Ayudantía 7

Buffer manager, índices y B+trees

IIC2413, Bases de Datos
Benjamín Saavedra

Notación y convenciones (recordatorio)

Buffer manager.

- Cada frame guarda una página, su `#pin` (cuántos procesos la están usando) y `dirty` (si fue modificada). `pin(p)` trae la página si no está (una **lectura**) y aumenta su `#pin`; `unpin(p, dirty)` lo disminuye y, si corresponde, marca la página como modificada.
- Cuando no hay frames libres, la **política de reemplazo** elige una víctima entre los frames con `#pin` $= 0$; si está `dirty`, primero se escribe en disco (una **escritura**). Si todos tienen `#pin` ≥ 1, el `pin` debe esperar.
- **FIFO** saca la página cargada hace más tiempo; **LRU**, la de último uso (`pin` o `unpin`) más antiguo; **Random**, una al azar; **MRU** (Ramakrishnan & Gehrke), la de último uso más reciente.
- `usar(X)` abrevia `pin(X)` seguido inmediatamente de `unpin(X, false)`.

B+trees de orden d.

- Un directorio es $p_0\, k_1\, p_1 \ldots k_n\, p_n$ con $n \leq 2d$; los datos k^* alcanzables desde p_i cumplen $k_i \leq k < k_{i+1}$. Las hojas están ordenadas y doblemente ligadas.
- Todas las hojas están al mismo nivel. Toda hoja tiene entre d y $2d$ datos y todo directorio que no es la raíz, entre d y $2d$ separadores. La raíz puede tener un solo separador.
- **Split de hoja**: $P_1 = k_1^* \ldots k_d^*$, $P_2 = k_{d+1}^* \ldots k_{2d+1}^*$ y el separador k_{d+1} se *copia* al padre. **Split de directorio**: quedan d separadores a cada lado y k_{d+1} *sube* al padre. La altura sólo crece cuando la raíz hace split.
- **Eliminación**: si la hoja queda con menos de d datos se intenta, en orden, redistribuir con el vecino izquierdo, con el derecho, unsplit con el izquierdo y unsplit con el derecho, siempre con vecinos *del mismo padre*. Al eliminar sin rebalancear, los separadores no cambian.
- Se inserta sin redistribución, salvo que se pida. Una búsqueda por valor lee una página por nivel.

Pregunta 1

Una consulta hace k recorridos completos (*scans*) de una relación de $N+1$ páginas, $1, 2, \ldots, N+1$, siempre en ese orden: la secuencia de peticiones es $\texttt{usar}(1), \ldots, \texttt{usar}(N+1)$ repetida k veces. El buffer tiene N frames y parte vacío; nadie más lo usa.

Por ejemplo, con $N = 3$ frames, la tabla tiene $N+1 = 4$ páginas (1, 2, 3 y 4). Con $k = 2$ recorridos, las peticiones son:

$$1,\ 2,\ 3,\ 4,\ \ 1,\ 2,\ 3,\ 4$$

Esto ocurre, por ejemplo, al calcular un join $R \bowtie S$ con dos `for` anidados: por cada página de R se recorre S completa buscando las tuplas que calzan.

```
for cada página r de R:
  for cada página s de S:   (se recorre S completa otra vez)
    agregar al resultado los pares de tuplas de r y s que calzan
```

Si R tiene k páginas y S tiene $N+1$, las páginas de S se piden en el orden de arriba, k veces.

(a) Explique y argumente por qué con LRU *toda* petición produce una lectura a disco, es decir, por qué se hacen $k(N+1)$ lecturas. ¿Ocurre lo mismo con FIFO?

(b) Considere la política MRU. Para $N = 3$ y $k = 3$, muestre el contenido del buffer en cada petición y cuente las lecturas. Luego estime el número de lecturas para N y k arbitrarios.

(c) Dé una secuencia de peticiones (sobre un buffer de 3 frames) en la que MRU haga *muchas* más lecturas que LRU. ¿Qué conclusión saca?

(d) A diferencia de un sistema operativo, un DBMS *sabe* qué patrón de acceso va a tener cada operador. Proponga cómo extender la interfaz `pin`/`unpin` para aprovechar este conocimiento, y dé un ejemplo de operador que se beneficiaría.

(adaptada de Ramakrishnan & Gehrke, Ej. 9.12 y §9.4.1)

Pregunta 2

Considere la relación **Players**(id, name, team, age) con la siguiente instancia:

id	name	team	age
1	Pedro	Arsenal	23
2	Ana	Chelsea	27
3	Carlos	Liverpool	25
4	Ana	Barcelona	22
5	Carlos	Barcelona	30
6	Ana	Real Madrid	31

En clases vimos que se puede crear un índice (un B+tree) sobre *más de un* atributo:

```
CREATE INDEX PlayersNameTeamIndex ON Players (name, team);
```

(a) ¿Qué significa un índice sobre dos atributos? ¿Se crea un B+tree para name y otro para team? Escriba las llaves del índice `(name, team)` en el orden en que quedan en las hojas.

(b) ¿Es `INDEX(name, team)` lo mismo que `INDEX(team, name)`? Para responder, escriba las llaves de `INDEX(team, name)` en el orden en que quedan en las hojas y compare con (a).

(c) Para cada consulta, indique si `INDEX(name, team)` permite responderla *sin recorrer todo el índice*, y explique brevemente.

(I) `SELECT * FROM Players WHERE name = 'Ana'`

(II) `SELECT * FROM Players WHERE name = 'Ana' AND team = 'Chelsea'`

(III) `SELECT * FROM Players WHERE team = 'Chelsea'`

¿Qué índice usaría para la consulta (iii)?

(d) Suponga que se tiene un índice sobre tres atributos, `INDEX(A, B, C)`. ¿En cuáles de las siguientes condiciones del `WHERE` el índice permite encontrar directamente las tuplas buscadas?

(1) `A = a` (2) `A = a AND B = b` (3) `A = a AND B = b AND C = c`
(4) `B = b` (5) `C = c`

(e) Considere la consulta

```
SELECT * FROM Players WHERE name = 'Ana' AND age >= 25
```

y los índices `INDEX(name, age)` e `INDEX(age, name)`. Para cada uno, explique cómo se respondería la consulta (dónde empieza la búsqueda y hasta dónde se hace el scan) y cuántas entradas del índice se revisan. ¿Cuál conviene?

Pregunta 3

En clases supusimos que los valores indexados son distintos. Sin embargo, es muy común crear un índice sobre un atributo que no es llave (por ejemplo, `CREATE INDEX ON Players(age)`).

(a) Considere un B+tree de orden $d = 2$ vacío. Se inserta 3^* y luego cuatro veces el valor 7^*, aplicando literalmente el algoritmo de inserción. Muestre el árbol resultante, indique qué propiedad del B+tree deja de cumplirse y dé una consulta que se responda incorrectamente con `busquedaEnArbol`.

(b) Modifique la condición sobre los separadores y el algoritmo de búsqueda para que la consulta $A = k$ encuentre *todas* las ocurrencias de k, aunque estén repartidas en varias hojas y los separadores se repitan en el directorio.

(c) Otra solución es indexar el par (A, rid), donde rid identifica la tupla. Explique por qué esto resuelve el problema, cómo se responde $A = k$, y qué ventaja tiene al *eliminar* una tupla específica.

(d) Una tercera solución usa *páginas de overflow* en las hojas. Explique la idea y compare con las anteriores. ¿Se sigue cumpliendo que el costo de una búsqueda es $O(\log_d n)$?

(Examen 2020, P2; Ayudantía 1 (semestre pasado); Ramakrishnan & Gehrke, Ej. 10.11)

Pregunta 4

En clases se dijo que las políticas de reemplazo del buffer manager "son heurísticas y todas tienen un peor caso". En esta pregunta vamos a encontrarlos.

Considere un buffer manager con **3 frames**, inicialmente vacío. Escribimos `usar(X)` para abreviar `pin(X)` seguido inmediatamente de `unpin(X, false)`; así, entre petición y petición todas las páginas tienen `#pin` $= 0$ y cualquiera puede ser víctima. Recuerde:

- **FIFO** saca la página que lleva *más tiempo cargada* en el buffer.
- **LRU** saca la página que lleva *más tiempo sin usarse*.
- **Random** saca una página al azar.

Cada secuencia indica las páginas sobre las que se hace `usar`, en orden.

(a) Para la secuencia $S_1 = A, B, C, D, A, B, C, D$, muestre el contenido del buffer después de cada petición con FIFO y con LRU, y cuente las lecturas a disco. ¿Por qué este es el *peor caso* posible para ambas?

(b) Repita (a) con $S_2 = A, B, C, A, D, A, E, A, B$. ¿Qué política hace menos lecturas? ¿Qué tiene de especial la página A?

(c) Repita (a) con $S_3 = A, B, C, A, D, B$. ¿Qué política hace menos lecturas? ¿Por qué falla la otra?

(d) ¿Se puede repetir el argumento de (a) con Random, es decir, garantizar que *toda* petición de S_1 sea una lectura a disco? ¿Cuál es la ventaja y cuál la desventaja de elegir al azar?

(e) Con lo anterior, complete la tabla: para cada política, describa en una línea cuándo funciona *bien* y cuándo funciona *mal*.

Política	Funciona bien cuando...	Funciona mal cuando...
FIFO		
LRU		
Random		

Pregunta 5

Un buffer manager está pensado para gestionar la memoria RAM, pero la memoria de un computador tiene múltiples niveles. Suponga que desea diseñar un buffer jerárquico de dos niveles: en el **nivel 2** está el buffer de la RAM, que puede leer y escribir páginas del disco duro; en el **nivel 1** está el buffer de la memoria caché, que sólo puede leer y escribir páginas en la RAM (el nivel 1 nunca accede directamente al disco; siempre pasa por el nivel 2).

(a) Diseñe una estructura de buffer manager jerárquico de dos niveles. Debe definir al menos (i) la estructura de datos de los frames de cada nivel, (ii) el funcionamiento de `pin` y `unpin`, y (iii) la política de reemplazo de cada nivel. Justifique.

(b) Suponga que ambos niveles tienen la misma cantidad de frames. ¿Qué ocurre en su diseño si la política de reemplazo del nivel 1 es distinta a la del nivel 2?

(Control 1, 2020)

Pregunta 6

(a) Se quiere garantizar que las hojas estén al menos a *dos tercios* de su capacidad en vez de a la mitad. Para esto, cuando una hoja P está llena y su vecino Q (con el mismo padre) también, en lugar de dividir P en dos se reparten los $4d + 1$ datos de P, Q y el nuevo valor en *tres* hojas. Si Q tiene espacio, se redistribuye. Explique en detalle cómo quedan las tres hojas y qué cambia en el padre. ¿Qué ocupación mínima garantiza para las hojas creadas de esta forma? ¿Cómo cambia la cota de la altura? ¿Qué costo adicional tiene?

(b) Un compañero propone guardar en cada nodo un puntero a su *padre*, para no tener que recordar el camino durante la inserción: se busca la hoja, se inserta, y si hay split se sube por los punteros al padre. Suponga que un directorio N hace split en N y N_2. ¿Qué pasa con los punteros al padre de los hijos de N? Proponga dos formas de arreglarlo, identifique una desventaja importante de cada una y concluya.

((a) propia, en el estilo de C1 2020 P2; (b) Ramakrishnan & Gehrke, Ej. 10.9)