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

Base de Datos

Segundo Cuatrimestre de 2012

Trabajo práctico 2

Buffer Manager y estrategias de reemplazo de páginas

Grupo 2

Integrante	LU	Correo electrónico
Cammi, Martín	676/02	martincammi@gmail.com
De Sousa, Mariano	389/08	marian_sabianaa@hotmail.com

Índice

1. Investigación sobre estrategia de reemplazo de páginas de Oracle

1.1. Introducción

A lo largo de este trabajo analizaremos el algoritmo de *Touch Count* diseñado por *Oracle* para administrar en memoria las páginas que se traen de la base de datos con el objetivo de evitar los accesos a disco y por consiguiente, obtener mejoría en el rendimiento del motor.

Inicialmente Oracle utilizaba para el algoritmo de manejo de *caché* lo que se conoce como algoritmo LRU (Least Recently Used). Este algoritmo, descripto más adelante se basa en priorizar en la caché las páginas de disco más referenciadas descartando las menos usadas recientemente (definición de LRU).

Un caso muy común que presenta un problema a este algoritmo son los full scan, los cuales recorren todos los registros de una tabla colocándolos en la *caché* y pisando todo historial previo. Así por ejemplo, si la cache del buffer tiene 300 bloques y un escaneo completo de una tabla está recibiendo 400 bloques en la cache del buffer, todos los bloques populares desaparecerán.

Para superar este problema, Oracle propuso un algoritmo modificado de LRU al que denominó *Touch Count*

1.2. Algoritmo *Touch Count*

El algoritmo *Touch Count* utiliza las idea de los algoritmos de LRU y MRU. Prioriza mantener en la caché las páginas más referenciadas y descartar las menos, manteniendo una lista LRU y reordenándola con un criterio que se asemeja con el desalojo de MRU: quien sea el más referenciado se reubicará, para ello el algoritmo llevará un conteo de cuantas veces fue referenciada. Este número de referencias a la página se denomina *Touch Count*.

1.3. Problemática que resuelve

Uno de los problemas que el *Touch Count* resuelve es el de las ráfagas de acceso a páginas. En un full scan por ejemplo múltiples páginas son referenciadas las que potencialmente pueden borrar la caché. Que una página sea referenciada esporádicamente no implica que permanecerá en memoria, deberá volver a pedirla, que aumente su *Touch Count* para así poder alejarse de la zona fría donde es más propensa a ser desalojada.

1.4. Ventajas y Desventajas

Como se comentó en el párrafo anterior y se verá más adelante, el algoritmo de *Touch Count* posee algunas ventajas con respecto a LRU y MRU, con respecto al mejor tratamiénto de ráfagas o la posibilidad de no depender del último acceso de las páginas sino de la cantidad de accesos que haya tenido.

Una desventaja que podría considerarse es el reacomodo de las páginas en el momento de un findVictim que los algoritmos de LRU y MRU no necesitan. Además un detalle a tener en cuenta es que los otros algoritmos pueden ser utilizados de forma *Plug & Play* mientras que el *Touch Count* debe ser previamente configurado para obterner un mejor rendimiento acorde las necesidades específicas de las consultas. Esto último es también una ventaja ya que una vez configurado correctamente puede presentar mejoras considerables.

2. Implementación estrategias de reemplazo de páginas

2.1. Algoritmo de LRU

El algoritmo LRU (Least Recently Used) funciona de la siguiente manera, mantiente en memoria las páginas más recientemente usadas y en caso de necesitar remover alguna, elige la que se posee fecha de referencia más vieja.

El siguiente es un ejemplo de como la caché se va actualizando mediante el algoritmo LRU.

El gráfico a continuación se lee de izquierda a derecha de arriba abajo. La primera hilera de números corresponde a la traza de páginas que intentan ser accedidas. Debajo de ella aparece en forma vertical la caché y como se va llenando a medida que cáda página de la traza es referenciada.

Marcaremos con amarillo cuando ocurra un *Hit* y con gris cuando ocurra un *Miss*. Cuando una página sea referenciada la marcaremos en la caché en negrita para saber que fue la más recientemente accedida.

Ingreso de pagina i	0	3	1	0	3	7	0	8	0	4	3
Caché (tamaño 3)	**0**	0	0	**0**	0	0	**0**	0	**0**	0	0
		3	3	3	**3**	3	3	**8**	8	8	**3**
			1	1	1	**7**	7	7	7	**4**	4

Así en el ejemplo,

- Se referencia inicialmente la página 0, como la caché se encuentra vacía se produce un *Miss*, y se prodece a agregar la página 0 a la caché.
- Se referencia a la página 3, como no existe en la caché (ya que solo posee la página 0 agregada anterioremente) se produce un *Miss* y se prodece a agregar la página 3 a la caché.
- Se referencia a la página 1, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 1 a la caché.
- Se referencia a la página 0, como existe en la caché se produce un *Hit*.
- Se referencia a la página 3, como existe en la caché se produce un *Hit*.
- Se referencia a la página 7, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 7 a la caché reemplazando la menos recientemente usada que es la página 1.

- Se referencia a la página 0, como existe en la caché se produce un *Hit*.
- Se referencia a la página 8, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 8 a la caché reemplazando la menos recientemente usada que es la página 3.
- Se referencia a la página 0, como existe en la caché se produce un *Hit*.
- Se referencia a la página 4, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 4 a la caché reemplazando la menos recientemente usada que es la página 7.
- Se referencia a la página 3, como no existe en la caché (porque ya fue desalojada) se produce un *Miss* y se prodece a agregar la página 3 a la caché reemplazando la menos recientemente usada que es la página 8.

Es interesante notar que una página no será desalojada hasta tanto se hayan referenciado al menos una vez a todas las otras, ya que de esta forma la página inicial se convertiría en la menos recientemente usada.

Cantidad de Hits: 4
Cantidad de Miss: 7, con un total de 3 Miss iniciales y 4 Miss a lo largo de la traza.
Predicción: Este enfoque se basa en el principio de vecindad temporal; páginas que fueron referenciadas posiblemente lo sean en un período corto de tiempo.
Desventajas: Una desventaja es que una consulta que traiga muchas páginas que no vuelvan a ser utilizadas (ej: full scan sobre una tabla), barrerá gran parte, sino toda, las entradas existentes en la caché.
Ventajas: En un contexto donde el principio de vecindad temporal es válido, mientras un conjunto de páginas se referencie con periocidad (en ventanas cortas de tiempo), existe una alta probabilidad que la página se encuentre en memoria.

2.2. Algoritmo de MRU

El algoritmo MRU (Most Recently Used) funciona a la inversa de *LRU*, intentando mantener en memoria las páginas menos recientemente usadas y en caso de necesitar remover alguna eligiendo de entre alguna de las más recientemente usadas.

El siguiente es un ejemplo de como las caché se va actualizando mediante el algoritmo LRU.

El gráfico a continuación se lee de izquierda a derecha de arriba abajo. La primera hilera de números corresponde a la traza de páginas que intentan ser accedidas. Debajo de ella aparece en forma vertical la caché y como se va llenando a medida que cáda página de la traza es referenciada.

Marcaremos con amarillo cuando ocurra un *Hit* y con gris cuando ocurra un *Miss*. Cuando una página sea referenciada la marcaremos en la caché en negrita para saber que fue la más recientemente accedida.

Ingreso de pagina i	0	3	1	0	3	7	0	8	0	4	3
Caché (tamaño 3)	**0**	0	0	**0**	0	0	**0**	**8**	**0**	**4**	**3**
		3	3	3	**3**	**7**	7	7	7	7	7
			1	1	1	1	1	1	1	1	1

Así en el ejemplo,

- Se referencia inicialmente la página 0, como la caché se encuentra vacía se produce un *Miss*, y se prodece a agregar la página 0 a la caché.
- Se referencia a la página 3, como no existe en la caché (ya que solo posee la página 0 agregada anterioremente) se produce un *Miss* y se prodece a agregar la página 3 a la caché.
- Se referencia a la página 1, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 1 a la caché.
- Se referencia a la página 0, como existe en la caché se produce un *Hit*.
- Se referencia a la página 3, como existe en la caché se produce un *Hit*.
- Se referencia a la página 7, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 7 a la caché reemplazando la más recientemente usada que es la página 3.

- Se referencia a la página 0, como existe en la caché se produce un *Hit*.
- Se referencia a la página 8, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 8 a la caché reemplazando la más recientemente usada que es la página 0.
- Se referencia a la página 0, como no existe en la caché (porque ya fue desalojada) se produce un *Miss* y se prodece a agregar la página 0 a la caché reemplazando la más recientemente usada que es la página 8.
- Se referencia a la página 4, como no existe en la caché se produce un *Miss* y se prodece a agregar la página 4 a la caché reemplazando la más recientemente usada que es la página 0.
- Se referencia a la página 3, como no existe en la caché (porque ya fue desalojada) se produce un *Miss* y se prodece a agregar la página 3 a la caché reemplazando la más recientemente usada que es la página 4.

En este algoritmo se puede notar que cualquier referencia a una página existente en la caché la convierte en la potencial primera víctima para ser desalojada en caso de un próximo *Miss*.

Cantidad de Hits: 3
Cantidad de Miss: 8, con un total de 3 *Miss* iniciales y 5 *Miss* a lo largo de la traza.
Predicción: Este tipo de enfoque intenta basarse en que una vez referenciada una página no volverá a serlo al menos en un cierto período de tiempo en el cual si podrían llegar a serlo páginas más antiguas.
Desventajas: Si la caché se encuentra llena con páginas que no serán referenciadas en un tiempo cercano y se consultan nuevas, deberá continuamente estar desalojando una página para poder agregarlas (notar que en cada ráfaga sólo quedará la última de ellas).
Ventajas: Las páginas que se consultan de manera esporádica serán rápidamente desalojadas sin afectar al resto de la caché.

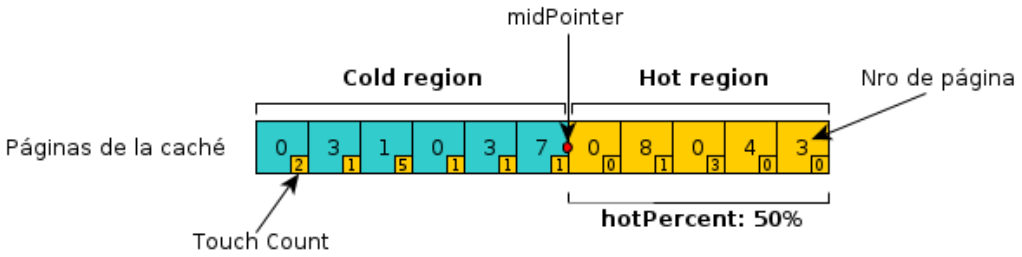
2.3. Algoritmo de Touch Count

Este algoritmo es la mejora del LRU implementada por *Oracle*. A continuación describiremos un poco de la estructura interna que utiliza el *Touch Count*.

El algoritmo Touch Count cuenta con tres elementos principales: una lista, un puntero, y un contador de referencias denominado *Touch Count*. La lista se utilizará para manejar las páginas y las prioridades que se le asignarán a cada una esta lista estará dividida en dos áreas o regiones. La región de la izquierda de la lista denominada *región fria* y la región inmediata contigua denominada *región caliente*.

Ambas regiones estarán separadas por un puntero, al que llamaremos midPointer, que se encargará de marcar dicha división. En realidad en la implementación el *midpointer* estará apuntando el primer elemento de la *región caliente* cuando haya elementos en dicha región o al final de la lista en caso contrario.

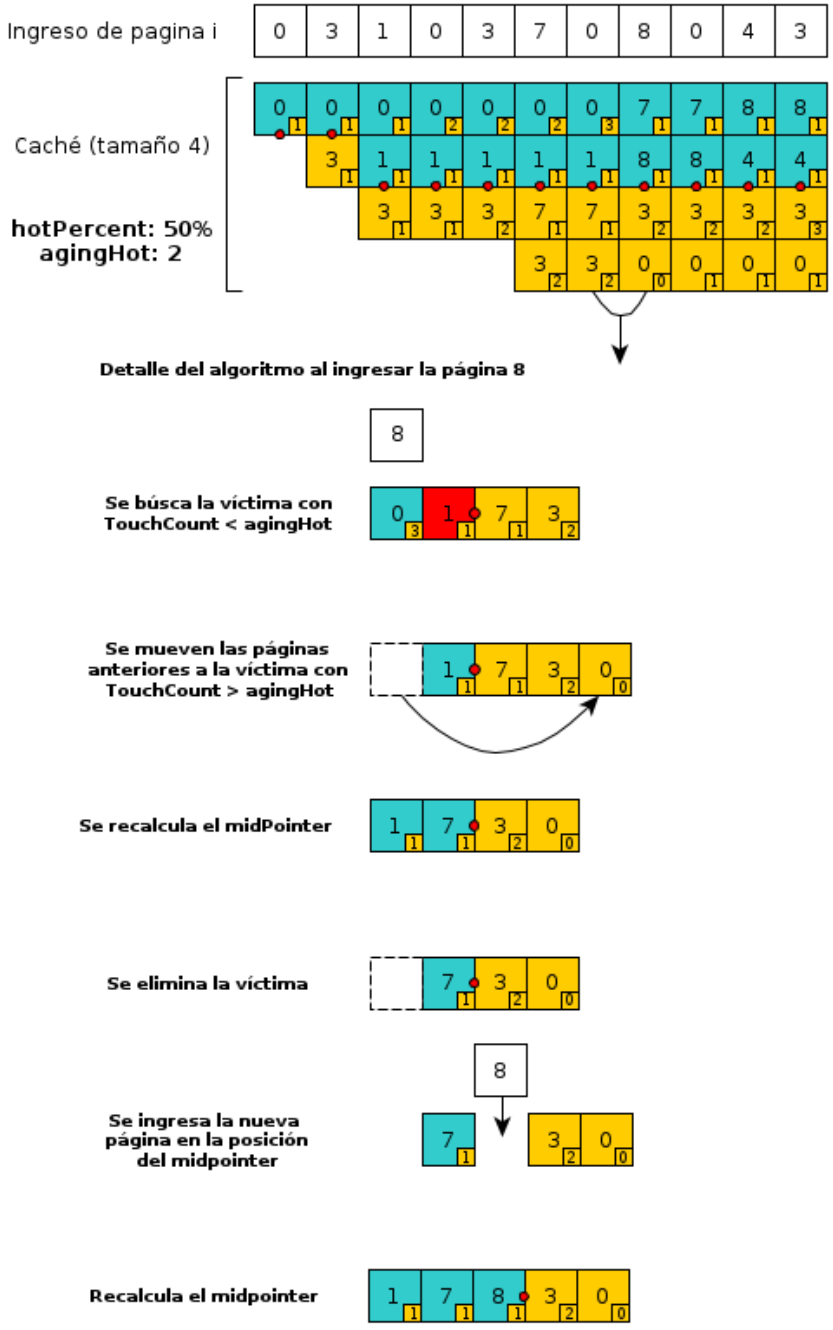
Por otro lado el algoritmo es configurable y cuenta con varios parámetros:



- hotPercent: Indica el porcentaje de páginas que se quieren mantener en la *región caliente*. Si el porcentaje no da exacto se toma la cantidad de páginas que conformen el porcentaje inmediato inferior.
- agingTouchTime: Es la cantidad mínima de tiempo a esperar antes de aumentar el *touch count* de una página ante otra referencia.
- agingWhenMoveToHot: Es el valor al cual se establece al *touch count* de una página cuando ésta pasa a la *región caliente.*
- agingToHotCriteria: Es el valor minimo de Touch Count que debe tener una página para que pueda ser enviada a la *región caliente.*
- agingCoolCount: Es el valor de *touch count* que pasa a tener la página cuando se mueve de la zona caliente a la fría.

Cabe aclarar que si la división del *midPointer* no deja un porcentaje exacto para la *región caliente* en base a la variable de *hotPercent*, el mismo se ajustará al porcentaje inmediato inferior, no superior. En el ejemplo, el *midPointer* deja 6 elementos a izquierda para la *zona fria* (54 %) y 5 elementos en la *zona caliente* (45 %), así el 45 % no supera el 50 % impuesto por el *hotPercent*.

Ahora describiremos su comportamiento en base a las operaciones que se realizan sobre la caché.



Como puede verse en el ejemplo anterior, las operaciones de *agregar*, *encontrar víctima* y *remover* hacen algunas extras que lo que indican a simple vista.

- Agregar una página: Si la caché todavía posee espacio disponible agregará la nueva página en la posición que indique el *midPointer*. Si la caché no posee espacio disponible procederá a buscar una víctima para removerla y agregar la nueva en la posición del *midPointer*.
- Encontrar víctima: El algoritmo comienza buscando una víctima que será la primera de la región fría que tenga Touch Count menor al *agingToHotCriteria* Una vez encontrada la víctima, moverá todas las páginas que al recorrer hayan superado el *agingToHotCriteria* al final de la *región caliente* asignándole un Touch Count de *agingWhenMoveToHot*. Al mismo tiempo por movimiento de estos otra página pasará el umbral de la caliente a la fría como ocurre con la página 7 del ejemplo anterior al moverse la página 0 de la *región fria* a la *región caliente*. A estas páginas que pasan a la *región fria* se les colocará un touch count de *agingCoolCount*.
- Remover una página: El remover una página borrará la página de la caché y recalculará la posición del *midpointer*.

Existe un caso particular sobre qué página remover; cuando todas ellas poseen un Touch Count mayor o igual a *agingToHotCriteria*. La información recolectada no especifica qué desición tomar. Se optó por buscar la página con Touch Count menor y en caso de existir más de una, seleccionar la de fecha de referencia más vieja.

Por otro lado, si se referencian set de datos diferentes cada vez, las páginas no alcanzarán el valor agingToHotCriteria suficiente para salvarse en el zona caliente. De esta manera, la zona fría se comportará como una LRU y las páginas que queden, por el corrimiento del mid point maker, en la zona caliente; no serán utilizadas desperdiciando así espacio de memoria.

3. Test de Unidad

3.1. MRUReplacementStrategyTest

3.1.1. findVictim tests

testNoPageToReplace: testea que no puede obtenerse una víctima porque todas las páginas están bloqueadas.
resultado esperado: lanzar una PageReplacementStrategyException indicando: *No page can be removed from pool.*

testOnlyOneToReplace: testea con una única página no bloqueada.
resultado esperado: devolver la página no bloqueada como víctima.

testMultiplePagesToReplace: testea con todas páginas no bloqueadas y no referenciadas.
resultado esperado: devolver la primera como víctima.

testMultiplePagesToReplaceButOldestOnePinned: testea con todas páginas no bloqueadas salvo la última.
resultado esperado: devolver la segunda como víctima.

testMultiplePagesToReplaceWithPinAndUnpin: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una.
resultado esperado: devolver la última como víctima.

testMultiplePagesToReplaceWithPinAndUnpinButOneDouble: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin, siendo que la segunda se le hizo un pin unpin más.
resultado esperado: devolver la segunda página como víctima.

testMultiplePagesToReplaceButNewestOnePinnedWithPin: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una salvo la última a la que sólo se le hizo pin al final.
resultado esperado: devolver la ante última página como víctima.

testMultiplePagesToReplaceButOldestOnePinnedWithPin: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una salvo la primera a la que sólo se le hizo pin al final.
resultado esperado: devolver la segunda página como víctima.

3.2. LRUReplacementStrategyTest

3.2.1. findVictim tests

testNoPageToReplace: testea que no puede obtenerse una víctima porque todas las páginas están bloqueadas.
resultado esperado: lanzar una PageReplacementStrategyException indicando que *No page can be removed from pool.*

testOnlyOneToReplace: testea con una única página no bloqueada.
resultado esperado: devolver la página no bloqueada como víctima.

testMultiplePagesToReplace: testea con todas páginas no bloqueadas y no referenciadas.
resultado esperado: devolver la primera como víctima.

testMultiplePagesToReplaceButOldestOnePinned: testea con todas páginas no bloqueadas salvo la última.
resultado esperado: devolver la primera como víctima.

testMultiplePagesToReplaceWithPinAndUnpin: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una.
resultado esperado: devolver la primera como víctima.

testMultiplePagesToReplaceWithPinAndUnpinFirstReferencedLast: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una siendo la primera referenciada última.
resultado esperado: devolver la segunda página como víctima.

testMultiplePagesToReplaceButOldestOnePinnedWithPinAndUnpin: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una salvo la primera a la que sólo se le hizo pin al principio.
resultado esperado: devolver la segunda página como víctima.

testMultiplePagesToReplaceButOldestOnePinnedWithPinAndUnpinFirstReferencesLast: testea con todas las páginas no bloqueadas habiéndoselas hecho pin y unpin a cada una salvo la primera a la que sólo se le hizo pin al final.
resultado esperado: devolver la segunda página como víctima.

3.3. TouchCountBufferFrameTest

3.3.1. *Touch Count* tests with agingTouchTime = 3

testTouchCountOnePin: testea un pin.
resultado esperado: el touch count sea 1.

testTouchCountPinUnpinEquals: testea un pin unpin seguidos.
resultado esperado: el touch count sea 1.

testTouchCountPinUnpingDistinct: testea un pin, espera de 3 segundos y luego un unpin.
resultado esperado: el touch count sea 2.

testTouchCountBlast: testea pin, espera de más de medio segundo, pin, espera de más de medio segundo, pin.
resultado esperado: el touch count sea 1 luego del primer pin.
resultado esperado: el touch count sea 1 luego del segundo pin.
resultado esperado: el touch count sea 2 luego del tercer pin.

3.4. TouchCountBufferPoolTest

3.4.1. *Touch Count* tests

testFindVictimWithSpace: testea la búsqueda de víctima con espacio en la caché.
resultado esperado: lanzar una BufferPoolException indicando: *The buffer still has space.*

testFindVictimWithExistingPage: testea pasando como parámetro una página a agregar existente en la caché.
resultado esperado: lanzar una BufferPoolException indicando: *The page is already in the buffer.*

testFindVictimInOrder: testea con todas páginas no bloqueadas.
resultado esperado: devolver la primera como víctima.

testFindVictimAllWithLowTouchCount: testea con todas páginas con bajo touch count.
resultado esperado: devolver la primera como víctima.

testFindVictimMovingOneToHot: testea con la primer página con alto touch count, la segunda con bajo touch count y la tercera bloqueada.
resultado esperado: devolver la segunda página como víctima, además la primera página debe tener touch count 0 (por haberse movido a la *región caliente*), la segunda debe tener un touch

count 1 (por haberse movido a la *región fria*) y la tercera no debió modificar su touch count puesto que no se movió

testFindVictimMovingTwoToHot: testea con la primer y tercer página con alto touch count y la segunda con bajo touch count.
resultado esperado: devolver la segunda página como víctima, además la primera página debe tener touch count 1 (por haberse movido a la *región caliente* primero y a la *región fria* después), la segunda debe tener touch count 1 (por haberse movido a la *región fria*) y la tercera touch count 1 por haber terminado en la *región fria*.

testMidPointerWithFiftyHotPercentage: testea el valor del midPointer con hotRegion = 50 %.
resultado esperado: devolver posicion del midPointer en 0 con la caché vacía.
resultado esperado: devolver posicion del midPointer en 1 habiendo agregado un elemento.
resultado esperado: devolver posicion del midPointer en 1 habiendo agregado dos elementos.
resultado esperado: devolver posicion del midPointer en 2 habiendo agregado tres elementos.
resultado esperado: devolver posicion del midPointer en 2 habiendo agregado cuatro elementos.

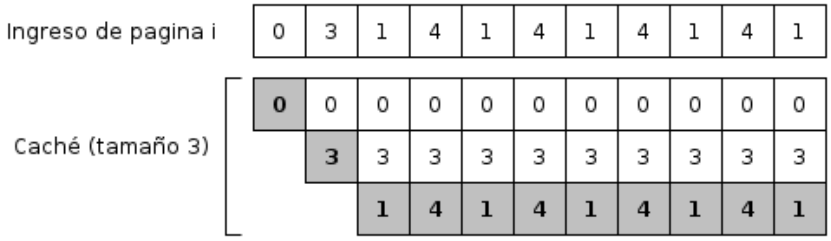
testMidPointerWithThirdHotPercentage: testea el valor del midPointer con hotRegion = 34 %.
resultado esperado: devolver posicion del midPointer en 0 con la caché vacía.
resultado esperado: devolver posicion del midPointer en 1 habiendo agregado un elemento.
resultado esperado: devolver posicion del midPointer en 2 habiendo agregado dos elementos.
resultado esperado: devolver posicion del midPointer en 2 habiendo agregado tres elementos.

4. Trazas evaluadas

A continuación hemos confeccionado una serie de trazas para evaluar el comportamiento de cada algoritmo. Algunas trazas intentan identificar los patrones patológicos para cada algoritmo analizando cual sería el escenario de peor caso.

4.1. Peor caso MRU: Traza MRUPathological

La siguiente traza describe la referencia a páginas que siempre de *Miss*. Para ello se llena la caché (referenciando a las páginas 0, 3, 1) que darán *Miss* de inicialización y luego se referencia una página que no exista en la caché, la página 4. Como la caché está llena, debe desalojar una página, la 1. Esa será la página que se referenciará a continuación la cual dará *Miss* y desalojará la página 4 que fue la última en ser referenciada y será la página que se pedirá a continuación la cual dará *Miss* y así sucesivamente.



Se generó una traza llamada *mruPathological-Music.trace* para probar este caso donde se utilizó un buffer de 10 elementos. Una vez lleno se cargó una nueva página (página de overflow) forzando el desalojo de una página. Durante un ciclo se pidió alternadamente la página de overflow y la desalojada.

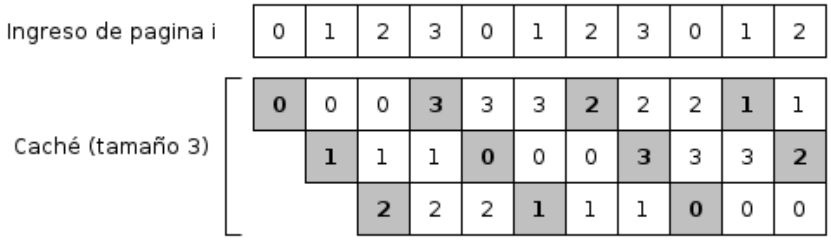
Los resultados obtenidos fueron:

MRU	**LRU**	**Touch Count**
Hits: 0	**Hits: 9**	**Hits: 9**
Misses: 20	**Misses: 11**	**Misses: 11**
Hit rate: 0.0	**Hit rate: 0.45**	**Hit rate: 0.45**

LRU y *Touch Count* obtuvieron el mismo rendimiento ya que ambos algoritmos alojaron en posiciones distintas de memoria tanto la página de overflow como la otra utilizada para generar el caso borde. Luego, ambos obtuvieron la misma cantidad hits. Además, sus resultados fueron óptimos, ya que generaron la mínima cantidad de miss, correspondiente a las once páginas utilizadas en la traza.

4.2. Peor caso LRU: Traza LRUPathological

En la siguiente traza para LRU, se intenta maximizar la cantidad de *Miss* para el algoritmo. El patrón para lograrlo se basa en completar la caché con una traza y luego repetirla varias veces de forma de desalojar las mismas páginas que se mantuvieron al principio. Al completar la caché se refencia una página nueva no existente, en el ejemplo la página 3, se desaloja la página más antigua, la 0, que justamente es la siguiente a ser referenciada por la traza y que vendrá a desalojar a la página 1, que será justamente la siguiente en ser desalojada. De esta forma se irán desalojando las páginas que serán inmediatamente referenciadas ocacionando una serie de *Miss en cascada*.



Se generó una traza llamada *lruPathological-Music.trace* para probar este caso donde se utilizó un buffer de 10 elementos. Una vez lleno se cargó una nueva página (página de overflow) y se fueron llamando a las páginas existentes previas a la de overflow. De esta manera LRU desaloja en el paso i la página requerida en el paso i+1.

Los resultados fueron:

MRU	**LRU**	**Touch Count**
Hits: 9	**Hits: 0**	**Hits: 4**
Misses: 11	**Misses: 20**	**Misses: 16**
Hit rate: 0.45	**Hit rate: 0.0**	**Hit rate: 0.2**

MRU obtuvo un buen rendimiento ya que sólo generó miss cuando cargó las páginas en la caché, en todos los pedidos de páginas algunas vez cargadas obtuvo hit. Este tipo de escenario no generan un caso borde en MRU.

Por otro lado es interesante marcar que si bien Touch Count se basa en el algoritmo de LRU, obtuvo una mejora. Esto sucede ya que la zona caliente poseía elementos y estos nunca fueron desalojados, por lo tanto; al ser requeridos por segunda vez generaron un hit.

4.3. LRU vs *Touch Count*

Para comparar en detalle las diferencias entre LRU y *Touch Count*, se generó la siguiente traza. A continuación se describe un escenario en el cual el algoritmo de *Touch Count* mejora con respecto a LRU.

El ejemplo consiste en potenciar las páginas que el Touch Count considera importantes, e intentar desalojarlas al mismo tiempo de la caché en el algoritmo LRU.

Inicialmente se utilizará una traza cualquiera para completar la caché. En ambos algoritmos estar inicialización dará *Miss* ya que las páginas no existirán.
Un escenario posible sería:

1) full scan sobre la tabla A: para cargar las páginas en la caché (se asume que los datos de A no superan el tamaño de la caché.

2) varios file scan sobre la tabla A condicionando a los registros con id par: en ambos algoritmos habrá *Hit*.

3) file scan sobre la tabla A condicionando a los registros con id impar: en ambos algoritmos habrá *Hit*.

4) file scan sobre la tabla B: en ambos algoritmos habrá *Miss*.

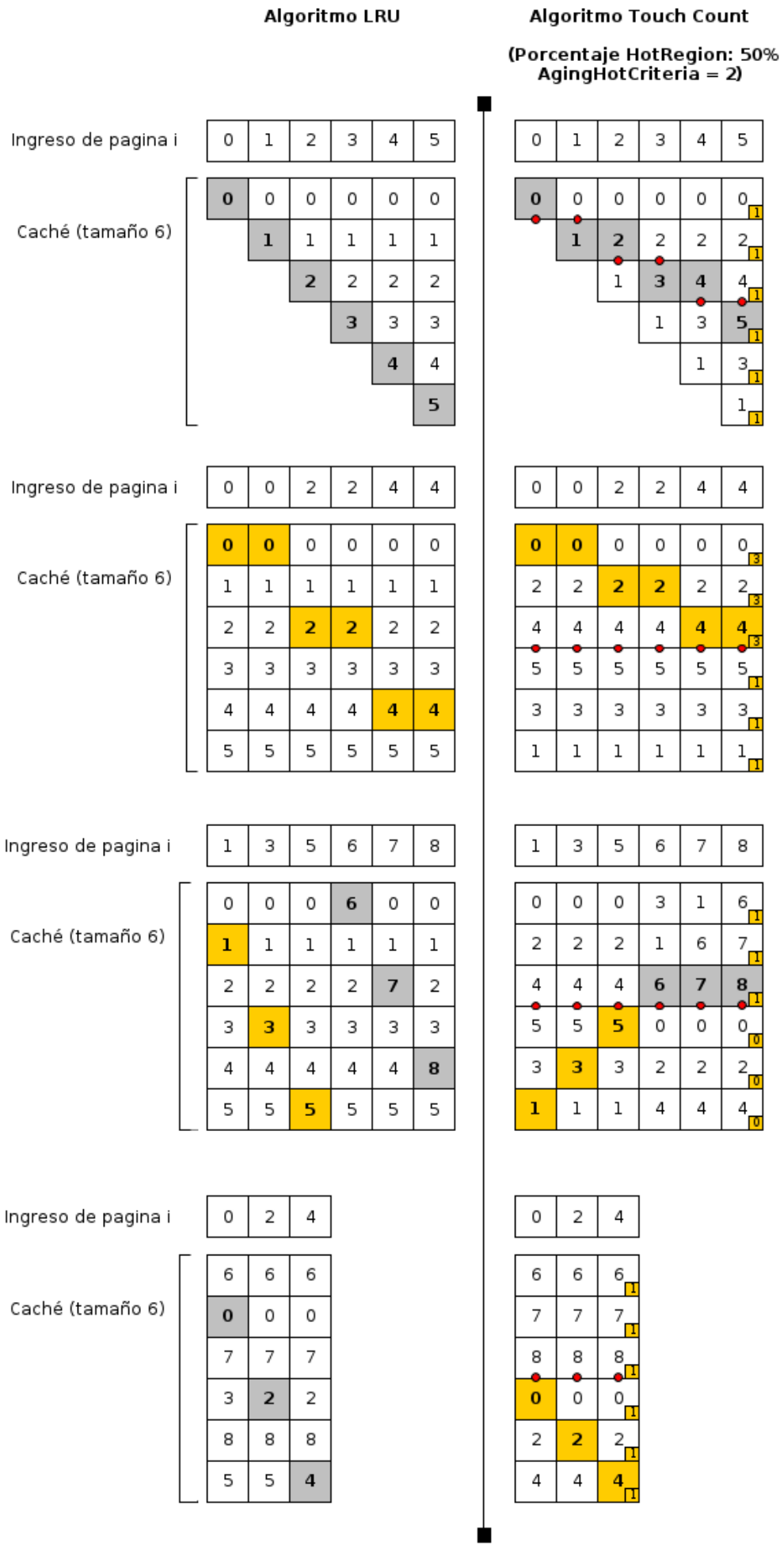
5) file scan sobre la tabla A: condicionando a los registros con id par: sólo en Touch Count las páginas todavía exisitŕan en caché. ya que las reiteradas referncias del punto 2 habrán hecho que Touch Count les diera un pero mayor y por ende las habría salvado en la *región caliente*.

La conclusión que se intenta describir es que el Touch Count prioriza las páginas por *peso* mientras que LRU las prioriza por último acceso.

Si una tabla tuvo sus páginas accedidas recientemente, el algoritmo LRU tendrá sus páginas como relevantes mientras que para Touch Count solo lo serán si son refenciadas en más ocaciones.

Por ello si una tabla fue accedida muchas veces, pero no últimamente, posiblemente no sean tenida en cuenta por el algoritmo LRU mientras que Touch Count las tendrá como relevantes ya que tuvieron una frecuencia alta de accesos.

En conclusión Touch Count se basa en que si una página fue accedida muchas veces es probable que vuelva a ser referenciada, por eso les asigna un peso y pondera respecto a él. Por el contrario LRU se basa en que las páginas más recientes son las más probables de volver a ser referenciadas y son éstas las que decide priorizar.


Algoritmo LRU
Algoritmo Touch Count
(Porcentaje HotRegion: 50%
AgingHotCriteria = 2)
Ingreso de pagina i
Caché (tamaño 6)
Ingreso de pagina i
Caché (tamaño 6)
Ingreso de pagina i
Caché (tamaño 6)
Ingreso de pagina i
Caché (tamaño 6)

4.4. Traza: badMRUAndNotGodLRU

Esta traza se generó llevando al extremo las diferencias entre los tres algortimos.

Como vimos anteriormente el algoritmo de MRU no quita una página de la caché hasta que esta no posea la referencia más reciente. Inmediatamente se desprende que las páginas que puedan llegar a la caché y sean poco utilizadas tienen altas chances de evitar el desalojo.

El algoritmo de LRU si bien corrige el caso de MRU recién mencionado, presenta como inconveniente que grandes pedidos de páginas que producen miss realizados de manera esporádica terminan barriendo la caché, perdiendo páginas que potencialmente pueden ser reutilizadas.

Touch count puede ser útil para evitar ambos problemas: la parte fría actúa como una LRU, corrigiendo el problema de MRU. Por otro lado, da peso a las páginas que son más usadas evitando que por ejemplo, los file scans grandes las terminen por excluir de la caché.

De esta manera y utilizando una caché de 200 páginas, se realizaron 7800 pedidos de la siguiente manera:

- Se llena la caché y se vuelve a pedir cada página de la caché tres veces (generando 200 miss y 600 hits). Estas páginas no vuelven a ser pedidas y al no pedir una misma página dos veces consecutivas, la cantidad de hits final máxima para MRU será de 600.

- Se crea un ciclo donde la caché se llena con páginas siempre nuevas, provocando que LRU realice un flush completo. A su vez se referencia a un conjunto (fijo y distinto de todas las anteriores) de 50 páginas (25 % de la caché) agingHotCriteria + 1 veces, con el objeto de mantener en caché en el algoritmo de *Touch Count*.

La traza se corre con el algoritmo de MRU, LRU y *Touch Count*, este último aceptando ráfagas (para simplificar el problema) y con cuatro porcentajes de hot region (25 %, 50 %, 75 %, 90 %).

A continuación se detallan los resultados obtenidos.

MRU
Hits: 600
Misses: 7200
Hit rate: 0.0769

LRU
Hits: 2971
Misses: 4829
Hit rate: 0.380

TouchCount (25 % HOT)
Hits: 3550
Misses: 4250
Hit rate: 0.455

TouchCount (50 % HOT)
Hits: 3550
Misses: 4250
Hit rate: 0.455

TouchCount (75 % HOT)
Hits: 3550
Misses: 4250
Hit rate: 0.455

TouchCount (90 % HOT)
Hits: 600
Misses: 7200
Hit rate: 0.0769

El rendimiento del algoritmo de MRU queda muy por debajo de los otros dos debido a que almacenó paginas que no volvieron a ser referenciadas como se explicó anteriormente.

El algoritmo de *Touch Count* gana más de un 7 % de mejora, al poder mantener ese conjunto de 50 páginas aún cuando se efectúa el file scan.

Es muy interesante notar que no existe mejora de performance entre las variantes de *Touch Count* de 25 %, contra las de 50 % y 75 % de hot region. Esto es debe a que las páginas que son continuamente referenciadas ocupan el 25 % de la caché y nunca bajan a la cold region; además, como nada de la cold region sube (exceptuándo las del 25 %), el espacio entre el mid point marker y la primer página que se referencia continuamente queda con la carga inicial (utilizada para MRU) que al no ser vuelta a referenciar, no añade hits.

Por otro lado, cuando se utiliza el porcentaje de hot region en 90 %, se obtienen los mismos resultados que en MRU. Como ya mencionamos, los elementos son insertados como último elemento de la cold region. La parte fría funciona como una LRU. Tenemos entonces una lista de tamaño 20 (10 % de la caché) en la cual en una ráfaga le agregamos 50 elementos. Luego de la carga, quedan los 20 últimos y en la siguiente iteración cuando queremos referenciarlos, para aumentar el touch count en el mismo orden, poseemos el mismo problema que en LRU, nunca obtenemos un hit.

Se desprende necesariamente que el tamaño óptimo de la hot region depende del tipo de carga que el servidor posee. Además, tener una zona fría muy chica fuerza a que para poder una página pasar a la zona caliente debe ser referenciada en una gran cantidad en poco tiempo y/o no pueden generarse gran cantidad de miss en caché ya que forzarían a que la página sea desalojada. Por otro lado, cuando más chica es la zona cálida, mayor es la cantidad de llamadas que reciben las páginas que posee, ya que si no son rápidamente desalojadas.

No se considera en la traza el parámetro de cantidad máxima de hits por segundo. Esto no significa que el tamaño óptimo de la hot region sea independiente de él, así como de la carga del servidor. Evitar las ráfagas dificulta mucho más que una página pueda mantenerse en caché, agregando un factor de periodicidad, es decir; no alcanza con haber sido requerida muchas veces en poco tiempo, periódicamente debe ser llamada para evitar el desalojo.

Es válido notar que tener igual hit rate con distinto porcentaje de hot region es un caso prácticamente exclusivo de nuestra carga de datos. Al tener una mayor cantidad de páginas salvadas en la zona cálida, existe mayor probabilidad que se referencie alguna de ellas una nueva vez (excluyendo los casos extremos considerados en el párrafo anterior).

A continuación veremos un test en donde los porcentajes difieren.

4.5. Traza: smallQueriesAndOneBigFileScan

Se creó una nueva traza con el objeto de analizar qué sucede cuando existen pedidos de sets pequeños de datos de manera periódica y se produce un pedido grande que supera el tamaño de la caché de páginas, las cuales todas producen miss, en nuestro caso emulado con un file scan.

Para ellos se generaran cinco conjuntos de páginas:

- Páginas de la tabla *music* (tamaño 5 % de la caché).
- Páginas de la tabla *artist* (tamaño 5 % de la caché).
- Páginas de la tabla *label* (tamaño 15 % de la caché).
- Páginas aleatóreas en cada iteración de la tabla *countrysales* (tamaño 3 % de la caché).
- Páginas de la tabla *género* (tamaño 2 * (tamaño de cache + offset)).

En cada iteración, se generan páginas aleatóreas para la tabla countrysales y se piden todas las páginas menos la de genre (ya que estas se corresponden con nuestro *big file scan*). Cabe destacar, que con el objeto de no condicionar el resultado, el orden en la que se encolan los pedidos de páginas de cada tabla es randomizado. En la mitad de las iteraciones, se pide el file scan de género

Se permitieron por simplicidad de la traza, ráfagas. El parámetro agingHotCriteria es de 10 y el tamaño de caché es de 400.
La traza consta de 8272 pedidos de páginas y los resultados obtenidos fueron:

MRU
Hits: 6613
Misses: 1659
Hit rate: 0.799

LRU
Hits: 6692
Misses: 1580
Hit rate: 0.808

TouchCount (25 % HOT)
Hits: 6792
Misses: 1480
Hit rate: 0.821

TouchCount (50 % HOT)
Hits: 6849
Misses: 1423
Hit rate: 0.827

TouchCount (75 % HOT)
Hits: 6907
Misses: 1365
Hit rate: 0.834

TouchCount (90 % HOT)
Hits: 6902
Misses: 1370
Hit rate: 0.834

El algoritmo MRU presenta mejor resultado al de la traza anterior. Esto se debe a que puede resolver el problema del file scan grande, como la página más reciente es la que acaba de agregar (perteneciente además al único pedido grande), la remueve y evita perder el resto de las páginas perteneciente a las *small queries* que son las que generan hits.

La diferencia de performance entre MRU y LRU se debe al pedido de las páginas aleatorias de countrysales. Por como está implementado el algoritmo, el pedido de estas páginas se realiza en un entorno temporal cercano al pedido del resto de las consultas pequeñas. Si la caché se encuentra llena, como sucede luego del file scan, MRU tendrá a desalojar a páginas de las consultas pequeñas que luego generarán miss, en cambio LRU y a medida que pasen los pedidos, tendrá a remover las páginas procedentes al file scan.

No es menor la mejora que genera el algoritmo de *Touch Count* en compación con LRU y MRU. Al poseer un 25 % de páginas que se frecuentan y un 3 % de páginas random que se van agregando, de las cuales posiblemente se repitan en el tiempo; estas se salvan en la zona caliente y el file scan no provoca su desalojo.

Se puede inferir que a medida que el porcentaje de zona caliente aumenta, la mejora del hit rate se va estabilizando. Nuevamente esto depende de la carga a la que se esté sometiendo al motor de base de datos. Para este ejemplo en particular, parecería sensato tomar como hot region algún valor entre 25 % y 50 % y dejar el resto de la caché como zona fría suponiendo que la carga continuara de similar manera y dejando espacio tanto en caliente como frío para algún caso atípico que pueda suceder.

4.6. Traza: pageBlast

El algoritmo de *Touch Count* posee varios parámetros que como ya mencionamos, no son independientes entre sí.

Por simplicidad, en las trazas anteriores, excluímos el parámetro de cantidad de segundos mínima para poder permitir aumento en el touch count desde el último realizado. De esta manera al procesar las trazas, ningún aumento de su valor era excluído, aunque forme parte de una ráfaga.

Oracle considera que ráfagas de pedidos pueden ensuciar el peso real que una página tiene en la caché. Ejemplificando este problema, se generó la siguiente traza compuesta por una caché de 50 elementos.

Se corrió variando únicamente la cantidad de segundos necesarias antes de permitir un nuevo aumento en el touch count; los valores son:

- 0 (es decir permite ráfagas),
- 1 aumento de touch count por segundo,
- 1 aumento cada dos segundos y uno cada tres.

La traza se generó de la siguiente manera:

Se posee la caché llena suponiendo que los elementos que se encuentran en la zona caliente han sido, y van a ser referenciados periódicamente en el contexto de ejecución.

Se genera una ráfaga sobre los elementos de la zona fría, cada elemento se le efectúa un pin y unpin diez veces. En el supuesto contexto de ejecución, estas páginas se trajeron por un file scan viejo, y se necesitaron llamar para una pequeña consulta y no se van a necesitar más.

El touch count de estos elementos aumenta, así como su probabilidad de pasar a la zona caliente. Si estas páginas pasan a la zona caliente, las que se encontraban previamente en ella, deberán pasarse a la zona fría (por funcionamiento intrínseco del algoritmo). Una vez que las páginas que queremos preservar pasaron a la fría, se efectúa una consulta sobre una tabla que no se encuentra en caché, obligando a desalojar las páginas que no querían perderse. Por ejemplo, podría existir un contexto donde algún cliente solicita ver un ABM que no suele visualizarse. Por el contexto de ejecución, vuelven a ser requeridas las páginas recientemente desalojadas y generan todas ellas miss.

Nota: el hilo de ejecución que levanta la traza y la ejecuta fue detenido en cada iteración durante medio segundo. Se utilizó un *agingHotCriteria* de 10:

Touch Count (allow blast)
Hits: 157
Misses: 89
Hit rate: 0.638

Touch Count (1 per second)
Hits: 157
Misses: 89
Hit rate: 0.638

Touch Count (1 per 2 seconds)
Hits: 169
Misses: 77
Hit rate: 0.686

Touch Count (1 per 3 seconds)
Hits: 169
Misses: 77
Hit rate: 0.686

Este ejemplo adhoc fue generado de forma tal que, tanto permitiendo ráfagas como limitando el aumento de touch count por segundo en uno, el rendimiento sea el mismo. Vemos que cuando el algoritmo se corre con ventanas de tiempo mayores, el touch count de cada página no supera el umbral y por lo tanto no se pasa a la *región caliente*, y por consiguiente; no se desalojan las páginas importantes al contexto de ejecución del problema.

Se decidió forzar a que el límite de un aumento por segundo no sea suficiente para mejorar el rendimiento para ejemplificar que los valores de los parámetros del algoritmo dependen fuertemente del contexto de ejecución, y su mínima alteración afecta fuertemente el rendimiento del todo.

A modo de comentario cabe destacar que en las pruebas comprobamos que si el sleep del thread desciende de 0.5 segundos a 0.38, el algoritmo utilizando una ventana de un segundo mejora su rendimiento e iguala al de la ventana de 2 y 3 segundos.

5. Código fuente

Los tests descriptos anteriormente tienen como título el nombre del archivo java con el que pueden encontrarse en el proyecto, así los test para las estrategias de MRU, LRU y *Touch Count* pueden encontrarse en las clases **MRUReplacementStrategyTest**, **LRUReplacementStrategyTest** y **TouchCountReplacementStrategyTest** respectivamente.

Tests adicionales para *Touch Count* se agrupan en **TouchCountBufferFrameTest** y **TouchCountBufferPoolTest** donde testean las funcionalidades más importantes de este algoritmo.

En la clase **MainEvaluator** figuran los test de todas las trazas que hemos generado, las cuales se encuentran también en el proyecto. Estas pueden correrse descomentando el test correspondiente en el método main que leerán las trazas de los archivos y mostrarán los resultados analizados en este informe; en caso de producirse alguna falla con los archivos de trazas se deberá correr el main de la clase **MainTraceGenerator** descomentando los métodos **patholigicalDataSet** y **benchmarksDataSets** para volver a generarlas.

Las trazas generadas son las siguientes:

- mruPathological-Music.trace
- badMRUAndNotGodLRU.trace
- lruPathological-Music.trace
- lruVSTouchCount-Music.trace
- smallQueriesAndOneBigFileScan.trace
- pageBlast.trace

El método *evaluate* fue modificado para realizar las pruebas y puede que tenga líneas de sleep u otras adaptaciones.

6. Referencias

- **Paper All About Oracles Touch Count Algorithm**
- **Oracle Touch Count Algotithm PPT** - www.bgoug.org
- **Touch-Count Algorithm** - http://oracle-online-help.blogspot.com.ar/2006/11/touch-count-algorithm-in-advancement-to.html