Pauta 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)

Solución.

(a) Con el buffer lleno, LRU guarda exactamente las N páginas pedidas más recientemente (siempre saca la que lleva más tiempo sin usarse). En el recorrido cíclico, justo antes de pedir una página, las últimas N peticiones fueron las otras N páginas (por ejemplo, con $N = 3$, antes de pedir 1 se pidieron 2, 3 y 4). Por lo tanto toda petición falla, porque el buffer guarda las N más recientes y la pedida es la $(N+1)$-ésima más reciente. Las primeras también fallan porque el buffer parte vacío: en total, $k(N+1)$ lecturas.

Con FIFO ocurre lo mismo: como cada página se usa una sola vez entre dos cargas, el orden de carga coincide con el orden de uso, y FIFO elige la misma víctima que LRU.

(b) Con $N = 3$, $k = 3$ (en negrita, las peticiones que leen de disco):

$$\mathbf{1}\ \mathbf{2}\ \mathbf{3}\ \mathbf{4}\ 1\ 2\ \mathbf{3}\ 4\ 1\ \mathbf{2}\ 3\ 4$$

Detalle: tras el primer scan el buffer tiene $\{1, 2, 4\}$ (al pedir 4 se sacó 3, la usada más recientemente). En el segundo scan, 1 y 2 están; al pedir 3 se saca 2 (la más reciente); 4 está. En el tercero 1 está, 2 falla (sale 1), 3 y 4 están. Total: **6 lecturas**, contra 12 de LRU.

En general, MRU "congela" $N-1$ páginas en el buffer y hace rotar sólo un frame: después del primer scan hay aproximadamente una lectura cada N peticiones. Exactamente: $(N+1) + \left\lfloor \frac{(k-1)(N+1)}{N} \right\rfloor$ lecturas (verificado por simulación para $N = 3, 4, 5$ y $k \leq 4$), es decir, $\approx (N+1)(1 + (k-1)/N)$ contra $k(N+1)$.

(c) Por ejemplo `usar` de $A, B, C, D, C, D, C, D, \ldots$ (alternar C y D). Con LRU, tras cargar D (sale A) todas las peticiones siguientes son aciertos. Con MRU, al pedir D se saca C (la más reciente), luego al pedir C se saca D, y así: *cada* petición falla. Conclusión: ninguna política es mejor en general; LRU funciona bien con localidad temporal y MRU con recorridos cíclicos mayores que el buffer.

(d) Por ejemplo, agregar un parámetro de *hint* a `pin` ("esta página se usará una sola vez", "esta página se volverá a leer pronto") o permitir que el operador elija la política de reemplazo para las páginas que él pide. Un *nested loops join* que recorre la relación interna una vez por cada página de la externa es exactamente el escenario de (a): con MRU (o fijando en el buffer una parte de la relación interna) se evita la inundación secuencial. Otro ejemplo: un scan que se hace una sola vez puede marcar sus páginas como víctimas preferidas para no expulsar del buffer los niveles superiores de los B+trees.

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
```

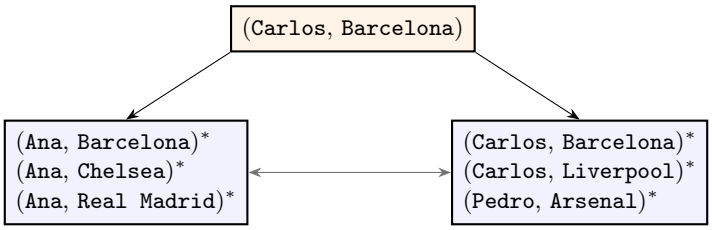
y los índices `INDEX(name, age)` e `INDEX(age, name)`. Para cada uno, explique cómo se respondería la consulta (dónde empieza la búsqueda y hasta dónde se hace el scan) y cuántas entradas del índice se revisan. ¿Cuál conviene?

Solución.

(a) Es **un solo** B+tree cuya llave es el *par* `(name, team)`. Las llaves se ordenan **lexicográficamente**, como las palabras en un diccionario: primero se compara `name` y, sólo si los nombres son iguales, se compara `team`. En las hojas quedan:

(Ana, Barcelona) < (Ana, Chelsea) < (Ana, Real Madrid) < (Carlos, Barcelona) < (Carlos, Liverpool) < (Pedro, Arsenal)

Por ejemplo, con orden $d = 2$ el árbol podría verse así (los separadores también son pares):



(b) **No.** Con `INDEX(team, name)` se compara primero `team`:

(Arsenal, Pedro) < (Barcelona, Ana) < (Barcelona, Carlos) < (Chelsea, Ana) < (Liverpool, Carlos) < (Real Madrid, Ana)

Ambos índices tienen las mismas tuplas, pero en *otro orden*: en (a) están agrupados los jugadores con el mismo nombre; acá, los del mismo equipo. Como veremos, eso cambia qué consultas aprovecha cada uno. **El orden de los atributos importa.**

(c)

(i) **Sí.** Todas las llaves que empiezan con `Ana` están *juntas* (las tres primeras). Es una búsqueda por rango: se baja con `busquedaEnArbol` hasta la primera llave con `Ana` y se hace scan hasta que el nombre deje de ser `Ana`.

(ii) **Sí.** Es una búsqueda por valor de la llave completa (`Ana`, `Chelsea`).

(III) **No.** Los equipos no están agrupados en todo el índice, sino sólo *dentro de cada nombre*: `Ana` → Barcelona, Chelsea, Real Madrid; `Carlos` → Barcelona, Liverpool; `Pedro` → Arsenal. Un `Chelsea` podría estar bajo cualquier nombre, así que hay que revisar todo el índice.

Para (iii) conviene `INDEX(team, name)` (o simplemente `INDEX(team)`): ahí todos los `Chelsea` están juntos.

(d) Sirven **(1), (2) y (3)**; no sirven **(4) ni (5)**. Es la *regla del prefijo izquierdo*: el índice `(A, B, C)` ordena por `A`, dentro de cada `A` por `B`, y dentro de cada `(A, B)` por `C`. Las tuplas que buscamos quedan contiguas sólo si fijamos un *prefijo* de los atributos: `A`, o `A, B`, o `A, B, C`. Si fijamos sólo `B` o sólo `C`, quedan repartidas por todo el índice.

(e)

- `INDEX(name, age)`. Orden de las llaves:
  (`Ana`, 22), (`Ana`, 27), (`Ana`, 31), (`Carlos`, 25), (`Carlos`, 30), (`Pedro`, 23). Se busca el punto de partida (`Ana`, 25): la primera llave mayor o igual es (`Ana`, 27). Luego se hace scan: (`Ana`, 27), (`Ana`, 31), y se detiene en (`Carlos`, 25) porque el nombre dejó de ser `Ana`. Se revisan **3 entradas** y *todas las que se devuelven son respuesta* (ids 2 y 6).
- `INDEX(age, name)`. Orden de las llaves:
  (22, `Ana`), (23, `Pedro`), (25, `Carlos`), (27, `Ana`), (30, `Carlos`), (31, `Ana`). Como el primer atributo es un *rango* (`age` ≥ 25), los `Ana` no están juntos: hay que hacer scan desde `age = 25` *hasta el final* y descartar a los que no se llaman `Ana`. Se revisan **4 entradas** para devolver 2, y con una tabla grande la diferencia es enorme.

Conviene `INDEX(name, age)`: **primero el atributo de la igualdad, después el del rango**. Así se busca un único punto de partida y el scan recorre sólo respuestas.

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.

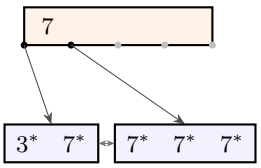
(c) Otra solución es indexar el par (A, rid), donde *rid* identifica la tupla. Explique por qué esto resuelve el problema, cómo se responde $A = k$, y qué ventaja tiene al *eliminar* una tupla específica.

(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)

Solución.

(a) 3^* y los primeros tres 7^* llenan la hoja raíz $\boxed{3^*\ 7^*\ 7^*\ 7^*}$. El cuarto 7 provoca un split $3^*\ 7^* \mid 7^*\ 7^*\ 7^*$ copiando el separador 7:



La hoja apuntada por p_0 contiene un 7^*, pero la regla exige que los datos bajo p_0 cumplan $k < k_1 = 7$. La consulta $A = 7$ hace `busquedaEnArbol`(7), que va a la derecha del separador ($7 \leq 7$), y hace scan hacia la derecha: *pierde* el 7^* de la primera hoja. Con más repeticiones el separador 7 incluso puede aparecer varias veces en un mismo directorio.

(b) Se relaja la condición a $k_i \leq k \leq k_{i+1}$ (los datos iguales a un separador pueden estar a ambos lados). Para buscar $A = k$ se busca la *ocurrencia más a la izquierda*: en cada directorio se baja por el puntero p_i con i mínimo tal que $k \leq k_{i+1}$ (es decir, a la *izquierda* del primer separador igual a k), y en la hoja se hace scan hacia la derecha por la lista ligada mientras el dato sea $\leq k$. Como en la hoja alcanzada puede no haber ningún k (si todas las ocurrencias quedaron en hojas siguientes), el scan debe continuar hasta encontrar un valor $> k$. El costo es $O(\log_d n + \#\text{hojas con } k)$.

(c) Los pares (A, rid) son todos distintos, así que el árbol vuelve a ser un B+tree sobre llaves únicas y se aplican los algoritmos de clases sin cambios. La consulta $A = k$ es la consulta de rango $(k, -\infty) \leq (A, rid) \leq (k, +\infty)$. *Ventaja al eliminar*: para borrar una tupla concreta se busca directamente el par (k, rid) en $O(\log_d n)$; con duplicados "puros" habría que recorrer todas las ocurrencias de k hasta encontrar la entrada de esa tupla. *Costo*: las llaves ocupan más espacio, luego d baja un poco.

(d) Cuando una hoja se llena de entradas con el *mismo* valor, en lugar de hacer split se encadena una página de overflow a esa hoja, donde se siguen guardando los repetidos; así los separadores nunca se repiten y la búsqueda del directorio sigue siendo la de clases. La búsqueda en el directorio sigue costando $O(\log_d n)$, pero recuperar todas las ocurrencias de un valor cuesta además la cadena de overflow, que puede ser arbitrariamente larga: el árbol deja de estar "balanceado" respecto a los datos, y buscar una tupla específica (para eliminarla) puede requerir recorrer toda la cadena. La solución (c) mantiene todas las garantías a cambio de llaves más grandes.

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		

Solución. En las tablas, cada celda muestra los tres frames $[F_1\ F_2\ F_3]$ después de la petición, siempre en la misma posición; en **negrita** la página que se acaba de cargar (reemplazando a la que estaba en ese frame) y • marca una lectura a disco.

(a) $S_1 = A, B, C, D, A, B, C, D$:

	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*
FIFO	[**A** – –] •	[A **B** –] •	[A B **C**] •	[**D** B C] •	[D **A** C] •	[D A **B**] •	[**C** A B] •	[C **D** B] •
LRU	[**A** – –] •	[A **B** –] •	[A B **C**] •	[**D** B C] •	[D **A** C] •	[D A **B**] •	[**C** A B] •	[C **D** B] •

Por ejemplo, al pedir D el buffer está lleno con $[A\ B\ C]$: ambas políticas sacan a A (la primera en entrar y también la que lleva más tiempo sin usarse), y D queda en su frame F_1. Luego se pide A, que acaba de salir, y hay que sacar a B; y así sucesivamente. Ambas hacen **8 lecturas**: *todas* las peticiones van a disco, así que el buffer no sirvió de nada. Es el peor caso porque se recorren cíclicamente 4 páginas con sólo 3 frames: cada vez que se pide una página, es justo la que se usó (y se cargó) hace más tiempo, que es exactamente la que ambas políticas acaban de sacar. Este patrón se llama *inundación secuencial* y aparece cuando una consulta recorre varias veces una tabla más grande que el buffer.

(b) $S_2 = A, B, C, A, D, A, E, A, B$:

	A	B	C	A	D	A	E	A	B
FIFO	[**A** – –]•	[A **B** –]•	[A B **C**]•	[A B C]	[**D** B C]•	[D **A** C]•	[D A **E**]•	[D A E]	[**B** A E]•
LRU	[**A** – –]•	[A **B** –]•	[A B **C**]•	[A B C]	[A **D** C]•	[A D C]	[A D **E**]•	[A D E]	[A **B** E]•

FIFO: 7 lecturas; LRU: 6. La página A es *popular*: se usa una y otra vez. LRU nunca la saca, porque siempre es una de las usadas más recientemente. FIFO, en cambio, sólo mira cuándo se *cargó*: al pedir D, A es la más antigua del buffer y la saca, aunque se acababa de usar, y hay que volver a leerla de inmediato. Es el peor caso típico de FIFO.

(c) $S_3 = A, B, C, A, D, B$:

	A	B	C	A	D	B
FIFO	[**A** – –]•	[A **B** –]•	[A B **C**]•	[A B C]	[**D** B C]•	[D B C]
LRU	[**A** – –]•	[A **B** –]•	[A B **C**]•	[A B C]	[A **D** C]•	[A D **B**]•

FIFO: 4 lecturas; LRU: 5. Al pedir D, LRU "apuesta" a que A, recién usada, se volverá a usar, y saca a B. Pero la siguiente petición es justamente B. LRU falla cuando una página que lleva rato sin usarse vuelve a pedirse *antes* que las usadas recientemente: el pasado no predijo el futuro. Junto con (b), esto muestra que ninguna de las dos es siempre mejor.

(d) **No.** En (a) pudimos construir la secuencia mala porque FIFO y LRU son *deterministas*: sabemos de antemano qué página van a sacar, así que basta con pedir justamente esa. Con Random no se puede saber cuál saldrá; en S_1, por ejemplo, al pedir D podría salir C en vez de A, y entonces la petición siguiente de A sería un acierto. Que *todas* fallen es posible, pero sólo por mala suerte, no porque la secuencia la obligue.

- *Ventaja*: no tiene un peor caso fijo que alguien pueda "armar", y es muy simple (no guarda ninguna información sobre el uso de las páginas).
- *Desventaja*: ignora completamente el uso, así que puede sacar una página popular como A en (b). Nunca aprovecha la localidad.

(e)

Política	Funciona bien cuando…	Funciona mal cuando…
FIFO	las páginas se usan por un rato y luego no se vuelven a pedir (por ejemplo, un recorrido de una sola pasada).	hay una página popular que se cargó temprano: la saca aunque se siga usando (S_2). También con recorridos cíclicos mayores que el buffer (S_1).
LRU	hay *localidad temporal*: lo usado recientemente se vuelve a usar (por ejemplo, la raíz de un B+tree, que se lee en cada búsqueda) (S_2).	se recorre cíclicamente un conjunto de páginas más grande que el buffer (S_1), o vuelve una página que llevaba rato sin usarse (S_3).
Random	no se conoce el patrón de acceso y se quiere evitar un peor caso sistemático; es barata de implementar.	hay páginas mucho más usadas que otras: puede sacarlas al azar, porque no aprovecha la localidad.

Conclusión: **ninguna política es la mejor para todas las secuencias**. Cada una es una apuesta sobre cómo se van a pedir las páginas en el futuro, y para cada política determinista existe una secuencia que la hace fallar en todas las peticiones.

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)

Solución. Hay más de un diseño correcto; lo importante es que sea consistente. Una solución posible (*inclusiva*: toda página en el nivel 1 está también en el nivel 2):

(a)

- *Frames.* Nivel 2: `(pageno, #pin, dirty)` como en clases. Nivel 1: `(pageno, #pin, dirty)` más una referencia al frame del nivel 2 que contiene la página.
- `pin(p)` en el nivel 1: si p está en el nivel 1, se incrementa su `#pin` y se retorna. Si no, se elige una víctima del nivel 1 (con `#pin` $= 0$); si está `dirty`, se copia su contenido al frame del nivel 2 y se llama $\texttt{unpin}_2$(víctima, true); si no, $\texttt{unpin}_2$(víctima, false). Luego se llama $\texttt{pin}_2(p)$, que puede leer p de disco con el algoritmo usual, y se copia la página al frame del nivel 1 con `#pin` $:= 1$.
- Invariante clave: mientras una página esté en el nivel 1, está *pineada* en el nivel 2 (su `#pin` en el nivel 2 cuenta al nivel 1 como un usuario). Así el nivel 2 nunca expulsa una página que el nivel 1 está usando y la escritura de una página `dirty` del nivel 1 siempre tiene un frame destino.
- `unpin(p, dirty)` en el nivel 1: decrementa el `#pin` y marca `dirty`; no toca el nivel 2 (la página sigue cacheada en el nivel 1).
- *Políticas*: LRU en el nivel 1 (accesos con localidad temporal). En el nivel 2 LRU sobre las páginas que no están en el nivel 1 (las demás están pineadas y no son candidatas).

(b) Con el diseño inclusivo y tamaños iguales, cuando el nivel 1 está lleno *todas* las páginas del nivel 2 están pineadas por el nivel 1. Para traer una página nueva, el nivel 1 primero expulsa su víctima (lo que hace $\texttt{unpin}_2$ de esa página) y recién entonces pide la nueva: la *única* candidata del nivel 2 es la página que el nivel 1 acaba de soltar. Por lo tanto la política del nivel 2 es irrelevante: el nivel 2 siempre replica exactamente el contenido del nivel 1 y no aporta ningún acierto adicional (sólo agrega costo). Si en cambio el diseño no fuera inclusivo, políticas distintas harían que ambos niveles guardaran conjuntos distintos, lo que puede ser bueno (el nivel 2 cubre lo que el 1 no tiene) pero complica mantener la consistencia de las páginas `dirty`. Conclusión esperada: el nivel 2 sólo tiene sentido si es estrictamente más grande que el nivel 1.

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)

Solución.

(a) Sean los $4d+1$ valores ordenados $v_1 < \cdots < v_{4d+1}$ (los $2d$ de P, los $2d$ de Q y el nuevo; suponga Q a la derecha de P). Se crean tres hojas consecutivas con $\lfloor (4d+1)/3 \rfloor$ o $\lceil (4d+1)/3 \rceil$ valores cada una, reemplazando a P y Q en la lista doblemente ligada. En el padre, el separador que había entre P y Q se reemplaza por el primer valor de la segunda hoja, y se agrega un separador nuevo (el primer valor de la tercera hoja) con un puntero nuevo: el padre gana *un* separador, igual que con un split normal, y si se llena hace split como siempre (o también $2 \to 3$ si se quiere la garantía en directorios).

Ocupación: cada hoja nueva tiene al menos $\lfloor (4d+1)/3 \rfloor \geq \lfloor 4d/3 \rfloor$ datos, es decir, $\approx \frac{2}{3}$ de su capacidad $2d$ (por ejemplo, $d=3$: 13 valores en hojas de 4, 4 y 5 sobre capacidad 6). La primera división, cuando la raíz es una sola hoja sin vecinos, tiene que ser un split normal $1 \to 2$. Para mantener la garantía también al eliminar, el unsplit debe ser $3 \to 2$ (fusionar tres hojas en dos).

Altura: si cada nodo (salvo la raíz) tiene al menos $\approx \frac{4d}{3}$ hijos, la cota queda $H \leq \log_{4d/3}(n) + O(1)$ en lugar de $\log_d(n) + O(1)$: algo más baja, pero sigue en $O(\log_d n)$.

Costo: hay que leer (y escribir) el vecino Q incluso para descubrir que está lleno, y se escriben tres hojas en lugar de dos. La eliminación es más compleja. A cambio, se usa mejor el espacio y el árbol tiene menos páginas. (Esta variante se conoce como B*-tree.)

(b) Tras el split, los hijos que quedaron en N_2 (son $d+1$ punteros) siguen apuntando a N como su padre: sus punteros quedan incorrectos.

- *Arreglarlos de inmediato*: leer y reescribir cada uno de esos $d+1$ hijos. Desventaja: un split de directorio pasa de costar $O(1)$ escrituras a $O(d)$ lecturas y escrituras extra (cientos de I/O), y puede repetirse en cada nivel.
- *Arreglarlos de forma perezosa*: corregir el puntero cuando el hijo se lea por otra razón (por ejemplo, al bajar desde N_2). Desventaja: una página que sólo se iba a leer ahora queda `dirty` y hay que escribirla; además, hasta entonces el puntero está mal y hay que poder detectarlo.

Conclusión: los punteros al padre no valen la pena. El algoritmo recursivo (o recordar el camino en una pila durante la bajada, que tiene largo $H+1$) entrega el padre gratis.