

Bases de Datos

SQL desde un programa y SQL avanzado

Modelo relacional y SQL

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: SQL desde Node.js

1. El programa y el DBMS; JDBC y `pg`
2. `Client`, `Pool`, resultados y tipos
3. Inyección SQL y consultas parametrizadas
4. Transacciones con un mismo client

Sesión B: SQL avanzado

5. Funciones y procedimientos
6. Triggers
7. Consultas recursivas
8. Window functions

Lectura

Silberschatz, secs. 5.1 y 9.8.1 (Sesión A); 5.2, 5.3, 5.4 y 5.5 (Sesión B). Además: node-postgres (*Connecting*, *Queries*, *Pooling*, *Transactions*), OWASP *SQL Injection Prevention Cheat Sheet*, la documentación de PostgreSQL sobre funciones, triggers, `WITH` y window functions, y CMU 15-445, clase #02 *Modern SQL*.

SECCIÓN 01

El programa y el DBMS

01



Una conversación de texto



objetos, números, textos, fechas

- **SQL dinámico**: el programa arma el texto y lo envía con una biblioteca (JDBC, `pg`).
- **SQL embebido**: el SQL va dentro del código y un preprocesador lo traduce. Hoy es raro.

Lo que ya saben de JDBC

JDBC (Java)	pg (Node.js)
`Connection`	`pg.Client`
`DataSource` con connection pool	`pg.Pool`
`PreparedStatement` con `?`	`query(texto, [valores])` con `$1`, `$2`
`ResultSet` y `next()`	`result.rows`, un arreglo
`executeUpdate()`	`result.rowCount`
try-with-resources, `close()`	`client.release()`, `pool.end()`
`setAutoCommit(false)`, `commit()`	`BEGIN`, `COMMIT` con el mismo client
`SQLException.getSQLState()`	`err.code`

Diferencia de fondo: en Node.js todo es asíncrono, con `await`.

SECCIÓN 02

Conectarse con pg

02



`Client`: una conexión

connect.mjs

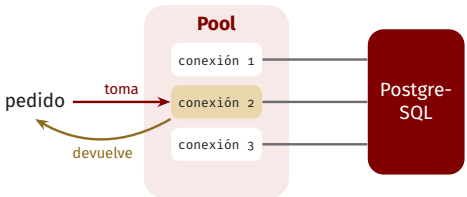
```
1  const client = new pg.Client();
2  await client.connect();
3  try {
4    const result = await client.query(
5      'SELECT name, city FROM agency WHERE region =
         $1',
6      ['Puno']);
7    console.log('rows:', result.rows);
8    console.log('rowCount:', result.rowCount);
9  } finally {
10   await client.end();
11 }
```

node

```
rows: [
  { name: 'Puno - Centro', city: 'Puno
      ' },
  { name: 'Juliaca - Centro', city: '
      Juliaca' }
]
rowCount: 2
```

- Sin opciones, `pg` usa `PGHOST`, `PGUSER`, `PGDATABASE`..., igual que `psql`.
- Un `Client` suelto sirve para un script corto.

`Pool`: conexiones que se prestan



- Abrir una conexión cuesta: el servidor crea un proceso y autentica.
- `pool.query`: toma una conexión libre, ejecuta y la devuelve.
- Hasta `max` conexiones (10 por defecto).
- `pool.end()` las cierra. Sin él, un script de prueba tardó 10 s en terminar (`idleTimeoutMillis`).

Las funciones reciben el pool

queries.mjs

```
export async function findCustomerByDni(db, dni) {
  const result = await db.query(
    'SELECT customer_id, dni, first_name, last_name, phone, city
     FROM customer
     WHERE dni = $1',
    [dni]);
  return result.rows;
}
```

Quien crea el pool, lo cierra

Un pool por aplicación, creado al arrancar. Las funciones lo reciben (o reciben un client: los dos tienen `query`) y no lo cierran. Así se pueden probar.

Los tipos que llegan a JavaScript

node connect.mjs

```
{
  shipment_id: 1,
  weight_kg: '64.60',
  price: '88.17',
  created_at: 2026-01-29T00
    :46:00.000Z,
  delivered_at: 2026-02-04T23
    :30:00.000Z,
  events: '4'
}
suma con texto: 88.1764.60
suma con Number: 152.76999999999998
```

PostgreSQL	JavaScript
`integer`	`number`
`numeric`	`string` (exacto)
`bigint`, `count(*)`	`string`
`timestamptz`	`Date` (se muestra en UTC)
`NULL`	`null`

Ojo

`assert.equal(row.price, 88.17)` falla: llega `'88.17'`. Los montos se suman en SQL, no con `Number`.

Errores: `err.code`

el error de un DNI repetido

```
{
  code: '23505',
  constraint: 'customer_dni_key',
  detail: 'Key (dni)=(61830975) already exists
      .',
  table: 'customer'
}
```

queries.mjs

```
} catch (err) {
  if (err.code === UNIQUE_VIOLATION &&
      err.constraint === '
      customer_dni_key') {
    throw new CustomerError(
      `el DNI ${dni} ya está registrado`,
      { cause: err });
  }
  throw err; // conexión caída, etc.
}
```

`23505` único, `23503` clave foránea, `23514` `CHECK`, `42P01` tabla inexistente.

Las restricciones con nombre (semana 4) dicen *qué* regla se violó. Al usuario, un mensaje útil; nunca el SQL.

SECCIÓN 03

Inyección SQL

03



Un buscador que pega el texto del usuario

unsafe.mjs

```
export async function findCustomerByDniUnsafe(db, dni) {
  return db.query(
    `SELECT customer_id, dni, first_name, last_name, phone, city
     FROM customer
     WHERE dni = '${dni}'`);
}
```

Con `dni = "' OR '1'='1"`, al servidor le llega:

lo que recibe PostgreSQL

```
SELECT ... FROM customer WHERE dni = '' OR '1'='1'
```

La comilla del usuario cierra la cadena; lo demás es SQL. Todas las filas cumplen.

Dos sentencias en un mismo texto

Con `dni = "'; DELETE FROM shipment_event; --"`:

lo que recibe PostgreSQL

```
SELECT ... FROM customer WHERE dni = ''; DELETE FROM shipment_event; --'
```

- `;` termina el `SELECT`; `--` convierte en comentario la comilla que sobraba.
- Una consulta sin parámetros viaja por el **protocolo simple**, que acepta varias sentencias. `query` devuelve un arreglo: un resultado por sentencia.

El libro

Silberschatz, 9.8.1: el mismo ataque con JDBC, concatenando en un `Statement`. El error es tratar datos como código.

Lo que pasó en `chasqui_w05`

injection.mjs

```
$ PGDATABASE=chasqui_w05 node injection.mjs
dni = "' OR '1'='1"
  vulnerable:    filas = 1800
  parametrizada: filas = 0
dni = "'; DELETE FROM shipment_event; --"
  eventos antes:  46421
  la vulnerable devolvió 2 resultados: SELECT 0, DELETE 46421
  eventos después: 0
  eventos tras ROLLBACK: 46421
  parametrizada: filas = 0, eventos = 46421
```

El script se niega a correr si la base de datos no es `chasqui_w05`, y el `DELETE` va dentro de una transacción que termina en `ROLLBACK`.

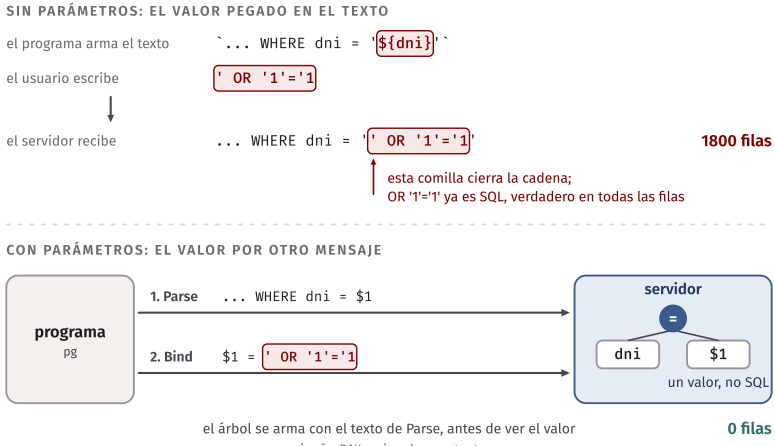
La defensa: consultas parametrizadas

queries.mjs

```
1  db.query('SELECT ... FROM customer WHERE dni = $1', [dni]);
```

- El valor nunca se interpreta como SQL: `' OR '1'='1` es un DNI que no existe.
- Con parámetros no se pueden mandar dos sentencias:
  `42601 cannot insert multiple commands into a prepared statement`

Pegado en el texto o enviado aparte



Lo que $1 no puede reemplazar

queries.mjs (resumido)

```
const ROUTE_ORDER = {
  tarifa: 'r.base_fare',
  distancia: 'r.distance_km',
  destino: 'd.name',
};
if (!Object.hasOwn(ROUTE_ORDER, order)) {
  throw new RangeError('orden no permitido');
}
const column = ROUTE_ORDER[order];
db.query('SELECT ... WHERE o.city = $1
  ORDER BY ${column} ${dir}', [city]);
```

- `$1` es un **valor**. No un nombre de columna ni `ASC`/`DESC`.
- `ORDER BY $1` ordena por una constante: no ordena.
- El usuario elige una clave; el texto sale de una lista que escribió el programador.
- No sirve escapar comillas a mano. Validar el formato ayuda, como segunda capa.

SECCIÓN 04

Transacciones desde Node.js

04



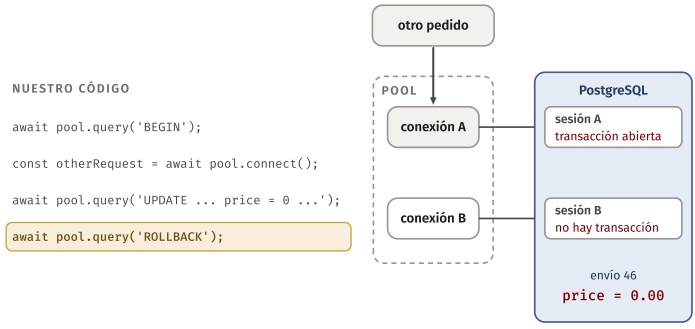
Entregar un envío: todo o nada

deliver.mjs (resumido)

```
export async function deliverShipment(pool, id, agencyId,
      at) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');
    await client.query('UPDATE shipment ...', [id, at]);
    await client.query('INSERT INTO shipment_event ...', [
      id, at, agencyId]);
    await client.query('COMMIT');
  } catch (err) {
    await client.query('ROLLBACK');
    throw err;
  } finally { client.release(); }
}
```

- `pool.connect()`: una conexión para nosotros solos.
- Todo por `client.query`.
- Error ⇒ `ROLLBACK` y se relanza.
- `release()` en el `finally`: el try-with-resources.
- Con la agencia 999 falla el `INSERT` (23503) y el envío sigue `in_transit`.

El error de `pool.query('BEGIN')`



el ROLLBACK va por B: "there is no transaction in progress", y el precio queda en 0.00

Cada `pool.query` puede ir por otra conexión. Con un solo usuario suele tocar la misma, y el código "funciona" en las pruebas.

Lo que pasó

wrong-transaction.mjs

```
$ PGDATABASE=chasqui_w05 node wrong-transaction.mjs
precio del envío 46 al empezar: 153.50
ROLLBACK...
  aviso de PostgreSQL: there is no transaction in progress
precio del envío 46 después del ROLLBACK: 0.00
```

El `UPDATE` ya había hecho commit; el `ROLLBACK` no deshizo nada. Y la conexión A volvió al pool con una transacción abierta: lo que haga el otro pedido queda adentro.

Regla

`BEGIN`, las sentencias y `COMMIT`/`ROLLBACK` van por el **mismo client**, obtenido con `pool.connect()`.

Dos olvidos que cuelgan el programa

Olvidar `client.release()`

La conexión queda prestada para siempre. Cuando se prestan las `max`, el siguiente `pool.connect()` espera una conexión que nunca llega: el programa se cuelga sin error.

Olvidar `pool.end()`

El script hace todo su trabajo y no termina: las conexiones abiertas mantienen vivo el proceso hasta que el pool las cierra por inactividad.

Por eso

`release()` va en el `finally`, no al final del `try`; `pool.end()`, en el `finally` del script.

SECCIÓN 05

Funciones y procedimientos

05

Una función en SQL

functions.sql

```sql
CREATE OR REPLACE FUNCTION
  shipment_fare(p_route_id integer,
                p_weight numeric,
                p_declared numeric DEFAULT NULL)
RETURNS numeric
LANGUAGE sql
STABLE
AS $$
  SELECT round(r.base_fare + 0.9 * p_weight
               + 0.01 * coalesce(p_declared, 0), 2)
  FROM route r
  WHERE r.route_id = p_route_id;
$$;
```

- Una consulta con **nombre** y **parámetros**.
- `$$ ... $$`: una cadena sin escapar comillas.
- `STABLE`: no modifica datos. `IMMUTABLE` si no lee tablas; `VOLATILE` por defecto.
- Ruta inexistente ⇒ `NULL`.

¿La fórmula es la correcta?

psql

```
chasqui_w05=# SELECT shipment_fare(1, 10);
 shipment_fare
---------------
         42.00
(1 row)

chasqui_w05=# SELECT count(*) AS total,
  count(*) FILTER (WHERE s.price = shipment_fare(r.route_id, s.weight_kg, s.declared_value)) AS
     iguales
FROM shipment s
  JOIN route r ON r.origin_agency_id = s.origin_agency_id AND r.dest_agency_id = s.dest_agency_id;
 total | iguales
-------+---------
 12000 |   12000
(1 row)
```

Ruta 1: tarifa base 33.00, y $33 + 0.9 \cdot 10 = 42$. Los 12 000 precios coinciden.

PL/pgSQL: variables, `IF` y `RAISE`

functions.sql

```sql
CREATE OR REPLACE FUNCTION quote_fare(p_origin integer, p_dest integer,
                                      p_weight numeric, p_declared numeric DEFAULT NULL)
RETURNS numeric LANGUAGE plpgsql STABLE AS $$
DECLARE v_fare numeric;
BEGIN
  IF p_weight IS NULL OR p_weight <= 0 THEN
    RAISE EXCEPTION 'peso inválido: %', p_weight USING ERRCODE = 'check_violation';
  END IF;
  SELECT base_fare INTO v_fare FROM route
  WHERE origin_agency_id = p_origin AND dest_agency_id = p_dest;
  IF NOT FOUND THEN
    RAISE EXCEPTION 'no hay ruta de la agencia % a la agencia %', p_origin, p_dest
      USING ERRCODE = 'no_data_found';
  END IF;
  RETURN round(v_fare + 0.9 * p_weight + 0.01 * coalesce(p_declared, 0), 2);
END; $$;
```

`RAISE EXCEPTION`: el error llega al programa

```
psql

chasqui_w05=# SELECT quote_fare(1, 7, 10, 500);
 quote_fare
------------
      47.00
(1 row)

chasqui_w05=# SELECT quote_fare(1, 29, 10);
ERROR:  no hay ruta de la agencia 1 a la agencia 29
CONTEXT:  PL/pgSQL function quote_fare(integer,
      integer,
numeric,numeric) line 15 at RAISE
chasqui_w05=# SELECT quote_fare(1, 7, -2);
ERROR:  peso inválido: -2
```

- $33 + 0.9 \cdot 10 + 0.01 \cdot 500 = 47$.
- `SELECT ... INTO` guarda en una variable; `FOUND` dice si encontró una fila.
- `RAISE EXCEPTION` aborta la sentencia que llamó a la función.
- `ERRCODE` elige el SQLSTATE: en Node llegan `'P0002'` y `'23514'`.

Funciones que devuelven tablas: van en el `FROM`

functions.sql (resumido)

```sql
CREATE OR REPLACE FUNCTION routes_from(p_city text)
RETURNS TABLE (route_id integer, origin text, dest text, base_fare numeric, shipments bigint)
LANGUAGE sql STABLE
AS $$ SELECT r.route_id, o.name, d.name, r.base_fare, count(s.shipment_id) FROM ... $$;
```

psql

```
chasqui_w05=# SELECT * FROM routes_from('Cusco') ORDER BY shipments DESC;
 route_id |     origin      |           dest            | base_fare | shipments
----------+-----------------+---------------------------+-----------+-----------
       44 | Cusco - Wanchaq | Lima - La Victoria        |     38.00 |       138
       46 | Cusco - Wanchaq | Lima - Los Olivos         |     38.50 |       123
      109 | Cusco - Wanchaq | Puerto Maldonado - Centro |     23.00 |        92
 ...
(6 rows)
```

Procedimientos: `CALL` y `COMMIT` por dentro

functions.sql

```
1  CREATE OR REPLACE PROCEDURE
       refresh_agency_stats()
2  LANGUAGE plpgsql AS $$
3  BEGIN
4    DELETE FROM agency_stats;
5    INSERT INTO agency_stats
6      (agency_id, shipments, revenue,
       refreshed_at)
7    SELECT a.agency_id, count(s.shipment_id),
8           coalesce(sum(s.price), 0), now()
9    FROM agency a LEFT JOIN shipment s
10     ON s.origin_agency_id = a.agency_id
11   GROUP BY a.agency_id;
12   COMMIT;
13 END;
14 $$;
```

No devuelve un valor; se llama con `CALL`. A diferencia de una función, puede hacer `COMMIT`…

psql

```
chasqui_w05=# BEGIN;
BEGIN
chasqui_w05=*# CALL refresh_agency_stats
     ();
ERROR:  invalid transaction termination
CONTEXT:  PL/pgSQL function
     refresh_agency_stats()
line 9 at COMMIT
```

…pero no si quien lo llama abrió una transacción: es suya.

¿En la base de datos o en la aplicación?

En la base de datos

La regla que debe cumplirse sin importar qué programa escriba los datos: la aplicación, una migración, alguien en `psql`.

En la aplicación

Lo que depende de la interfaz, de servicios externos o de reglas que cambian seguido: ahí es más fácil probar, versionar y depurar.

Ejemplo

`shipment_fare` en la base de datos garantiza que todos calculen igual el precio; el costo es que cambiar la tarifa es un cambio en la base de datos.

SECCIÓN 06

Triggers

06

Qué es un trigger

Trigger

Una función que el DBMS ejecuta sola ante un `INSERT`, `UPDATE` o `DELETE` sobre una tabla.

Decisión	Opciones	Qué cambia
Evento	`INSERT`, `UPDATE [OF col]`, `DELETE`	qué lo dispara
Momento	`BEFORE` / `AFTER`	`BEFORE` puede cambiar `NEW`
Granularidad	`FOR EACH ROW` / `FOR EACH STATEMENT`	una vez por fila o por sentencia
Condición	`WHEN (...)`	filtra antes de llamar a la función

En un trigger de fila: `OLD` (la fila como estaba) y `NEW` (como va a quedar). En PostgreSQL: una función que `RETURNS trigger` y `EXECUTE FUNCTION nombre()`.

AFTER: un log de cambios de estado

triggers.sql

```sql
CREATE OR REPLACE FUNCTION log_status_change()
RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  INSERT INTO shipment_status_log
    (shipment_id, old_status, new_status)
  VALUES (NEW.shipment_id, OLD.status, NEW.status)
    ;
  RETURN NULL;  -- en AFTER se ignora
END;
$$;

CREATE OR REPLACE TRIGGER shipment_status_log_trg
  AFTER UPDATE OF status ON shipment
  FOR EACH ROW
  WHEN (OLD.status IS DISTINCT FROM NEW.status)
  EXECUTE FUNCTION log_status_change();
```

- La tabla de log tiene `changed_at DEFAULT now()` y `changed_by DEFAULT current_user`.
- `UPDATE OF status` se dispara si `status` está en el `SET`, aunque no cambie.
- El `WHEN` deja pasar solo los cambios reales.

BEFORE: cambiar la fila antes de guardarla

triggers.sql

```sql
CREATE OR REPLACE FUNCTION fill_delivered_at()
RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  IF NEW.status = 'delivered'
     AND NEW.delivered_at IS NULL THEN
    NEW.delivered_at := now();
  END IF;
  RETURN NEW;  -- la fila que se guarda
END;
$$;

CREATE OR REPLACE TRIGGER
      shipment_fill_delivered_at_trg
  BEFORE UPDATE OF status ON shipment
  FOR EACH ROW
  EXECUTE FUNCTION fill_delivered_at();
```

- Lo que devuelve es lo que se guarda.
- `RETURN NULL` en un `BEFORE` de fila cancela esa fila, sin error.
- En un `AFTER` ya es tarde: la fila está guardada.
- Corre antes de verificar `CHECK` y `NOT NULL`.

Los dos triggers en acción

psql, dentro de BEGIN ... ROLLBACK

```
chasqui_w05=*# UPDATE shipment SET status = 'delivered' WHERE shipment_id = 46
               RETURNING shipment_id, status, delivered_at;
 shipment_id |  status   |         delivered_at
-------------+-----------+-------------------------------
          46 | delivered | 2026-10-31 20:54:29.512552-05
(1 row)

chasqui_w05=*# SELECT shipment_id, old_status, new_status, changed_by FROM shipment_status_log;
 shipment_id | old_status | new_status | changed_by
-------------+------------+------------+------------
          46 | in_transit | delivered  | postgres
(1 row)
```

El `UPDATE` no mencionaba `delivered_at`: lo llenó el `BEFORE`. La fila del log la puso el `AFTER`. `now()` es la hora de inicio de la transacción.

FOR EACH STATEMENT y tablas de transición

triggers.sql

```
1  CREATE OR REPLACE TRIGGER
       shipment_update_batch_trg
2    AFTER UPDATE ON shipment
3    REFERENCING NEW TABLE AS changed
4    FOR EACH STATEMENT
5    EXECUTE FUNCTION log_update_batch();
6  -- la función: INSERT INTO
       shipment_update_batch
7  --   (rows_count) SELECT count(*) FROM
       changed;
```

Sin `NEW` ni `OLD`: las filas de toda la sentencia están en `changed`.

psql, misma transacción

```
=*# UPDATE shipment SET status = 'returned'
    WHERE status = 'in_transit'
      AND origin_agency_id = 1;
UPDATE 14
=*# UPDATE shipment SET price = price
    WHERE shipment_id = 2;
UPDATE 1
=*# SELECT batch_id, rows_count
    FROM shipment_update_batch;
 batch_id | rows_count
----------+------------
        1 |          1
        2 |         14
        3 |          1
(3 rows)
```

Tres sentencias, tres filas. El log de estados tiene 15: una por envío que cambió (1 + 14).

Un trigger que se dispara a sí mismo

trigger_loop.sql (resumido; va dentro de BEGIN ... ROLLBACK)

```sql
CREATE FUNCTION touch_shipment() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  UPDATE shipment SET price = NEW.price WHERE shipment_id = NEW.shipment_id;
  RETURN NULL;
END; $$;
CREATE TRIGGER touch_shipment_trg AFTER UPDATE ON shipment
  FOR EACH ROW EXECUTE FUNCTION touch_shipment();
```

psql

```
psql:/work/code/week-05/trigger_loop.sql:22: ERROR:  stack depth limit exceeded
CONTEXT:  SQL statement "UPDATE shipment SET price = NEW.price WHERE shipment_id = NEW.shipment_id"
PL/pgSQL function touch_shipment() line 3 at SQL statement
...
```

¿Trigger o restricción?

Restricción

"El peso es positivo" es un `CHECK`; "la agencia existe" es una clave foránea. Declarativa, siempre se verifica, el optimizador la conoce, su error tiene código estándar.

Trigger

Código que hay que escribir bien y que se puede desactivar. Para lo que una restricción no puede expresar: reglas sobre otras filas u otras tablas, o efectos como un log.

Cuidado

Al cargar un backup o replicar, un trigger puede repetir una acción que ya se hizo (Silberschatz, 5.3). Que haga poco y se entienda leyendo su definición.

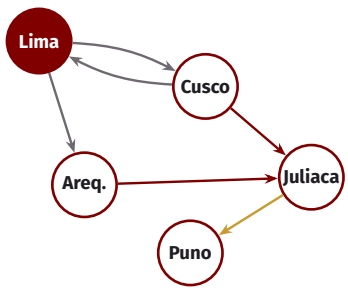
SECCIÓN 07

Consultas recursivas

07



¿A dónde se llega desde Lima?



- Un tramo: un `SELECT` sobre `route`.
- Dos tramos: un join. Tres: dos joins…
- ¿Cuántos tramos hacen falta? No se sabe.

Clausura transitiva

(a, b) está si hay ruta de a a b, o ruta de a a un c y (c, b) está. El álgebra relacional no la expresa con un número fijo de joins.

WITH RECURSIVE

recursive.sql

```sql
WITH RECURSIVE reachable (agency_id) AS (
    SELECT dest_agency_id
    FROM route
    WHERE origin_agency_id = 1
  UNION
    SELECT r.dest_agency_id
    FROM reachable re
      JOIN route r
        ON r.origin_agency_id = re.agency_id
)
SELECT count(*) AS agencies
FROM reachable;
```

psql

```
 agencies
----------
       28
(1 row)
```

- Arriba, la parte **no recursiva**: 14 agencias a un tramo.
- Abajo, la parte **recursiva**: un tramo más desde lo ya alcanzado.
- Falta Iquitos, que no tiene rutas. La 1 está: se va y se vuelve.

Cómo se evalúa: hasta el punto fijo

1. Se ejecuta la parte no recursiva: va al resultado y a la **tabla de trabajo**.
2. Se ejecuta la parte recursiva con solo la tabla de trabajo (lo nuevo de la vuelta anterior).
3. Con `UNION` se descarta lo que ya estaba. Lo que queda es la nueva tabla de trabajo.
4. Si la tabla de trabajo quedó vacía, **termina**. Si no, vuelta al paso 2.

Punto fijo

El resultado no cambia al aplicarle otra vez la parte recursiva. Con 29 agencias, no puede haber agencias nuevas para siempre.

Monotonía (sec. 5.4)

En la parte recursiva no va agregación ni `NOT EXISTS`/`EXCEPT` sobre la relación recursiva: si ella crece, el resultado no debe achicarse.

Ciclos: la recursión que no termina

recursive.sql

```sql
SET statement_timeout = '2s';
WITH RECURSIVE trip_path (agency_id, hops) AS (
    SELECT dest_agency_id, 1
    FROM route WHERE origin_agency_id = 1
  UNION ALL
    SELECT r.dest_agency_id, tp.hops + 1
    FROM trip_path tp
      JOIN route r ON r.origin_agency_id = tp.
      agency_id
)
SELECT count(*) FROM trip_path;
```

psql

```
ERROR:  canceling statement due to
      statement timeout
```

Lima → Cusco → Lima… cada vuelta trae filas distintas, y ni `UNION` las descartaría. **Tres controles**

1. Un límite: `tp.hops < 4`.
2. El camino en un arreglo y `<> ALL (path)`.
3. La cláusula `CYCLE`.

El camino en un arreglo

recursive.sql

```sql
WITH RECURSIVE trip_path (agency_id, hops, path) AS (
    SELECT dest_agency_id, 1, ARRAY[1, dest_agency_id]
    FROM route WHERE origin_agency_id = 1
  UNION ALL
    SELECT r.dest_agency_id, tp.hops + 1, tp.path || r.
      dest_agency_id
    FROM trip_path tp
      JOIN route r ON r.origin_agency_id = tp.agency_id
    WHERE r.dest_agency_id <> ALL (tp.path) AND tp.hops < 4
)
SELECT a.name, min(tp.hops) AS hops
FROM trip_path tp JOIN agency a ON a.agency_id = tp.
     agency_id
GROUP BY a.name HAVING min(tp.hops) >= 3 ORDER BY hops, a.
     name;
```

psql

```
        name         | hops
---------------------+------
 Pucallpa - Centro   |    3
 Puno - Centro       |    3
 Tacna - Centro      |    3
(3 rows)
```

- 14 agencias a un tramo, 10 a dos, 3 a tres.
- El `min` va afuera de la CTE.

La cláusula CYCLE

recursive.sql

```
1  WITH RECURSIVE trip_path (agency_id, hops) AS (
2      SELECT 1, 0
3    UNION ALL
4      SELECT r.dest_agency_id, tp.hops + 1
5      FROM trip_path tp
6        JOIN route r ON r.origin_agency_id = tp.
         agency_id
7      WHERE tp.hops < 4
8  ) CYCLE agency_id SET is_cycle USING path
9  SELECT count(*) FILTER (WHERE NOT is_cycle) AS
       paths,
10       count(*) FILTER (WHERE is_cycle) AS cycles
         ,
11       count(DISTINCT agency_id) FILTER (WHERE
       NOT is_cycle) AS agencies
12 FROM trip_path;
```

psql

```
 paths | cycles | agencies
-------+--------+----------
  1018 |    604 |       28
(1 row)
```

- PostgreSQL agrega `path` e `is_cycle`; la fila que repite una agencia no se expande.
- Arranca en la agencia 1 con 0 tramos, para que ella también esté en el camino.
- Estándar SQL; en PostgreSQL desde la versión 14.

Ejemplo: lo más barato de Lima a Puno

recursive.sql (resumido)

```sql
WITH RECURSIVE trip_path (agency_id, hops, fare, path) AS (
    SELECT dest_agency_id, 1, base_fare::numeric, ARRAY[1, dest_agency_id]
    FROM route WHERE origin_agency_id = 1
  UNION ALL
    SELECT r.dest_agency_id, tp.hops + 1, tp.fare + r.base_fare, tp.path || r.dest_agency_id
    FROM trip_path tp JOIN route r ON r.origin_agency_id = tp.agency_id
    WHERE r.dest_agency_id <> ALL (tp.path) AND tp.hops < 4
)
SELECT tp.hops, tp.fare, ... AS cities FROM trip_path tp
WHERE tp.agency_id = 18 ORDER BY tp.fare, tp.hops LIMIT 3;
```

- Se suma la tarifa tramo por tramo y se guarda el camino.
- La subconsulta de `cities` convierte el arreglo en texto (`unnest ... WITH ORDINALITY`).

Lo más barato de Lima a Puno: el resultado

psql

```
 hops | fare  |                cities
------+-------+-------------------------------------
    3 | 66.50 | Lima → Cusco → Juliaca → Puno
    3 | 76.00 | Lima → Arequipa → Juliaca → Puno
    4 | 76.00 | Lima → Lima → Cusco → Juliaca → Puno
(3 rows)
```

Lo más barato *entre los caminos de a lo más 4 tramos*. “Lima → Lima”: de La Victoria a otra agencia de Lima.

Los tipos de las dos partes deben coincidir

Sin el `::numeric` de la parte no recursiva:

psql

```
ERROR:  recursive query "trip_path" column 3 has type numeric(8,2) in non-recursive term but type
      numeric overall
LINE 2:     SELECT dest_agency_id, 1, base_fare, ARRAY[1, dest_agency_id]...
                                     ^
HINT:  Cast the output of the non-recursive term to the correct type.
```

- `base_fare` es `numeric(8,2)`; la suma `tp.fare + r.base_fare` es `numeric` sin precisión fija.
- Se convierte en la parte no recursiva, como dice el `HINT`.

SECCIÓN 08

Window functions

08



Un valor por fila, mirando a otras filas

GROUP BY: colapsa

```
=# SELECT city, count(*) FROM customer
   WHERE city IN ('Puno','Tacna')
   GROUP BY city;
 city  | count
-------+-------
 Puno  |    47
 Tacna |    46
(2 rows)
```

OVER: cada fila sigue en su lugar

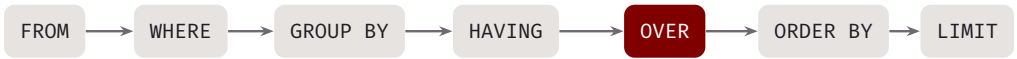
```
=# SELECT customer_id, city,
     count(*) OVER (PARTITION BY city) AS in_city
   FROM customer WHERE city IN ('Puno','Tacna')
   ORDER BY city, customer_id LIMIT 4;
 customer_id | city | in_city
-------------+------+---------
           2 | Puno |      47
           6 | Puno |      47
          54 | Puno |      47
          91 | Puno |      47
(4 rows)
```

Silberschatz, sec. 5.5: el puesto, el acumulado o la fila anterior necesitan un valor *por fila* que depende de otras filas.

La ventana

`función(...) OVER (PARTITION BY ... ORDER BY ... frame)`

- `PARTITION BY`: los grupos; cada fila ve solo los suyos. Sin él, una sola partición.
- `ORDER BY`: el orden dentro de cada partición. Lo necesitan los rankings.
- El **frame**: qué filas, relativas a la actual, entran en un `sum` o `avg`.



Por eso una ventana puede usar agregados (`rank() OVER (ORDER BY sum(price) DESC)`) y no puede ir en el `WHERE`.

rank, dense_rank, row_number

window.sql

```sql
SELECT route_id, base_fare,
  rank()       OVER w AS rank,
  dense_rank() OVER w AS
      dense_rank,
  row_number() OVER w AS
      row_number
FROM route
WINDOW w AS (ORDER BY base_fare
     DESC)
ORDER BY base_fare DESC, route_id
LIMIT 10;
```

`WINDOW w AS (...)` le pone nombre a una ventana para no repetirla.

psql

```
 route_id | base_fare | rank | dense_rank | row_number
----------+-----------+------+------------+------------
       37 |     49.50 |    1 |          1 |          2
       38 |     49.50 |    1 |          1 |          3
       39 |     49.50 |    1 |          1 |          4
       40 |     49.50 |    1 |          1 |          5
       41 |     49.50 |    1 |          1 |          1
       42 |     49.50 |    1 |          1 |          6
       25 |     45.50 |    7 |          2 |          8
       26 |     45.50 |    7 |          2 |          7
       23 |     45.00 |    9 |          3 |          9
       24 |     45.00 |    9 |          3 |         10
(10 rows)
```

Qué hace cada una con los empates

- `rank()`: 1 + el número de filas *estrictamente* antes. Seis empatan con 49.50 y tienen 1; la siguiente tiene 7. Deja huecos, como en una competencia.
- `dense_rank()`: 1 + el número de valores *distintos* antes. Sin huecos: las de 45.50 tienen 2.
- `row_number()`: 1, 2, 3… sin repetir. Entre empates el orden es arbitrario: la ruta 41 quedó primera; otro plan podría numerarlas distinto.

Error común

“Los tres primeros” con `row_number()` y empates: el resultado cambia entre ejecuciones. Usen `rank()` si los empates deben entrar, o desempaten con la clave primaria en el `ORDER BY`.

Los primeros de cada grupo

window.sql

```sql
WITH spent AS (
  SELECT c.city, c.customer_id, c.last_name, sum(s.price) AS total,
         rank() OVER (PARTITION BY c.city ORDER BY sum(s.price) DESC) AS pos
  FROM customer c
    JOIN shipment s ON s.sender_id = c.customer_id
  WHERE c.city IN ('Arequipa', 'Cusco', 'Puno', 'Tacna')
  GROUP BY c.city, c.customer_id, c.last_name
)
SELECT city, pos, customer_id, last_name, total
FROM spent
WHERE pos <= 2
ORDER BY city, pos, customer_id;
```

Los dos clientes que más gastaron como remitentes en cada ciudad del sur.

Los primeros de cada grupo: el resultado

psql

```
   city    | pos | customer_id |     last_name     |  total
-----------+-----+-------------+-------------------+---------
 Arequipa |   1 |        1637 | Condori Ramírez   | 3362.86
 Arequipa |   2 |         847 | Huanca Mamani     | 3315.56
 Cusco    |   1 |        1066 | Torres Sánchez    | 4107.83
 Cusco    |   2 |        1742 | Rojas García      | 1553.43
 Puno     |   1 |        1588 | Paredes Yupanqui  | 3212.32
 Puno     |   2 |        1373 | Gutiérrez Silva   | 2734.15
 Tacna    |   1 |        1412 | Quispe Vargas     | 3454.62
 Tacna    |   2 |         623 | Villanueva Vargas | 2900.55
(8 rows)
```

- Orden: `JOIN`, `WHERE`, `GROUP BY` (la suma), y recién el `rank()` sobre las sumas.
- El filtro `pos <= 2` va en la consulta de afuera.

Una window function no va en el `WHERE`

psql

```
chasqui_w05=# SELECT route_id FROM route WHERE rank() OVER (ORDER BY base_fare DESC) <=
    3;
ERROR:  window functions are not allowed in WHERE
LINE 1: SELECT route_id FROM route WHERE rank() OVER (ORDER BY base_...
                                         ^
```

Error común

Filtrar por una window function en el `WHERE` (o en el `HAVING`). Se evalúan antes que la ventana: calculen el ranking en una CTE o subconsulta y filtren afuera.

El acumulado: `sum(...) OVER (... ORDER BY ...)`

window.sql

```sql
SELECT origin_agency_id AS agency, shipment_id, created_at, price,
       sum(price) OVER (PARTITION BY origin_agency_id
                        ORDER BY created_at) AS running
FROM shipment
WHERE origin_agency_id IN (18, 20)
  AND created_at >= '2026-02-01' AND created_at < '2026-02-08'
ORDER BY origin_agency_id, created_at;
```

- Con `ORDER BY` en la ventana, `sum` suma desde el inicio de la partición hasta la fila actual.
- Puno - Centro (18) y Tacna - Centro (20): dos particiones, cada una desde cero.

El acumulado: el resultado

psql

```
 agency | shipment_id |       created_at       | price  | running
--------+-------------+------------------------+--------+---------
     18 |        9123 | 2026-02-01 18:23:00-05 |  99.69 |   99.69
     18 |        1752 | 2026-02-02 23:42:00-05 |  60.57 |  160.26
     18 |        8820 | 2026-02-03 09:38:00-05 |  13.20 |  173.46
 ...
     18 |        8736 | 2026-02-07 17:37:00-05 |  12.21 |  308.31
     20 |        7301 | 2026-02-01 03:40:00-05 |  87.93 |   87.93
     20 |        3296 | 2026-02-01 11:53:00-05 |  45.63 |  133.56
     20 |       10679 | 2026-02-01 21:27:00-05 |  79.76 |  213.32
 ...
     20 |        6008 | 2026-02-06 13:13:00-05 |  40.65 |  448.77
(14 rows)
```

- Tacna vuelve a empezar en 87.93.
- La ventana ve solo las filas que dejó el `WHERE`: es el acumulado desde el 1 de febrero.

El frame de cada fila

```
sum(price) OVER (PARTITION BY origin_agency_id ORDER BY created_at) AS running
```

PARTICIÓN ORIGIN_AGENCY_ID = 18

	shipment_id	created_at	price	running	
frame de la fila 1564	9123	2026-02-01 18:23:00-05	99.69	99.69	
	1752	2026-02-02 23:42:00-05	60.57	160.26	
	8820	2026-02-03 09:38:00-05	13.20	173.46	
	1564	2026-02-05 16:45:00-05	37.59	211.05	fila actual
	11769	2026-02-06 18:05:00-05	38.20	249.25	
	10605	2026-02-07 13:36:00-05	46.85	296.10	
	8736	2026-02-07 17:37:00-05	12.21	308.31	

```
running de 1564 = 99.69 + 60.57 + 13.20 + 37.59 = 211.05
```

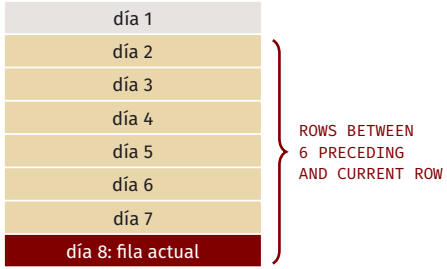
PARTICIÓN ORIGIN_AGENCY_ID = 20 el acumulado vuelve a empezar

	shipment_id	created_at	price	running	
frame de la fila 10679	7301	2026-02-01 03:40:00-05	87.93	87.93	
	3296	2026-02-01 11:53:00-05	45.63	133.56	
	10679	2026-02-01 21:27:00-05	79.76	213.32	fila actual
	11919	2026-02-04 22:07:00-05	105.66	318.98	
	2176	2026-02-06 05:30:00-05	59.25	378.23	
	8415	2026-02-06 11:54:00-05	29.89	408.12	
	6008	2026-02-06 13:13:00-05	40.65	448.77	

```
running de 10679 = 87.93 + 45.63 + 79.76 = 213.32
```

Frames

- Sin `ORDER BY`: la partición entera.
  `sum(price) OVER ()` es el total.
- Con `ORDER BY`, por defecto:
  `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
  hasta la fila actual y sus empates (peers).
- Fila por fila: `ROWS BETWEEN ...` y un desempate en el `ORDER BY`.



Promedio móvil de 7 días

window.sql

```sql
WITH daily AS (
  SELECT (created_at AT TIME ZONE 'America/Lima')::
      date AS day,
         count(*) AS shipments
  FROM shipment WHERE origin_agency_id = 1
  GROUP BY 1
), smoothed AS (
  SELECT day, shipments,
         round(avg(shipments) OVER (ORDER BY day
           ROWS BETWEEN 6 PRECEDING AND CURRENT ROW),
       2) AS avg_7
  FROM daily
)
SELECT day, shipments, avg_7
FROM smoothed WHERE day BETWEEN '2026-03-01' AND '
     2026-03-07'
ORDER BY day;
```

psql

```
    day     | shipments | avg_7
------------+-----------+-------
 2026-03-01 |         9 |  9.71
 2026-03-02 |         5 |  9.14
 2026-03-03 |         7 |  9.43
 2026-03-04 |        11 | 10.00
 2026-03-05 |        12 |  9.86
 2026-03-06 |        10 |  9.86
 2026-03-07 |        11 |  9.29
(7 rows)
```

El filtro va afuera: si no, el 1 de marzo no vería febrero.

`lag`: la fila anterior (`lead`: la siguiente)

window.sql

```sql
SELECT shipment_id, seq, kind, at,
       at - lag(at) OVER (PARTITION BY shipment_id ORDER BY seq) AS since_previous
FROM shipment_event
WHERE shipment_id = 1
ORDER BY seq;
```

psql

```
 shipment_id | seq |   kind    |           at           | since_previous
-------------+-----+-----------+------------------------+-----------------
           1 |   1 | received  | 2026-01-28 19:46:00-05 |
           1 |   2 | departed  | 2026-02-02 12:00:00-05 | 4 days 16:14:00
           1 |   3 | arrived   | 2026-02-02 16:30:00-05 | 04:30:00
           1 |   4 | delivered | 2026-02-04 18:30:00-05 | 2 days 02:00:00
(4 rows)
```

ROLLUP y CUBE

psql

```
1  SELECT a.region, a.city,
2         grouping(a.region, a.city) AS g,
3         sum(s.price) AS revenue
4  FROM shipment s
5    JOIN agency a ON a.agency_id = s.
       origin_agency_id
6  WHERE a.region IN ('Puno', 'Tacna')
7  GROUP BY ROLLUP (a.region, a.city)
8  ORDER BY a.region, a.city;
```

psql

```
 region |  city   | g | revenue
--------+---------+---+----------
 Puno   | Juliaca | 0 | 19904.93
 Puno   | Puno    | 0 |  7279.80
 Puno   |         | 1 | 27184.73
 Tacna  | Tacna   | 0 |  8393.39
 Tacna  |         | 1 |  8393.39
        |         | 3 | 35578.12
(6 rows)
```

- `ROLLUP (a, b)` = `GROUPING SETS ((a, b), (a), ())`.
- `CUBE (a, b)`: además `(b)`. Aquí, 9 filas.

El vacío es un `NULL` que significa “todas”. `grouping` lo distingue de un `NULL` real: un bit por columna agregada.

En resumen

- Un `Pool` por aplicación; las funciones lo reciben y no lo cierran.
- `numeric` y `bigint` llegan como texto.
- Siempre `$1`, `$2`. Lo que no es un valor se elige de una lista fija.
- Transacción: `pool.connect()`, `BEGIN`, `COMMIT`/`ROLLBACK`, `release()` en el `finally`.
- Funciones (`sql`, `plpgsql`, `RETURNS TABLE`); procedimientos con `CALL`.
- Triggers: `BEFORE` cambia `NEW`; `AFTER` ve el cambio; fila o sentencia.
- `WITH RECURSIVE`: punto fijo; los ciclos se controlan.
- `OVER`: ranking, acumulados, frames, `lag`; se filtra en una CTE.

Para la próxima clase

- Laboratorio: una aplicación pequeña con `pg`, con pruebas en `node:test`.
- P1 (Esquema y consultas) se entrega la semana 6.
- Leer Silberschatz, secs. 6.1 a 6.6 (el modelo entidad-relación).
- Correr `make test` en `code/week-05` y leer los apuntes de la semana.

Pregunta

Para cada agencia de origen quieren los tres envíos más caros. ¿Por qué no basta con `GROUP BY`? ¿Qué cambia si usan `rank()` en vez de `row_number()` y hay empates?