

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Almacenamiento físico, páginas y buffer pool

Almacenamiento, índices y consultas

Gerson Garrido

Semana 9

Hoy

Sesión A: almacenamiento físico

1. La jerarquía de almacenamiento y el bloque
2. Discos magnéticos: seek, rotación, transferencia
3. Flash y SSD
4. Acceso a bloques y el modelo de costo

Sesión B: archivos, páginas y buffer pool

5. Registros de longitud fija y variable
6. Slotted pages y el record id
7. Heap files, archivos secuenciales, diccionario
8. El buffer pool: pin, dirty, reemplazo

Lectura

Silberschatz, secs. 12.1–12.4 y 12.6 (sesión A; 12.5, RAID, complementaria) y 13.1–13.5 (sesión B; 13.6 opcional). CS186, notas 3 (*Disks and Files*) y 5 (*Buffer Management*). CMU 15-445, clases #03 y #04. PostgreSQL, *Database Page Layout*. Además: PostgreSQL *Database File Layout*, *TOAST* y `shared_buffers`; CMU #05; Crotty, Leis y Pavlo, *Are You Sure You Want to Use MMAP…* (CIDR 2022); opcional, CMU #06.

SECCIÓN 01

La jerarquía de almacenamiento

01

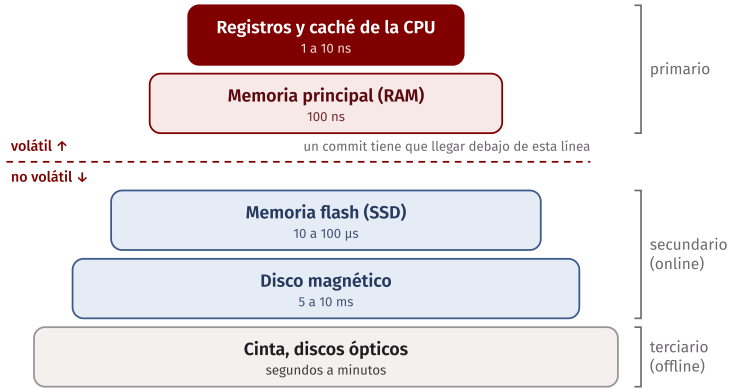
Cambiamos de lado

- Ocho semanas usando PostgreSQL desde arriba: esquema, SQL, restricciones, `pg`.
- Hasta la semana 12 lo abrimos: cómo guarda, lee, indexa y ejecuta.
- Un DBMS existe porque los datos no entran en memoria o tienen que sobrevivir a un apagón.
- En los dos casos viven en un dispositivo miles de veces más lento que la RAM.

La pregunta de este tramo

¿Cuántas veces hay que ir al disco? Páginas con slots, buffer pool, índices, joins y optimizador existen para que ese número sea pequeño.

Del registro de la CPU a la cinta



hacia abajo: más lento, más barato por byte y más grande

El bloque: la unidad de transferencia

- El disco no entrega una fila: entrega un **bloque** entero.
- Para leer 60 bytes, se trae el bloque donde están.
- PostgreSQL: páginas de 8 KiB, fijas al compilar.
- `quipu` (P3): páginas de 4 KiB.

psql

```
chasqui_w09=# SHOW block_size;
 block_size
------------
 8192
(1 row)
```

Desde hoy

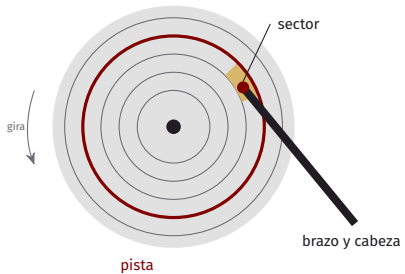
La RAM es entre 100 y 100 000 veces más rápida que el almacenamiento no volátil. El costo de un algoritmo se mide en bloques leídos y escritos, no en comparaciones ni en tuplas.

SECCIÓN 02

Discos magnéticos

02

Cómo está hecho un disco



- **Platos** que giran a 5400–15 000 RPM.
- **Pistas** concéntricas, divididas en **sectores** (512 o 4096 bytes).
- Una cabeza por cara, todas en un mismo **brazo**: un **cilindro** a la vez.

Para leer un sector

1. **Seek**: mover el brazo a la pista (4–10 ms en promedio).
2. **Latencia rotacional**: esperar el sector (media vuelta).
3. **Transferencia**: leer los bytes (50–200 MB/s).

Un ejemplo con números

Disco de 7200 RPM, seek promedio 4 ms, 200 MB/s, bloques de 4 KiB.

- Una vuelta: $60\,\text{s}/7200 = 8{,}33$ ms. Latencia rotacional: 4,17 ms.
- Transferir 4 KiB: $4096/(200 \times 10^6)\,\text{s} =$ 0,02 ms.
- Un bloque al azar: $4 + 4{,}17 + 0{,}02 \approx$ 8,2 ms: unas 120 lecturas por segundo.

120 bloques al azar

$120 \times 8{,}2$ ms ≈ 984 ms

120 bloques seguidos

$8{,}17 + 120 \times 0{,}02 \approx 10{,}6$ ms

Casi cien veces menos

Casi todo el costo está en posicionarse. Leer el bloque siguiente es casi gratis: el acceso **secuencial** le gana al **aleatorio**.

Fallos: MTTF

- **MTTF** *(mean time to failure)*: por ejemplo, 1 000 000 de horas.
- No es que un disco dure 114 años.
- Con 1000 discos nuevos, se espera un fallo cada 1000 horas: cada mes y medio.
- Por eso los datos importantes no viven en un solo disco: RAID, replicación.

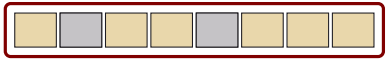
SECCIÓN 03

Flash y SSD

03

Sin partes móviles

- Sin brazo ni platos: no hay seek ni latencia rotacional.
- Leer 4 KiB al azar: decenas de µs. NVMe: cientos de miles de IOPS.



bloque de borrado (cientos de KiB a MiB)

páginas de flash (4 KiB): se leen y se escriben

gris: páginas inválidas

- Una página no se sobrescribe: hay que borrarla, y se borra por **bloques**.
- Borrar toma milisegundos; cada bloque aguanta de miles a cientos de miles de borrados.

La FTL y la amplificación de escritura

- Actualizar = escribir la versión nueva en otra página ya borrada e invalidar la vieja.
- La **flash translation layer** (FTL) mapea dirección lógica → página física.
- **Recolección de basura**: copiar las páginas válidas de un bloque y borrarlo entero.
- **Nivelación de desgaste** *(wear leveling)*: repartir los borrados.

Amplificación de escritura ***(write amplification)***

$\frac{\text{bytes escritos en la flash}}{\text{bytes que pidió el sistema}}$. Si guardar 1 GB obliga a escribir 3 GB, es 3: menos rendimiento y tres veces más desgaste.

En un SSD

La brecha entre secuencial y aleatorio es menor, pero no desaparece.

Valores de referencia

Operación	Tiempo aprox.
Caché L1 de la CPU	1 ns
Memoria principal	100 ns
4 KiB al azar, SSD NVMe	10–100 µs
4 KiB al azar, disco magnético	5–10 ms
Red dentro de un centro de datos	0,5 ms

Órdenes de magnitud típicos, no mediciones.

A escala humana

Si leer de memoria tomara 1 segundo: un SSD, minutos; un disco magnético, más de un día.

- Interfaces: SATA y SAS (de la época del disco), **NVMe** (PCIe, miles de pedidos en vuelo).
- En red: SAN (bloques), NAS (archivos), nube. Mismo modelo, otros números.

SECCIÓN 04

Acceso a bloques y costo

04

Ir menos al disco (sec. 12.6)

- **Buffering**: guardar bloques leídos (el buffer pool).
- **Read-ahead**: traer los siguientes antes de que se pidan.
- **Planificación**: atender en el orden del brazo (ascensor).
- **Organización**: lo que se lee junto, junto.
- **Buffers no volátiles** de escritura.
- **El log en su propio disco**: escrituras secuenciales.

El sistema operativo lo hace; el DBMS lo hace mejor

El sistema operativo adivina el patrón de acceso. El DBMS lo sabe: cuando empieza un sequential scan, sabe que va a leer toda la tabla.

El modelo de costo del libro

Costo de transferir b bloques con S posicionamientos

$$b \cdot t_T + S \cdot t_S$$

t_T: transferir un bloque. t_S: seek + latencia rotacional.

	t_S	t_T
Disco magnético	4 ms	0,1 ms
SSD	90 µs	10 µs

Deja fuera la CPU y la diferencia entre leer y escribir. Tosco, pero ordena bien los algoritmos. Semanas 10 a 12 (sec. 15.2).

Los 148 bloques de `shipment` en `chasqui_wo9`:

- Disco: en orden $4 + 148 \times 0{,}1 = 18{,}8$ ms; al azar $148 \times 4{,}1 = 606{,}8$ ms (32 veces).
- SSD: en orden $0{,}09 + 148 \times 0{,}01 = 1{,}57$ ms; al azar $148 \times 0{,}1 = 14{,}8$ ms (9 veces).

Tuplas no; bloques y seeks

Error común

Medir el costo en tuplas leídas. 1000 tuplas en 15 páginas seguidas: 1 seek y 15 transferencias. 1000 tuplas en 1000 páginas distintas: hasta .

RAID en una diapositiva (sec. 12.5, repaso de Sistemas Operativos)

- **Striping**: repartir bloques entre discos para leer en paralelo.
- **Mirroring** (RAID 1) y **paridad** (RAID 5) para tolerar fallos.
- Escritura pequeña en RAID 5: leer dato y paridad viejos, escribir dato y paridad nuevos: 4 E/S. En RAID 1: 2 escrituras en paralelo.
- Muchas escrituras pequeñas: RAID 1 (o 10) mejor que RAID 5.

SECCIÓN 05

De la tabla al archivo

05

La arquitectura de almacenamiento (sec. 13.1)



- Una relación → un **archivo**; un archivo → una secuencia de **páginas**; una página → **registros**.
- **Tupla**: concepto lógico. **Registro** *(record)*: los bytes que la representan.
- El buffer manager es la única puerta entre el disco y el resto del motor.

Dónde está una tabla de PostgreSQL

psql

```
chasqui_w09=# SELECT pg_relation_filepath('agency') AS path;
       path
------------------
 base/54184/54185
(1 row)
```

- `base/`*OID de la base/relfilenode*.
- Cambian con cada `make`.

ls (otro make, otros números)

```
-rw------- 1 postgres postgres 1212416 Nov  1 18:44 base/56475/56565
-rw------- 1 postgres postgres   24576 Nov  1 18:44 base/56475/56565_fsm
-rw------- 1 postgres postgres    8192 Nov  1 18:44 base/56475/56565_vm
```

El archivo principal de `shipment`, su **free space map** y su *visibility map* (semana 14). Más de 1 GiB: varios archivos.

Tamaños en páginas

pages.sql: sizes

```
    relname     |  bytes  | pages | relpages | reltuples | rows_per_page
----------------+---------+-------+----------+-----------+---------------
 shipment_event | 2801664 |   342 |      342 |     46421 |           136
 shipment       | 1212416 |   148 |      148 |     12000 |            81
 trip           |  360448 |    44 |       44 |      5914 |           134
 customer       |  155648 |    19 |       19 |      1800 |            95
 vehicle        |    8192 |     1 |        1 |        24 |            24
 route          |    8192 |     1 |        1 |       120 |           120
 agency         |    8192 |     1 |        1 |        29 |            29
(7 rows)
```

- Todo es múltiplo de 8192. `pg_relation_size`: el archivo ahora. `relpages`, `reltuples`: estadísticas de `VACUUM` y `ANALYZE`.
- Sin el `VACUUM` del `Makefile`: `trip` medía 64 páginas con filas en 44, y `shipment` 200 con filas en 148. `COPY` agrega páginas de a varias; `VACUUM` corta las vacías del final.

SECCIÓN 06

Registros

06

Registros de longitud fija

`route` con un formato propio:

- 4 enteros de 4 bytes + `base_fare` en 8 bytes = 24 bytes.
- En 4 KiB: $\lfloor 4096/24 \rfloor = 170$ registros; sobran 16 bytes.
- El registro i está en el byte $i \times 24$.
- **Cruzar el límite del bloque**: costaría dos lecturas. Solo registros completos; lo que sobra se pierde.

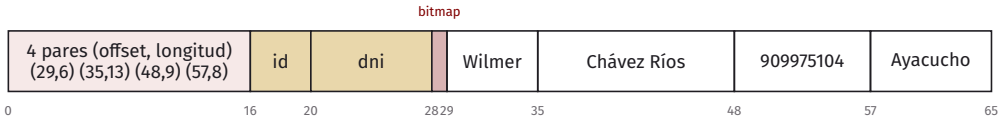
Borrar deja un hueco

1. Correr todos los siguientes: caro.
2. Mover el último al hueco: cambia su posición.
3. **Lista de libres** *(free list)*: el encabezado apunta al primer hueco, cada hueco al siguiente.

> **Ojo**
>
> Mover un registro es un problema si alguien (un índice) guarda su dirección.

Registros de longitud variable: el formato del libro

Primera fila de `customer`: `1, 61830975, Wilmer, Chávez Ríos, 909975104, Ayacucho`



- Parte fija: un par por atributo variable, los atributos fijos y el **bitmap de nulos**.
- Después, los bytes variables. Total: 65 bytes. (UTF-8: `á` e `í` ocupan 2 bytes.)
- `phone` en `NULL`: su bit en 1 y ningún byte para él.
- Llegar al atributo *k* es directo: no hay que recorrer los anteriores.

En PostgreSQL: varlena

psql

```
 attnum |   attname   |   type    | attlen | attalign
--------+-------------+-----------+--------+----------
      1 | customer_id | integer   |      4 | i
      2 | dni         | character |     -1 | i
      3 | first_name  | text      |     -1 | i
      4 | last_name   | text      |     -1 | i
      5 | phone       | text      |     -1 | i
      6 | city        | text      |     -1 | i
(6 rows)
```

- Encabezado de tupla (23 bytes) + bitmap + relleno; `t_hoff` dice dónde empiezan los datos.
- Atributos en el orden de la tabla, sin tabla de offsets.
- Cada varlena (`attlen = -1`) lleva su longitud delante.
- `char(8)` también es varlena.

Los bytes de verdad

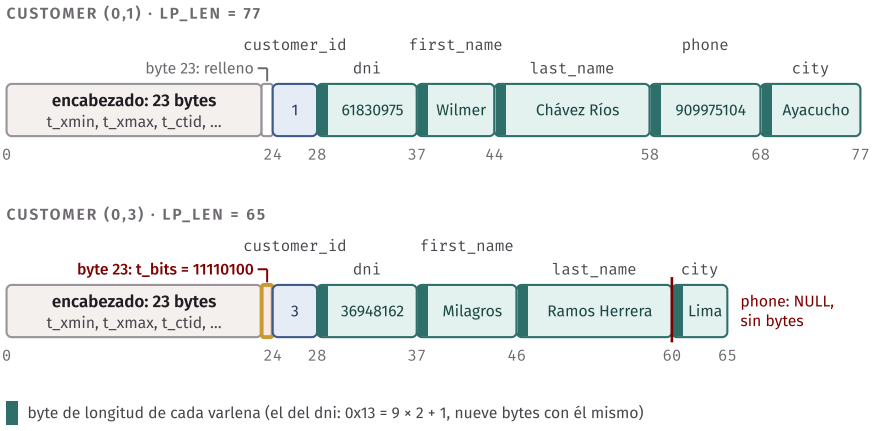
```
heap_page_items(get_raw_page('customer', 0))

 lp | lp_len | t_hoff |  t_bits  |         primeros_bytes
----+--------+--------+----------+--------------------------------
  1 |     77 |     24 |          | \x01000000133631383330393735 0f
  2 |     75 |     24 |          | \x02000000133137323332323035 0d
  3 |     65 |     24 | 11110100 | \x03000000133336393438313632 13
(3 rows)
```

- `01000000`: `customer_id` = 1 (little-endian).
- `13`: varlena corto, $\texttt{0x13} = 9 \times 2 + 1$: 9 bytes con el encabezado.
- `3631383330393735`: `61830975`.
- `0f`: $7 \times 2 + 1$, 6 bytes: `Wilmer`…

- Fila 1: $24 + 53 = 77$ bytes.
- Fila 3: `t_bits = 11110100`, 1 = no es `NULL` (al revés que el libro). El quinto, `phone`, en 0: no ocupa nada.
- El bitmap cabe en el byte 23: `t_hoff` sigue en 24.

Dos tuplas de `customer`, byte por byte



Alineación y registros grandes

- Un entero de 4 bytes va en un offset múltiplo de 4. Un varlena con encabezado de 1 byte no se alinea; uno de 4 bytes (más de 126 bytes de texto), sí.
- Sin offsets, llegar al atributo k obliga a recorrer los anteriores. PostgreSQL cachea los offsets fijos del principio.
- El orden de las columnas importa: `smallint` y luego `bigint` dejan 6 bytes de relleno.

Registros grandes: TOAST

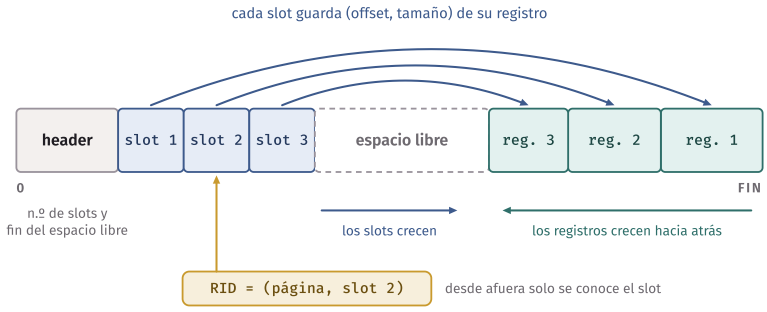
Si una fila pasa de unos 2 KB, PostgreSQL comprime sus varlena más grandes y, si no basta, los parte en trozos en una tabla TOAST aparte. En la fila queda un puntero de 18 bytes.

SECCIÓN 07

Slotted pages

07

La slotted page (sec. 13.2.2)



- **Header**: número de slots, fin del espacio libre.
- **Slot**: offset y tamaño de un registro.
- **Insertar**: el registro antes del último, un slot más al final.
- **Borrar**: marcar el slot libre; **compactar** mueve registros y corrige offsets.

El record id

Record id (RID)

El par (número de página, número de slot). Es lo que guardan los índices.

- Compactar mueve un registro dentro de su página: cambia el offset en el slot, no el RID.
- Un slot libre no se quita del arreglo si hay slots después: cambiaría sus números.

El slot es un nivel de indirección

Afuera de la página: (página, slot). Adentro: el slot dice el offset. La página se reorganiza sin invalidar ningún puntero externo.

La página de `agency`: el encabezado

pages.sql: header

```sql
SELECT lower, upper, special, pagesize,
       upper - lower AS free
FROM page_header(get_raw_page('agency', 0));
```

psql

```
 lower | upper | special | pagesize | free
-------+-------+---------+----------+------
   140 |  6280 |    8192 |     8192 | 6140
(1 row)
```

- Header de **24 bytes**; un *line pointer* de **4 bytes** por slot.
- $24 + 29 \times 4 = 140 =$ `lower`: fin de los slots.
- `upper`: inicio de la primera tupla.
- Libre: $6280 - 140 = 6140$ bytes, el 75 %.
- `special` = 8192: sin espacio especial (lo usan los índices).

Los slots y el `ctid`

pages.sql: slots y last_slots

```
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_ctid
----+--------+----------+--------+--------+--------+--------
  1 |   8128 |        1 |     57 |  13129 |      0 | (0,1)
  2 |   8072 |        1 |     56 |  13129 |      0 | (0,2)
  3 |   8016 |        1 |     49 |  13129 |      0 | (0,3)
  4 |   7952 |        1 |     64 |  13129 |      0 | (0,4)
  5 |   7880 |        1 |     72 |  13129 |      0 | (0,5)
(5 rows)

 lp | lp_off | lp_len
----+--------+--------
 29 |   6280 |     60
 28 |   6344 |     85
 27 |   6432 |     63
(3 rows)
```

- `lp_off`: la primera al final, cada una antes que la anterior; la 29 en `upper`.
- $8192 - 57 = 8135 \to 8128$: las tuplas empiezan en múltiplos de 8.
- `lp_flags`: 0 libre, 1 normal, 2 redirect, 3 dead.
- `ctid` = `(0,1)`: página 0, slot 1. El RID del libro.

La misma página, leída desde Node.js

pgpage.mjs: get_raw_page llega como un Buffer de 8192 bytes

```
  lower: page.readUInt16LE(12),
  upper: page.readUInt16LE(14),
  // un line pointer: 15 bits de offset, 2 de banderas, 15 de longitud
  const v = page.readUInt32LE(PAGE_HEADER_SIZE + (lp - 1) * LINE_POINTER_SIZE);
  return { lp, lp_off: v & 0x7fff, lp_flags: (v >>> 15) & 0x3, lp_len: v >>> 17 };
```

node

```
$ PGDATABASE=chasqui_w09 node dump.mjs agency 0
agency, página 0: lower=140 upper=6280 special=8192 slots=29 libre=6140 bytes
  lp=1 off=8128 flags=1 len=57 xmin=14018 xmax=0 ctid=(0,1)
  ...
  lp=29 off=6280 flags=1 len=60 xmin=14018 xmax=0 ctid=(0,29)
```

Sin magia. Las pruebas lo comparan con `page_header()` y `heap_page_items()`.

Un UPDATE no sobrescribe en el lugar

update.sql

```sql
BEGIN;
UPDATE agency SET name = name || ' (centro)' WHERE agency_id = 1;  -- (0,1) -> (0,30)
```

psql

```
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_ctid
----+--------+----------+--------+--------+--------+--------
  1 |   8128 |        1 |     57 |  13129 |  13231 | (0,30)
  2 |   8072 |        1 |     56 |  13129 |      0 | (0,2)
 29 |   6280 |        1 |     60 |  13129 |      0 | (0,29)
 30 |   6208 |        1 |     66 |  13231 |      0 | (0,30)
(4 rows)
```

- La vieja sigue: `t_xmax` = quien la reemplazó; `t_ctid` apunta a la nueva.
- La nueva: slot 30, en 6208. Un `INSERT` más una marca (MVCC: semana 14).

`COMMIT`, `VACUUM` y la página compactada

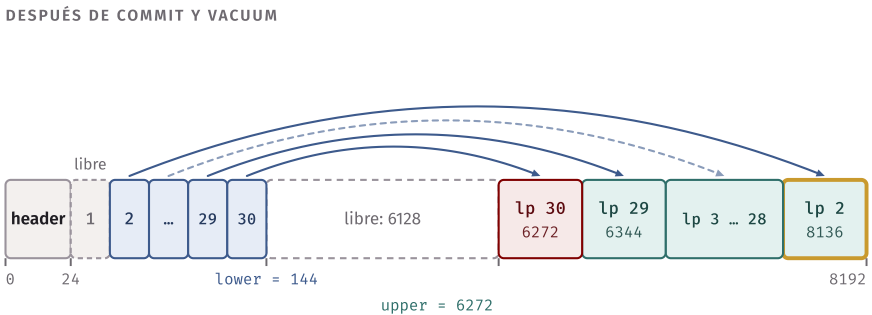
después de COMMIT y VACUUM agency

```
 lp | lp_off | lp_flags | lp_len | t_ctid
----+--------+----------+--------+--------
  1 |      0 |        0 |      0 |
  2 |   8136 |        1 |     56 | (0,2)
 29 |   6344 |        1 |     60 | (0,29)
 30 |   6272 |        1 |     66 | (0,30)
(4 rows)

 lower | upper | free
-------+-------+------
   144 |  6272 | 6128
(1 row)
```

- Slot 1 libre (`lp_flags = 0`), pero sigue en el arreglo: `lower` sigue en 144.
- La página se compactó: el slot 2 pasó de 8072 a 8136.
- Y su `ctid` sigue siendo `(0,2)`: el registro se movió, el RID no.
- 64 bytes libres más: los 57 de la vieja con su alineación.

El UPDATE y el VACUUM, en la página



el slot 1 queda libre y las tuplas se corren al final; la agencia 2 sigue en (0,2)

Dos detalles de PostgreSQL

HOT update

`name` tiene un índice `UNIQUE`. Si se cambia una columna sin índice y la versión nueva cabe en la página, no se toca ningún índice y el slot viejo queda como `LP_REDIRECT`. Prueben con `city` (ejercicio 6).

`t_xmax` no siempre es un borrado

Justo después de `chasqui-load`, las 29 tuplas de `agency` tienen `t_xmax = t_xmin`: son los locks de fila de las verificaciones de clave foránea, con la bandera `HEAP_XMAX_LOCK_ONLY`. El `VACUUM (FREEZE)` del `Makefile` los limpia.

P3: la slotted page de `quipu`

Lo que fija el enunciado

- Páginas de 4 KiB.
- Los primeros 8 bytes del encabezado, reservados para el pageLSN de P5 (como `pd_lsn`).
- `compact()` no cambia los números de slot.
- El formato, en su README con un diagrama.

Lo que deciden ustedes

- ¿Qué más va en el encabezado? ¿De qué tamaño cada campo?
- ¿Cómo distinguen un slot libre?
- ¿`insert` reutiliza un slot libre del medio?
- ¿`freeSpace()` cuenta el slot nuevo?
- ¿Qué hace `update` si el registro crece pero cabe?

`pgpage.mjs` es un lector de solo lectura de otro formato, con otra API: sirve de referencia para leer bytes con `Buffer`, no para copiar.

SECCIÓN 08

Organización de archivos

08

Heap files y el free space map

- **Heap file**: cualquier registro en cualquier página. Así son las tablas de PostgreSQL y el heap file de P3.
- Insertar exige encontrar espacio: recorrer 24 667 páginas por `INSERT` es absurdo.
- Lista de páginas con espacio, o un **free space map**: una entrada aproximada por página.

pg_freespacemap

```
chasqui_w09=# SELECT blkno, avail
chasqui_w09-#   FROM pg_freespace('agency');
 blkno | avail
-------+-------
     0 |  6112
(1 row)
```

- En PostgreSQL, el archivo `_fsm`: 1 byte por página, en unidades de $8192/256 = 32$ bytes. La página tiene 6140 libres; el mapa dice 6112.
- Un árbol de máximos encima: una página con *n* bytes libres en pocos pasos.
- No tiene que ser exacto: si se equivoca, se corrige y se prueba otra página.

Otras organizaciones (sec. 13.3)

Archivo secuencial

Ordenado por una **clave de búsqueda**. Si no hay espacio, un **bloque de overflow**; hay que reorganizar. (`CLUSTER` ordena una vez.)

- **Clustering multitabla**: cada `shipment` con sus `shipment_event` en el mismo bloque. Leer solo `shipment` cuesta más.
- **Particionamiento**: `shipment` por mes de `created_at`. `PARTITION BY RANGE` en PostgreSQL.

Row store y column store (sec. 13.6, opcional)

Column store: cada columna en su archivo; `avg(price)` lee solo `price`.
Transacciones: row store. Análisis: column store.

El diccionario de datos (sec. 13.4)

Metadatos en tablas del propio sistema: `pg_class`, `pg_attribute`, `pg_index`.

psql

```
      index       |  tabla   | indkey | indisunique | indisprimary
------------------+----------+--------+-------------+--------------
 customer_pkey    | customer | 1      | t           | t
 customer_dni_key | customer | 2      | t           | f
(2 rows)

  oid  |     relname      | relfilenode | relkind | relpages
-------+------------------+-------------+---------+----------
 56489 | customer         |       56489 | r       |       19
 56501 | customer_pkey    |       56501 | i       |        7
 56503 | customer_dni_key |       56503 | i       |       11
(3 rows)
```

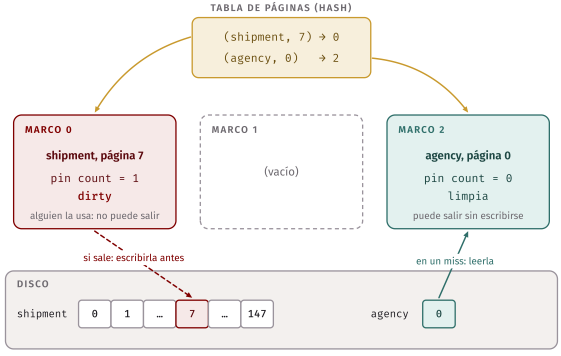
Cada índice es una relación más, con su archivo. Huevo y gallina: la ubicación de unas pocas tablas del catálogo es fija.

SECCIÓN 09

El buffer pool

09

El buffer pool (sec. 13.5)



- Un arreglo de **marcos** *(frames)* del tamaño de una página.
- Una **tabla de páginas**: qué página está en qué marco (no la del sistema operativo).
- Por marco: la página, el **pin count**, la bandera **dirty** y lo que pida la política de reemplazo.

Pin, unpin y páginas sucias

1. **Pedir** (pin): si está, hit: pin count + 1. Si no, miss: elegir un marco con pin count 0; si está sucio, escribirlo; leer la página; pin count = 1.
2. **Soltar** (unpin): pin count $-$ 1, y "la modifiqué" $\Rightarrow$ dirty.
3. **Escribir** (flush): llevar una página sucia al disco sin sacarla.

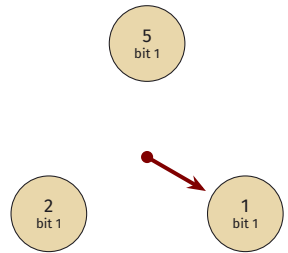
- Pin count > 0: alguien tiene un puntero al marco. No se desaloja.
- Todo fijado y llega un miss: es un bug de quien olvidó soltar páginas.
- Latches: locks breves sobre el marco mientras se copian bytes. No son los locks de la semana 14.

Error común

Unpin no escribe a disco: solo dice "ya no la uso". La escritura ocurre al desalojar una página sucia, en un checkpoint o cuando lo pide la recuperación.

Políticas de reemplazo

- **LRU**: sale la usada hace más tiempo. Exacto, cuesta tocar una lista compartida en cada hit.
- **Clock**: un bit de referencia por marco y una manecilla. Bit en 1: bajarlo y avanzar (segunda oportunidad). Bit en 0 y sin pin: víctima.
- **MRU**: sale la usada más recientemente.
- **Toss-immediate**: sacarla apenas se termina de usar.



antes de pedir 3: la manecilla en el marco de 1

Clock paso a paso

node

```
$ node replay.mjs clock 3
clock, 3 marcos
  pide  1  miss  [ 1  -  - ]
  pide  2  miss  [ 1  2  - ]
  pide  3  miss  [ 1  2  3 ]
  pide  4  miss  [ 4  2  3 ]  sale 1
  pide  1  miss  [ 4  1  3 ]  sale 2
  pide  2  miss  [ 4  1  2 ]  sale 3
  pide  5  miss  [ 5  1  2 ]  sale 4
  pide  1  hit   [ 5  1  2 ]
  pide  2  hit   [ 5  1  2 ]
  pide  3  miss  [ 5  3  2 ]  sale 1
  pide  4  miss  [ 5  3  4 ]  sale 2
  pide  5  hit   [ 5  3  4 ]
hits=3 misses=9
```

- `replacement.mjs`: solo la decisión de a quién desalojar; sin pins ni disco.
- Al pedir 3: los tres bits en 1. La manecilla baja el de 1, el de 2, el de 5, vuelve a 1 y lo desaloja.
- LRU habría sacado 5. Con LRU: 2 hits; con MRU: 5.
- Clock aproxima LRU: a veces gana, a veces pierde (ejercicio 8).

Clock: la manecilla



pide 5: hit; su bit queda en 1 y la manecilla no se mueve

termina con 3 hits y 9 misses, como replay.mjs clock 3

Sequential flooding

node

```
$ node replay.mjs mru 3 1 2 3 4 1 2 3 4 1 2 3 4
mru, 3 marcos
  pide  1  miss  [ 1  -  - ]
  pide  2  miss  [ 1  2  - ]
  pide  3  miss  [ 1  2  3 ]
  pide  4  miss  [ 1  2  4 ]  sale 3
  pide  1  hit   [ 1  2  4 ]
  pide  2  hit   [ 1  2  4 ]
  pide  3  miss  [ 1  3  4 ]  sale 2
  pide  4  hit   [ 1  3  4 ]
  pide  1  hit   [ 1  3  4 ]
  pide  2  miss  [ 2  3  4 ]  sale 1
  pide  3  hit   [ 2  3  4 ]
  pide  4  hit   [ 2  3  4 ]
hits=6 misses=6
```

- Un scan repetido de una tabla de 4 páginas con 3 marcos (la tabla de adentro de un nested loop join).
- LRU: sale justo la página que se va a pedir. 0 hits. Clock igual.
- MRU: 6 hits.

El DBMS sabe

El sistema operativo usa una política genérica. El DBMS conoce el patrón de cada operador y puede elegir.

El buffer pool de PostgreSQL

buffers.sql

```
 buffers | unit | total
---------+------+--------
 16384   | 8kB  | 128 MB
(1 row)
```

- `shared_buffers`: 16 384 marcos, compartidos por todos los procesos.
- **Clock sweep**: un **usage count** de 0 a 5 que sube con cada uso; la manecilla resta 1. Sale el primero en 0 y sin pin.

buffers.sql: top_relations

```
            relname             | buffers | dirty
--------------------------------+---------+-------
 shipment_event                 |     346 |   346
 shipment                       |     152 |   152
 shipment_event_pkey            |     129 |   129
 pg_proc                        |     108 |   108
 pg_toast_2618                  |      68 |    68
 pg_attribute                   |      62 |    62
 pg_description                 |      50 |    50
 trip                           |      48 |    48
 pg_statistic                   |      45 |    45
 pg_proc_proname_args_nsp_index |      36 |    36
(10 rows)
```

Recién cargada: todo sucio. Lo escriben el checkpointer y el background writer.

Caché fría y caliente

después de `pg_buffercache_evict_relation`

```
                        QUERY PLAN
------------------------------------------------------------
 Aggregate (actual rows=1.00 loops=1)
   Buffers: shared read=148
   ->  Seq Scan on shipment (actual rows=12000.00 loops=1)
         Buffers: shared read=148
 Planning:
   Buffers: shared hit=27
 Planning Time: 0.068 ms
 Execution Time: 0.989 ms
(8 rows)
```

la segunda vez

```
                        QUERY PLAN
------------------------------------------------------------
 Aggregate (actual rows=1.00 loops=1)
   Buffers: shared hit=148
   ->  Seq Scan on shipment (actual rows=12000.00 loops=1)
         Buffers: shared hit=148
 Planning Time: 0.015 ms
 Execution Time: 0.409 ms
(6 rows)
```

- `EXPLAIN (ANALYZE, BUFFERS)`: `hit` = estaba en shared buffers; `read` = se pidió afuera.
- 148 = las páginas de la tabla: un sequential scan lee cada una una vez. Después de los dos scans, las 148 tienen usage count 2.

Una tabla más grande que un cuarto del pool

big.sql: segundo scan de shipment en chasqui_big

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate (actual time=151.120..151.120 rows=1.00 loops=1)
   Buffers: shared hit=94 read=24573
   ->  Seq Scan on shipment (actual time=0.078..82.713 rows=2012000.00 loops=1)
         Buffers: shared hit=94 read=24573
```

24 667 páginas: el segundo scan vuelve a leer casi todo. `customer` (2040 páginas): `hit=2040`.

Buffer ring

Un scan de una tabla de más de 1/4 de `shared_buffers` usa un anillo pequeño de buffers y solo desaloja sus propias páginas: quedan 188 en memoria.

`read` no siempre es el disco

- `read` = “no estaba en shared buffers y se la pedí al sistema operativo”.
- En PostgreSQL 18 (`io_method = worker`, por defecto) leen los procesos `io worker`. En `/proc/<pid>/io`: `rchar` (bytes pedidos con `read`) y `read_bytes` (bytes traídos del dispositivo).

`io workers, antes y después de un scan con read=24385`

```
rchar=1871282623 read_bytes=0
rchar=2071533508 read_bytes=0
```

- `rchar` subió $200\,250\,885 \approx 24\,385 \times 8192$. `read_bytes`: ni un byte del disco. Venía de la caché del sistema operativo.
- **Doble buffering**: la misma página en shared buffers y en la caché del sistema operativo. Por eso `shared_buffers` empieza en un 25 % de la RAM, no en toda.

¿Por qué no basta el sistema operativo, ni `mmap`?

1. **El orden de las escrituras.** WAL (semana 15): una página sucia no puede llegar al disco antes que su log. Cada página guarda su LSN (`pd_lsn`; en `quipu`, los 8 bytes reservados). Con `mmap`, el sistema operativo escribe cuando quiere.
2. **El patrón de acceso.** Prefetch exacto, MRU o anillos donde LRU falla, fijar la raíz de un índice.
3. **Fallos de página y errores.** Con `mmap`, un fallo detiene el hilo sin aviso; un error de E/S llega como `SIGBUS`.
4. **Muchos hilos.** TLB shootdowns y estructuras compartidas del kernel: el sistema operativo se vuelve el cuello de botella.

> **Crotty, Leis y Pavlo (CIDR 2022)**
>
> `mmap` parece un atajo y termina siendo más difícil de hacer correcto y más lento que un buffer pool propio.

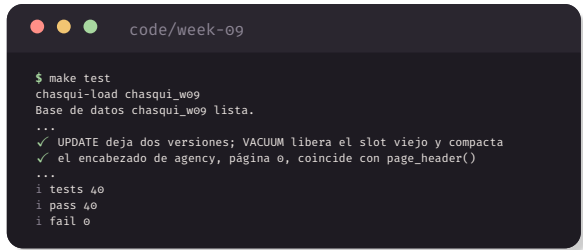
SECCIÓN 10

Cierre

10

El código de la semana

- `pages.sql`, `update.sql`, `buffers.sql`: sobre `chasqui_w09`.
- `big.sql`: sobre `chasqui_big` (no cambia datos).
- `pgpage.mjs` y `dump.mjs`: la página leída a mano.
- `replacement.mjs` y `replay.mjs`: LRU, MRU, clock.
- `make` hace `VACUUM (FREEZE, ANALYZE)`: estado conocido.



Resumen

- Se lee y escribe por **bloques**; el costo se mide en bloques y seeks: $b \cdot t_T + S \cdot t_S$.
- Disco magnético: casi todo es posicionarse; lo secuencial puede ser 100 veces más barato. SSD: sin seek, pero borrado por bloques, FTL y amplificación de escritura.
- Registro variable: pares (offset, longitud) y bitmap de nulos. PostgreSQL: varlena y bitmap en el encabezado.
- **Slotted page**: slots desde el inicio, registros desde el final. **RID** = (página, slot) = `ctid`. Compactar no cambia el RID.
- `UPDATE` = versión nueva + marca en la vieja; `VACUUM` libera y compacta.
- Heap file + free space map. Archivo secuencial, particionamiento, catálogo en tablas.
- **Buffer pool**: pin, dirty, unpin $\neq$ escribir. LRU, clock, MRU; sequential flooding; clock sweep y buffer rings en PostgreSQL.
- El DBMS administra su memoria: WAL, patrón de acceso, sin `mmap`.

Para la próxima clase

- Laboratorio: las páginas de PostgreSQL por dentro (`pageinspect`, `pg_buffercache`).
- Se entrega P2. Se publica **P3** (`quipu`: almacenamiento); se entrega en la semana 12.
- Leer Silberschatz, secs. 14.1 a 14.3 (índices y B+ trees) y la nota 4 de CS186.
- Correr `make test` en `code/week-09` y leer los apuntes de la semana.

Pregunta

Un índice guarda, para cada `shipment_id`, el RID de su fila. Después de un `UPDATE` que cambia el `ctid` de la fila, ¿qué tiene que pasar con el índice? ¿Y después de un `VACUUM` que compacta la página?