

Bases de Datos

Optimización de consultas

Almacenamiento, índices y consultas

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: equivalencias y cardinalidades

1. Del SQL al plan
2. Reglas de equivalencia
3. Estadísticas del catálogo
4. Selectividad de selecciones y joins
5. Cuando falla la independencia

Sesión B: elección de planes

6. Programación dinámica y System R
7. El optimizador de PostgreSQL
8. Leer `EXPLAIN ANALYZE`
9. Arreglar un plan

Lectura

Silberschatz, secs. 16.1–16.3 (sesión A) y 16.4 (sesión B; 16.5 y 16.6 opcionales). CS186, nota 10 (*Query Optimization*). PostgreSQL, *Statistics Used by the Planner*, *Row Estimation Examples*, *Using EXPLAIN* y *Planner/Optimizer*. Además: CMU 15-445 #16; PostgreSQL `pg_stats`, `CREATE STATISTICS`, *Multivariate Statistics Examples*, *Explicit JOIN Clauses*, *GEQO* y *Planner Cost Constants*; Winand, *Use The Index, Luke!*; Leis et al., *How Good Are Query Optimizers, Really?* (VLDB 2015).

SECCIÓN 01

Del SQL al plan

01



La semana pasada elegíamos nosotros

- Semana 11: el costo de cada algoritmo. Para el mismo `customer` ⋈ `shipment`, de $6{,}6 \cdot 10^9$ transferencias (nested loop) a 35 000 (hash join en memoria).
- Pero el plan lo decidíamos nosotros: “block nested loop, `customer` afuera”.
- SQL dice **qué** filas, no **cómo**.
- El **optimizador** *(query optimizer)* decide.

Tres pasos (sec. 16.1)

1. Generar expresiones equivalentes (reglas del álgebra relacional).
2. Anotarlas con algoritmos: planes alternativos.
3. Estimar el costo de cada plan y quedarse con el más barato.

La consulta de la Sesión A

equivalences.sql

```sql
SELECT c.first_name, c.last_name,
       s.shipment_id, a.name
FROM customer c, shipment s, agency a
WHERE c.customer_id = s.sender_id
  AND s.dest_agency_id = a.agency_id
  AND c.city = 'Arequipa'
  AND a.region = 'Cusco';
```

Envíos de clientes de Arequipa a agencias de la región Cusco.

Traducción directa:

$$\Pi_{\dots}\left(\sigma_{\theta_{cs} \wedge \theta_{sa} \wedge city \wedge region}(c \times s \times a)\right)$$

Al pie de la letra

$1800 \times 12\,000 \times 29 = 626\,400\,000$ tuplas en el producto.

Lo que hace PostgreSQL

EXPLAIN (where_form)

```
Hash Join  (cost=43.99..345.34 rows=20 width=43)
  Hash Cond: (s.dest_agency_id = a.agency_id)
  ->  Hash Join  (cost=42.61..342.19 rows=593 width=29)
        Hash Cond: (s.sender_id = c.customer_id)
        ->  Seq Scan on shipment s  (cost=0.00..268.00 rows=12000 width=12)
        ->  Hash  (cost=41.50..41.50 rows=89 width=25)
              ->  Seq Scan on customer c  (cost=0.00..41.50 rows=89 width=25)
                    Filter: (city = 'Arequipa'::text)
  ->  Hash  (cost=1.36..1.36 rows=1 width=22)
        ->  Seq Scan on agency a  (cost=0.00..1.36 rows=1 width=22)
              Filter: (region = 'Cusco'::text)
```

- Sin producto: dos **hash joins**. Los filtros, pegados a sus scans: antes de los joins.
- Costo 345, contra 7 830 989 del plan del producto.
- Con `JOIN ... ON` sale el mismo plan, línea por línea: se optimiza la expresión, no el texto.

El costo es una estimación

> **Idea clave**
>
> El optimizador no ejecuta los planes para compararlos. Los costea con fórmulas (semana 11) y con estadísticas de las tablas.

- Si las estadísticas se equivocan, elige **con confianza** un plan malo.
- Casi todo plan malo que van a diagnosticar es un error de estimación, no de algoritmo.
- Hoy:
  - cómo se generan las alternativas (reglas de equivalencia),
  - cómo se estima cuántas tuplas salen de cada operador *(cardinality estimation)*,
  - cómo se elige sin probarlas todas.

SECCIÓN 02

Reglas de equivalencia

02

Expresiones equivalentes

Equivalencia

Dos expresiones del álgebra relacional son **equivalentes** si generan el mismo resultado en toda instancia válida de la base de datos.

- No basta con los datos de hoy: tiene que valer para cualquier contenido.
- El orden de las tuplas no importa; los atributos se toman por nombre.
- SQL usa multiconjuntos: las reglas que vemos valen también con la versión multiconjunto de cada operador.
- Una **regla de equivalencia** permite reemplazar una forma por la otra en cualquier parte del árbol: es un movimiento para generar alternativas.

Las reglas (sec. 16.2.1), 1 a 6

Regla	Forma
1. Cascada de σ	$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. Conmutatividad de σ	$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
3. Cascada de Π	$\Pi_{L_1}(\Pi_{L_2}(\ldots \Pi_{L_n}(E))) = \Pi_{L_1}(E)$ si $L_1 \subseteq \ldots \subseteq L_n$
4. σ sobre producto	$\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
	$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. Conmutatividad del join	$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
6. Asociatividad del join	$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
	$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$ si θ_2 usa solo atributos de E_2 y E_3

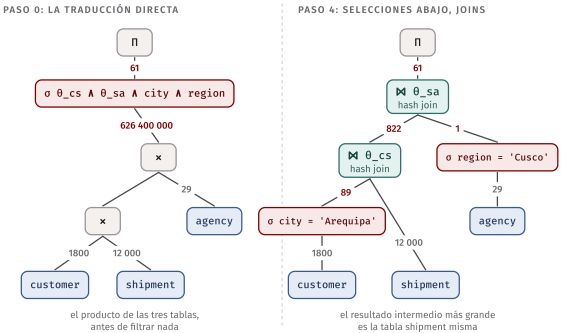
Las reglas, 7 a 12

Regla	Forma
7. Bajar σ por un join	$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_0}(E_1) \bowtie_\theta E_2$ si θ_0 usa solo E_1
	$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie_\theta \sigma_{\theta_2}(E_2)$
8. Bajar Π por un join	$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$ si θ usa solo $L_1 \cup L_2$
9, 10. $\cup$ y $\cap$	conmutativas y asociativas (la diferencia, no)
11. σ sobre $\cup$, $\cap$, $-$	$\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$ (igual con $\cap$ y $-$)
	$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$ (igual con $\cap$; no con $\cup$)
12. Π sobre $\cup$	$\Pi_L(E_1 \cup E_2) = \Pi_L(E_1) \cup \Pi_L(E_2)$

Ojo

Π se distribuye sobre la diferencia: dos tuplas distintas pueden coincidir al proyectar.

Aplicar las reglas: del producto al plan



1. Reglas 1 y 2: partir el σ en cuatro y ordenarlo.
2. Regla 7: *city* baja a `c`, *region* baja a `a`.
3. Regla 4: θ_{cs} y θ_{sa} convierten los productos en joins.
4. Regla 8: proyectar lo justo (`width` 12 en vez de 67 en el scan de `shipment`).

Cuánto se gana

Árbol	Resultado intermedio	Tuplas
Paso 0	$c \times s \times a$	626 400 000
Filtros al final	$c \bowtie s \bowtie a$	12 000
Filtros abajo	$\sigma_{city}(c)$	89
	$\sigma_{region}(a)$	1
	$\sigma_{city}(c) \bowtie s$	822
	resultado	61

Heurística

Bajar las selecciones y cambiar productos por joins casi siempre conviene: se aplica sin costear. El orden de los joins, en cambio, depende de los datos.

La trampa del outer join

equivalences.sql

```sql
-- condición en el ON: 33 filas
SELECT count(*) FROM agency a
  LEFT JOIN route r
    ON r.origin_agency_id = a.agency_id
   AND r.distance_km > 800;
-- condición en el WHERE: 10 filas
SELECT count(*) FROM agency a
  LEFT JOIN route r
    ON r.origin_agency_id = a.agency_id
WHERE r.distance_km > 800;
```

- ON: `agency LEFT JOIN` σ(`route`). Las agencias sin rutas largas siguen, con `NULL`.
- WHERE: σ(`agency LEFT JOIN route`). `NULL > 800` es *unknown*: se descartan.
- PostgreSQL convierte el segundo en un inner join: el plan dice `Hash Join`, no `Hash Left Join`.
- Error común: mover una condición del WHERE al ON es inofensivo en un inner join; en un outer join cambia el resultado.

¿Cuántos órdenes de join hay?

Solo con las reglas 5 y 6, para n relaciones:

$$\frac{(2(n-1))!}{(n-1)!}$$

n	órdenes
3	12
5	1680
7	665 280
10	$\approx 1{,}76 \cdot 10^{10}$

- $n = 3$: 3 pares para empezar, $\times 2$ lados, $\times 2$ lados con la tercera.
- Enumerar todas las expresiones equivalentes (sec. 16.2) es posible compartiendo subárboles (el *memo* de Volcano/Cascades), pero crece demasiado rápido.
- Ningún optimizador los prueba todos: lo vemos en la Sesión B.

SECCIÓN 03

Estadísticas del catálogo

03

Lo que guarda el catálogo (sec. 16.3.1)

- n_r: tuplas de r; b_r: bloques.
- l_r: bytes por tupla; f_r: tuplas por bloque.
- $V(A, r)$: valores distintos de A (= tamaño de $\Pi_A(r)$).
- Histogramas.

No se actualizan en cada INSERT

Los recalcula `ANALYZE` (a mano o el autovacuum).

stats.sql: sizes

```
 relname  |  n_r  | b_r
----------+-------+-----
 agency   |    29 |   1
 customer |  1800 |  19
 shipment | 12000 | 148
 trip     |  5914 |  44
 vehicle  |    24 |   1
```

n_r y b_r son `reltuples` y `relpages` de `pg_class`.

Histogramas: equi-width y equi-depth

weight_kg, buckets de 15 kg

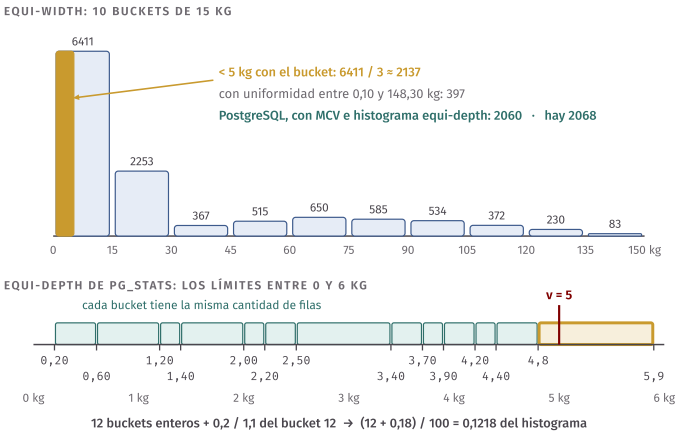
bucket	from_kg	to_kg	count
1	0	15	6411
2	15	30	2253
3	30	45	367
4	45	60	515
5	60	75	650
6	75	90	585
7	90	105	534
8	105	120	372
9	120	135	230
10	135	150	83

- **Equi-width**: rangos del mismo ancho; se guarda cuántas tuplas tiene cada uno. Más de la mitad cae en el primero.
- **Equi-depth**: buckets con las mismas tuplas; se guardan los límites. Donde hay muchos valores, buckets angostos.
- PostgreSQL: equi-depth de 100 buckets (120 envíos cada uno):

weight_bounds

```
{0.20,0.60,1.20,1.40,2.00,2.20,2.50,3.40,3.70,3.90,4.20,4.40}
```

Tres estimaciones de `weight_kg < 5`



pg_stats

stats.sql: columns

attname	null_frac	n_distinct	n_mcv	n_bounds	correlation
created_at	0	-0.977	100	101	-0.001
sender_id	0	-0.14316666	100	101	0.020
shipment_id	0	-1		101	1.000
status	0	4	4		0.857
trip_id	0.041666668	-0.40633333	100	101	0.000
weight_kg	0	-0.110416666	100	101	0.010

- `n_distinct` $= V(A, r)$; si es negativo, es menos una fracción de las filas (-1: todos distintos; $-0{,}143 \times 12\,000 \approx 1718$ remitentes).
- `most_common_vals`/`freqs`: los MCV con su frecuencia exacta.
- `histogram_bounds`: equi-depth de los valores que no son MCV.
- `correlation`: orden físico frente a orden de los valores.

ANALYZE trabaja con una muestra

- Muestra de 300× `default_statistics_target` = 30 000 filas.
- `chasqui_w12`: 12 000 envíos, la muestra es toda la tabla.
- `chasqui_big`: 2 millones; ya no.
- `n_distinct` es lo más difícil: 183 076 remitentes según `pg_stats`, 201 789 de verdad.

chasqui_big

```
Seq Scan on shipment  (cost=0.00..49817.00 rows=229435 width=68)
  Filter: (status = 'in_transit'::text)

   status   |  count
------------+---------
 registered |  222556
 returned   |  221985
 in_transit |  222334
 delivered  | 1345125
```

Error común

Tratar las estimaciones como exactas: vienen de una muestra, con supuestos, y envejecen.

SECCIÓN 04

Selectividad

04

Igualdad

Libro: uniformidad, $\mathrm{sel}(A = a) = 1/V(A, r)$.

PostgreSQL:

- a en la lista de MCV: su frecuencia.
- Si no: lo que no cubre la lista, repartido entre los demás valores:

$$\frac{1 - \mathit{null_frac} - \sum \mathit{freq}_{\mathrm{MCV}}}{V(A, r) - \#\mathrm{MCV}}$$

```
eq_mcv: status = 'in_transit'

Seq Scan on shipment  (cost=0.00..298.00 rows=66 width=67)
```

$0{,}0055 \times 12\,000 = 66$. Uniforme: 3000.

```
eq_other: sender_id = 7

Seq Scan on shipment  (cost=0.00..298.00 rows=5 width=67)
```

$(1 - 0{,}2889)/(1718 - 100) \times 12\,000 = 5{,}3$. Hay 6.

Comparaciones: el libro

$$\text{size}(\sigma_{A \le v}(r)) = \begin{cases} 0 & v < \text{mín}(A, r) \\ n_r & v \ge \text{máx}(A, r) \\ n_r \cdot \dfrac{v - \text{mín}(A, r)}{\text{máx}(A, r) - \text{mín}(A, r)} & \text{en otro caso} \end{cases}$$

- `weight_kg < 5`, de 0,10 a 148,30 kg: 397 filas.
- Con el equi-width de 15 kg: un tercio de 6411 = 2137.
- Hay 2068.

Comparaciones: PostgreSQL

1. MCV que cumplen el predicado, sin suponer nada: 25 de los 100 son < 5; suman $0{,}0985$.
2. El histograma describe el resto: $1 - 0 - 0{,}3955 = 0{,}6045$.
3. 5 cae en el bucket 12, entre 4,8 y 5,9. Interpolación lineal dentro del bucket:

$$\frac{12 + 0{,}2/1{,}1}{100} = 0{,}1218$$

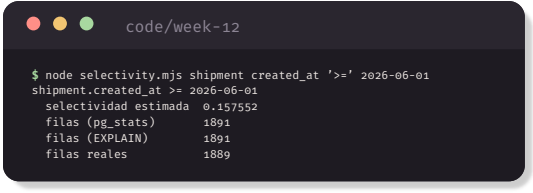
4. Corrección de PostgreSQL: el histograma estima $\leq v$; para $< v$ resta $1/(1325 - 100) = 0{,}0008$.
5. $0{,}0985 + 0{,}6045 \times (0{,}1218 - 0{,}0008) = 0{,}1716$; por 12 000: 2060.

EXPLAIN dice 2060

`Seq Scan on shipment (cost=0.00..298.00 rows=2060 width=67)`. Hay 2068.

selectivity.mjs

- Lee `pg_stats`, hace la cuenta y la compara con `EXPLAIN` y con un `count(*)`.
- Funciones puras: `histogramFraction`, `inequalitySelectivity`, `equalitySelectivity`.
- El valor va como `$1`; tabla y columna, con `escapeIdentifier`.
- Diez predicados en las pruebas: coincide con `EXPLAIN` en todos.



Conjunción, disyunción, negación

Supuesto de **independencia**:

$\theta_1 \wedge \theta_2$	$s_1 \cdot s_2$
$\theta_1 \vee \theta_2$	$1 - (1 - s_1)(1 - s_2)$
$\neg\theta$	$1 - s - \mathit{null_frac}$

`NULL <> 'delivered'` es *unknown*: la negación no devuelve los nulos.

`status` y `weight_kg` no tienen relación:

	estim.	**real**
`returned` AND < 5	55	45
`returned` OR < 5	2324	2342
`<> 'delivered'`	885	885

Sin estadísticas: 0,5 % para $=$ y un tercio para $<$.

Tamaño de un join (sec. 16.3.3)

- $R \cap S = \emptyset$: producto, $n_r \cdot n_s$.
- $R \cap S$ clave de R: a lo más n_s.
- Clave foránea de S hacia R, sin nulos: exactamente n_s.
- En general, con uniformidad:

$$\text{size}(r \bowtie s) \approx \frac{n_r \cdot n_s}{\text{máx}(V(A, r), V(A, s))}$$

- `customer` $\bowtie$ `shipment`: $1800 \times 12\,000 / \text{máx}(1800, 1718) = 12\,000$. Exacto.
- Con nulos en la clave foránea (`trip_id`), se restan: no se unen con nadie.

Un join después de un filtro

stats.sql: join_fk

```
Hash Join  (cost=42.61..342.19 rows=593 width=19) (actual rows=822.00 loops=1)
  Hash Cond: (s.sender_id = c.customer_id)
  ->  Seq Scan on shipment s  (cost=0.00..268.00 rows=12000 width=8) (actual rows=12000.00 loops=1)
  ->  Hash  (cost=41.50..41.50 rows=89 width=19) (actual rows=89.00 loops=1)
        ->  Seq Scan on customer c  (cost=0.00..41.50 rows=89 width=19) (actual rows=89.00 loops=1)
              Filter: (city = 'Arequipa'::text)
```

- $89 \times 12\,000/1800 = 593$: si todos los clientes enviaran lo mismo.
- Hay 822: los de Arequipa envían más. Uniformidad, no la fórmula.
- El join con la agencia de Cusco estima $593/29 = 20$, y hay 61.

Los errores se multiplican

$\times 1{,}4$ en un join y $\times 3$ en el siguiente: $\times 4$ arriba. Con 6 joins, a ciegas (Leis et al., 2015).

SECCIÓN 05

Cuando falla la independencia

05

Columnas que dependen una de otra

correlated.sql

```
Seq Scan on shipment  (cost=0.00..298.00 rows=820 width=67) (actual rows=0.00 loops=1)
  Filter: ((delivered_at IS NULL) AND (status = 'delivered'::text))

Seq Scan on shipment  (cost=0.00..328.00 rows=8 width=67) (actual rows=92.00 loops=1)
  Filter: ((origin_agency_id = 6) AND (dest_agency_id = 28))
```

Condición	estim.	real
`delivered` y sin fecha de entrega	820	0
`registered` y sin viaje	21	500
origen Cusco y región Cusco (`shipment_origin`)	31	610
de Cusco (6) a Puerto Maldonado (28)	8	92
`GROUP BY origin_agency_id, dest_agency_id`	784	120

En `agency` (29 filas) city y region también dependen, pero toda estimación se redondea a 1.

CREATE STATISTICS: tres tipos

- **dependencies**: grado de dependencia funcional $A \to B$. Para `A = x AND B = y`.

$$\mathrm{sel}(A{=}a \wedge B{=}b) = \mathrm{sel}(A{=}a) \cdot \big(f + (1-f) \cdot \mathrm{sel}(B{=}b)\big)$$

- **ndistinct**: $V(A_1 A_2, r)$. Para `GROUP BY` y `DISTINCT`.
- **mcv**: las combinaciones más comunes con su frecuencia.

Solo dentro de una tabla

No ayudan con la correlación entre columnas de `customer` y de `shipment`. Se crean cuando `EXPLAIN ANALYZE` muestra el error; cuestan tiempo de `ANALYZE`.

city → region: dependencies

correlated.sql

```sql
CREATE STATISTICS shipment_origin_dep (dependencies)
  ON origin_city, origin_region FROM shipment_origin;
ANALYZE shipment_origin;
```

pg_stats_ext y el plan

```
              dependencies
-----------------------------------------
 {"2 => 3": 1.000000, "3 => 2": 0.884583}

 Seq Scan on shipment_origin  (cost=0.00..308.00 rows=610 width=18) (actual rows=610.00 ...
```

- La ciudad determina la región en todas las filas ($f = 1$): la segunda condición no filtra. De 31 a 610, exacto.
- La región casi determina la ciudad (0,88): Puno tiene Puno y Juliaca; Áncash, Chimbote y Huaraz.

Origen y destino: el tipo correcto

- `(dependencies)`: grados 0,018 y 0,020. El origen no determina el destino. La estimación pasa de 8 a 11.
- `(mcv, ndistinct)`:

```
CREATE STATISTICS shipment_od
  (mcv, ndistinct)
  ON origin_agency_id, dest_agency_id
  FROM shipment;
ANALYZE shipment;
```

después

```
 n_distinct
---------------
 {"4, 5": 120}

Seq Scan on shipment  (cost=0.00..328.00 rows=92 ...
  Filter: ((origin_agency_id = 6) AND (dest_agency_id = 28))

HashAggregate  (cost=358.00..359.20 rows=120 width=16)
  Group Key: origin_agency_id, dest_agency_id
```

92 envíos y 120 grupos: exactos.

Estadísticas viejas

stale.sql: antes de ANALYZE

```
Hash Join  (cost=252.11..303.02 rows=41 width=19)
  ->  Seq Scan on customer c  (... rows=1800 ...)
  ->  Hash  (cost=251.60..251.60 rows=41 width=8)
        ->  Seq Scan on shipment_copy s  (... rows=41 ...)
              Filter: (status = 'delivered'::text)

Hash (actual rows=11115.00 loops=1)
  Buckets: 16384 (originally 1024)  Batches: 1 ...
```

después de ANALYZE

```
Hash Join  (cost=59.50..386.74 rows=11115 width=19)
  ->  Seq Scan on shipment_copy s  (... rows=11115 ...)
        Filter: (status = 'delivered'::text)
  ->  Hash  (cost=37.00..37.00 rows=1800 width=19)
        ->  Seq Scan on customer c  (... rows=1800 ...)
```

- Sin `pg_stats`: filas por el tamaño del archivo (8288) y 0,5 % para la igualdad: 41.
- La hash table se construye con el lado que cree pequeño, que tiene 11 115 filas, y tiene que crecer.
- Autovacuum analiza tras $50 + 0{,}1 \cdot n_r$ cambios. Después de una carga masiva: `ANALYZE` ya.

SECCIÓN 06

Elegir el plan

06

El costo de un plan

- Suma de los costos de sus operadores; cada uno depende del tamaño de sus entradas.
- Elegir el mejor algoritmo de cada operador por separado no alcanza:
  - un merge join puede costar más que un hash join,
  - pero deja la salida **ordenada**, y un `GROUP BY`, un `ORDER BY` u otro merge join se ahorran un sort.
- El problema difícil: el orden de los joins.

Principio de optimalidad

El mejor plan para S se arma con el mejor plan de S_1 y el mejor de $S - S_1$. Basta guardar un plan por subconjunto.

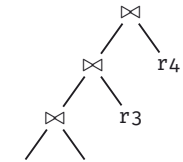
Programación dinámica (sec. 16.4)

findBestPlan

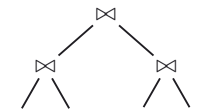
```
function findBestPlan(S) {
  if (bestPlan.has(key(S))) return bestPlan.get(key(S));
  let best = { plan: null, cost: Infinity };
  if (S.length === 1) best = bestAccessPath(S[0]);      // scan o índice
  else for (const S1 of nonEmptyProperSubsets(S)) {
    const p1 = findBestPlan(S1), p2 = findBestPlan(minus(S, S1));
    for (const alg of joinAlgorithms) {                 // nested loop, hash, merge...
      const cost = p1.cost + p2.cost + joinCost(alg, p1, p2);
      if (cost < best.cost) best = { plan: [alg, p1.plan, p2.plan], cost };
    }
  }
  bestPlan.set(key(S), best);
  return best;
}
```

- 2^n subconjuntos: 128 para 7 tablas, frente a 665 280 árboles.
- Cada S de k tablas: $2^k - 2$ divisiones. Total $O(3^n)$: $3^{10} \approx 59\,000$, frente a $1{,}76 \cdot 10^{10}$.

Left-deep trees y System R



left-deep



bushy

- **Left-deep**: la entrada de adentro de cada join es una tabla base.
- System R (Selinger et al., 1979) solo considera estos:
  - pipeline del lado de afuera e **index nested loop** adentro;
  - k divisiones por subconjunto: $O(n \cdot 2^n)$.
- $n!$ órdenes left-deep: 5040 para 7 tablas.

Órdenes interesantes ***(interesting orders)***

Para cada subconjunto se guarda el mejor plan para cada orden útil después (clave de un join, `GROUP BY`, `ORDER BY`), además del mejor sin orden.

Las pasadas, con los costos de PostgreSQL

Pasada 1: mejor acceso a cada tabla.

	costo	filas
σ_{city}(customer)	41,50	89
shipment	268,00	12 000
σ_{region}(agency)	1,36	1

Pasada 2: cada par.

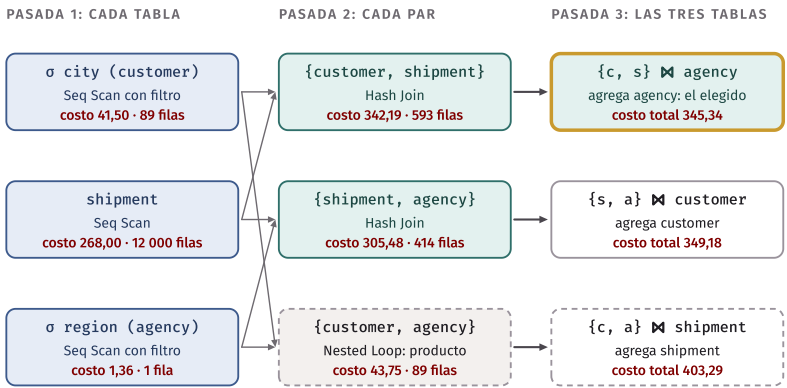
	costo	filas
{c, s} hash join	342,19	593
{s, a} hash join	305,48	414
{c, a} producto	43,75	89

Pasada 3: agregar la tercera.

	costo
({c, s}) ⋈ a	345,34
({s, a}) ⋈ c	349,18
({c, a}) ⋈ s	403,29

- Gana el plan del principio.
- Cerca: el costo lo domina leer `shipment`, una vez en los tres.
- El producto no es un desastre aquí: `agency` filtrada tiene 1 fila.

Las pasadas, de un vistazo



System R no genera el producto: hay alternativas con condición de join
se guarda solo el mejor plan de cada subconjunto

Heurísticas y subconsultas

- Selecciones y proyecciones lo antes posible.
- Evitar productos cartesianos.
- Unir primero lo que deja resultados chicos.
- Solo left-deep trees.
- **Decorrelar**: `EXISTS` e `IN` pasan a **semi join** ($r \ltimes s$); `NOT EXISTS`, a anti join.

equivalences.sql

```sql
SELECT c.customer_id, c.last_name
FROM customer c
WHERE c.city = 'Cusco'
  AND EXISTS (SELECT 1 FROM shipment s
              WHERE s.sender_id = c.customer_id
                AND s.status = 'returned');
```

plan

```
Hash Right Semi Join
  Hash Cond: (s.sender_id = c.customer_id)
  ->  Seq Scan on shipment s
        Filter: (status = 'returned'::text)
  ->  Hash
        ->  Seq Scan on customer c
              Filter: (city = 'Cusco'::text)
```

El optimizador de PostgreSQL

- Programación dinámica, también con **bushy trees**; guarda planes por orden útil (*pathkeys*).
- Desde `geqo_threshold` = 12 tablas en el `FROM`: **GEQO**, un optimizador genético, sin garantía de óptimo.
- `join_collapse_limit` = 8: cuántos `JOIN` explícitos reordena.

plans.sql: settings

```
            name             | setting
-----------------------------+---------
 cpu_index_tuple_cost        | 0.005
 cpu_operator_cost           | 0.0025
 cpu_tuple_cost              | 0.01
 default_statistics_target   | 100
 from_collapse_limit         | 8
 geqo_threshold              | 12
 join_collapse_limit         | 8
 random_page_cost            | 4
 seq_page_cost               | 1
```

Las unidades de cost=

- Una página leída en secuencia = 1. Al azar = 4. Una tupla = 0,01. Una comparación = 0,0025.
- Comprobación a mano:

Seq Scan on shipment: $148 \times 1 + 12\,000 \times 0{,}01$	268,00
con `WHERE status = 'x'`: $+12\,000 \times 0{,}0025$	298,00
con dos condiciones: $+12\,000 \times 0{,}0025 \times 2$	328,00
Seq Scan on customer: $19 + 1800 \times 0{,}01$	37,00

Error común

Leer `cost=` como milisegundos. Son unidades arbitrarias: solo sirven para comparar planes de .

SECCIÓN 07

Leer EXPLAIN ANALYZE

07



Anatomía de un nodo

EXPLAIN ANALYZE

```
Hash Join  (cost=42.61..342.19 rows=593 width=19) (actual time=0.394..1.481 rows=822.00 loops=1)
```

Estimado (`EXPLAIN`):

- `cost=inicio..total`: el hash join paga la hash table antes de la primera tupla.
- `rows`: filas que devuelve el nodo.

Real (`ANALYZE`, que ejecuta):

- `actual time`: ms a la primera y a la última fila.
- `rows`, `loops`: por iteración.
- PG 18: `Buffers` e `Index Searches` sin pedirlos.

Con UPDATE o DELETE: hace el cambio

```
BEGIN; EXPLAIN ANALYZE UPDATE ...; ROLLBACK;
```

Loops: lo que hay que multiplicar

plans.sql: nested_loop

```
Nested Loop  (cost=0.29..1129.05 rows=298 width=16) (actual time=0.011..0.884 rows=467.00 loops=1)
  ->  Seq Scan on shipment s  (cost=0.00..328.00 rows=77 width=4) (actual time=0.005..0.466 rows=117.00 loops=1)
        Filter: ((origin_agency_id = 1) AND (dest_agency_id = 4))
  ->  Index Scan using shipment_event_pkey on shipment_event e  (cost=0.29..10.36 rows=4 width=16)
                                                 (actual time=0.003..0.003 rows=3.99 loops=117)
        Index Cond: (shipment_id = s.shipment_id)
```

- El de adentro corre 117 veces, una por envío de afuera.
- $3{,}99 \times 117 = 467$ filas; $\approx 0{,}003 \times 117 \approx 0{,}35$ ms.
- El costo de adentro (10,36) también es por iteración: se multiplicó por 77.

Error común

Leer `rows` o `actual time` de adentro como el total: hay que multiplicar por `loops`.

La señal más útil

Estimado contra real

Una diferencia de 10 veces o más entre `rows` estimadas y reales en un nodo bajo casi siempre explica la mala decisión de arriba.

Síntoma	Causa probable	Arreglo
Tabla recién cargada	estadísticas viejas	`ANALYZE`
Dos columnas de una tabla	correlación	`CREATE STATISTICS`
0,5 % o un tercio exactos	expresión sin estadísticas	índice en la expresión
Error moderado, muchos valores	muestra chica	`SET STATISTICS`
Join entre tablas distintas	uniformidad, correlación	a veces ninguno; reescribir

Se busca el nodo más bajo donde empieza la diferencia: los de arriba la heredan.

SECCIÓN 08

Arreglar un plan

08

Una condición que el índice no puede usar

```
created_at::date = '2026-03-02'

Seq Scan on shipment (actual rows=73.00 loops=1)
  Filter: ((created_at)::date = '2026-03-02'::date)
  Rows Removed by Filter: 11927
  Buffers: shared hit=148
```

```
created_at >= '2026-03-02' AND <
'2026-03-03'

Bitmap Heap Scan on shipment (actual rows=73.00 loops=1)
  Heap Blocks: exact=55
  Buffers: shared hit=55 read=2
  ->  Bitmap Index Scan on shipment_created_at_idx ...
```

- El índice está ordenado por `created_at`, no por `created_at::date`.
- Estimación: 60 (el 0,5 %) contra 74 con el histograma. Hay 73.
- 148 páginas contra 57.
- ¿Índice sobre `created_at::date`?

```
psql

ERROR:  functions in index expression
        must be marked IMMUTABLE
```

El día de un `timestamptz` depende de la zona horaria de la sesión.

Un índice sobre una expresión

plans.sql

```sql
CREATE INDEX customer_lower_last_name_idx
  ON customer (lower(last_name));
ANALYZE customer;
```

lower_index_analyzed

```
Index Scan using customer_lower_last_name_idx
  on customer  (cost=0.28..8.29 rows=1 width=51)
  (actual time=0.017..0.017 rows=1.00 loops=1)
  Index Cond: (lower(last_name) = 'quispe flores'::text)
```

- Sin índice: 9 de 1800 (0,5 %). En `chasqui_big`: 1009; hay 3080.
- Con índice y `ANALYZE`: estima 1; hay 1.

Dos arreglos en uno

`ANALYZE` junta estadísticas de las expresiones indexadas.

Orden de joins: libre

plans.sql: join_order_free (costo 504, 2,1 ms)

```
Hash Join (actual rows=138.00 loops=1)
  ->  Hash Join (actual rows=138.00 loops=1)
        ->  Seq Scan on shipment s (actual rows=12000.00 loops=1)
        ->  Hash (actual rows=91.00 loops=1)
              ->  Hash Join (actual rows=91.00 loops=1)
                    ->  Seq Scan on trip t (actual rows=5914.00 loops=1)
                    ->  Hash (actual rows=1.00 loops=1)
                          ->  Seq Scan on vehicle v (actual rows=1.00 loops=1)
                                Filter: (plate = 'K2U-314'::text)
  ->  Hash (actual rows=1800.00 loops=1)
        ->  Seq Scan on customer c (actual rows=1800.00 loops=1)
```

- Empieza por `trip` $\bowtie$ `vehicle`: el filtro por placa deja 91 viajes (estimaba $5914/24 = 246$).
- Los 12 000 envíos se reducen a 138 antes de unir `customer`.

Orden de joins: forzado

SET join_collapse_limit = 1 (costo 606, 4,8 ms)

```
Hash Join (actual rows=138.00 loops=1)
  ->  Hash Join (actual rows=11500.00 loops=1)
        ->  Hash Join (actual rows=12000.00 loops=1)
              ->  Seq Scan on shipment s (actual rows=12000.00 loops=1)
              ->  Hash (actual rows=1800.00 loops=1)
                    ->  Seq Scan on customer c (actual rows=1800.00 loops=1)
        ->  Hash (actual rows=5914.00 loops=1)
              ->  Seq Scan on trip t (actual rows=5914.00 loops=1)
  ->  Hash (actual rows=1.00 loops=1)
        ->  Seq Scan on vehicle v (actual rows=1.00 loops=1)
              Filter: (plate = 'K2U-314'::text)
```

- Orden escrito: $((c \bowtie s) \bowtie t) \bowtie v$, con intermedios de 12 000 y 11 500.
- Forzar sirve para diagnosticar. Para arreglar, darle al optimizador lo que le falta.

Opcional: vistas materializadas (sec. 16.5)

- Guardan el resultado de una consulta como una tabla.
- Hay que **mantenerlas**: el libro describe el mantenimiento incremental (calcular solo el cambio del resultado) y cómo el optimizador puede usarlas sin que la consulta las nombre.
- PostgreSQL: `CREATE MATERIALIZED VIEW` y `REFRESH MATERIALIZED VIEW`, que recalcula todo. Sin mantenimiento incremental.
- Útiles para reportes que aceptan datos de hace un rato.

SECCIÓN 09

Cierre

09

El código de la semana

- `stats.sql`, `equivalences.sql`, `correlated.sql`, `stale.sql`, `plans.sql`: sobre `chasqui_w12`.
- `selectivity.mjs`: la estimación de un predicado desde `pg_stats`.
- `examples.test.mjs` corre cada `.sql` entero y revisa las cifras.
- `make` hace `VACUUM (ANALYZE)`: estadísticas siempre iguales.

code/week-12

```
$ make test
chasqui-load chasqui_w12
Base de datos chasqui_w12 lista.
...
✓ stats.sql: tamaños y estadísticas del catálogo
✓ equivalences.sql: el optimizador baja los filtros y arma joins
✓ correlated.sql: independencia y CREATE STATISTICS
✓ stale.sql: sin ANALYZE, 41 filas estimadas para 11 115
✓ plans.sql: orden de joins, loops y reescrituras
...
ℹ tests 29
ℹ pass 29
ℹ fail 0
```

Resumen

- El optimizador genera planes **equivalentes**, los costea con **estimaciones** y elige.
- Reglas: cascada de σ, σ sobre producto = join, conmutar y asociar joins, bajar σ y Π. Con outer joins, cuidado con el lado de los nulos.
- Catálogo: n_r, b_r, $V(A, r)$, histogramas equi-depth. PostgreSQL: `pg_stats` (MCV, `histogram_bounds` sin los MCV, `n_distinct`), de una muestra.
- Selectividad: $1/V$ o la frecuencia del MCV; histograma con interpolación; $\wedge$ producto, $\vee$, $\neg$ menos nulos. Join por clave foránea: n_s; en general $n_r n_s / \max(V)$.
- La independencia falla con columnas relacionadas: `CREATE STATISTICS` (`dependencies`, `mcv`, `ndistinct`). Estadísticas viejas: `ANALYZE`.
- Programación dinámica: $O(3^n)$; left-deep $O(n2^n)$; órdenes interesantes. PostgreSQL: GEQO desde 12 tablas.
- `EXPLAIN ANALYZE`: estimado contra real, por `loops`. Arreglos: estadísticas, reescribir, índices sobre expresiones.

Para la próxima clase

- Laboratorio: leer y mejorar planes en `chasqui_big` (estadísticas, una copia sin `ANALYZE`, `CREATE STATISTICS`, reescrituras, orden de joins).
- Se entrega P3. Se publica **P4** (`quipu`: operadores con iteradores y comparación con `EXPLAIN`); se entrega en la semana 14.
- Leer Silberschatz, secs. 17.1 a 17.6 (transacciones) y la nota 11 de CS186.
- Correr `make test` en `code/week-12` y leer los apuntes de la semana.

Pregunta

El lado de afuera de un `Nested Loop` estima `rows=20` y devuelve 40 000. ¿Qué plan habría elegido el optimizador con la estimación correcta? ¿Por dónde empezarían a buscar la causa?