TouchBufferFrame bufferFrame : cold) {
                if (bufferFrame.canBeReplaced()){
                        cold.remove(bufferFrame);

                        if(Math.abs(cold.size() - hot.size())>0){ //Si
                            HOT es mzs grande
                                cold.addLast(hot.removeLast());
                                    //Mantener balanceo
                        }//Si no, es que son iguales. No puede haber mas
                            de 1 de diferencia

                        return bufferFrame;
                }
        }
        throw new PageReplacementStrategyException("No hay Cold Buffer
            reemplazable");
}

//Una vez que tengamos cold y hoy, pasar de cold a hot segun parametro
    del paper
private void hotNColdMovement(){
                LinkedList<TouchBufferFrame> coldToRemove = new
                    LinkedList<TouchBufferFrame>();
                LinkedList<TouchBufferFrame> hotToRemove = new
                    LinkedList<TouchBufferFrame>();
        for(TouchBufferFrame bufferFrame : cold){
                if(bufferFrame.count >= 2 ){ // Dato del paper
                        bufferFrame.count = 0; //Se tiene que resetear

                        coldToRemove.addLast(bufferFrame);

                        TouchBufferFrame toColdToMantainBalanceFrame =
                            hot.removeLast();
                                toColdToMantainBalanceFrame.count = 1;
                                    //Si no en la proxima iteracion lo
                                    va a mandar a Hot de vuelta
                                hotToRemove.addLast(toColdToMantainBalanceFrame)
                                    //Mantener balanceo
                }
        }

        //Elimino los que tienen count>2 de cold
        cold.removeAll(coldToRemove);
        //Concateno de forma tal que quede al principio coldToRemove y
            despues hot
        coldToRemove.addAll(hot);
        //Hot es esta nueva lista
        hot = coldToRemove;

        //MantenerBalanceo agregando los que saque de Hot al final de
            cold
        cold.addAll(hotToRemove);
}

public BufferFrame createNewFrame(Page page) {
        TouchBufferFrame bufferFrame = new TouchBufferFrame(page);
```

```
                cold.addFirst(bufferFrame);

                if (Math.abs(cold.size() - hot.size())>1){ //Mantener balanceo,
                    maxima diferencia 1
                        TouchBufferFrame coldToHodFrame = cold.removeLast();
                        hot.addLast(coldToHodFrame);
                }

                return bufferFrame;
        }

        public String toString() {
            return "Touch Count Replacement Strategy";
        }
}
```

5. Comparaciones

Para analizar el comportamiento de las estrategias, generamos distintos patrones de acceso a la base de datos (trazas). Corrimos las diferentes estrategias para cada traza, haciendo variar el tamaño del Pool. Para cada una de estas corridas, calculamos el hitrate de cada estrategia. Luego representamos estos resultados en gráficos comparativos que utilizamos para verificar nuestras hipótesis.

5.1. Generación de Trazas

Touch Count intenta resolver casos en los que tanto LRU como MRU, algoritmos predecesores, no tienen una buena performance, por lo que generamos trazas imitando el comportamiento apropiado de dichos casos. Éstas fueron generadas con el mixer otorgado por la cátedra y con trazas previamente generadas.

A continuación, se explica cada una de ellas

- La primera traza combina distintos tipos de acceso pero con una relevancia mayor al FileScan. (referirse a Motivación de Touch Count pará más información)
- La segunda traza, al igual que el anterior, combina otros tipos, pero esta vez su mayoría radica en IndexScan
- La tercera traza realiza solo FileScan ya que es el problema principal a resolver dentro de LRU, y como se comporta delante la decisión de separación del buffer en Hot y Cold

5.2. Análisis

Cada una de las trazas antes mencionadas fueron probadas ante distintos tamaños de buffer para comparar los HitRate obtenidos

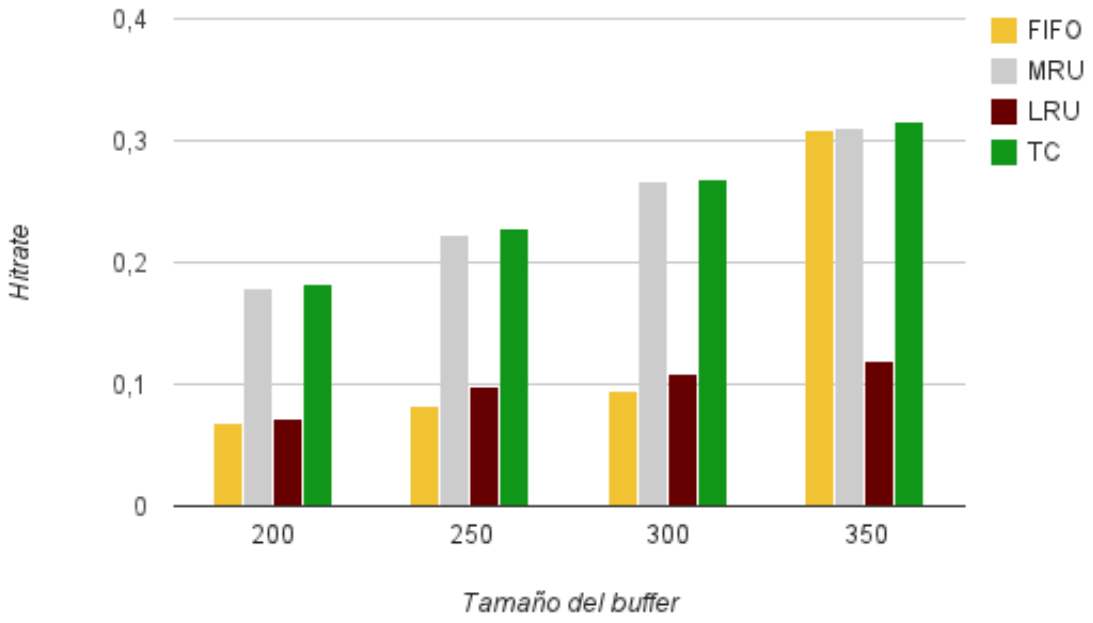

Figura 1: Mixed Trace con más File Scan

Como veníamos prediciendo, Touch Count muestra una mejor performance frente al resto. Si bien las diferencias son pequeñas, se mantienen para los distintos tamaños.

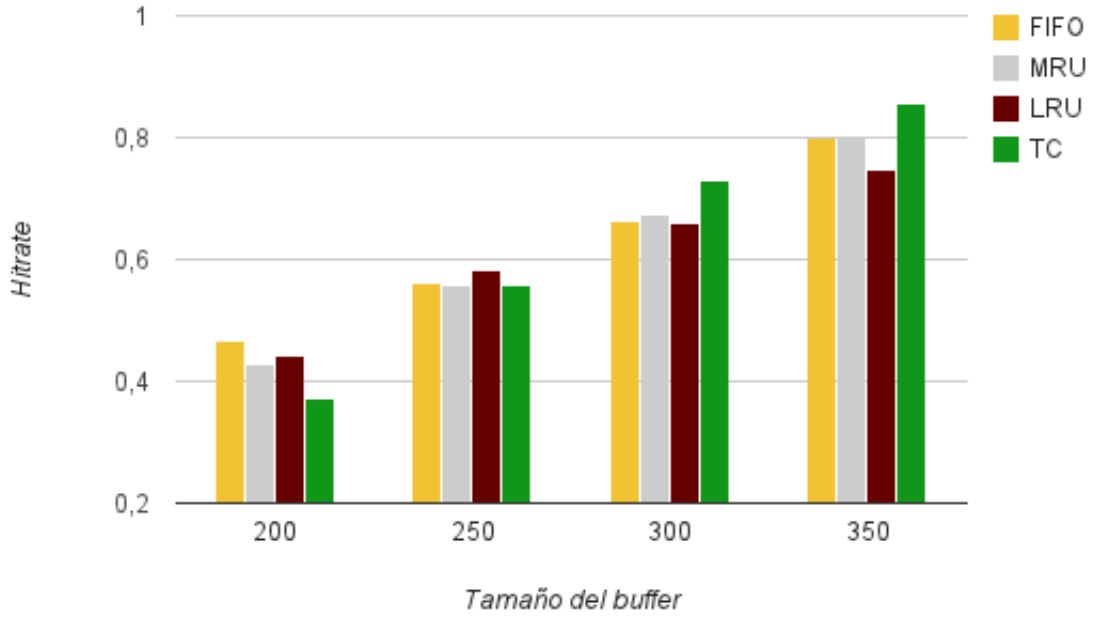

Figura 2: Random File Scan +File Scan

En este caso para tamaños más chicos de buffer, obtenemos mejores resultados con LRU y MRU. Como el paper indica, este algoritmo fue pensado ante el nuevo avance computacional de buffers más grandes comparado cuando se empezaron a utilizar los algoritmos LRU y MRU, por lo que es lógico que, tal como vemos en el gráfico, Touch Count se comporte mejor ante tamaños de buffer más grandes.

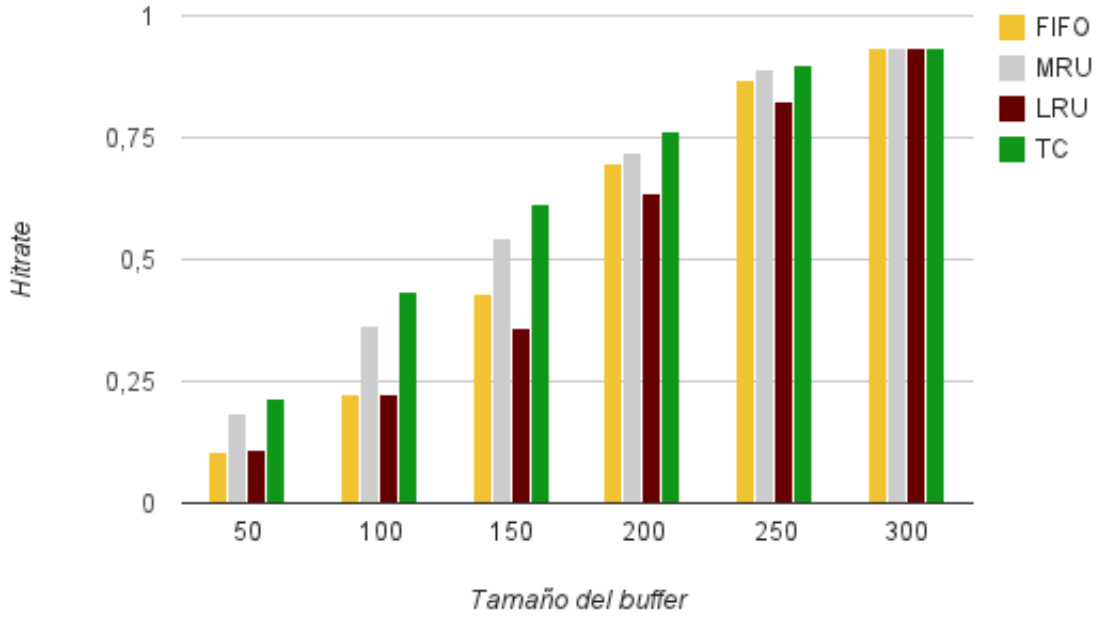

Figura 3: Mixed Trace +Index Scan

Tanto como para el primer caso, el resultado predicho fue el ganador. Touch Count obtiene una mejor performance frente al resto.

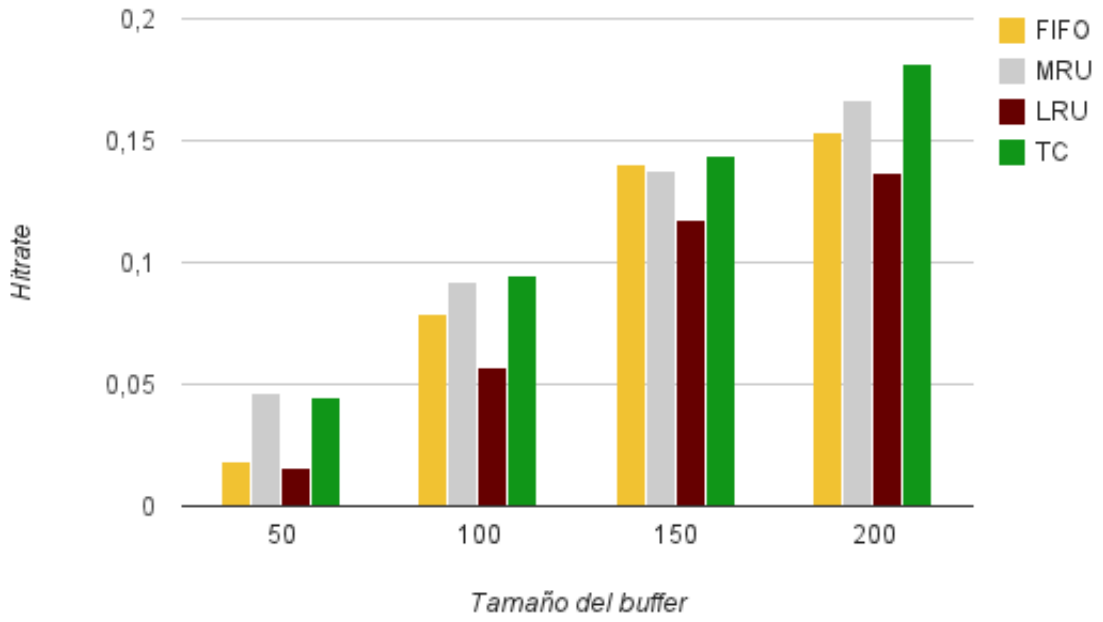

Figura 4: Mixed Trace solo File Scan

Al ser todo FileScan, los algoritmos LRU y FIFO tienen problemas (referirse a Motivación de Touch Count pará más información), por lo que deberían dar casi 0 HitRate. Y MRU y Touch Count deberían dar mejores valores. Esto se ve reflejado en el gráfico de arriba.

6. Conclusión

Observando estos gráficos podemos observar que el Touch Count es el que tiene un comportamiento más regular en todos los casos. No obstante siguen existiendo combinaciones de trazas que tendrían un mayor hitrate si se utilizara MRU o LRU. Aunque el Touch Count nos demuestra que abarca una espectro más amplio de casos podría inducirnos a que quizás siga existiendo espacio para mejorar la selección de víctimas de un buffer pool. Una posibilidad sería el tratar de que el algoritmo identifique los casos en los que el Touch Count no representa la mejor opción antes de aplicarlo.