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 utilizando dicha memoria de a manera más eficiente.

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 el mismo espíritu que sus predecesores LRU y MRU, solo que en una mezcla de ambos. Por un lado priorizará mantener en la caché las páginas más referenciadas y descartar las menos pero a su modo, cualquier página que ingrese deberá competir con otras existentes en la caché por permanecer en la misma, 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 y en algoritmos como

1.4. Ventajas y desventajas

2. Implementación estrategias de reemplazo de páginas

2.1. Algoritmo de LRU

El algoritmo LRU (Least Recently Used) funciona de la siguiente manera, intentando mantener en memoria las páginas más recientemente usadas y en caso de necesitar remover alguna elegir de entre alguna de las menos recientemente usadas.

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

El siguiente gráfico 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.

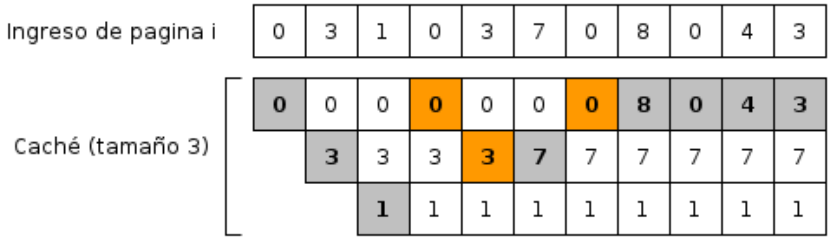
Es interesante notar que una página no será desalojada hasta tanto se hayan referenciado al menos una vez a todas las otras, ya que de esta forma la página inicial se convertirÃa en la menos recientemente usada.

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 intenta predecir que páginas que fueron referenciadas posiblemente lo sean en un período corto de tiempo.
Desventajas: Una desventaja es que un full scan sobre una tabla barrerá por completo con todas las entradas de la caché.
Ventajas:

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 elegir 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 anterior 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.

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:
Ventajas:

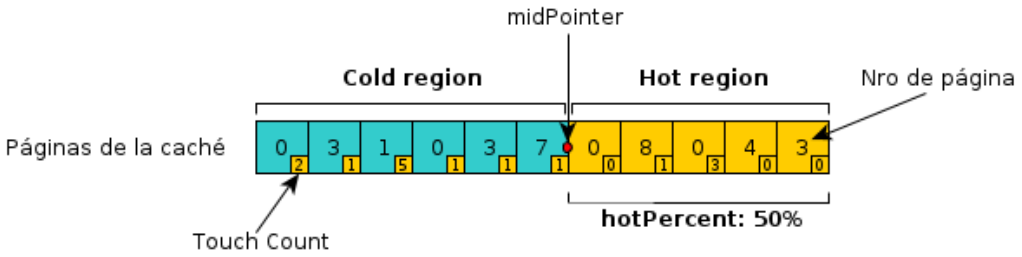
2.3. Algoritmo de Touch Count

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

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*.

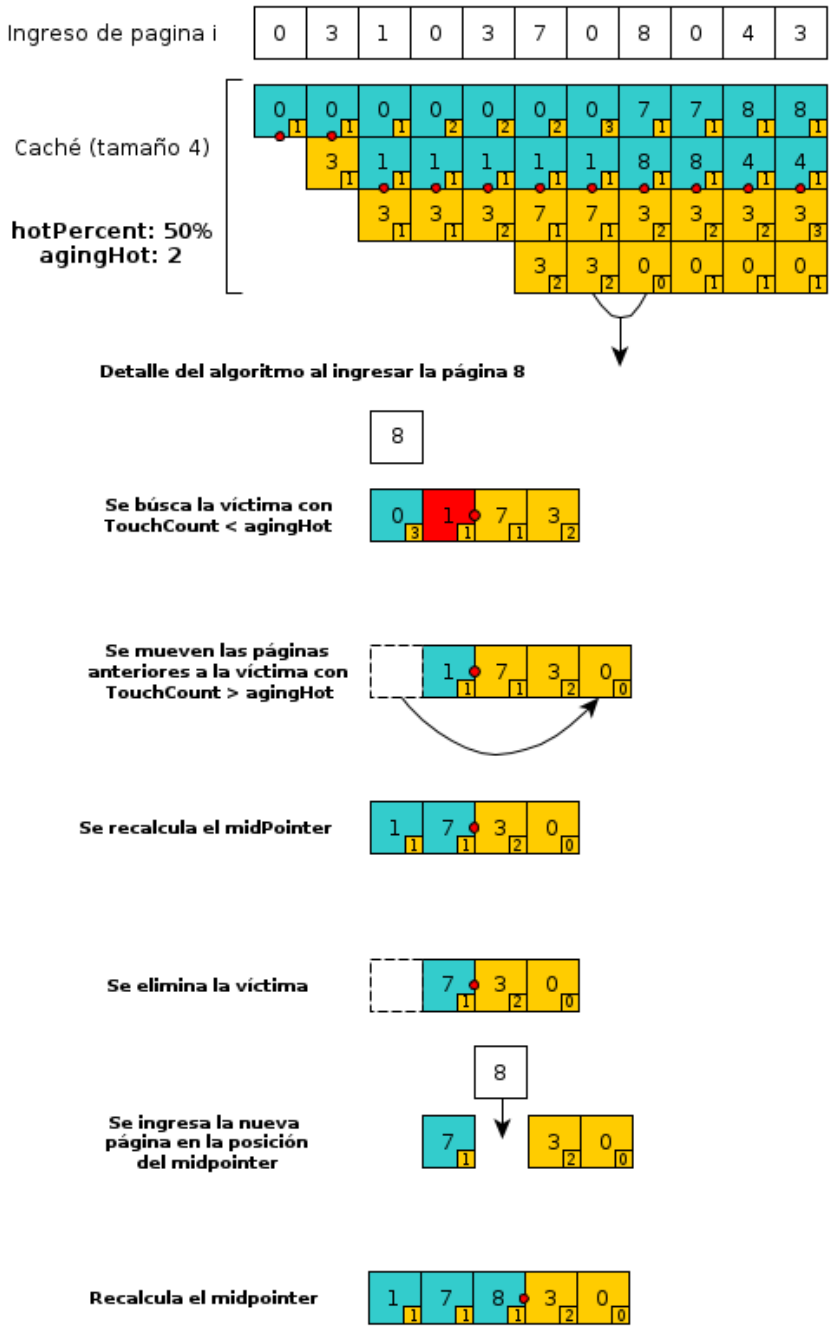
Ambas regiones estarán separadas por un puntero, al que llamaremos midPointer, que se encargará de marcar dicha división. En realidad en la implementación el *midpointer* estará apuntando el primer elemento de la *región caliente* cuando haya elementos en dicha región o al final de la lista en caso contrario.

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
- agingToHotCriteria: Es el valor al cual se establece al Touch Count de una página cuando ésta pasa a la *región caliente*
- agingWhenMoveToHot: Es el valor minimo de Touch Count que debe tener una página para que pueda ser enviada a la *región caliente*

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 vÃctima* 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 fria 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 0. Al mismo tiempo por

movimiento de estos otra página pasará el umbral de la caliente a la fria 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 1.

- Remover una página: El remover una página borrará la página de la caché y recalculará la posición del *midpointer*.

Un caso particular respecto al Touch Count es que sucede cuando todas las páginas poseen un touch count mayor al agingToHotCriteria

Peor caso del TouchCount

Si se referencian set de datos diferentes (tablas diferentes) cada vez, no se aclcanzaran para las paginas un aging count suficiente para salvarlas y todo el tiempo estaremos pisando las página de la cachÃ© con páginas nuevas (al menos en la cola de frias)

Si nada se pasa del agingHot lo que estÃ© en la Hot nunca se va a desaloar, sino que la cola cold será la que se renueve. Se le va a dar más importancia a las páginas con touch counte mayor al aging count más nuevas.

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 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.
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.

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 Ãºltima a la que sólo se le hizo pin al final.
resultado esperado: devolver la segunda 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.
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.

testMultiplePagesToReplaceWithPinAndUnpinFirstReferncedLast: 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. TouchCountBufferPoolTest

3.3.1. TouchCount tests

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

testFindVictimWithExistingPage: testea pasando como parámetro una página a agregar existente en la caché.
resultado esperado: lanzar una BufferPoolException indicando que *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, ademÃ¡s la primera página debe tener touch count 0 (por haberse movido a la *región caliente*), la segunda debe tener 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, ademÃ¡s 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.

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

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*.

de en que se basa parte de llenar la caché con una traza inicial y luego repetirla nuevamente.

4.2. Peor caso LRU

Ingreso de pagina i	0	1	2	3	0	1	2	3	0	1	2
Caché (tamaño 3)	**0**	0	0	**3**	3	3	**2**	2	2	**1**	1
		1	1	1	**0**	0	0	**3**	3	3	**2**
			2	2	2	**1**	1	1	**0**	0	0

4.3. LRU vs TouchCount

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*.

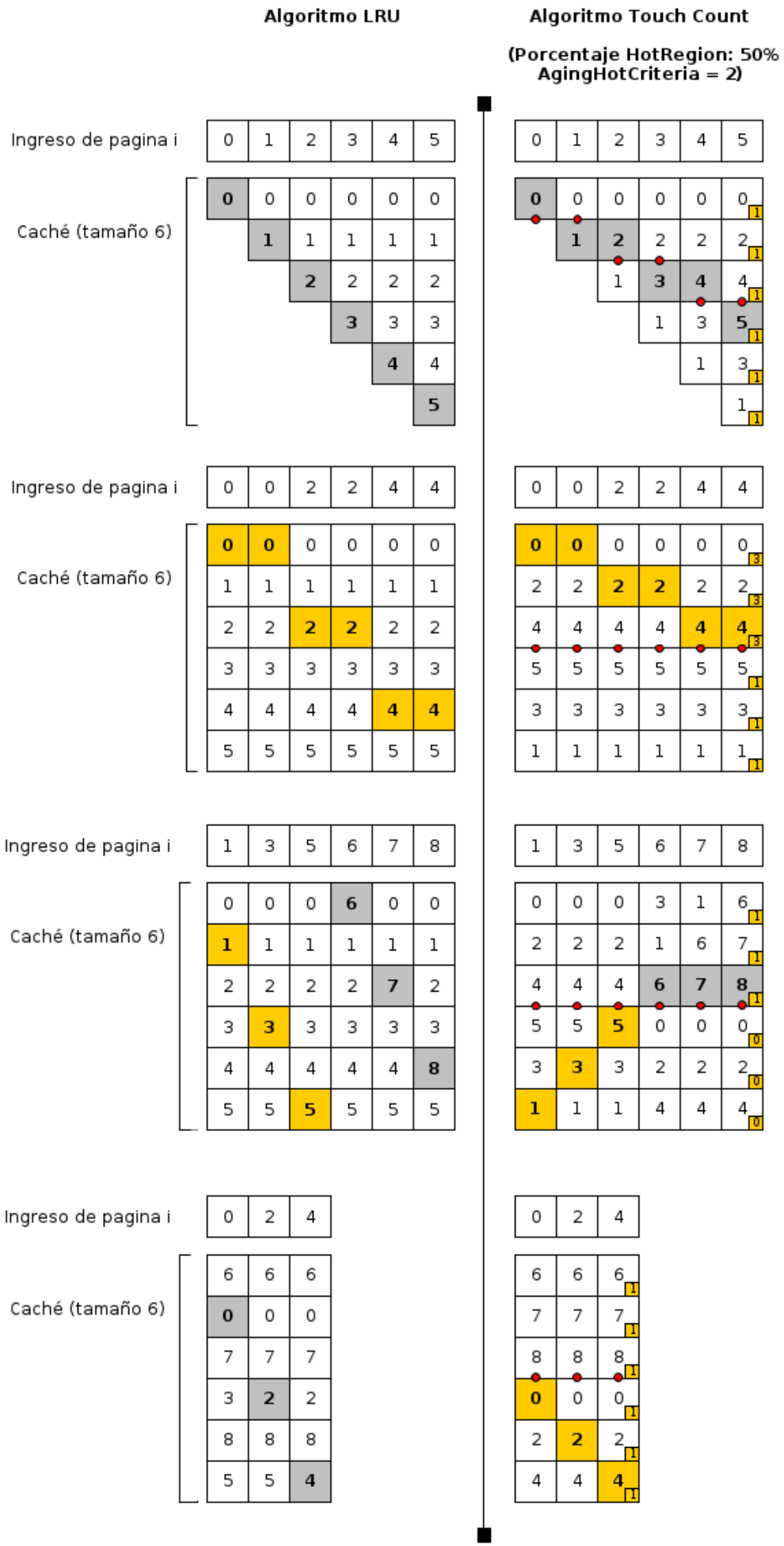
5) file scan sobre la tabla A: condicionando a los registros con id par: sólo en Touch Count las páginas todavía exisitŕan en caché. ya que las reiteradas referncias del punto 2 habrán hecho que Touch Count les diera un pero mayor y por ende las habría salvado en la *región caliente*.

La conclusion 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 ultimamente, posiblemente no sean tenida en cuenta por el algoritmo LRU mientras que Touch Count las tendrÃ¡ 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. Mejor caso LRU

4.5. Mejor caso MRU

4.6. Mejor caso *Touch Count*

4.7. Peor caso *Touch Count*