

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Conjuntos, NULL, agregación y subconsultas

Modelo relacional y SQL

Gerson Garrido

Semana 3

Hoy

Sesión A: SQL II

1. `UNION`, `INTERSECT`, `EXCEPT` y `ALL`
2. `NULL` y la lógica de tres valores
3. Funciones de agregación
4. `GROUP BY` y `HAVING`

Sesión B: SQL III

5. Subconsultas: `IN`, `SOME`, `ALL`, `EXISTS`
6. Subconsultas en `FROM`, `WITH` y escalares
7. `INSERT`, `UPDATE` y `DELETE`

Lectura

Silberschatz, secs. 3.5–3.7 (sesión A) y 3.8–3.9 (sesión B). Berkeley CS186, notas 1 y 2. PostgreSQL: *Aggregate Functions*, *Combining Queries*, *Subquery Expressions* y *WITH Queries*.

SECCIÓN 01

Operaciones de conjuntos

01



Unión, intersección y diferencia

Relaciones compatibles

Misma aridad y dominios compatibles, columna por columna. Solo entre ellas se definen $r \cup s$, $r \cap s$ y $r - s$.

- SQL: `UNION`, `INTERSECT`, `EXCEPT`.
- Entre dos consultas completas; comparan filas completas.

sets-null-aggregates.sql

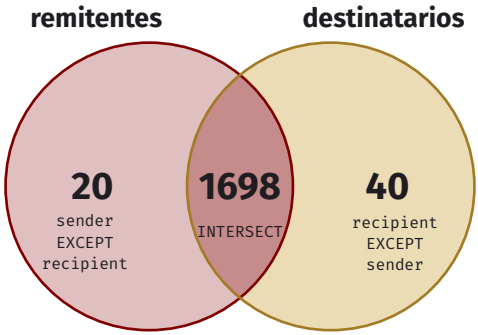
```sql
SELECT count(*) AS ciudades
FROM (SELECT city FROM agency
      UNION
      SELECT city FROM customer) AS t;
```

salida

```
ciudades
----------
        26
```

29 agencias, 1800 clientes, 26 ciudades: `UNION` elimina duplicados.

`EXCEPT` no es simétrico



$20 + 1698 + 40 = 1758$ clientes aparecen en algún envío; los otros 42, en ninguno.

Y `city FROM customer EXCEPT city FROM agency` da 0 filas: todo cliente vive donde hay agencia.

Multiconjuntos: la versión ALL

Si una fila aparece m veces a la izquierda y n a la derecha:

Operación	Copias
`UNION ALL`	$m + n$
`INTERSECT ALL`	$\min(m, n)$
`EXCEPT ALL`	$\max(m - n, 0)$
sin `ALL`	0 o 1

`UNION ALL` de ciudades: 1829 filas ($29 + 1800$). A la derecha: Lima tiene 3 agencias y Arequipa 2; quedan las repeticiones.

sets-null-aggregates.sql

```sql
SELECT city FROM agency
EXCEPT ALL
SELECT DISTINCT city FROM agency
ORDER BY city;
```

salida

```
city
----------
 Arequipa
 Lima
 Lima
```

Dos detalles de PostgreSQL

- `INTERSECT` tiene más precedencia que `UNION` y `EXCEPT`:
  `a UNION b INTERSECT c` es `a UNION (b INTERSECT c)`. Ante la duda, paréntesis.
- Un `ORDER BY` o `LIMIT` al final se aplica al resultado completo.
- Si las partes no tienen la misma forma:
  `ERROR:  each UNION query must have the same number of columns`

Costo

`UNION ALL` no busca duplicados (no ordena ni arma una hash table). Si las repeticiones importan, o no puede haberlas, es la opción barata.

SECCIÓN 02

NULL y la lógica de tres valores

02

`NULL`: falta un valor

Columna que admite `NULL`	Filas `NULL`
`customer.phone`	259
`shipment.declared_value`	4864
`shipment.trip_id`	500
`shipment.delivered_at`	885
`trip.arrival_at`	14

- No es 0 ni ”: `declared_value = 0` declaró que no vale nada.
- La pregunta de siempre: ¿qué debe responder la consulta para una fila cuyo valor no se conoce?
- En las salidas, `\pset null '∅'`.

Aritmética y comparaciones con `NULL`

sets-null-aggregates.sql

```sql
SELECT 10 + NULL AS suma, NULL = NULL AS igual,
       NULL <> NULL AS distinto, NULL IS NULL AS es_null;
```

salida

```
suma | igual | distinto | es_null
------+-------+----------+---------
   ∅ | ∅     | ∅        | t
```

- Aritmética con `NULL`: `NULL`.
- Comparación con `NULL`, incluso `NULL = NULL`: **unknown** (en PostgreSQL, el `NULL` de `boolean`).
- La única pregunta con respuesta segura: `IS NULL`.

Lógica de tres valores

AND	**V**	**F**	**U**
V	V	F	U
F	F	F	F
U	U	F	U

OR	**V**	**F**	**U**
V	V	V	V
F	V	F	U
U	V	U	U

	NOT
V	F
F	V
U	U

Regla para recordarlas

U es “quizás V, quizás F”. El resultado es U solo si de verdad depende de cuál sea. `false AND` U es F. `true OR` U es V. `NOT` U es U.

La tabla completa, calculada por PostgreSQL, está en `sets-null-aggregates.sql` (consulta 2.2).

`WHERE` se queda solo con lo verdadero

sets-null-aggregates.sql

```sql
SELECT count(*) AS total,
       count(*) FILTER (WHERE declared_value > 500) AS mayor,
       count(*) FILTER (WHERE declared_value <= 500) AS menor_igual,
       count(*) FILTER (WHERE declared_value > 500
                        OR declared_value <= 500) AS cualquiera,
       count(*) FILTER (WHERE declared_value IS NULL) AS sin_valor
FROM shipment;
```

salida

```
total | mayor | menor_igual | cualquiera | sin_valor
-------+-------+-------------+------------+-----------
 12000 |  6039 |        1097 |       7136 |      4864
```

$x > 500$ `OR` $x \leq 500$ no es una tautología: con x `NULL`, U `OR` U es U. $7136 = 12\,000 - 4864$.

`NOT` no rescata a los `NULL`

sets-null-aggregates.sql

```sql
SELECT count(*) AS no_mayor
FROM shipment
WHERE NOT (declared_value > 500);
```

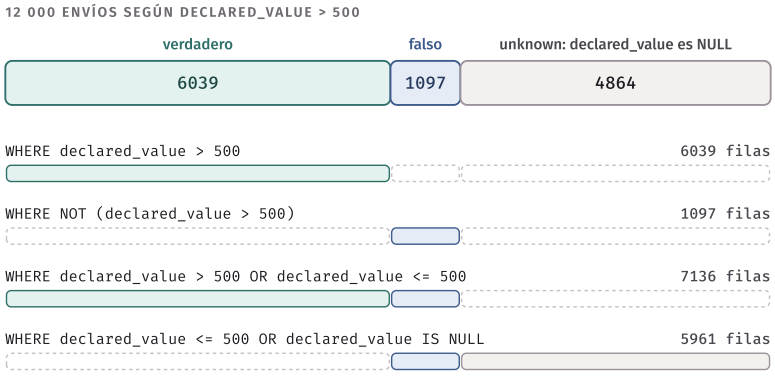
salida

```
no_mayor
----------
      1097
```

- No son $12\,000 - 6039 = 5961$: `NOT` U es U.
- p y `NOT` p parten las filas en tres: p, `NOT` p y "p es U".
- Para incluir los `NULL` hay que pedirlo:

```
... OR declared_value IS NULL
```

Tres tramos, no dos



ningún predicado hecho de comparaciones, AND, OR y NOT conserva la parte gris; hay que pedirla con IS NULL

Error común: `= NULL`

mal

```sql
SELECT count(*) AS igual_null
FROM customer
WHERE phone = NULL;
```

salida

```
igual_null
------------
          0
```

bien

```sql
SELECT count(*) AS is_null
FROM customer
WHERE phone IS NULL;
```

salida

```
is_null
---------
      259
```

`phone = NULL` es unknown para *todas* las filas: cero filas, sin error ni advertencia.

Predicados que nunca dan unknown

- `IS NULL`, `IS NOT NULL`.
- *p* `IS TRUE`, `IS FALSE`, `IS UNKNOWN`: `(declared_value > 500) IS UNKNOWN` cuenta 4864 envíos.
- `a IS DISTINCT FROM b`: como `<>`, pero dos `NULL` son iguales.
- `coalesce(a, b, ...)`: el primero que no es `NULL`.

  `coalesce(phone, 'sin teléfono')`

salida

```
d1 | d2 | d3
---+----+----
 f  | t  | t
```

```
d1: NULL IS DISTINCT FROM NULL
d2: NULL IS DISTINCT FROM 5
d3: 5 IS NOT DISTINCT FROM 5
```

Donde dos `NULL` sí son iguales

Contexto	¿`NULL` igual a `NULL`?
`=`, `IN`, `WHERE`, `ON`, `HAVING`	no: unknown
`IS NOT DISTINCT FROM`	sí
`DISTINCT`, `UNION`, `INTERSECT`, `EXCEPT`	sí
`GROUP BY`	sí: un solo grupo

sets-null-aggregates.sql

```sql
SELECT NULL AS x
UNION
SELECT NULL;
```

salida

```
x
---
 ∅
```

En `WHERE`: “¿estos valores son iguales?” (no se sabe). Al eliminar duplicados: “¿estas filas son copias?” (sí).

SECCIÓN 03

Funciones de agregación

03



Cinco funciones, dos reglas

Reciben un multiconjunto de valores y devuelven uno:

- `count`
- `sum`, `avg` (numéricos)
- `min`, `max` (cualquier tipo con orden)

`count(DISTINCT city)`: elimina duplicados antes.

Regla 1

Las cinco ignoran los `NULL`, salvo `count(*)`, que cuenta filas.

Regla 2

Sobre una colección vacía, `count` da 0 y las demás dan `NULL`. Una colección de puros `NULL` cuenta como vacía.

Tres maneras de contar

sets-null-aggregates.sql

```sql
SELECT count(*) AS filas,
       count(phone) AS con_telefono,
       count(DISTINCT city) AS ciudades
FROM customer;
```

salida

```
filas | con_telefono | ciudades
------+--------------+----------
 1800 |         1541 |       26
```

- `count(*)`: filas.
- `count(phone)`: valores no `NULL`. `count(DISTINCT city)`: valores distintos no `NULL`.

Error común: contar filas con `count(columna)`

Si la columna admite `NULL`, la cuenta baja sin aviso.

avg promedia solo los valores que hay

sets-null-aggregates.sql

```sql
SELECT round(avg(declared_value), 2) AS promedio,
       round(sum(declared_value) / count(*), 2) AS sum_entre_filas,
       round(sum(declared_value) / count(declared_value), 2) AS sum_entre_valores
FROM shipment;
```

salida

```
promedio | sum_entre_filas | sum_entre_valores
----------+-----------------+-------------------
  1512.33 |          899.33 |           1512.33
```

- `avg(x)` = `sum(x) / count(x)`, no `sum(x) / count(*)`.
- Dividir entre `count(*)` supone que los envíos sin valor declarado valen 0.

Cero filas: ¿una fila o ninguna?

sin GROUP BY

```sql
SELECT count(*) AS n, sum(price) AS suma,
       avg(price) AS promedio, max(price) AS
     maximo
FROM shipment
WHERE price < 0;
```

salida

```
n | suma | promedio | maximo
---+------+----------+--------
 0 |    ∅ |        ∅ |      ∅
(1 row)
```

con GROUP BY

```sql
SELECT status, count(*) AS n
FROM shipment
WHERE price < 0
GROUP BY status;
```

salida

```
status | n
--------+---
(0 rows)
```

Sin `GROUP BY` (ni `HAVING`), todo es un grupo, aunque vacío: siempre una fila. Con `GROUP BY`, cero filas son cero grupos.

Filas sí, valores no

sets-null-aggregates.sql

```sql
SELECT count(*) AS filas, count(declared_value) AS valores,
       sum(declared_value) AS suma,
       coalesce(sum(declared_value), 0) AS suma_o_cero
FROM shipment
WHERE declared_value IS NULL;
```

salida

```
filas | valores | suma | suma_o_cero
-------+---------+------+-------------
  4864 |       0 |    ∅ |           0
```

- 4864 filas, ningún valor: `sum` no tiene qué sumar y da `NULL`, no 0.
- Si el informe necesita un 0: `coalesce(sum(...), 0)`.

SECCIÓN 04

GROUP BY y HAVING

04

GROUP BY: una fila por grupo

sets-null-aggregates.sql

```sql
SELECT status, count(*) AS n
FROM shipment
GROUP BY status
ORDER BY n DESC;
```

salida

```
status    |   n
------------+-------
 delivered  | 11115
 registered |   500
 returned   |   319
 in_transit |    66
```

- Mismos valores en las columnas de agrupamiento $\Rightarrow$ mismo grupo.
- Los agregados se calculan por grupo.
- En el álgebra: ${}_{G}\gamma_{F}(r)$.
  Aquí: ${}_{status}\gamma_{count(*)}(shipment)$.
- Sin GROUP BY, G es vacío: un solo grupo.

La regla del SELECT

Cada expresión del SELECT debe tener un solo valor por grupo.

errors.sql

```
1  SELECT region, name, count(*) FROM agency GROUP BY region;
```

salida

```
ERROR:  column "agency.name" must appear in the GROUP BY clause or be used in an
     aggregate function
LINE 1: SELECT region, name, count(*)
                       ^
```

La región Lima tiene tres agencias con tres nombres: ¿cuál se mostraría?

Cada grupo da una fila

```
SELECT region, count(*) FROM agency
WHERE region IN ('Lima', 'Arequipa', 'Cusco', 'Puno') GROUP BY region;
```

FILAS DE AGENCY, AGRUPADAS POR REGION

agency_id	name	region
1	Lima - La Victoria	Lima
2	Lima - Los Olivos	Lima
3	Lima - Ate	Lima
4	Arequipa - Centro	Arequipa
5	Arequipa - Cerro Colorado	Arequipa
6	Cusco - Wanchaq	Cusco
18	Puno - Centro	Puno
19	Juliaca - Centro	Puno

RESULTADO

region	count
Lima	3
Arequipa	2
Cusco	1
Puno	2

¿cuál de los tres name iría en la fila de Lima?

por eso name no puede ir en el SELECT sin estar en el GROUP BY

Agrupar por la clave primaria

sets-null-aggregates.sql

```sql
SELECT a.agency_id, a.name, sum(s.price) AS
       ingresos
FROM agency a, shipment s
WHERE s.origin_agency_id = a.agency_id
GROUP BY a.agency_id
ORDER BY ingresos DESC LIMIT 5;
```

salida recortada

```
 agency_id |        name         | ingresos
-----------+---------------------+-----------
         1 | Lima - La Victoria | 116713.45
         3 | Lima - Ate         |  71252.76
...
```

En PostgreSQL

Agrupar por la clave primaria permite usar las demás columnas de la tabla: `name` depende de `agency_id`.

Para ser portable, agreguen `a.name` al `GROUP BY`.

Los `NULL` forman un grupo

sets-null-aggregates.sql

```sql
SELECT trip_id, count(*) AS n
FROM shipment
GROUP BY trip_id
ORDER BY n DESC, trip_id
LIMIT 3;
```

salida

```
trip_id |  n
---------+-----
      ∅ | 500
   2160 |  10
    307 |   9
```

Los 500 envíos sin viaje van juntos. Pero:

salida

```
valores_distintos | count_distinct
------------------+----------------
             4877 |           4876
```

`SELECT DISTINCT trip_id` cuenta el `NULL` como un valor más; `count(DISTINCT trip_id)`, como todo agregado, lo ignora.

HAVING: filtrar grupos

sets-null-aggregates.sql

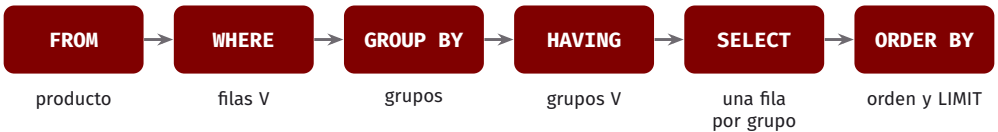
```sql
SELECT count(*) AS rutas
FROM (SELECT route_id
      FROM trip
      GROUP BY route_id
      HAVING count(*) > 50) AS t;
```

salida

```
rutas
-------
    58
```

- WHERE filtra filas, antes de agrupar.
- HAVING filtra grupos, después.
- En HAVING: columnas de agrupamiento y agregados (la regla del SELECT).
- Las rutas tienen 25, 26, 75 o 76 viajes: 58 de 120 pasan.

El orden lógico de una consulta



- Un alias del `SELECT` sirve en `ORDER BY`, no en `WHERE` ni en `HAVING`.
- Un agregado no puede ir en `WHERE`: todavía no hay grupos.
- El motor puede ejecutar en otro orden, pero el resultado debe ser este.

Error común: el agregado en `WHERE`

errors.sql

```sql
SELECT route_id
FROM trip
WHERE count(*) > 50
GROUP BY route_id;
```

salida

```
ERROR:  aggregate functions are not allowed in WHERE
LINE 3: WHERE count(*) > 50
              ^
```

- Va en `HAVING count(*) > 50`.
- El error silencioso es otro: olvidar el filtro de filas en `WHERE` y contar filas que no correspondían.

`WHERE` o `HAVING`: dos preguntas distintas

todos los viajes

```sql
SELECT route_id, count(*) AS viajes,
       count(arrival_at) AS llegados
FROM trip
GROUP BY route_id
HAVING count(*) > 75;
```

salida recortada

```
 route_id | viajes | llegados
----------+--------+----------
        7 |     76 |       75
       12 |     76 |       75
...
(10 rows)
```

solo los que llegaron

```sql
SELECT route_id, count(*) AS viajes
FROM trip
WHERE arrival_at IS NOT NULL
GROUP BY route_id
HAVING count(*) > 75;
```

salida

```
route_id | viajes
----------+--------
(0 rows)
```

Agregación con `FILTER`

sets-null-aggregates.sql

```sql
SELECT a.region, count(*) AS envios,
       count(*) FILTER (WHERE s.status = 'delivered') AS entregados,
       count(*) FILTER (WHERE s.status = 'returned') AS devueltos
FROM agency a, shipment s
WHERE s.origin_agency_id = a.agency_id
GROUP BY a.region ORDER BY envios DESC LIMIT 3;
```

salida

```
region   | envios | entregados | devueltos
----------+--------+------------+-----------
 Lima     |   3965 |       3706 |       107
 Áncash   |    873 |        816 |        21
 Arequipa |    842 |        778 |        18
```

SECCIÓN 05

Subconsultas anidadas

05



Pertenencia: `IN`

subqueries.sql

```sql
SELECT count(*) AS remitentes
FROM customer
WHERE customer_id IN (SELECT sender_id
                      FROM shipment);
```

salida

```
remitentes
------------
       1718
```

- La subconsulta repite `sender_id`; no importa: `IN` pregunta pertenencia.
- Un join con `shipment` repetiría al cliente una vez por envío.
- También: `(a, b) IN (SELECT ...)`.

`NOT IN` es una cadena de `AND`

$$x \texttt{ IN } (v_1, \ldots, v_n) \equiv x = v_1 \texttt{ OR } \cdots \texttt{ OR } x = v_n$$

$$x \texttt{ NOT IN } (v_1, \ldots, v_n) \equiv x \neq v_1 \texttt{ AND } \cdots \texttt{ AND } x \neq v_n$$

Si algún v_i es `NULL`, $x \neq v_i$ es U, y el `AND` queda en F o en U:

Expresión	Valor (de PostgreSQL)
`3 NOT IN (1, 2)`	`t`
`3 NOT IN (1, 2, NULL)`	∅
`1 NOT IN (1, 2, NULL)`	`f`
`3 IN (1, 2, NULL)`	∅

Un `NOT IN` cuya lista contiene un `NULL` nunca es verdadero.

La trampa en Chasqui: viajes sin envíos

subqueries.sql

```sql
SELECT count(*) AS viajes_vacios
FROM trip
WHERE trip_id NOT IN (SELECT trip_id FROM shipment);
```

salida

```
viajes_vacios
---------------
             0
```

- Hay 1038 viajes vacíos. Pero los 500 envíos registrados tienen `trip_id` `NULL`.
- Cero filas, sin error ni advertencia.
- Hoy funciona si no hay `NULL`; el día que alguien inserta uno, deja de funcionar.

Tres versiones correctas: 1038 viajes

subqueries.sql

```sql
-- 1. Sacar los NULL de la subconsulta
SELECT count(*) FROM trip
WHERE trip_id NOT IN (SELECT trip_id FROM shipment WHERE trip_id IS NOT NULL);

-- 2. NOT EXISTS (la recomendada)
SELECT count(*) FROM trip t
WHERE NOT EXISTS (SELECT 1 FROM shipment s WHERE s.trip_id = t.trip_id);

-- 3. Diferencia de conjuntos (ahí NULL = NULL)
SELECT count(*) FROM (SELECT trip_id FROM trip
                      EXCEPT
                      SELECT trip_id FROM shipment) AS t;
```

Las pruebas de `subqueries.test.mjs` comprueban que las tres devuelven las mismas 1038 filas.

Comparación con conjuntos: SOME y ALL

Cuantificadores

$x\ \theta$ SOME (S): $x\ \theta\ v$ para algún $v \in S$.

$x\ \theta$ ALL (S): $x\ \theta\ v$ para todo $v \in S$.

$\theta \in \{=, \neq, <, \leq, >, \geq\}$; ANY es sinónimo de SOME.

- `= SOME` es `IN`.
- `<> ALL` es `NOT IN`, con la misma trampa.
- `<> SOME` no es `NOT IN`: basta un valor distinto.

Envíos más caros que todos los de Tumbes - Centro (agencia 10):

```
WHERE price > ALL (SELECT price FROM shipment WHERE origin_agency_id = 10)
```

da 409; con `> SOME`, 11 902 de 12 000.

ALL y SOME con NULL y con vacío

5 > ALL / SOME (...)

```
 all_sin_null | all_unknown | all_falso | some_unknown | some_verdad
--------------+-------------+-----------+--------------+-------------
 t            | ∅           | f         | ∅            | t
```

`(1, 2)` `(1, NULL)` `(9, NULL)` `SOME (9, NULL)` `SOME (1, NULL)`

conjunto vacío

```
 all_vacio | some_vacio | max_vacio | null_not_in_vacio
-----------+------------+-----------+-------------------
 t         | f          | ∅         | t
```

- `ALL` vacío es V (“para todo”); `SOME` vacío es F (“existe”).
- `> ALL` y `> (SELECT max(...))` difieren si la subconsulta es vacía: `max` de nada es `NULL`.

`EXISTS` y las subconsultas correlacionadas

subqueries.sql

```sql
SELECT count(*) AS nunca_recibieron
FROM customer c
WHERE NOT EXISTS (
  SELECT 1
  FROM shipment s
  WHERE s.recipient_id = c.customer_id);
```

salida

```
nunca_recibieron
------------------
              62
```

- `EXISTS`: ¿hay al menos una fila? Nunca es U.
- Correlacionada: usa `c.customer_id`, de afuera.
- Conceptualmente, se evalúa una vez por cada cliente.
- El motor la convierte en un join (semanas 11 y 12).

“Para todo” con doble negación

subqueries.sql

```sql
SELECT count(*) AS a_todas_lima
FROM customer c
WHERE NOT EXISTS (
  SELECT 1
  FROM agency a
  WHERE a.region = 'Lima'
    AND NOT EXISTS (
      SELECT 1
      FROM shipment s
      WHERE s.sender_id = c.customer_id
        AND s.dest_agency_id = a.agency_id))
    ;
```

$$\forall a\, P(a) \iff \neg \exists a\, \neg P(a)$$

- Adentro: ¿c envió a a?
- Medio: una agencia de Lima sin envío: un contraejemplo.
- Afuera: clientes sin contraejemplo.

salida

```
a_todas_lima
--------------
           175
```

Es la división del álgebra.

El divisor vacío

Contando (`HAVING count(DISTINCT a.agency_id) = (SELECT count(*) ...)`) también sale 175.
Pero con una región sin agencias (`'Lima Metropolitana'`):

Versión	Resultado
Doble `NOT EXISTS` recorriendo `customer`	1800
Doble `NOT EXISTS` recorriendo los remitentes (la división)	1718
Contando, con `GROUP BY` y `HAVING`	0

- Sin contraejemplos, "para todo" es V para todo lo que recorre la consulta de afuera.
- $r \div s$ toma sus tuplas de r: con s vacío da $\Pi_{sender_id}(shipment)$.

En PostgreSQL, el predicado `UNIQUE (subconsulta)` del libro es estándar, pero no está implementado.

SECCIÓN 06

FROM, WITH y subconsultas escalares

06



Subconsultas en FROM

subqueries.sql

```sql
SELECT route_id, round(avg_price, 2) AS
    avg_price
FROM (SELECT t.route_id, avg(s.price) AS
    avg_price
      FROM shipment s, trip t
      WHERE s.trip_id = t.trip_id
      GROUP BY t.route_id) AS route_price
WHERE avg_price > 80
ORDER BY avg_price DESC;
```

salida recortada

```
 route_id | avg_price
----------+-----------
       40 |     88.47
       38 |     88.14
       31 |     87.35
...
(10 rows)
```

En PostgreSQL, desde la versión 16 el alias (`AS route_price`) es opcional. El estándar lo exige: pónganlo siempre.

`WITH`: nombrar pasos intermedios

subqueries.sql

```sql
WITH route_load AS (
  SELECT t.route_id,
         sum(s.weight_kg) AS total_kg
  FROM shipment s, trip t
  WHERE s.trip_id = t.trip_id
  GROUP BY t.route_id
)
SELECT count(*) AS rutas_sobre_promedio,
       (SELECT count(*) FROM route_load) AS
     rutas
FROM route_load
WHERE total_kg >
      (SELECT avg(total_kg) FROM route_load)
     ;
```

salida

```
rutas_sobre_promedio | rutas
---------------------+-------
                  58 |   120
```

- Una CTE *(common table expression)*: la asignación ← del álgebra.
- Se define una vez y se usa tres.
- Varias CTE separadas por comas; cada una ve las anteriores.

Subconsultas escalares

subqueries.sql

```sql
SELECT s.shipment_id, s.status,
       (SELECT count(*)
        FROM shipment_event e
        WHERE e.shipment_id = s.
     shipment_id)
         AS eventos
FROM shipment s
WHERE s.shipment_id IN (1, 36, 46, 63)
ORDER BY s.shipment_id;
```

salida

```
shipment_id |   status   | eventos
------------+------------+---------
          1 | delivered  |       4
         36 | registered |       1
         46 | in_transit |       2
         63 | returned   |       4
```

Una fila, una columna: va donde va un valor.

- Sin filas, vale `NULL`: `(SELECT name FROM agency WHERE agency_id = 99)` es $\emptyset$.
- Con más de una fila, error al ejecutar:

```
ERROR:  more than one row returned by a subquery used as an expression
```

LATERAL (opcional)

subqueries.sql

```sql
SELECT s.shipment_id, s.status,
       last.seq, last.kind
FROM shipment s
CROSS JOIN LATERAL (
  SELECT e.seq, e.kind
  FROM shipment_event e
  WHERE e.shipment_id = s.
      shipment_id
  ORDER BY e.seq DESC
  LIMIT 1) AS last
WHERE s.shipment_id IN (1, 36, 46,
      63)
ORDER BY s.shipment_id;
```

salida

```
shipment_id |   status   | seq |   kind
------------+------------+-----+-----------
          1 | delivered  |   4 | delivered
         36 | registered |   1 | received
         46 | in_transit |   2 | departed
         63 | returned   |   4 | returned
```

Una subconsulta en `FROM` que usa columnas de las tablas a su izquierda. A diferencia de una escalar, devuelve varias columnas.

SECCIÓN 07

Modificación de la base de datos

07

Una receta: `BEGIN` … `ROLLBACK`

modification.sql

```sql
BEGIN;

INSERT INTO customer (customer_id, dni,
    first_name,
                      last_name, phone, city)
VALUES (1801, '70000001', 'Rosa',
        'Quispe Mamani', NULL, 'Cusco');

-- ... más cambios; aquí se ven ...

ROLLBACK;   -- y aquí se deshacen
```

salida

```
BEGIN
INSERT 0 1
ROLLBACK
```

- Las columnas no nombradas reciben su valor por defecto (`NULL`).
- Las restricciones se revisan fila por fila.
- Qué es una transacción: semana 13.

INSERT ... SELECT

modification.sql

```sql
CREATE TABLE frequent_sender (
  customer_id integer PRIMARY KEY
              REFERENCES customer,
  shipments   integer NOT NULL,
  total_paid  numeric(10, 2)
);

INSERT INTO frequent_sender (customer_id,
    shipments)
SELECT sender_id, count(*)
FROM shipment
GROUP BY sender_id
HAVING count(*) >= 15;
```

salida

```
CREATE TABLE
INSERT 0 100
```

Primero el SELECT completo

Después se inserta. Por eso `INSERT INTO t SELECT * FROM t` duplica las filas una vez, y no se queda insertando para siempre.

UPDATE con CASE

modification.sql

```sql
UPDATE route
SET base_fare = CASE
      WHEN distance_km > 500
        THEN round(base_fare * 1.05, 2)
      ELSE round(base_fare * 1.03, 2)
    END;
```

salida

```
UPDATE 120
```

`sum(base_fare)`: de 2814.00 a 2920.34.

Con dos UPDATE, el orden importa

Si la regla mira la columna que cambia (“5 % hasta S/ 30, 3 % sobre S/ 30”): S/ 29.00 $\rightarrow$ 30.45 $\rightarrow$ 31.36.

Un CASE aplica una regla y actualiza cada fila una vez.

UPDATE con subconsultas

En el `WHERE`:

modification.sql

```sql
UPDATE route
SET base_fare = round(base_fare *
    1.10, 2)
WHERE route_id IN (SELECT route_id
                   FROM trip
                   GROUP BY route_id
                   HAVING count(*) >
       75);
```

salida

```
UPDATE 10
```

En el `SET`, escalar y correlacionada:

modification.sql

```sql
UPDATE frequent_sender f
SET total_paid = (SELECT sum(s.price
    )
                  FROM shipment s
                  WHERE s.sender_id
                        = f.
    customer_id);
```

salida

```
UPDATE 100
```

Sin envíos quedaría `NULL`: `coalesce(..., 0)`.

`DELETE`: la integridad referencial protege

errors.sql

```sql
DELETE FROM shipment
WHERE price < (SELECT avg(price) FROM shipment);
```

salida

```
ERROR:  update or delete on table "shipment" violates foreign key constraint "
     shipment_event_shipment_id_fkey" on table "shipment_event"
DETAIL:  Key (shipment_id)=(2) is still referenced from table "shipment_event".
```

- Primero los eventos de esos envíos, después los envíos.
- Tras un error dentro de `BEGIN`, todo se rechaza hasta el `ROLLBACK`.

DELETE: la condición se evalúa antes

modification.sql

```sql
DELETE FROM shipment_event
WHERE shipment_id IN (
  SELECT shipment_id FROM shipment
  WHERE price < (SELECT avg(price)
                 FROM shipment));

DELETE FROM shipment
WHERE price < (SELECT avg(price)
               FROM shipment);
```

salida

```
DELETE 29090
DELETE 7532
```

Antes: promedio S/ 60.50, 7532 envíos debajo.

después

```
 quedan | promedio_nuevo | bajo_promedio_nuevo
--------+----------------+---------------------
   4468 |          97.60 |                2508
```

- El promedio no se recalcula al borrar.
- Igual en `UPDATE` e `INSERT ... SELECT`.

`RETURNING` (de PostgreSQL)

modification.sql

```sql
UPDATE shipment
SET status = 'delivered', delivered_at = now()
WHERE shipment_id = 12001
RETURNING shipment_id, status, delivered_at IS NOT NULL AS tiene_fecha;
```

salida

```
shipment_id |  status   | tiene_fecha
------------+-----------+-------------
       12001 | delivered | t
```

- No es estándar, pero evita una segunda consulta para saber qué cambió.
- Desde Node.js:
  `pool.query('UPDATE ... WHERE shipment_id = $1 RETURNING *', [id])`

Error común: UPDATE o DELETE sin WHERE

modification.sql

```sql
BEGIN;
UPDATE route SET base_fare = 0;
ROLLBACK;
```

salida

```
BEGIN
UPDATE 120
ROLLBACK
```

Las 120 rutas, gratis

La sentencia es válida y toca *todas* las filas.

1. Escriban primero el `SELECT` con el mismo `WHERE`.
2. Revisen cuántas filas devuelve.
3. Recién entonces, la modificación.
4. En el laboratorio, entre `BEGIN` y `ROLLBACK`.

El código de la semana

bash · salida recortada

```
$ cd code/week-03
$ make test
chasqui-load chasqui_w03
Base de datos chasqui_w03 lista.
...
✓ un agregado no puede ir en WHERE
...
i tests 45
i pass 45
i fail 0
```

- Tres archivos `.sql`: sesión A, sesión B y modificación.
- Las pruebas fijan los números de los apuntes.
- `pg` devuelve `count` y `numeric` como texto: `count(*)::int`.
- `BEGIN` y `ROLLBACK` por la misma conexión: `pool.connect()`.

Para recordar

- Comparar con `NULL` da unknown; `WHERE` solo conserva lo verdadero.
- `IS NULL`, nunca `= NULL`.
- Los agregados ignoran `NULL` (salvo `count(*)`); sobre vacío, `count` da 0 y los demás `NULL`.
- Sin `GROUP BY` (ni `HAVING`): siempre una fila. Con `GROUP BY`: puede no haber ninguna.
- `WHERE` filtra filas; `HAVING`, grupos.
- `NOT IN` con un `NULL`: nunca verdadero. Usen `NOT EXISTS`.
- `ALL` sobre vacío es V; `SOME`, F.
- Las modificaciones evalúan su condición antes de cambiar nada.

Para la próxima clase

- Leer Silberschatz, secciones 3.8 y 3.9, si no lo hicieron, y las notas 1 y 2 de CS186.
- Laboratorio: `labo3.sql`, con las modificaciones entre `BEGIN` y `ROLLBACK`.
- Entregar **P0** y leer el enunciado de **P1**.

Pregunta

Escriban con `NOT EXISTS` los vehículos que han hecho viajes por todas las rutas que salen de Lima. ¿Qué devuelve la consulta si no hay ninguna ruta que salga de Lima?