Bases de Datos

Índices: B+ trees, hashing e índices en PostgreSQL

Almacenamiento, índices y consultas

Gerson Garrido

Semana 10

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: índices y B+ trees

1. Qué es un índice y cómo se evalúa
2. Agrupamiento y secundario, denso y disperso, varios niveles
3. B+ tree: estructura, búsqueda, fan-out y altura
4. Inserción con splits, borrado, claves repetidas

Sesión B: hashing e índices en la práctica

5. Hash estático, extendable y linear hashing
6. Bulk loading y claves compuestas
7. Índices de PostgreSQL: tipos, parciales, covering
8. Cuándo un índice no ayuda y cuánto cuesta (y LSM trees)

Lectura

Silberschatz, secs. 14.1–14.3 (sesión A); 14.5, 14.4 (bulk loading), 14.6, 14.7 y 24.5 (sesión B); opcional, 14.8. CS186, nota 4 (*B+Trees*). CMU 15-445, clases #08 y #07. PostgreSQL, *Index Types* y *Multicolumn Indexes*. Además: Winand, *Use The Index, Luke!*; PostgreSQL *B-Tree Indexes*, *Index-Only Scans*, *BRIN*, *GIN* y *Hash Indexes*; opcional, Silberschatz 24.1–24.2, CMU #05 y #09, O'Neil et al. (1996).

SECCIÓN 01

Qué es un índice

01

Como el índice de un libro

- Un sequential scan lee todas las páginas: 148 en `chasqui_w10`, 24 667 en `chasqui_big`.
- Para una fila, casi todo ese trabajo se pierde.
- Un índice es una estructura **auxiliar**, al lado del heap file.
- Pero la tabla cambia: cada `INSERT`, `UPDATE` y `DELETE` mantiene todos sus índices.

Clave de búsqueda ***(search key)***

El atributo (o atributos) por el que se busca. Puede repetirse: `sender_id`.

Entrada del índice

(valor de la clave, record id). El record id es el `ctid` de la semana 9.

Cómo se evalúa un índice

Criterio	Pregunta
Tipos de acceso	¿Igualdad? ¿Rangos?
Tiempo de acceso	¿Cuántas páginas para encontrar lo que se busca?
Tiempo de inserción	¿Cuánto trabajo extra cuesta un `INSERT`?
Tiempo de borrado	¿Y un `DELETE`?
Espacio	¿Cuántas páginas ocupa?

Como la semana pasada

Todo se cuenta en páginas leídas y escritas. Comparar claves en memoria cuesta nanosegundos; traer una página, micro o milisegundos.

Los envíos del cliente 42

sin índice

```
Seq Scan on shipment (actual rows=51.00 loops=1)
  Filter: (sender_id = 42)
  Rows Removed by Filter: 11949
  Buffers: shared hit=148
```

- PostgreSQL indexa `PRIMARY KEY` y `UNIQUE`, no las claves foráneas.
- `CREATE INDEX shipment_sender_idx ON shipment (sender_id);`

con índice

```
Bitmap Heap Scan on shipment (actual rows=51.00 loops=1)
  Recheck Cond: (sender_id = 42)
  Heap Blocks: exact=45
  Buffers: shared hit=45 read=2
  ->  Bitmap Index Scan on shipment_sender_idx (actual rows=51.00 loops=1)
        Index Cond: (sender_id = 42)
        Index Searches: 1
        Buffers: shared read=2
```

47 páginas en vez de 148

2 del índice y 45 del heap: los 51 envíos están repartidos por toda la tabla.

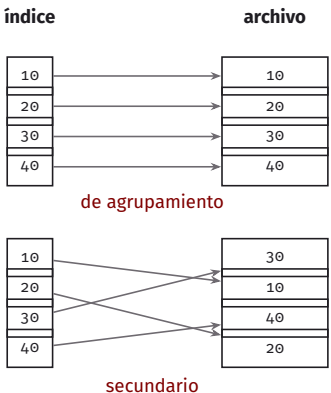
SECCIÓN 02

Índices ordenados

02



De agrupamiento y secundario

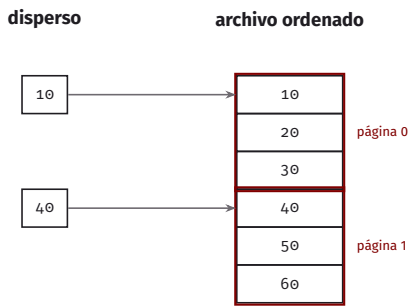


- **De agrupamiento** *(clustering)*: la clave del índice define el orden del archivo. Claves cercanas, páginas cercanas.
- **Secundario** *(nonclustering)*: el archivo tiene otro orden. Cada tupla puede estar en otra página.
- A lo más un índice de agrupamiento por tabla.

Error común

En el libro, *índice primario* = de agrupamiento, sobre cualquier clave; no “el de la clave primaria”.

Densos y dispersos

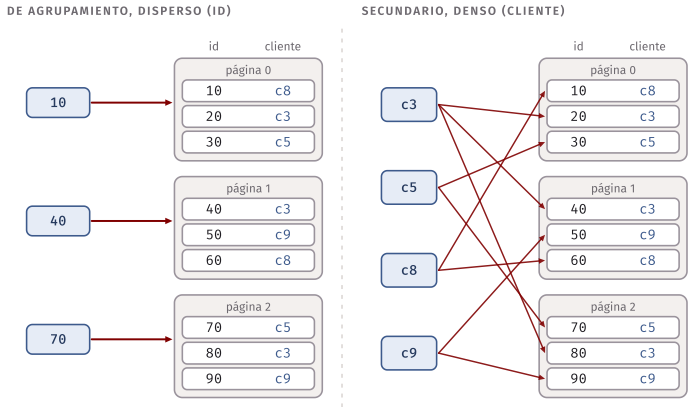


- **Denso**: una entrada por cada valor de la clave.
- **Disperso**: una entrada por página (su primera clave). Para buscar 50: la última entrada ≤ 50 y recorrer la página.
- Solo funciona si el archivo está ordenado por la clave.
- Un índice secundario tiene que ser denso.

En `chasqui_big`

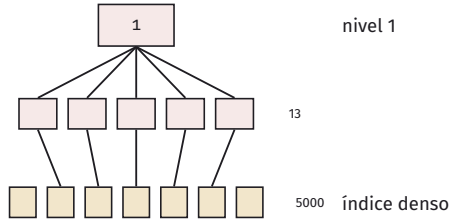
2 012 000 filas en 24 667 páginas: denso, 2 012 000 entradas; disperso, 24 667.

Disperso si el archivo está ordenado; si no, denso



Índices de varios niveles

- Índice denso de 2 012 000 entradas, 400 por página: $\approx$ 5000 páginas.
- Búsqueda binaria: $\lceil \log_2 5000 \rceil = 13$ lecturas.
- Un índice disperso sobre el índice: 5000 entradas en 13 páginas.
- Y otro encima: 1 página.
- Ahora una búsqueda lee 3 páginas, una por nivel.



Es la forma de un B+ tree

Le falta mantenerse al día al insertar y borrar.

En PostgreSQL todos los índices son secundarios

- Toda tabla es un heap file: las filas nuevas van donde hay espacio.
- Incluso `shipment_pkey` es secundario.
- `correlation` (`pg_stats`): 1 mismo orden, 0 ninguno.

correlation en chasqui_w10

```
   attname   |  correlation
-------------+---------------
 created_at  | -0.0012399433
 shipment_id |             1
```

indexes.sql

```sql
CLUSTER shipment
  USING shipment_created_idx;
-- created_at: 1
UPDATE shipment SET price = price
WHERE shipment_id % 10 = 0;
-- 1200 filas; created_at: 0.8181597
```

Error común

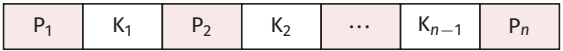
`CLUSTER` ordena la tabla **una vez**. Las escrituras siguientes no respetan el orden: no deja un clustered index permanente.

SECCIÓN 03

B+ trees

03

Un nodo es una página



Hoja

P_i apunta a la tupla con clave K_i (un record id). P_n apunta a la hoja siguiente: las hojas forman una lista en orden. Juntas son un índice **denso**.

Nodo interno

P_1: claves $< K_1$. P_i: claves $\geq K_{i-1}$ y $< K_i$. El último: claves $\geq$ la última clave. Son un índice **disperso** sobre las hojas.

Fan-out

Cada nodo tiene cientos de hijos, no dos. Eso hace que el árbol sea bajo.

Balanceado: la ocupación mínima

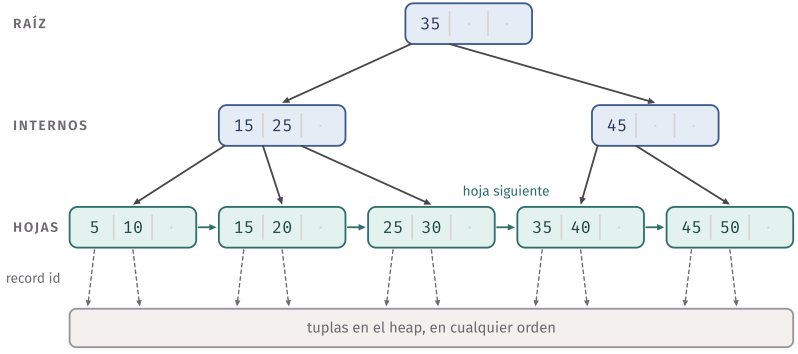
Nodo	Mínimo	Máximo
Hoja	$\lceil (n-1)/2 \rceil$ claves	$n-1$ claves
Interno	$\lceil n/2 \rceil$ punteros	n punteros
Raíz interna	2 punteros	n punteros

- Todo nodo salvo la raíz, al menos medio lleno.
- Todas las hojas a la misma profundidad: la altura h.
- Con $n = 4$: hojas de 2 a 3 claves, internos de 2 a 4 hijos.

B+ tree y B-tree

En un B+ tree todas las claves están en las hojas; las de los nodos internos son copias que separan. El B-tree original (sec. 14.4) guarda cada clave una vez, en cualquier nivel.

Un B+ tree con $n = 4$



- La raíz: $<$ 35 a la izquierda, $\geq$ 35 a la derecha. `[15 25]`: $<$ 15, de 15 a 24, $\geq$ 25.
- Las hojas, enlazadas en orden: un rango no vuelve a la raíz.

Un B-tree de PostgreSQL por dentro

pk_meta y pk_pages

```
 root | level
------+-------
    3 |     1

 type | level | pages | items | avg_items | avg_free
------+-------+-------+-------+-----------+----------
 r    |     1 |     1 |    33 |        33 |     7496
 l    |     0 |    33 | 12032 |       365 |      856
```

- `level` cuenta desde 0 en las hojas: altura 2.
- 12 032 = 12 000 claves + 32 **high keys**.

leaf_items: hoja 1

```
 itemoffset | ctid  | itemlen |          data           | htid
------------+-------+---------+-------------------------+-------
          1 | (4,1) |      16 | 6f 01 00 00 00 00 00 00 |
          2 | (0,1) |      16 | 01 00 00 00 00 00 00 00 | (0,1)
          3 | (0,2) |      16 | 02 00 00 00 00 00 00 00 | (0,2)
          4 | (0,3) |      16 | 03 00 00 00 00 00 00 00 | (0,3)
```

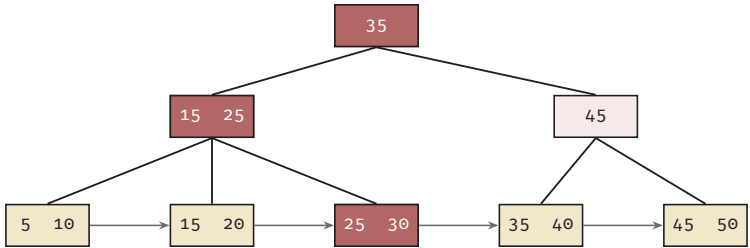
- Primera entrada: high key 0x016f = 367.
- Después (clave, `ctid`): el envío 1 está en (0,1).
- Es un B+ tree: hojas enlazadas (`btpo_prev`, `btpo_next`).

SECCIÓN 04

Búsqueda, fan-out y altura

04

Buscar el 30



1. En la raíz: la primera clave mayor que 30 es 35: bajar por su izquierda.
2. En `[15 25]`: ninguna es mayor que 30: bajar por el último puntero.
3. En la hoja: está el 30, y su puntero lleva a la tupla. Una página por nivel.

Rango `lo..hi`: buscar `lo` y seguir los enlaces de las hojas hasta pasar `hi`.

Fan-out y altura

Altura de un B+ tree de N claves

$$h = \lceil \log_f N \rceil \qquad f = \text{fan-out}$$

Mejor caso $f = n$; peor caso $f = \lceil n/2 \rceil$.

`chasqui_big`, $N \approx 2 \cdot 10^6$, clave `bigint` y puntero de 8 bytes: $8192/16 \approx 500$ pares por página.

- Mejor: $\lceil \log_{500} 2 \cdot 10^6 \rceil = \lceil 2{,}33 \rceil = 3$. Peor: $\lceil \log_{250} 2 \cdot 10^6 \rceil = \lceil 2{,}63 \rceil = 3$.
- Árbol binario: $\lceil \log_2 2 \cdot 10^6 \rceil = 21$.

En la práctica

Segundo nivel: entre 8 y 32 páginas. Raíz y segundo nivel viven en el buffer pool: 1 o 2 lecturas de disco por búsqueda.

Error común

Calcular con $\log_2$, o con el número de páginas en vez del de claves. La base es el fan-out.

La altura real de `shipment_pkey`

big.sql: pk_meta y pk_levels

```
 root | level
------+-------
  412 |     2

 type | level | pages |  items  | avg_items
------+-------+-------+---------+-----------
 r    |     2 |     1 |      19 |        19
 i    |     1 |    19 |    5516 |       290
 l    |     0 |  5498 | 2017497 |       367
```

pk_lookup (extracto)

```
Index Scan using shipment_pkey on shipment (...)
  Index Cond: (shipment_id = 1500000)
  Index Searches: 1
  Buffers: shared hit=3 read=1 written=1
```

- Altura 3, como la cuenta. Segundo nivel: 19 páginas.
- Entrada de 16 bytes + 4 del line pointer: caben $\approx$ 407 por página.
- `CREATE INDEX` llena las hojas al 90 % (fill factor): 366 claves + high key.
- Internos, alrededor del 70 %: 290.

193 MB, 2 millones de filas

Encontrar una: 4 páginas, y solo una se leyó.

SECCIÓN 05

Inserción

05

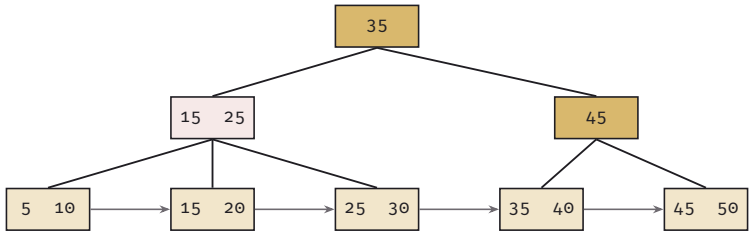
Insertar: split de hoja y split de nodo interno

1. Bajar hasta la hoja, recordando el camino. Si hay lugar, insertar en orden.
2. **Hoja llena**: con la nueva son n claves. Las primeras $\lceil n/2 \rceil$ se quedan, el resto va a una hoja nueva. La primera clave de la hoja nueva se copia al padre.
3. **Nodo interno lleno**: con $n+1$ punteros, se quedan los primeros $\lceil (n+1)/2 \rceil$. La clave del medio sube y no queda abajo.
4. Si se divide la raíz, raíz nueva: el árbol crece hacia arriba.

Por qué la asimetría

Las hojas son el índice denso: si la clave se fuera al padre, la tupla dejaría de estar indexada. En un nodo interno la clave solo separa: con dejarla en el padre basta.

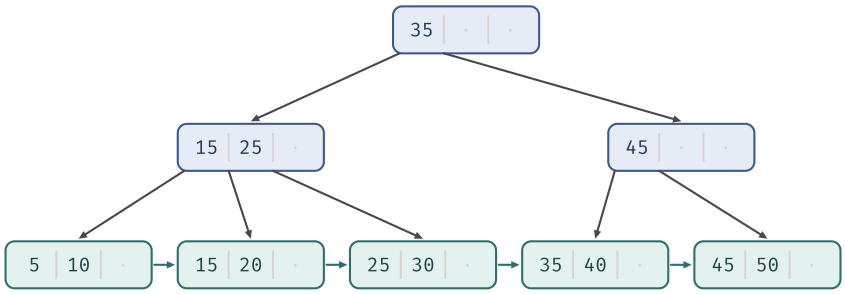
Paso a paso, $n = 4$: 10, 20, 5, 15, 25, 30, 35, 40, 45, 50



+50 divide [35 40 45 50]; la raíz tendría 15 25 35 45: se divide y el 35 sube

La inserción de 50: copiar y subir

el 35 queda solo en la raíz nueva y, como toda clave, en su hoja



la raíz se divide en [15 25] y [45]; el 35 SUBE a una raíz nueva

Copiar o subir

Error común

Copiar la clave del medio en el split de un nodo interno, o subirla sin copiar en el de una hoja. Prueba: después de insertar, **toda clave sigue en alguna hoja**.

- Una inserción: h lecturas y, sin splits, una escritura.
- Un split escribe 3 páginas y puede subir hasta la raíz. Son raros: una hoja de 400 lugares se divide cada $\sim$200 inserciones.
- Claves en orden (`shipment_id`): todas van a la última hoja. PostgreSQL deja la hoja vieja llena al fill factor, no a la mitad.

SECCIÓN 06

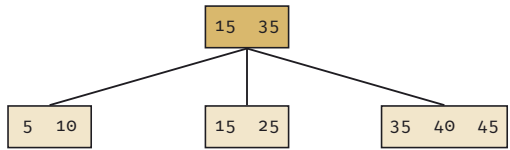
Borrado

06

Borrar: fusionar o redistribuir

1. Quitar la entrada de la hoja. Si quedan $\geq \lceil (n-1)/2 \rceil$ claves, listo.
2. Si no (**underflow**), mirar un hermano con el mismo padre:
   - Si caben juntos: **fusionar** *(coalesce)* en el de la izquierda y borrar del padre el separador y el puntero. Puede propagarse.
   - Si no: **redistribuir**, pasar una entrada del hermano y actualizar el separador del padre.
3. En un nodo interno, al fusionar, el separador del padre baja.
4. Si la raíz queda con un hijo, se borra: el árbol se achica por la raíz.

Tres borrados, $n = 4$



−30: [25] con [5 10 15] serían 4 claves: no caben.
Redistribución: el 15 pasa a la derecha y el separador cambia de 25 a 15.

Lo que hace PostgreSQL

- Nunca fusiona ni redistribuye páginas medio vacías.
- `DELETE` marca la tupla; `VACUUM` quita las entradas del índice que apuntan a tuplas muertas.
- Una página completamente vacía sale del árbol y se recicla. Una medio vacía se queda así.
- Un índice con muchas páginas casi vacías está hinchado *(bloated)*: `REINDEX` lo reconstruye.

Por qué

Fusionar cuesta escrituras y complica la concurrencia. En una tabla que crece, las páginas medio vacías se vuelven a llenar.

SECCIÓN 07

Claves repetidas y bulk loading

07

Claves que se repiten

- Sec. 14.3.5: hacer única la clave, indexar (`sender_id`, record id).
- PostgreSQL 12: el `ctid` desempata.
- PostgreSQL 13: **deduplicación**, el valor una vez y la *posting list* de sus `ctid`.

big.sql: sender_sizes

```
            relname            | relpages | size
-------------------------------+----------+-------
 shipment_sender_idx           |     2286 | 18 MB
 shipment_sender_nodedup_idx   |     5527 | 43 MB
```

En `chasqui_big`, 2,4 veces menos que sin deduplicación.

indexes.sql: posting_list, hoja 1 de shipment_sender_idx

```
 itemoffset | itemlen |  htid   |                                                                                                  tids
------------+---------+---------+-------------------------------------------------------------------------------------------------------------------------------------------
          1 |      16 |         |
          2 |      64 | (22,34) | {"(22,34)","(26,43)","(52,57)","(85,28)","(87,30)","(99,21)","(102,11)","(122,46)"}
          3 |      32 | (55,74) | {"(55,74)","(61,15)"}
```

Bulk loading

1. Armar las entradas y ordenarlas (ordenamiento externo, semana 11).
2. Llenar las hojas de izquierda a derecha hasta el fill factor.
3. Armar cada nivel con la primera clave de cada nodo de abajo.

De abajo hacia arriba

Cada página se escribe una vez, en orden. Es lo que hace `CREATE INDEX`.

2 012 000 envíos, índice sobre `sender_id`, sin deduplicación:

Cargar y después `CREATE INDEX`	1,3 + 1,0 s
Índice desde antes	4,6 s

big.sql: bulk_vs_incr

```
    build    | leaves | avg_items | avg_free
-------------+--------+-----------+----------
 bulk        |   5498 |       367 |      802
 incremental |   6899 |       293 |     2295
```

Primero los datos, después los índices.

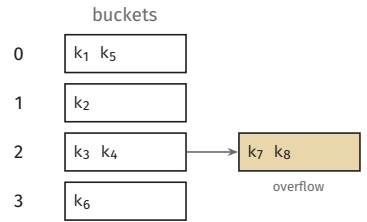
SECCIÓN 08

Índices hash

08

Hashing estático

- B buckets, cada uno una página; $h(k) \in \{0, \ldots, B-1\}$.
- Igualdad: calcular $h(k)$ y leer un bucket.
- Buena h: **uniforme** y **aleatoria**.
- Overflow por crecimiento o por **sesgo** *(skew)*: páginas de **overflow** encadenadas.
- El problema: B se fija al crear el índice.



Extendable hashing (sec. 24.5)

- Se usan los primeros i bits del hash: la **profundidad global**.
- Un **directorio** de 2^i entradas apunta a los buckets.
- Cada bucket tiene su **profundidad local** $i_j \leq i$; lo apuntan 2^{i-i_j} entradas.

Bucket lleno

- $i_j < i$: dividirlo y reasignar punteros.
- $i_j = i$: primero duplicar el directorio, después dividir.
- Reintentar: puede seguir lleno.

La gracia

Al crecer, solo se recalculan las claves del bucket que se divide, nunca todas.

Los DNI de Chasqui, buckets de 2

indexes.sql: dni_bits, clientes 1 a 9

customer_id	dni	h	bits
1	61830975	292428428	00010001011011100001101010001100
2	17232205	-1757271257	10010111101000010001010110100111
3	36948162	-309584369	11101101100011000001111000001111
4	15433383	621095809	00100101000001010010101110000001
5	23448138	-2013037940	10001000000001101111010100011100
6	75503630	1591562236	01011101101110101001111111111100
7	24986841	-1005250091	11000100000101010001100111010101
8	36581411	1096976409	01000001011000101000100000011001
9	58728634	-1185145636	10111001010111000001110011011100

+ 58728634

```
profundidad global 3
000 -> [00] 61830975 15433383
001 -> [00] (igual que 000)
010 -> [01] 75503630 36581411
011 -> [01] (igual que 010)
100 -> [100] 17232205 23448138
101 -> [101] 58728634
110 -> [11] 36948162 24986841
111 -> [11] (igual que 110)
```

Cinco buckets, directorio de 8. Lo produce `exthash-demo.mjs`.

Linear hashing

- Sin directorio. Buckets $0, 1, 2, \ldots$ y un **puntero de división** s.
- $h_0(k) = h(k) \bmod N$, $h_1(k) = h(k) \bmod 2N$.
- Buscar: $b = h_0(k)$; si $b < s$, usar $h_1(k)$.
- Cuando cualquier bucket hace overflow, se divide el bucket s con h_1 (sus claves van a s y a $s + N$), y s avanza.
- Cuando $s = N$: ronda nueva con $2N$ buckets.

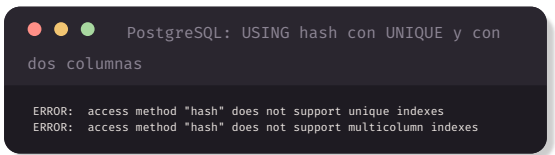
El compromiso

Crece un bucket por vez, sin duplicar nada de golpe. A cambio, cadenas de overflow temporales en los buckets que todavía no se dividieron.

Hash frente a B+ tree

	Hash	B+ tree
Igualdad	1 bucket	*h* páginas
Rango, orden	no	sí
Muy repetidos	overflow	bien

- Los niveles altos del B+ tree están en memoria: el hash ahorra 1 o 2 lecturas.
- Donde sí brilla: tablas hash en memoria (hash join, semana 11).



Solo `=`; guarda el hash de 4 bytes; crece de a un bucket. En `customer.dni` de `chasqui_big`: 842 páginas contra 782 del B-tree; 2 páginas por búsqueda contra 4 (ya leída la metapágina).

SECCIÓN 09

Acceso con varias claves

09

Dos índices de una columna: BitmapAnd

`origin_agency_id = 12` y una semana de `created_at`, con un índice sobre cada columna:

big.sql: bitmap_and (extracto)

```
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=1644 read=1331
  ->  Bitmap Heap Scan on shipment (actual rows=2837.00 loops=1)
        Heap Blocks: exact=2686
        ->  BitmapAnd (actual rows=0.00 loops=1)
              Buffers: shared read=289
              ->  Bitmap Index Scan on shipment_origin_idx (actual rows=67161.00 loops=1)
                    Index Cond: (origin_agency_id = 12)
                    Buffers: shared read=59
              ->  Bitmap Index Scan on shipment_created_idx (actual rows=82775.00 loops=1)
                    Buffers: shared read=230
```

- Cada índice arma un **bitmap** de las filas que cumplen; el AND bit a bit deja 2837 filas en 2686 páginas, leídas en orden.
- Mejor que cada índice solo, pero casi 3000 páginas.

Índice compuesto: el orden de las columnas

agencia 12, todo abril; índice (origin_agency_id, created_at), extracto

```
->  Index Only Scan using shipment_origin_created_idx on shipment (actual rows=11838.00 loops=1)
      Buffers: shared hit=1 read=48
```

la misma consulta; índice (created_at, origin_agency_id), extracto

```
->  Index Only Scan using shipment_created_origin_idx on shipment (actual rows=11838.00 loops=1)
      Buffers: shared hit=10 read=1370
```

- 49 páginas contra 1380.
- Agencia primero: la 12 en abril está junta.
  Fecha primero: mezclada con las otras 27.

Regla

Igualdad primero, rango después.
Sirve para cualquier prefijo.

Skip scan (PostgreSQL 18)

Solo `created_at` (una hora del 1 de abril), con el índice en (`origin_agency_id`, `created_at`):

big.sql: skip_scan (extracto)

```
->  Index Only Scan using shipment_origin_created_idx on shipment (actual rows=471.00 loops=1)
      Heap Fetches: 0
      Index Searches: 30
      Buffers: shared hit=38 read=56
```

- El índice se trata como 28 índices chicos, uno por agencia.
- `Index Searches: 30`: unas treinta bajadas desde la raíz, cerca de una por agencia. 94 páginas.
- Conviene si la primera columna tiene pocos valores distintos. Con `sender_id` serían 200 000 búsquedas.

Covering index e index-only scan

```
CREATE INDEX shipment_sender_status_idx ON shipment (sender_id) INCLUDE (status);
```

- `SELECT sender_id, status ... WHERE sender_id = 42`: todas las columnas están en el índice. 3 páginas, contra 47. `INCLUDE`: columnas solo en las hojas.

recién hecho VACUUM (extracto)

```
Index Only Scan using shipment_sender_status_idx on shipment (actual rows=51.00 loops=1)
  Heap Fetches: 0
  Buffers: shared hit=1 read=2
```

después de un UPDATE de esos 51 envíos (extracto)

```
Index Only Scan using shipment_sender_status_idx on shipment (actual rows=51.00 loops=1)
  Heap Fetches: 101
  Buffers: shared hit=50
```

La condición escondida: la página del heap tiene que estar *toda visible* en el visibility map (`VACUUM`). Si no, `Heap Fetches`.

SECCIÓN 10

Índices en PostgreSQL

10



CREATE INDEX

sintaxis

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] nombre
  ON tabla [USING método]
     (columna o (expresión) [ASC | DESC], ...)
  [INCLUDE (columnas)]
  [WITH (parámetro = valor)]
  [WHERE condición];
```

- `PRIMARY KEY` y `UNIQUE` crean un `UNIQUE INDEX`.
- Las claves foráneas no se indexan: `sender_id`, `trip_id`, ...
- `CREATE INDEX` bloquea las escrituras; `CONCURRENTLY` no, pero tarda más y no va dentro de una transacción.
- `WITH (fillfactor = 70)`: más lugar libre en las hojas.

Los tipos de índice

Método	Qué es	Para qué
`btree`	El B+ tree de hoy (por defecto)	`=`, `<`, `BETWEEN`, `IN`, `IS NULL`, `ORDER BY`
`hash`	Buckets que crecen de a uno	Solo `=`
`gin`	Índice invertido: elemento $\rightarrow$ filas	`jsonb` (semana 8), arreglos, texto completo
`brin`	Mínimo y máximo por rango de páginas	Tablas enormes ordenadas por la columna
`gist`, `spgist`	Árboles balanceados generales	Geometría, rangos, vecino más cercano

BRIN: el índice disperso de PostgreSQL

big.sql: brin_items (shipment_id) y brin_items_created

```
 blknum |       value
--------+--------------------
      0 | {1 .. 10396}
    128 | {10397 .. 108821}
    256 | {108822 .. 119263}
    384 | {119264 .. 129708}

 blknum |                                value
--------+-------------------------------------------------------------------
      0 | {2026-01-05 00:00:00-05 .. 2026-06-26 23:54:00-05}
    128 | {2026-01-05 01:20:23.52665-05 .. 2026-06-26 12:08:00-05}
    256 | {2026-01-05 00:05:41.394295-05 .. 2026-06-23 23:34:28.842958-05}
    384 | {2026-01-05 00:05:38.280809-05 .. 2026-06-23 23:54:42.92335-05}
```

3 páginas cada uno, contra 5519 de `shipment_pkey`.

- Como un índice disperso: sirve solo si la tabla sigue el orden de la columna.
- `shipment_id`: rangos angostos. `created_at`: todo el semestre en cada rango.

Un día, en `chasqui_w10` (rangos de 4 páginas):

- antes: `Heap Blocks: lossy=148` (todas)
- después de `CLUSTER`: `lossy=4`

Índices parciales y sobre expresiones

indexes.sql

```sql
CREATE INDEX shipment_pending_idx
  ON shipment (origin_agency_id)
  WHERE delivered_at IS NULL;
```

partial_sizes

```
       relname        | relpages | entries
----------------------+----------+---------
 shipment_origin_idx  |       13 |   12000
 shipment_pending_idx |        2 |     885
```

Se usa si la consulta implica la condición del índice.

indexes.sql

```sql
CREATE INDEX customer_lower_last_name_idx
  ON customer (lower(last_name));
SELECT customer_id FROM customer
WHERE lower(last_name)
  = 'villanueva ramírez';
```

Con un índice en `last_name`: `Seq Scan`. Con el de la expresión: `Bitmap Index Scan`, 6 filas. La expresión tiene que ser la misma.

SECCIÓN 11

Cuándo un índice no ayuda

11

La cuenta

`shipment` de `chasqui_big` (24 667 páginas, 2 012 000 filas) y una condición que deja pasar el 10 %, con un índice secundario.

Disco ($t_S = 4$ ms, $t_T = 0{,}1$ ms)

Sequential scan: $4 + 24\,667 \times 0{,}1 \approx 2{,}5$ s.
Índice: $201\,200 \times 4{,}1$ ms $\approx$ 825 s.

SSD ($t_S = 90$ µs, $t_T = 10$ µs)

Sequential scan: ≈ 247 ms.
Índice: $201\,200 \times 0{,}1$ ms $\approx$ 20 s.

- Equilibrio: $\approx$ 600 filas en disco (0,03 %), $\approx$ 2470 en SSD (0,12 %).
- **Bitmap scan**: lee cada página una vez, en orden físico.

Error común

Creer que un índice siempre ayuda. Ayuda si la condición es selectiva, si basta el índice o si evita un ordenamiento.

Medido: rangos de `created_at`

Rango	Filas	Págs.	Plan
10 minutos	74	73	bitmap, 0,5 ms
1 día	11 934	9489	bitmap, 15 ms
1 mes	367 202	24 667	bitmap, 108 ms
3 meses	1 052 486	24 667	seq scan

Mismo mes con seq scan obligado: 180 ms.

```
month_index_scan (extracto)

Aggregate (actual time=553.920..553.921 rows=1.00 loops=1)
  Buffers: shared hit=240559 read=127631 written=13
  ->  Index Scan using shipment_created_idx on shipment (...)
Execution Time: 571.043 ms
```

Index scan común (seq scan y bitmap scan apagados): 368 190 páginas para una tabla de 24 667, casi una por fila.

Lo que cuesta mantener un índice

Los mismos 512 000 envíos, insertados en tres tablas vacías:

Índices	Tiempo	Registros de WAL	Bytes de WAL
0	286 ms	512 002	65 MB
1 (clave primaria, en orden)	468 ms	1 025 405	99 MB
4 (+ `sender_id`, `recipient_id`, `created_at`)	2455 ms	2 577 921	227 MB

- 8,6 veces más tiempo: un registro por fila en el heap y uno por índice.
- Un índice sobre la columna que se actualiza impide los `UPDATE` HOT.
- ¿Se usa? `pg_stat_user_indexes.idx_scan`.

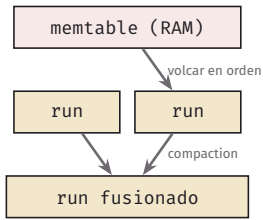
SECCIÓN 12

LSM trees (opcional)

12



LSM trees: optimizar las escrituras



- Escrituras a una **memtable** ordenada (y a un log).
- Llena, se escribe secuencialmente como un run inmutable.
- En segundo plano, los runs se fusionan (**compaction**).
- Leer: memtable y varios runs; un **bloom filter** por run evita los inútiles.

Write amplification frente a read amplification

RocksDB, LevelDB, Cassandra. PostgreSQL no usa LSM trees.

SECCIÓN 13

Laboratorio y cierre

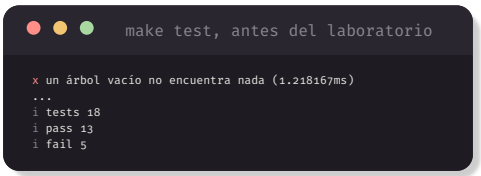
13

Laboratorio

Un B+ tree en memoria (`bptree.mjs`)

1. `search`: bajar con `upperBound`.
2. `insert`: split de hoja (copiar) y de nodo interno (subir).
3. `*range(lo, hi)`: seguir los `next`.
4. Una prueba de invariantes.

Es el algoritmo del índice de P3, sin las páginas.



En `chasqui_big`: `sender_id`, `bt_metap`, el orden de `(dest_agency_id, created_at)` y un índice inútil en `weight_kg`. Borren sus índices al terminar.

Resumen

- Índice: entradas (clave, record id). De agrupamiento o **secundario**; **denso** o **disperso** (solo con el archivo ordenado). En PostgreSQL, todos secundarios.
- **B+ tree**: nodos de una página, medio llenos, hojas a la misma profundidad y enlazadas. Altura $\lceil \log_f N \rceil$: 3 para 2 millones.
- Split de hoja: copiar. Split interno: subir. Crece y se achica por la raíz. PostgreSQL no fusiona.
- Clave repetida + record id = clave única; deduplicación. Bulk loading = `CREATE INDEX`.
- Hash: igualdad en una lectura, sin rangos. Extendable (directorio) y linear (puntero).
- Compuesto: igualdad primero. Skip scan. `INCLUDE` e index-only scan (visibility map).
- B-tree, hash, GIN, BRIN, GiST; parciales, sobre expresiones, `CONCURRENTLY`.
- Un índice secundario solo gana con condiciones muy selectivas, y cada uno encarece cada escritura.

Para la próxima clase

- Laboratorio: completar `bptree.mjs` y medir índices en `chasqui_big`.
- Seguir con P3: el B+ tree de `quipu` se guarda en páginas del buffer pool.
- Leer Silberschatz, secs. 15.1 a 15.4 (costo, selección y ordenamiento externo) y la nota de CS186 *Sorting*.
- Correr `make test` en `code/week-10` y leer los apuntes de la semana.

Pregunta

Una consulta busca los envíos *sin entregar* de una agencia de origen. ¿Qué índice crearían: uno sobre `(origin_agency_id)`, uno sobre `(origin_agency_id, delivered_at)` o uno parcial? ¿Cómo lo comprobarían con `EXPLAIN`?