

Bases de Datos

Procesamiento de consultas

Costo, selección, ordenamiento externo, joins e iteradores

Gerson Garrido

Semana 11

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: costo, selección y ordenamiento

1. El recorrido de una consulta
2. Medidas de costo: bloques y seeks
3. Selección: A1 a A10, bitmap scan
4. External merge sort

Sesión B: joins e iteradores

5. Nested loop, block nested loop, index nested loop
6. Merge join y hash join
7. Duplicados, conjuntos, agregación
8. Materialización, pipelining e iteradores

Lectura

Silberschatz, secs. 15.1–15.4 (sesión A) y 15.5–15.7 (sesión B). CS186, notas 8 (*Sorting*) y 9 (*Iterators and Joins*). CMU 15-445, clase #11 (notas), #13 (notas y diapositivas) y #14 (notas). Además: CS186, notas 6 (*Relational Algebra*) y 7 (*Hashing*); PostgreSQL, *Using EXPLAIN* y `work_mem`; CMU #15 (notas); Winand, *Use The Index, Luke!*: Nested Loops, Hash Join y Sort-Merge Join.

SECCIÓN 01

El recorrido de una consulta

01

Cuatro etapas

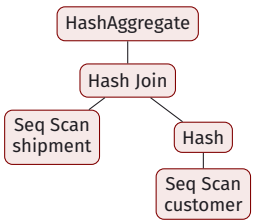


- Una misma consulta tiene muchas expresiones equivalentes, y cada operación tiene varios algoritmos.
- El **plan** es un árbol de operadores: cada nodo es una operación con su algoritmo *(evaluation primitive)*.
- **Hoy**: cuánto cuesta ejecutar cada nodo. **Semana 12**: cómo se elige el plan.

Un plan de evaluación

joins.sql

```sql
SELECT c.city, sum(s.price)
FROM customer c
  JOIN shipment s
    ON s.sender_id = c.customer_id
GROUP BY c.city;
```



chasqui_w11

```
HashAggregate (actual rows=25.00 loops=1)
  Group Key: c.city
  Batches: 1  Memory Usage: 32kB
  Buffers: shared read=167
  ->  Hash Join (actual rows=12000.00 loops=1)
        Hash Cond: (s.sender_id = c.customer_id)
        Buffers: shared read=167
        ->  Seq Scan on shipment s (actual rows=12000.00 loops=1)
              Buffers: shared read=148
        ->  Hash (actual rows=1800.00 loops=1)
              Buckets: 2048  Batches: 1  Memory Usage: 97kB
              Buffers: shared read=19
              ->  Seq Scan on customer c (actual rows=1800.00 loops=1)
                    Buffers: shared read=19
```

Se lee de abajo hacia arriba. `Buffers`: páginas tocadas, `hit` (en el buffer pool) o `read`.

SECCIÓN 02

Medidas de costo

02

El costo se mide en bloques y seeks

Modelo de costo (sec. 15.2)

$$\text{costo} = b \cdot t_T + S \cdot t_S$$

b: bloques transferidos; S: seeks.
Disco: $t_S = 4$ ms, $t_T = 0{,}1$ ms.
SSD: $t_S \approx 90$ µs, $t_T \approx 10$ µs.

n_r, b_r	tuplas y páginas de r
M	páginas de memoria del operador
h_i	altura del índice i

Tres simplificaciones del libro

- Se ignora la CPU.
- No se cuenta la escritura del resultado final: es igual para todos.
- Supuesto pesimista: el buffer pool empieza vacío.

Idea clave

Se cuentan páginas y seeks, no tuplas. Las mismas tuplas pueden costar cien veces más si están repartidas.

Los datos de referencia

Supuestos para las cuentas, inspirados en `chasqui_big`:

	n	b
`shipment`	$2 \cdot 10^6$	33 000
`customer`	200 000	2000

M en PostgreSQL: `work_mem` / 8 KiB. Por defecto, 4 MB = 512 páginas.

chasqui_w11

```
 relname  |  b  |   n
----------+-----+-------
 customer |  19 |  1800
 shipment | 148 | 12000
```

chasqui_w11_big

```
       relname       |   b   |    n
---------------------+-------+---------
 customer            |  2040 |  201800
 shipment            | 24667 | 2012000
 shipment_sender_idx |  2286 | 2012000
```

Medidos con `relpages` y `reltuples` de `pg_class`.

SECCIÓN 03

Selección

03

A1: búsqueda lineal

- Leer todas las páginas en orden y filtrar.
- Un seek y b_r transferencias:

  $$t_S + b_r \cdot t_T = 4 + 33\,000 \times 0{,}1\ \text{ms} \approx 3{,}3\ \text{s}$$

- Igualdad sobre una clave: en promedio, la mitad.
- Sirve siempre: cualquier condición, con o sin índice, con el archivo ordenado o no.

selection.sql: a1

```
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on shipment (actual rows=319.00 loops=1)
   Filter: (status = 'returned'::text)
   Rows Removed by Filter: 11681
   Buffers: shared hit=148
```

`status` no tiene índice: las 148 páginas.

A2–A6: con índices

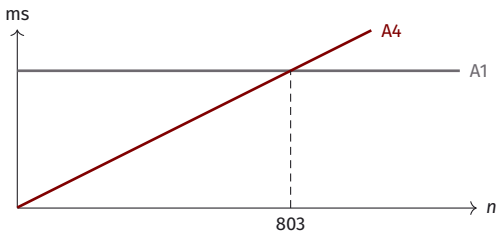
Algoritmo	Condición	Costo
A2, clustered index	igualdad, clave	$(h_i + 1)(t_T + t_S)$
A3, clustered index	igualdad, no clave	$h_i(t_T + t_S) + t_S + b \cdot t_T$
A4, índice secundario	igualdad, clave	$(h_i + 1)(t_T + t_S)$
A4, índice secundario	igualdad, no clave	$(h_i + n)(t_T + t_S)$
A5, clustered index	comparación	$h_i(t_T + t_S) + t_S + b \cdot t_T$
A6, índice secundario	comparación	$(h_i + n)(t_T + t_S)$

- b: bloques con tuplas que cumplen; n: tuplas que cumplen.
- Clustered: las tuplas que cumplen están juntas. Secundario: una lectura aleatoria por tupla.
- A2 con $h_i = 3$: $4 \times 4{,}1 = 16{,}4$ ms. A4, un cliente con 10 envíos: $13 \times 4{,}1 = 53{,}3$ ms.

¿Cuándo pierde el índice secundario?

$$(3 + n) \times 4{,}1 \text{ ms} > 3304 \text{ ms} \iff n > 802{,}9$$

- Desde unas 800 tuplas (el 0,04 % de `shipment`), A1 gana.
- Con 200 000 tuplas: A4 tarda $\approx$ 820 s; A1, 3,3 s.
- En un SSD ($t_S + t_T = 0{,}1$ ms) el punto sube a unas 3300 tuplas.



En PostgreSQL no hay clustered index

- Heap files: todos los índices son secundarios.
- `CLUSTER` reordena solo una vez.
- `pg_stats.correlation`: cuán ordenada está la tabla por cada columna.
- `bt_metap`: $h_i = \texttt{level} + 1$.

selection.sql: correlation

```
   attname   | correlation
-------------+-------------
 created_at  |      -0.001
 sender_id   |       0.020
 shipment_id |       1.000
```

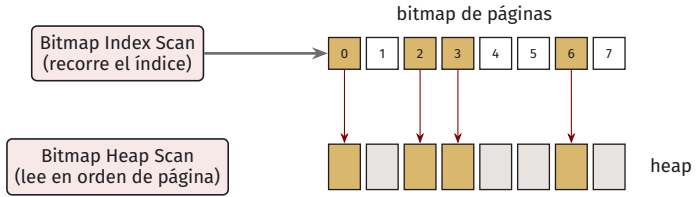
selection.sql: a4

```
Bitmap Heap Scan on shipment (actual rows=10.00 loops=1)
  Recheck Cond: (sender_id = 53)
  Heap Blocks: exact=10
  Buffers: shared hit=10 read=2
  ->  Bitmap Index Scan on shipment_sender_idx (actual rows=10.00 loops=1)
        Index Cond: (sender_id = 53)
        Index Searches: 1
        Buffers: shared read=2
```

A4 con $h_i = 2$: 2 páginas del índice + 1 por tupla = 12.

`shipment_id < 200` (correlación 1): 199 tuplas en 5 páginas.

El bitmap scan



- Cada página marcada se lee una vez y en orden físico: término medio entre A1 y A6.
- Si el bitmap no entra en `work_mem` se vuelve *lossy*: hay que reevaluar la condición (`Recheck Cond`).
- `created_at >= '2026-06-20'`: 507 tuplas (4 %) en 147 de 148 páginas. Con el 69 %, `Seq Scan`.

Conjunciones, disyunciones, negaciones

- **A7**: un índice para la condición más selectiva; las demás, en memoria.
- **A8**: un índice compuesto sobre las columnas.
- **A9**: un índice por condición, intersectar los RID.
- **A10**: disyunción, unir los RID, solo si *todas* tienen índice.
- Negación: casi siempre búsqueda lineal.

En PostgreSQL, A9 y A10 son `BitmapAnd` y `BitmapOr`.

`sender_id BETWEEN 100 AND 140`	216
`created_at >= '2026-06-01'`	1889
`BitmapAnd`: tuplas / páginas	39 / 37

`sender_id = 53 OR status = 'returned'`: `status` no tiene índice ⇒ `Seq Scan`.

Error común

Pensar que `OR` se resuelve igual que `AND`. En una conjunción, una condición sin índice se filtra al final; en una disyunción, obliga a leer **toda** la tabla.

`BitmapAnd` en el plan

selection.sql: conj (sin la línea Recheck Cond)

```
Bitmap Heap Scan on shipment (actual rows=39.00 loops=1)
  Heap Blocks: exact=37
  Buffers: shared hit=42 read=5
  ->  BitmapAnd (actual rows=0.00 loops=1)
        Buffers: shared hit=5 read=5
        ->  Bitmap Index Scan on shipment_sender_idx (actual rows=216.00 loops=1)
              Index Cond: ((sender_id >= 100) AND (sender_id <= 140))
              Index Searches: 1
              Buffers: shared hit=2 read=1
        ->  Bitmap Index Scan on shipment_created_at_idx (actual rows=1889.00 loops=1)
              Index Cond: (created_at >= '2026-06-01 00:00:00-05'::timestamp with time zone)
              Index Searches: 1
              Buffers: shared hit=3 read=4
```

- Cada `Bitmap Index Scan` arma un bitmap; `BitmapAnd` los intersecta.
- El `actual rows=0.00` del `BitmapAnd` no es un error: produce un bitmap, no tuplas.

SECCIÓN 04

Ordenamiento externo

04

Ordenar lo que no entra en memoria

- Se ordena para `ORDER BY`, para el **merge join**, para eliminar duplicados, para agrupar y para construir un B+ tree.
- `shipment`: 33 000 páginas; $M = 512$. No entra.

Fase 1: runs

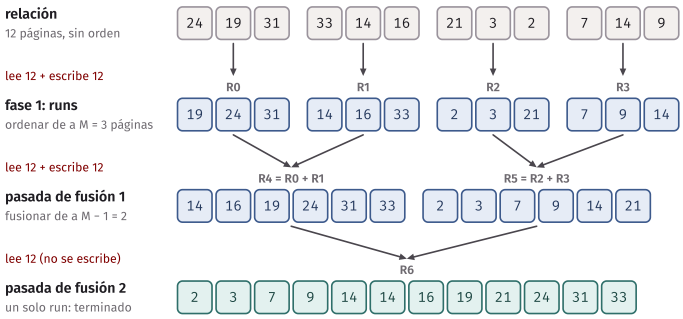
Leer M páginas, ordenarlas en memoria, escribirlas como un **run**. Quedan $\lceil b_r/M \rceil$ runs.

Fase 2: fusión

Fusionar de a $M - 1$ runs (una página de entrada por run y una de salida). Cada **pasada** divide los runs por $M - 1$.



Un ejemplo pequeño: 12 páginas, $M = 3$



Fusión de $M - 1 = 2$ vías: $\lceil \log_2(12/3) \rceil = 2$ pasadas. Transferencias: $24 + 24 + 12 = 60 = 12 \times (2 \times 2 + 1)$.

El costo

External merge sort (sin la escritura final)

$$\text{pasadas} = \lceil \log_{M-1}(b_r/M) \rceil \qquad \text{transferencias} = b_r\,(2 \cdot \text{pasadas} + 1)$$

- Con b_b páginas de buffer por run se fusionan $\lfloor M/b_b \rfloor - 1$ a la vez:

$$\text{seeks} = 2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2 \cdot \text{pasadas} - 1)$$

- Un b_b más grande: menos seeks, a veces más pasadas.

M	runs	pasadas	transferencias
20	1650	3	231 000
102	324	2	165 000
512	65	1	99 000
$\geq$ 33 000	1	0	33 000

`shipment`, $b_r = 33\,000$. Con dos pasadas se ordenan hasta $M(M-1) \approx M^2$ páginas: con $M = 512$, unos 2 GiB.

Error común

Aplicar $\log_2$ a la fusión

La base del logaritmo es $M - 1$: el número de runs que se fusionan a la vez. Con $\log_2$, ordenar `shipment` con $M = 512$ parecería necesitar $\lceil \log_2 65 \rceil = 7$ pasadas; necesita **una**.

Con poca memoria, la memoria extra vale mucho; con bastante, casi nada

De $M = 20$ a $M = 512$, las transferencias bajan de 231 000 a 99 000. De $M = 512$ a $M = 33\,000$, de 99 000 a 33 000, con 64 veces más memoria.

El ordenamiento externo en PostgreSQL

sort.sql: in_memory (work_mem = 4MB)

```
Sort (actual rows=12000.00 loops=1)
  Sort Key: price
  Sort Method: quicksort  Memory: 1413kB
  Buffers: shared hit=151
  ->  Seq Scan on shipment (actual rows=12000.00 loops=1)
        Buffers: shared hit=148
```

sort.sql: external (work_mem = 64kB)

```
Sort (actual rows=12000.00 loops=1)
  Sort Key: price
  Sort Method: external merge  Disk: 1000kB
  Buffers: shared hit=148, temp read=246 written=271
  ->  Seq Scan on shipment (actual rows=12000.00 loops=1)
        Buffers: shared hit=148
```

- `price` no tiene índice: hay que ordenar.
- `external merge`: los runs van a archivos temporales (`temp read/written`).
- Con `LIMIT 10`: `Sort Method: top-N heapsort Memory: 26kB`.

`trace_sort`: los runs y las pasadas

sort.sql: traced (sin los tiempos de CPU ni los runs 3 a 24)

```
LOG:  begin tuple sort: nkeys = 1, workMem = 64, randomAccess = f
LOG:  worker -1 switching to external sort with 6 tapes
LOG:  worker -1 finished writing run 1 to tape 1
LOG:  worker -1 finished writing run 2 to tape 2
LOG:  worker -1 finished writing run 25 to tape 1
LOG:  performsort of worker -1 starting
LOG:  worker -1 finished writing run 26 to tape 2
LOG:  worker -1 using 56 KB of memory for tape buffers
LOG:  starting merge pass of 26 input runs on 6 tapes, 2 KB of memory for each input tape
LOG:  starting merge pass of 5 input runs on 5 tapes, 9 KB of memory for each input tape
LOG:  performsort of worker -1 done (except 5-way final merge)
LOG:  external sort of worker -1 ended, 125 disk blocks used
```

- 26 runs; una pasada de 26 a 5; fusión final de 5. Dos pasadas, como la fórmula con $M = 8$: $\lceil \log_7(148/8) \rceil = 2$.
- La fusión final no se escribe: se entrega al nodo de arriba.
- Los runs no son de M páginas exactas: el formato en memoria es otro.

SECCIÓN 05

Algoritmos de join

05

El ejemplo de la sesión

$\texttt{customer} \bowtie_{\texttt{customer_id}=\texttt{sender_id}} \texttt{shipment}$

		n	b
r	`customer`	200 000	2000
s	`shipment`	$2 \cdot 10^6$	33 000

$M = 102$ páginas (chico a propósito).

- Equi-join: la condición es una igualdad.
- r es la relación de afuera *(outer)*; s, la de adentro *(inner)*.
- Contamos transferencias; los seeks, cuando cambian la conclusión.
- No se cuenta la escritura del resultado.

Nested loop join

pseudocódigo

```
para cada tupla t_r en r:
    para cada tupla t_s en s:
        si (t_r, t_s) cumple θ:
            agregar t_r · t_s
```

Sirve para cualquier condición, sin índices.

Peor caso: una página por relación

transferencias $= n_r \cdot b_s + b_r$
seeks $= n_r + b_r$

$200\,000 \times 33\,000 + 2000 \approx 6{,}6 \cdot 10^9$
transferencias: a 0,1 ms, más de 7 días.

Si la relación chica entra en memoria: $b_r + b_s$.

Block nested loop join

pseudocódigo

```
para cada bloque B_r de r (M - 2 páginas):
    para cada página B_s de s:
        para cada t_r en B_r, t_s en B_s:
            si (t_r, t_s) cumple θ:
                agregar t_r · t_s
```

Costo

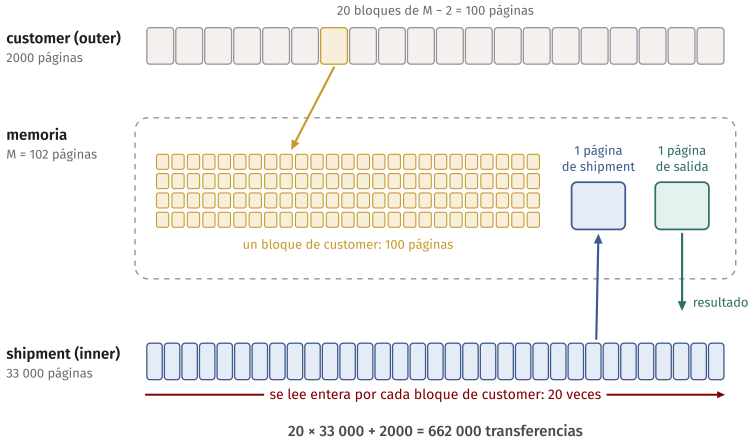
transferencias
$= \lceil b_r/(M-2) \rceil \cdot b_s + b_r$
seeks $= 2\lceil b_r/(M-2) \rceil$

$\lceil 2000/100 \rceil = 20$ bloques:
$20 \times 33\,000 + 2000 =$ 662 000 transferencias,
40 seeks, $\approx$ 66 s.

Con `shipment` afuera:
$330 \times 2000 + 33\,000 = 693\,000$.
Afuera va la relación más chica en páginas.

Con $M \geq b_r + 2 = 2002$: $b_r + b_s = 35\,000$.

Block nested loop join, con $M = 102$



Index nested loop join

pseudocódigo

```
para cada tupla t_r en r:
    buscar en el índice de s las t_s
        con sender_id = t_r.customer_id
    agregar t_r · t_s por cada una
```

Costo

$b_r(t_T + t_S) + n_r \cdot c$, con c el costo de una búsqueda.

Búsqueda en `shipment(sender_id)`: $h_i = 3$ y ≈ 10 envíos: 13 páginas.

- Todos los clientes: $2000 + 200\,000 \times 13 \approx 2{,}6 \cdot 10^6$, aleatorias.
- 200 clientes de una ciudad pequeña: $2000 + 200 \times 13 = 4600$.
- 10 000 clientes: 132 000, aleatorias: vuelve a ganar el hash join.

Idea clave

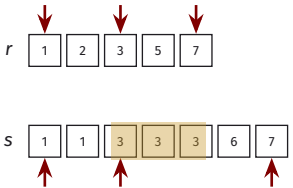
Recorrer el inner menos veces (bloques) o no entero (índice).

Merge join

pseudocódigo

```
mientras p_r y p_s no se acaben:
    si p_r.customer_id < p_s.sender_id: avanzar p_r
    si p_r.customer_id > p_s.sender_id: avanzar p_s
    si son iguales (valor v):
        S_s = las tuplas de s con valor v
        combinar cada tupla de r con valor v
            con cada tupla de S_s
```

Las dos entradas ordenadas por la clave del join. Solo equi-joins.



- S_s (las tuplas con el mismo valor) tiene que caber en memoria.
- Ordenadas: $b_r + b_s$ transferencias.

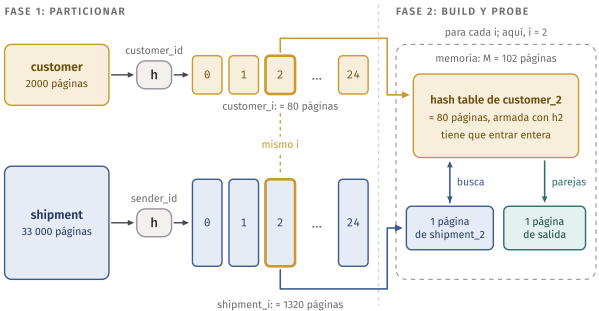
Merge join: el costo de ordenar

- Si no vienen ordenadas: external sort de cada una y escribir el resultado ordenado.

	cuenta	transferencias
Ordenar `customer` (20 runs, 1 pasada)	$2000 \times 3 + 2000$	8000
Ordenar `shipment` (324 runs, 2 pasadas)	$33\,000 \times 5 + 33\,000$	198 000
Fusionar	$2000 + 33\,000$	35 000
Total		$\approx$ 241 000

- Atractivo si las entradas ya vienen ordenadas: por otro merge join, por un `ORDER BY` sobre la clave…
- …o por un índice. Pero con un índice secundario, cada tupla es una página distinta del heap (*hybrid merge join* en el libro).

Hash join: la idea



1. **Particionar** las dos con la misma función h en n_h particiones.
2. Por cada i: armar una hash table en memoria con r_i (**build**) y recorrer s_i (**probe**).

Build: la relación más chica. Cada r_i tiene que entrar en memoria.

Hash join: el costo

Sin particionamiento recursivo

transferencias $= 3(b_r + b_s) + 4n_h$ seeks $= 2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$

- Particionar: leer y escribir las dos, $2(b_r + b_s)$. Build y probe: leerlas otra vez.
- $n_h = \lceil 2000/102 \rceil \times 1{,}2 = 24 \approx 25$ particiones de unas 80 páginas.
- $3 \times 35\,000 + 4 \times 25 = 105\,100$ transferencias.
- Si el build entra entero en memoria: $b_r + b_s = 35\,000$.
- **Memoria**: $n_h < M$ y cada partición de build $\leq M \Rightarrow M > \sqrt{b_{\text{build}}}$. Con `customer`: $\sqrt{2000} \approx 45$.
- Si no alcanza: **particionamiento recursivo**. Con $M = 40$, 60 particiones y solo 39 páginas.

Overflow y hybrid hash join

Overflow

Una partición de build no entra en memoria: hash malo o valores muy repetidos.

- *Resolution*: volver a particionar esa partición.
- *Avoidance*: más particiones desde el inicio, y juntar las chicas.

Un solo valor repetido no lo separa ninguna función hash.

Hybrid hash join

Si sobra memoria ($M \gg \sqrt{b_{\text{build}}}$), la partición 0 de build se queda en memoria como hash table. Las tuplas de probe de la partición 0 se combinan en el momento. Se ahorra escribir y releer las dos particiones 0.

Comparación con $M = 102$

Algoritmo	Transferencias
Nested loop join	$\approx 6{,}6 \cdot 10^9$
Block nested loop join (`customer` afuera)	662 000
Index nested loop join (todo `customer`)	$\approx 2{,}6 \cdot 10^6$, aleatorias
Merge join (ordenando las dos)	$\approx$ 241 000
Hash join	$\approx$ 105 000
Index nested loop join (200 clientes)	$\approx$ 4600

Con $M \geq 2002$: block nested loop y hash join, 35 000 los dos. Cuentas en `costs.mjs`.

Error común

Elegir el join por la cantidad de tuplas y no por la de páginas, o creer que hash join siempre gana. Depende de la memoria, de los índices y de los filtros.

Joins complejos

- **Conjunción** $\theta_1 \wedge \theta_2$: join con la igualdad (hash o merge), el resto se comprueba en cada pareja.
- **Disyunción** $\theta_1 \vee \theta_2$: nested loop, o la unión de los joins por separado.
- **Sin igualdad** (`r.distance_km > a.agency_id * 40`): ni hash ni merge. Solo nested loop.

`joins.sql`	**plan de PostgreSQL**
`equi_filter`: igualdad y desigualdad	`Hash Join` + `Join Filter`
`theta`: solo desigualdad	`Nested Loop` + `Materialize`

Los joins en PostgreSQL

`Nested Loop`, `Merge Join` y `Hash Join`; `enable_*` los desalientan. En `chasqui_w11` gana el hash join (167 páginas).

joins.sql: merge (enable_hashjoin = off), extracto

```
->  Merge Join (actual rows=12000.00 loops=1)
      Merge Cond: (c.customer_id = s.sender_id)
      ->  Index Scan using customer_pkey on customer c (actual rows=1800.00 loops=1)
            Buffers: shared hit=22 read=3
      ->  Index Scan using shipment_sender_idx on shipment s (actual rows=12000.00 loops=1)
            Buffers: shared hit=11343 read=13
```

joins.sql: nestloop (y enable_mergejoin = off), extracto

```
      ->  Memoize (actual rows=1.00 loops=12000)
            Cache Key: s.sender_id
            Hits: 10282  Misses: 1718  Evictions: 0  Overflows: 0  Memory Usage: 191kB
            ->  Index Scan using customer_pkey on customer c (actual rows=1.00 loops=1718)
                  Index Cond: (customer_id = s.sender_id)
                  Buffers: shared hit=5154
```

Merge join: 11 356 accesos para una tabla de 148 páginas. `Memoize`: 1718 búsquedas.

Hybrid hash join en PostgreSQL

- Memoria: `work_mem` × `hash_mem_multiplier` (2).
- Si el build no entra: `Batches`, una potencia de 2. El batch 0 se queda en memoria.
- Si un batch crece demasiado, duplica los batches.
- Con los tres `enable_*` en `off`: el plan igual tiene un join, marcado `Disabled: true`.

joins.sql: batches (work_mem = 64kB)

```
Hash Join (actual rows=46421.00 loops=1)
  Hash Cond: (e.shipment_id = s.shipment_id)
  Buffers: shared hit=148 read=342, temp read=233 written=233
  ->  Seq Scan on shipment_event e (actual rows=46421.00 loops=1)
        Buffers: shared read=342
  ->  Hash (actual rows=12000.00 loops=1)
        Buckets: 4096  Batches: 8  Memory Usage: 84kB
        Buffers: shared hit=148, temp written=28
        ->  Seq Scan on shipment s (actual rows=12000.00 loops=1)
              Buffers: shared hit=148
```

En los archivos temporales solo van las columnas que se necesitan.

En la tabla grande: el join completo

big.sql: hash (chasqui_w11_big, work_mem = 4MB), extracto

```
->  Hash Join (actual rows=2012000.00 loops=1)
      Hash Cond: (s.sender_id = c.customer_id)
      Buffers: shared read=26707, temp read=4350 written=4350
      ->  Seq Scan on shipment s (actual rows=2012000.00 loops=1)
            Buffers: shared read=24667
      ->  Hash (actual rows=201800.00 loops=1)
            Buckets: 262144  Batches: 2  Memory Usage: 6664kB
            Buffers: shared read=2040, temp written=426
Execution Time: 698.488 ms
```

- 2 batches: uno en memoria, el otro a disco. Cada tabla se lee una vez (26 707).
- Con `work_mem = '256MB'`: `Batches: 1` y sin `temp`.
- Merge join ordenando (`merge_sort`): las mismas 26 707 y 11 479 páginas temporales escritas.

En la tabla grande: un join selectivo

big.sql: selective (Iquitos, 38 clientes)

```
Nested Loop (actual rows=370.00 loops=1)
  Buffers: shared hit=2342 read=182
  ->  Seq Scan on customer c (actual rows=38.00 loops=1)
        Filter: (city = 'Iquitos'::text)
        Rows Removed by Filter: 201762
        Buffers: shared hit=2040
  ->  Index Scan using shipment_sender_idx on shipment s (actual rows=9.74 loops=38)
        Index Cond: (sender_id = c.customer_id)
        Index Searches: 38
        Buffers: shared hit=302 read=182
Execution Time: 9.681 ms
```

- 2040 páginas de `customer` y 38 búsquedas de $\approx 3 + 10$.
- 2524 páginas y 10 ms, contra 26 707 y 700 ms.
- Huaraz (7083 clientes): hash join otra vez.

Con los números reales (b = 2040 y 24 667, $M = 512$)

Block nested loop 100 708; merge join 133 535; hash join 80 141. PostgreSQL midió $26\,707 + 2 \times 4350 \approx 35\,400$ (hybrid, columnas necesarias).

SECCIÓN 06

Otras operaciones

06

Duplicados, proyección y conjuntos

- **Ordenando**: los iguales quedan seguidos. Se pueden descartar durante el external sort.
- **Con hash**: particionar y armar una hash table por partición.
- Proyección: quitar columnas, tupla por tupla; con `DISTINCT`, eliminar duplicados después.
- `UNION`, `INTERSECT`, `EXCEPT`: ordenando o con hash.

other.sql: distinct_sort

```
(enable_hashagg = off)

Unique (actual rows=26.00 loops=1)
  ->  Sort (actual rows=1800.00 loops=1)
        Sort Key: city
        Sort Method: quicksort  Memory: 49kB
        ->  Seq Scan on customer (actual rows=1800.00 loops=1)
```

other.sql: intersect

```
HashSetOp Intersect (actual rows=26.00 loops=1)
  ->  Seq Scan on customer (actual rows=1800.00 loops=1)
  ->  Seq Scan on agency (actual rows=29.00 loops=1)
```

Sin `enable_hashagg = off`, `DISTINCT` es un `HashAggregate`.

Outer join y agregación

other.sql: outer_join

```
HashAggregate (actual rows=1800.00 loops=1)
  Group Key: c.customer_id
  Batches: 1  Memory Usage: 217kB
  ->  Hash Right Join (actual rows=12082.00 loops=1)
        Hash Cond: (s.sender_id = c.customer_id)
        ->  Seq Scan on shipment s (actual rows=12000.00 loops=1)
        ->  Hash (actual rows=1800.00 loops=1)
              Buckets: 2048  Batches: 1  Memory Usage: 80kB
              ->  Seq Scan on customer c (actual rows=1800.00 loops=1)
```

Build = el lado que se preserva: al final del probe se emiten los 82 clientes sin pareja, con `NULL`.

- **Agregación**: ordenando (`GroupAggregate`) o con hash (`HashAggregate`).
- Un acumulador por grupo: `count`, `sum`, `min`, `max`; `avg` = suma y cantidad.
- 12 000 envíos, 4 estados: la hash table tiene 4 entradas.
- `HashAggregate` es un operador de P4.

SECCIÓN 07

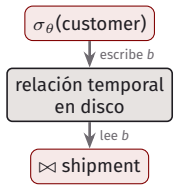
Evaluación de expresiones

07



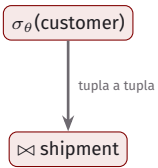
Materialización y pipelining

Materialización



$+2b$ transferencias por resultado intermedio.

Pipelining



Sin escrituras intermedias; el resultado empieza a salir antes.

- **Por demanda** *(pull)*: el de arriba pide la siguiente tupla. Iteradores; PostgreSQL.
- **Por producción** *(push)*: cada operador produce y llena un buffer.

Operadores bloqueantes

Operador bloqueante

Necesita toda su entrada antes de producir la primera tupla.

- **Sort**: la primera tupla ordenada puede ser la última que llega.
- **Build del hash join**. El probe, en cambio, sí es pipeline.
- **Agregación**, salvo con la entrada ordenada por el grupo.

Un plan es un conjunto de pipelines separados por operadores bloqueantes.

Error común

Confundir “pipeline” con “no usa memoria”. El hash join es pipeline del lado de probe, pero guarda todo el build en memoria. El pipeline ahorra escribir los resultados **intermedios**.

Pipelining en PostgreSQL

other.sql: limit_pipeline

```
Limit (actual rows=5.00 loops=1)
  Buffers: shared hit=3
  ->  Seq Scan on shipment (actual rows=5.00 loops=1)
        Filter: (status = 'returned'::text)
        Rows Removed by Filter: 141
        Buffers: shared hit=3
```

El scan se detiene en 3 páginas de 148.

other.sql: limit_blocking (ORDER BY price)

```
Limit (actual rows=5.00 loops=1)
  Buffers: shared hit=151
  ->  Sort (actual rows=5.00 loops=1)
        Sort Key: price
        Sort Method: top-N heapsort  Memory: 26kB
        Buffers: shared hit=151
        ->  Seq Scan on shipment (actual rows=319.00 loops=1)
              Filter: (status = 'returned'::text)
              Rows Removed by Filter: 11681
              Buffers: shared hit=148
```

El sort necesita las 319: las 148 páginas.

Un CTE `MATERIALIZED` se guarda en un tuplestore, pero se llena a pedido: con `LIMIT 5`, también 3 páginas.

SECCIÓN 08

El modelo de iteradores

08

open, next, close

- `open()`: prepara y abre a los hijos. Un sort ordena aquí; un hash join arma la hash table.
- `next()`: la siguiente tupla o fin. Pide a sus hijos lo necesario y guarda su estado.
- `close()`: libera y cierra a los hijos.
- Un plan es un **árbol de iteradores**: la raíz pide hasta el fin (modelo Volcano).

pseudocódigo

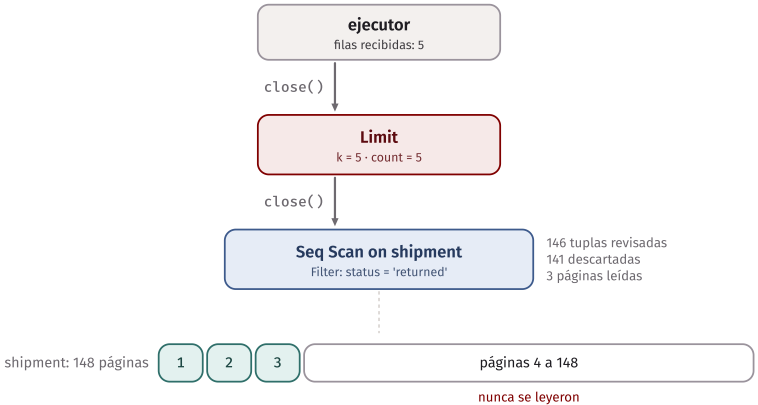
```
Sort.open():
    hijo.open()
    T = todas las tuplas de hijo
    ordenar T;  pos = 0

Sort.next():
    si pos = |T|: devolver fin
    pos = pos + 1;  devolver T[pos - 1]
```

Bloqueante: todo el trabajo en `open()`. Un operador pipeline hace su trabajo en cada `next()`.

Costo del modelo: una llamada por tupla y por operador. Algunos sistemas devuelven lotes (vectorización, CMU #14).

El plan con `LIMIT 5`, como árbol de iteradores



con 5 filas, el Limit responde fin sin pedir más; close() baja

Un iterador en Node.js: `Limit`

limit.mjs (sin las líneas en blanco)

```
export class Limit {
  constructor(child, k) {
    this.child = child;
    this.k = k;
  }
  async open() {
    await this.child.open();
    this.count = 0;
  }
  next() {
    if (this.count >= this.k) return null;
    const row = this.child.next();
    if (row !== null) this.count++;
    return row;
  }
  close() {
    this.child.close();
  }
}
```

- La interfaz de `iterators.mjs`: `open()` es `async` (habla con PostgreSQL); `next()` devuelve una fila o `null`.
- El estado entre llamadas: `this.count`.
- `Counter` cuenta las llamadas a su hijo.

pipeline.mjs

```
$ PGDATABASE=chasqui_w11 node pipeline.mjs
filas del plan: 1, 2, 3, 4, 5
llamadas al SeqScan: { opens: 1, nexts: 5, rows: 5, closes: 1 }
filas que SeqScan.open() trajo de PostgreSQL: 12000
```

Nuestro `SeqScan` trae todo en `open()`; uno de verdad lee página por página.

Iteradores con generadores

limit.mjs

```
1  export function* limit(input, k) {
2    if (k <= 0) return;
3    let count = 0;
4    for (const row of input) {
5      yield row;
6      if (++count === k) return;
7    }
8  }
```

`values(rows, log)` entrega las filas de un arreglo, anota cada una y, en un `finally`, que lo cerraron.

pipeline.mjs, segunda parte

```
values: entrega {"id":1}
raíz: recibe {"id":1}
values: entrega {"id":2}
raíz: recibe {"id":2}
values: cerrado
```

- Cada `next()` corre hasta el siguiente `yield`.
- Al terminar `limit`, `for...of` cierra a `values`: el `finally` es el `close()`.
- El estado es una variable local.

El laboratorio

En `code/week-11/iterators.mjs`:

- `Filter(child, predicate)`
- `Project(child, columns)`
- `NestedLoopJoin(outer, inner, predicate)`
- `HashJoin(build, probe, buildKey, probeKey)`

`iterators.test.mjs` compara el join de los clientes de una ciudad con PostgreSQL. Es el esqueleto de **P4**.

Decisión de diseño

¿Cómo vuelve a empezar el inner del `NestedLoopJoin`?

- Cerrarlo y abrirlo otra vez, o
- guardar sus filas en memoria.

Justifiquen la suya con las fórmulas de hoy.

`make test` (con `make big` hecho)

code/week-11

```
$ make test
chasqui-load chasqui_w11
Base de datos chasqui_w11 lista.
...
✓ SeqScan devuelve todas las agencias (9.096542ms)
x Filter y Project (0.989083ms)
x NestedLoopJoin: los envíos de los clientes de una ciudad (0.57225ms)
x HashJoin da el mismo resultado que NestedLoopJoin (0.123459ms)
✓ Limit(2) sobre Values: dos filas y dos llamadas a next() (0.831042ms)
...
i tests 41
i pass 38
i fail 3
```

- Las tres que fallan son las del laboratorio: pasan cuando completen los operadores. Sin `make big`, la de `big.test.mjs` se salta y pasan 37.
- `selection.sql`, `sort.sql`, `joins.sql`, `other.sql`: las pruebas corren el mismo texto y revisan los planes.
- `make big`: la copia de `chasqui_big` con el índice, para `big.sql`.

SECCIÓN 09

Cierre

09

Resumen

- Costo: $b \cdot t_T + S \cdot t_S$; M = memoria del operador.
- A1 sirve siempre; un índice secundario cuesta una lectura por tupla.
- PostgreSQL: todos los índices son secundarios; bitmap scan, `BitmapAnd`, `BitmapOr`.
- External merge sort: $b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$; base $M - 1$.
- Joins: block nested loop (la chica afuera), index nested loop (outer filtrado), merge join (ordenadas), hash join ($M > \sqrt{b_{\text{build}}}$).
- Duplicados, conjuntos y agregación: ordenando o con hash.
- Pipelining por demanda con iteradores; sort y build son bloqueantes.

Para la próxima clase

- Laboratorio: joins a mano y en el motor; los iteradores en Node.js.
- Leer Silberschatz, secs. 16.1 a 16.3 (equivalencias y estimación de cardinalidades), la nota 10 de CS186 (*Query Optimization*) y, de PostgreSQL, *Statistics Used by the Planner* y *Row Estimation Examples*.
- Semana 12: se entrega P3 y se publica **P4** (`quipu`: consultas con iteradores).
- Correr `make test` en `code/week-11` y leer los apuntes de la semana.

Pregunta

Para `customer` ⋈ `shipment` el optimizador eligió hash join, y para los clientes de Iquitos, index nested loop. ¿Cómo supo, *antes* de ejecutar, que Iquitos tiene 38 clientes y Huaraz 7083?