NOMBRE: Benjamín Farías Valdés

N.ALUMNO: 17642531

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2020

Control 1

Doy mi palabra que la siguiente solución de la pregunta 1 fue desarrollada y escrita individualmente por mi persona según el código de honor de la Universidad.

Figura 1: Firma

1. Buffer Manager

1.1. Jerarquía

La estructura de datos de los frames se **dividirá en 2 buffers**, uno para la memoria caché y otro para la principal. Ambos tendrán la estructura típica de un buffer, guardando las páginas en frames de su mismo tamaño, donde los frames se podrán guardar dentro de un arreglo en memoria. Al poseer 2 niveles, el nivel 1 (caché) estará conectado al 2 (RAM), y el nivel 2 estará conectado al disco. Adicionalmente, como el procesador tiene acceso directo tanto al caché como a la RAM, este podrá leer y escribir en páginas que estén en ambos tipos de memoria, por lo tanto, en caso de no poder colocar una página en el caché, se podrá acceder desde la RAM de todas formas.

Como se puede acceder directamente a las páginas del nivel 1 y 2, el funcionamiento del **pinning** será aplicado para cada nivel de forma **independiente**. Para que esto funcione correctamente, se debe evitar tener la misma página guardada tanto en la RAM como en el caché, de forma que al copiar una página desde el nivel 2 al 1, esta debe eliminarse del nivel 2 después del traslado. Esto además permite utilizar más eficientemente el espacio en el nivel 2 (ya que no tiene copias redundantes de páginas). Dicho esto, cada página será **marcada con pin** cuando ingrese a su frame respectivo, y será **'soltada'** cuando deje de ser solicitada por procesos. Al estar sin pin, podrá ser devuelta desde el nivel 1 al 2 (al realizar un reemplazo en el caché), o desde el nivel 2 al disco (en caso de que estuviese siendo accedida a través de la RAM). Como una página solo puede estar en un 1 nivel de memoria a la vez, ante cualquier modificación de su contenido, será marcada como **dirty**, y cuando sea devuelta al disco se escribirán los cambios realizados en su ubicación respectiva.

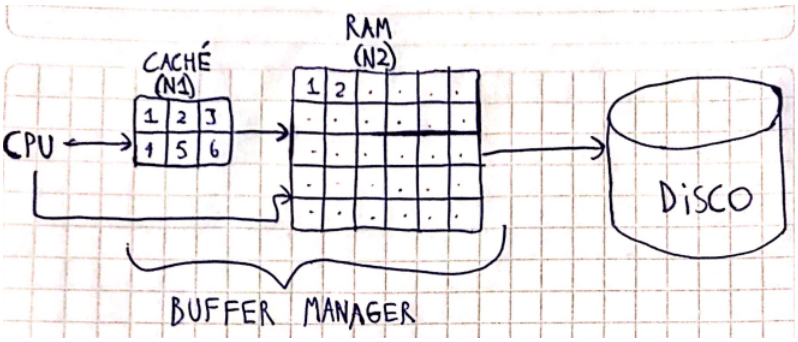

Figura 2: Estructura del Buffer Manager Jerárquico

Cuando se solicite una página, primero será buscada en el caché. Si no está, entonces se buscará en la RAM, y en caso de encontrarla, será copiada al caché en algún frame libre. Si al copiarla no habían frames libres en el caché, se utilizará la política de reemplazo del nivel 1 para **intercambiarla** con otra página del nivel 1, quedando esta última ahora en el nivel 2. En caso de que el caché posea todas sus páginas **pinneadas**, no se podrá copiar la página solicitada, y será entonces accedida directamente a través del nivel 2 (gracias a la conexión CPU-RAM). Finalmente, en caso de que la página solicitada no se encuentre en ninguno de los 2 niveles, esta será copiada desde el disco a la RAM (si el nivel 2 tiene todas sus páginas **pinneadas**, entonces deberá esperar que alguna se libere), ocupando la política de reemplazo del nivel 2 si es que corresponde.

En cuanto a las políticas de reemplazo, para ambos niveles se utilizará la política **LRU**. Esto permite que, al momento de trasladar una página entre niveles, se pueda también copiar el atributo de su frame que indica el último acceso a dicha página, evitando así que una página que no ha sido accedida en mucho tiempo se priorize por sobre otras debido a que venía desde otro nivel.

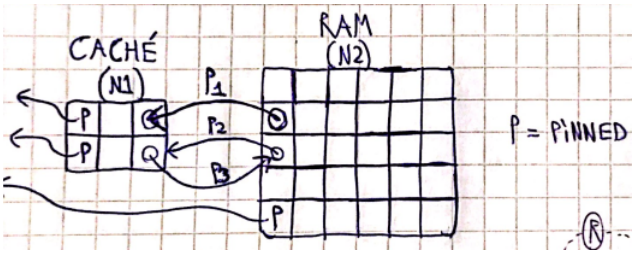

Figura 3: Traslado de Páginas entre Niveles

1.2. Políticas

Como se mencionó anteriormente, si se utilizan 2 políticas distintas entre niveles, puede ocurrir que la incompatibilidad entre ambas políticas cause que una página que no es necesaria para un proceso actualmente sea priorizada sobre otras que si son necesarias, simplemente porque llegó **'como nueva'** a un nivel luego de trasladarse desde otro. Un ejemplo más concreto se puede hacer con **LRU** en el caché y **FIFO** en la RAM. Si una página del caché no ha sido accedida recientemente, será la primera en ser reemplazada en el caché cuando se solicite una nueva página que no se encuentra en el nivel 1. Al ocurrir esto, la página reemplazada ingresará al nivel 2, y debido a FIFO, será priorizada respecto a todas las otras que ya estaban en la RAM. Si estas otras páginas están siendo accedidas recurrentemente de forma directa (CPU-RAM), entonces claramente son más importantes que la nueva página que acaba de ingresar, pero por la política FIFO serán reemplazadas primero debido a que ingresaron primero al nivel (**First In, First Out**).

NOMBRE: Benjamín Farías Valdés

N.ALUMNO: 17642531

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2020

Control 1

Doy mi palabra que la siguiente solución de la pregunta 2 fue desarrollada y escrita individualmente por mi persona según el código de honor de la Universidad.

Figura 4: Firma

2. Índices

2.1. B+Tree

Se realizará una demostración por **Inducción**:

- **Caso Base:** El árbol tiene 1 solo nodo directorio, por lo tanto el acceso a todas las páginas de datos toma tiempo constante ($O(1)$). Cumple con tener el mismo costo de búsqueda para cualquier tupla.
- **Hipótesis Inductiva:** Tras realizar i inserciones/eliminaciones, el árbol sigue cumpliendo con tener el mismo costo de acceso a cualquier tupla.
- **Caso Inductivo:** Por demostrar que tras aplicar la operación de inserción/eliminación número $i+1$, el árbol sigue cumpliendo con el mismo costo de acceso a cada dato.

  Esto se demostrará por casos:

  - **A) Inserción:** Al realizar una inserción, primero se debe buscar la página que le corresponde al dato, bajando por una rama del árbol hasta llegar a ella (mediante comparaciones con los separadores de los directorios). Una vez encontrada la página, puede ocurrir que la inserción sea directa o también puede que no haya suficiente espacio y se deba realizar un **split**. En este último caso, el split divide la página en 2, y sube el nuevo valor separador al directorio que apunta a estas páginas. Luego, se aplica este algoritmo de forma recursiva, recorriendo el árbol hacia arriba

hasta llegar a un directorio que tenga espacio libre o hasta llegar a la raíz. Si llega hasta la raíz, y esta se encuentra llena, se volverá a realizar un último split, el que generará una nueva raíz y aumentará la altura del árbol en 1. Este es el único caso de inserción en el que se modifican los costos de acceso (debido al cambio en la altura del árbol). Aun así, la búsqueda de cualquier tupla siempre parte desde la raíz del árbol (por construcción), lo que significa que para cualquier búsqueda el costo aumentó en 1, y por lo tanto, sigue siendo **exactamente el mismo costo** de acceso para cualquier dato.

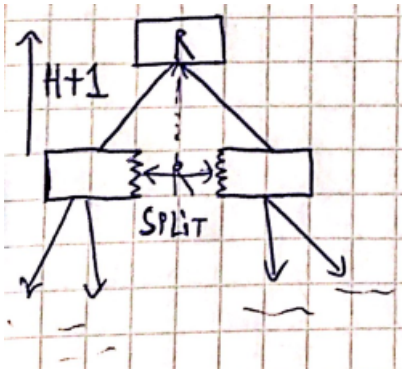

Figura 5: Aumento de Costo tras Inserción en B+Tree

- **B) Eliminación:** Al igual que en la inserción, primero se debe buscar la página que le corresponde al dato. Una vez encontrada la página, puede ocurrir que la eliminación sea directa o también puede que no se cumpla con la condición del mínimo de uso de cada página y se deba realizar un **unsplit** (asumiendo que no es posible rebalancear). En este último caso, el unsplit combina las páginas en 1, y elimina uno de los valores separadores del directorio que apunta a esta página. Luego, se aplica este algoritmo de forma recursiva, recorriendo el árbol hacia arriba hasta llegar a un directorio que tenga más de la mitad del espacio ocupado, o hasta llegar a la raíz. Si llega hasta la raíz, y esta se encuentra con 1 solo elemento, se volverá a realizar un último unsplit, el que eliminará la raíz (combinándola con sus hijos) y disminuirá la altura del árbol en 1. Este es el único caso de eliminación en el que se modifican los costos de acceso (debido al cambio en la altura del árbol). Aun así, análogo a el caso de la inserción, la búsqueda de cualquier tupla siempre parte desde la raíz del árbol (por construcción), lo que significa que para cualquier búsqueda el costo disminuyó en 1, y por lo tanto, sigue siendo **exactamente el mismo costo** de acceso para cualquier dato.

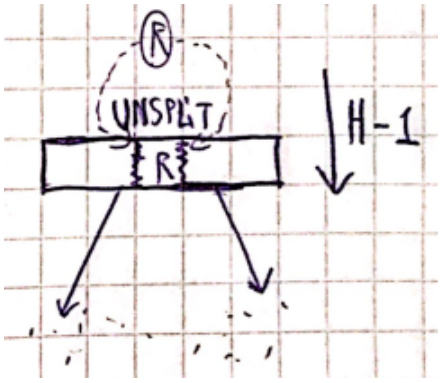

Figura 6: Disminución de Costo tras Eliminación en B+Tree

Debido a que en todos los casos de inserción/eliminación el costo de acceso se **mantiene o cambia en la misma proporción** para todas las tuplas, se cumple que tras $i+1$ cambios, el árbol seguirá cumpliendo la hipótesis inductiva.

Queda demostrado entonces, que el costo de búsqueda para cualquier par de tuplas en un B+Tree es exactamente el mismo ($O(log_{\frac{B}{2}}(\frac{2*T}{B}))$), independiente de la cantidad de modificaciones realizadas sobre el árbol.

2.2. Extendable Hashing

Como se requiere triplicar el tamaño del directorio, la representación binaria ya no será muy útil, debido a que aumentar el número de bits a utilizar no calzará siempre con la cantidad requerida de entradas del directorio. Se debe ocupar una representación **ternaria** (codificando los números con base 3). Esto puede ser adaptado binariamente (usando sólo 0's y 1's), por ejemplo, utilizando un conjunto adicional de bits del mismo largo que las entradas del directorio, cuyos bits sean 1 si es que el número en esa misma posición dentro de la palabra en el directorio debería corresponder a un 2 (ya que el 2 no puede ser representado con 1 sólo bit). También podría ser adaptado utilizando otros tipos, tales como int.

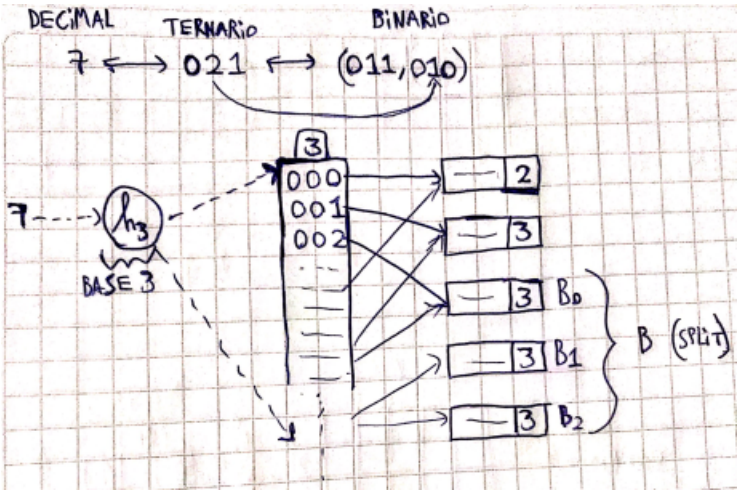

Figura 7: Representación Ternaria para Extendable Hashing

Asumiendo que se implementó la representación en base 3, entonces al momento de que ocurra un **bucket split** cuya profundidad local era igual a la global, el directorio podrá utilizar un dígito adicional ternario para triplicar la cantidad de entradas. El hecho de trabajar con esta representación para la **inserción** afecta también al **bucket split** mismo, ya que como ahora cada dígito de la representación puede tomar 3 valores (0, 1 y 2), se deben separar en 3 buckets nuevos (en vez de 2). El aumento de la profundidad local seguirá siendo de 1 tras un split. En cuanto a la búsqueda, seguirá de forma similar a la representación binaria: aplicar hash, obtener los primeros n dígitos **ternarios** y obtener el bucket correspondiente.

Este cambio mantiene la misma complejidad del **Extendable Hashing** normal, ya que el tiempo de acceso al directorio y a la página (siguiendo los punteros) es constante, y al triplicar el tamaño del directorio la complejidad de recorrer las nuevas entradas es de $2D$, lo que sigue siendo $O(D)$.