

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Joins, vistas e integridad

Modelo relacional y SQL

Gerson Garrido

Semana 4

Hoy

Sesión A: joins y vistas

1. `JOIN ... ON`, `USING` y `NATURAL JOIN`
2. Outer joins: `LEFT`, `RIGHT`, `FULL`
3. `ON` frente a `WHERE`
4. Vistas, vistas materializadas, vistas actualizables

Sesión B: integridad, tipos y autorización

5. `NOT NULL`, `UNIQUE`, `CHECK`
6. Claves foráneas, acciones y restricciones diferidas
7. Reglas que miran otras tablas
8. Fechas y horas, dominios, esquemas
9. Roles, `GRANT` y `REVOKE`

Lectura

Silberschatz, secs. 4.1–4.2 (sesión A); 4.3 como presentación y 4.4–4.7 (sesión B). Berkeley CS186, nota 2 (joins). PostgreSQL: tutoriales *Joins Between Tables* y *Views*; manual: *Constraints* y *Privileges*.

SECCIÓN 01

Expresiones de join

01

Del producto cartesiano al join

Theta join

$r \bowtie_\theta s = \sigma_\theta(r \times s)$

Si θ es una conjunción de igualdades: **equi-join**.

- $r \times s$ tiene $|r| \cdot |s|$ tuplas.
- En SQL, el producto explícito es `CROSS JOIN`.
- La coma del `FROM` es lo mismo.

psql

```
=# SELECT count(*)
-#  FROM agency CROSS JOIN vehicle;
 count
-------
   696
(1 row)
```

29 agencias $\times$ 24 vehículos.

`JOIN ... ON`: la condición junto a la tabla

semana 2

```sql
SELECT count(*)
FROM shipment s, agency a
WHERE s.origin_agency_id = a.agency_id
  AND a.region = 'Cusco';
```

joins.sql

```sql
SELECT count(*)
FROM shipment s
  JOIN agency a ON a.agency_id = s.
      origin_agency_id
WHERE a.region = 'Cusco';
```

- Las dos dan **610**, y `EXPLAIN` muestra el mismo plan.
- El `ON` dice cómo se relacionan las tablas; el `WHERE` queda para los filtros.
- Con cinco tablas, olvidar una condición en un `WHERE` largo es un producto cartesiano silencioso.
- `JOIN` = `INNER JOIN`: solo las combinaciones que cumplen la condición.

`USING`: columnas del mismo nombre

joins.sql

```sql
SELECT trip_id, shipment_id, route_id, weight_kg, departure_at
FROM shipment JOIN trip USING (trip_id)
WHERE shipment_id = 1;
```

psql

```
 trip_id | shipment_id | route_id | weight_kg |      departure_at
---------+-------------+----------+-----------+------------------------
     996 |           1 |       58 |     64.60 | 2026-02-02 12:00:00-05
(1 row)
```

- Equivale a `ON shipment.trip_id = trip.trip_id`, pero `trip_id` aparece una sola vez y se nombra sin prefijo.
- Con `ON` hay dos columnas `trip_id`; nombrarla a secas da error por ambigüedad.

NATURAL JOIN

Natural join

Si $R \cap S = \{A_1, \ldots, A_k\}$: $\quad r \bowtie s = \Pi_{R \cup S}\left(\sigma_{r.A_1 = s.A_1 \wedge \cdots \wedge r.A_k = s.A_k}(r \times s)\right)$

- Iguala todas las columnas con el mismo nombre, sin nombrarlas.
- Cada columna común aparece una vez. Si no hay ninguna, es el producto cartesiano.
- En SQL: `r NATURAL JOIN s` es un `USING` con la lista calculada por el motor.
- Con un esquema real, la lista depende de nombres que nadie miró…

Peligro 1: `customer NATURAL JOIN agency`

joins.sql

```sql
SELECT c.customer_id, c.last_name, a.name
FROM customer c NATURAL JOIN agency a
WHERE c.customer_id = 3;
```

psql

```
 customer_id |   last_name   |        name
-------------+---------------+--------------------
           3 | Ramos Herrera | Lima - La Victoria
           3 | Ramos Herrera | Lima - Los Olivos
           3 | Ramos Herrera | Lima - Ate
(3 rows)
```

- La única columna común es `city`.
- Cada cliente con cada agencia de su ciudad: 3433 filas que nadie pidió.
- La consulta corre sin error. El error está en el significado.

Peligro 2: alguien agrega una columna

joins.sql

```sql
BEGIN;
ALTER TABLE route
  ADD COLUMN created_at timestamptz NOT NULL DEFAULT now
    ();
SELECT count(*) FROM shipment NATURAL JOIN route;
SELECT count(*) FROM shipment
  JOIN route USING (origin_agency_id, dest_agency_id);
ROLLBACK;
```

psql

```
 count
-------
     0
(1 row)

 count
-------
 12000
(1 row)
```

Error común: confiar en `NATURAL JOIN`

De 12000 filas a cero, sin un error. En el curso: `ON` o `USING`.

La misma tabla dos veces

psql

```
SELECT s.shipment_id,
       snd.first_name || ' ' || snd.last_name AS sender,
       rcp.first_name || ' ' || rcp.last_name AS recipient
 FROM shipment s
   JOIN customer snd ON snd.customer_id = s.sender_id
   JOIN customer rcp ON rcp.customer_id = s.recipient_id
 WHERE s.shipment_id <= 2 ORDER BY s.shipment_id;
```

psql

```
 shipment_id |        sender         |       recipient
-------------+-----------------------+-----------------------
           1 | Luis Mamani Chávez    | Gonzalo Herrera Flores
           2 | Kevin Quispe Sánchez  | Rosa Choque Ramos
(2 rows)
```

SECCIÓN 02

Outer joins

02

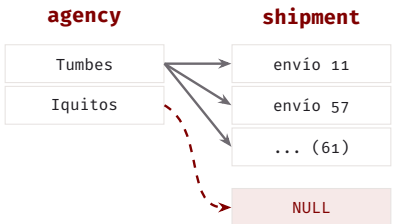


Conservar las filas sin pareja

Left outer join

$$r \bowtie_\theta s \cup \left(r - \Pi_R(r \bowtie_\theta s)\right) \times \{(\text{null}, \ldots, \text{null})\}$$

- El inner join, más cada tupla de r sin pareja, completada con nulos.
- `RIGHT`: lo mismo con s. `FULL`: con los dos lados.



Todas las agencias con su cantidad de envíos

joins.sql

```sql
SELECT a.name, count(s.shipment_id) AS
     shipments
FROM agency a
  LEFT JOIN shipment s
    ON s.origin_agency_id = a.agency_id
GROUP BY a.agency_id, a.name
ORDER BY shipments
LIMIT 3;
```

psql

```
          name          | shipments
------------------------+-----------
 Iquitos - Centro       |         0
 Tumbes - Centro        |        61
 Huancavelica - Centro  |        75
(3 rows)
```

Con un inner join, Iquitos no aparece: no tiene ningún envío con el que emparejarse.

Error común: `count(*)` después de un `LEFT JOIN`

joins.sql

```sql
SELECT a.name, count(*) AS wrong,
       count(s.shipment_id) AS right
FROM agency a
  LEFT JOIN shipment s
    ON s.origin_agency_id = a.agency_id
WHERE a.agency_id IN (10, 29)
GROUP BY a.agency_id, a.name;
```

psql

```
        name       | wrong | right
-------------------+-------+-------
 Iquitos - Centro |     1 |     0
 Tumbes - Centro  |    61 |    61
(2 rows)
```

La fila de nulos también es una fila

`count(*)` cuenta filas: una agencia sin envíos aparece con 1. Se cuenta una columna de la derecha que no sea nula si hay pareja: `count(s.shipment_id)`.

`ON` frente a `WHERE`: envíos entregados por agencia

en el WHERE

```sql
SELECT a.name, count(s.shipment_id)
FROM agency a LEFT JOIN shipment s
    ON s.origin_agency_id = a.agency_id
WHERE a.agency_id IN (10, 29)
  AND s.status = 'delivered'
GROUP BY a.agency_id, a.name;
```

en el ON

```sql
SELECT a.name, count(s.shipment_id)
FROM agency a LEFT JOIN shipment s
    ON s.origin_agency_id = a.agency_id
   AND s.status = 'delivered'
WHERE a.agency_id IN (10, 29)
GROUP BY a.agency_id, a.name;
```

psql

```
      name       | delivered
-----------------+-----------
 Tumbes - Centro |        56
(1 row)
```

psql

```
       name       | delivered
------------------+-----------
 Iquitos - Centro |         0
 Tumbes - Centro  |        56
(2 rows)
```

ON o WHERE, paso a paso

```
FROM agency a
  LEFT JOIN shipment s ON s.origin_agency_id = a.agency_id  AND s.status = 'delivered'
WHERE a.agency_id IN (10, 29)
```

FILAS DEL LEFT JOIN CON EL ON COMPLETO

a.name	s.status	
Tumbes - Centro	delivered	56 filas
Iquitos - Centro	NULL	1 fila

los 5 envíos de Tumbes no entregados
no cumplen el ON: no se emparejan
Iquitos no tiene pareja: queda con NULL

RESULTADO

name	delivered
Iquitos - Centro	0
Tumbes - Centro	56

en el ON, la condición decide qué se empareja: Iquitos queda, con 0 entregados

La regla

A la izquierda, Iquitos sale del join con `s.status` nulo; `NULL = 'delivered'` es *unknown* y el `WHERE` la descarta: quedó un inner join.

Idea clave

En un **inner** join, una condición en el `ON` o en el `WHERE` da lo mismo. En un **outer** join, no:

- el `ON` decide qué filas se emparejan;
- el `WHERE` decide qué filas quedan después.

- Orden lógico: primero el `FROM` con sus joins, después el `WHERE`.
- Un filtro sobre la tabla que se completa con nulos va en el `ON`…
- …salvo que sea justamente `IS NULL`: el anti-join.

El error al revés: filtrar la izquierda en el `ON`

joins.sql

```sql
SELECT count(*) AS rows,
       count(s.shipment_id) AS with_shipment
FROM agency a
  LEFT JOIN shipment s
    ON s.origin_agency_id = a.agency_id
   AND a.region = 'Tumbes';
```

psql

```
 rows | with_shipment
------+---------------
   89 |            61
(1 row)
```

- Se querían los envíos de Tumbes. Salieron los 61… y las otras 28 agencias, cada una con una fila de nulos.
- Una condición en el `ON` no quita filas de la tabla que se conserva entera: solo impide que se emparejen. Ese filtro va en el `WHERE`.

Anti-join y `RIGHT JOIN`

clientes que nunca enviaron nada

```sql
SELECT count(*)
FROM customer c
  LEFT JOIN shipment s
    ON s.sender_id = c.customer_id
WHERE s.shipment_id IS NULL;
```

lo mismo, al revés

```sql
SELECT count(*)
FROM shipment s
  RIGHT JOIN customer c
    ON s.sender_id = c.customer_id
WHERE s.shipment_id IS NULL;
```

psql

```
 count
-------
    82
(1 row)
```

- Las dos dan 82, igual que `NOT EXISTS` (semana 3); el plan puede ser otro.
- `s.shipment_id` es nulo solo en las filas completadas: en las otras es una clave.
- Se prefiere `LEFT`: la tabla que se conserva, primero.

`FULL JOIN`: sin pareja de los dos lados

joins.sql

```sql
SELECT k.agency_id, k.counter_no,
       c.full_name
FROM demo.counter k
  FULL JOIN demo.clerk c
    ON c.agency_id = k.agency_id
   AND c.counter_no = k.counter_no
ORDER BY k.agency_id, k.counter_no;
```

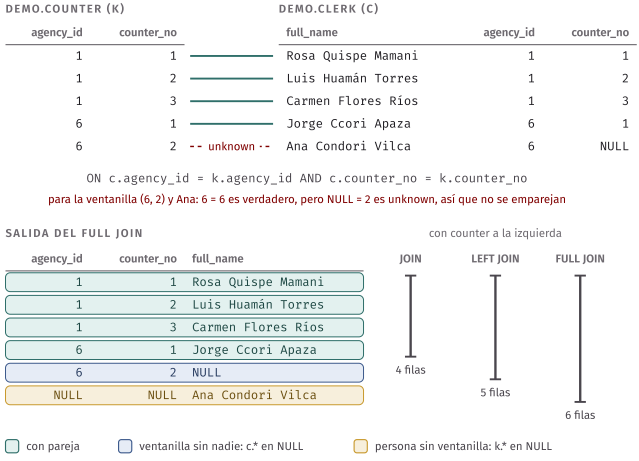
Esquema `demo`: ventanillas (`counter`) y personal (`clerk`). Lo usamos en la Sesión B.

psql

```
 agency_id | counter_no |     full_name
-----------+------------+--------------------
         1 |          1 | Rosa Quispe Mamani
         1 |          2 | Luis Huamán Torres
         1 |          3 | Carmen Flores Ríos
         6 |          1 | Jorge Ccori Apaza
         6 |          2 |
           |            | Ana Condori Vilca
(6 rows)
```

Una ventanilla sin nadie; una persona sin ventanilla.

Inner, left y full



Las formas de join

Forma	Qué conserva	Columnas comunes
`r CROSS JOIN s`	todas las combinaciones	repetidas
`r JOIN s ON` θ	las que cumplen θ	repetidas
`r JOIN s USING (a)`	las que coinciden en `a`	`a` una vez
`r NATURAL JOIN s`	las que coinciden en todas las comunes	una vez
`r LEFT JOIN s ...`	más las de *r* sin pareja	según `ON`/`USING`
`r RIGHT JOIN s ...`	más las de *s* sin pareja	según `ON`/`USING`
`r FULL JOIN s ...`	más las sin pareja de ambos	según `ON`/`USING`

SECCIÓN 03

Vistas

03

Una vista es una consulta con nombre

views.sql

```sql
CREATE VIEW route_detail AS
SELECT r.route_id,
       o.name AS origin_name,
       d.name AS dest_name,
       r.distance_km,
       r.base_fare
FROM route r
  JOIN agency o ON o.agency_id = r.
      origin_agency_id
  JOIN agency d ON d.agency_id = r.
      dest_agency_id;
```

Vista *(view)*

Una relación virtual definida por una consulta. Se guarda la consulta, no su resultado.

- Cada uso se reemplaza por la definición (*view expansion*) y se calcula con los datos de ese momento.
- Sirve para no repetir y para mostrar a cada usuario solo lo suyo.

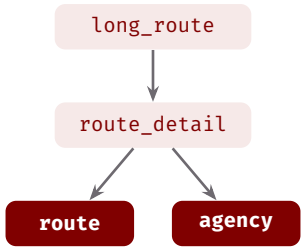
Vistas sobre vistas

views.sql

```sql
CREATE VIEW long_route AS
SELECT route_id, origin_name, dest_name,
       distance_km
FROM route_detail
WHERE distance_km > 800;
```

psql

```
=# SELECT count(*) FROM long_route;
 count
-------
    10
(1 row)
```



la expansión termina si ninguna vista depende de sí misma

Una vista no guarda datos

views-demo.sql

```sql
BEGIN;
UPDATE route SET distance_km = 900
WHERE route_id = 1;
SELECT count(*) FROM long_route;
```

psql

```
 count
-------
    11
(1 row)
```

- Antes eran 10. No hubo que “actualizar” la vista: no hay nada guardado.
- `\sv long_route` muestra la definición guardada.
- Con `SELECT *`, el asterisco se expande al crear la vista: una columna que se agregue después a la tabla no aparece.

Vistas materializadas: guardan el resultado

views.sql y views-demo.sql

```sql
CREATE MATERIALIZED VIEW route_trip_stats AS
SELECT r.route_id, count(t.trip_id) AS trips,
       avg(t.arrival_at - t.departure_at)
         AS avg_duration
FROM route r
  LEFT JOIN trip t ON t.route_id = r.route_id
GROUP BY r.route_id;

INSERT INTO trip VALUES (90001, 1, 1,
  '2026-10-30 07:00-05', '2026-10-30 19:00-05');
SELECT trips FROM route_trip_stats WHERE route_id
      = 1;
REFRESH MATERIALIZED VIEW route_trip_stats;
SELECT trips FROM route_trip_stats WHERE route_id
      = 1;
ROLLBACK;
```

psql

```
 trips
-------
    75
(1 row)

 trips
-------
    76
(1 row)
```

- Rápida de leer, pero desactualizada hasta el REFRESH.
- PostgreSQL siempre recalcula todo.

Vistas actualizables y `WITH CHECK OPTION`

views.sql

```sql
CREATE VIEW truck AS
SELECT vehicle_id, plate,
       kind,
         capacity_kg
FROM vehicle
WHERE kind = 'truck'
WITH CHECK OPTION;
```

Una tabla, solo columnas, sin `DISTINCT` ni `GROUP BY`: cada fila de la vista es una fila de `vehicle`.

psql

```
=# UPDATE truck SET capacity_kg = capacity_kg + 500
-#  WHERE vehicle_id = 11 RETURNING *;
 vehicle_id |  plate  | kind  | capacity_kg
------------+---------+-------+-------------
         11 | P5Z-632 | truck |       11800
(1 row)

UPDATE 1
=# UPDATE truck SET capacity_kg = 1
-#  WHERE vehicle_id = 19;       -- 19 es una van
UPDATE 0
=# UPDATE truck SET kind = 'van' WHERE vehicle_id = 11;
ERROR:  new row violates check option for view "truck"
DETAIL:  Failing row contains (11, P5Z-632, van, 11800).
```

Una vista con join no es actualizable

psql

```
=# UPDATE route_detail SET base_fare = 50 WHERE route_id = 1;
ERROR:  cannot update view "route_detail"
DETAIL:  Views that do not select from a single table or view are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an unconditional
ON UPDATE DO INSTEAD rule.
```

- ¿A cuál de las tres tablas iría el cambio?
- PostgreSQL actualiza automáticamente una vista con una sola tabla en el `FROM`, sin `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT`, `UNION` ni agregados.
- Para lo demás, un trigger `INSTEAD OF` (semana 5).
- La violación de `WITH CHECK OPTION` tiene SQLSTATE `44000`.

SECCIÓN 04

Restricciones de integridad

04

Transacciones, en una frase

constraints.sql

```sql
SELECT count(*) FROM shipment_event
WHERE shipment_id = 1;                -- 4
BEGIN;
DELETE FROM shipment_event
WHERE shipment_id = 1;                --
      DELETE 4
SELECT count(*) FROM shipment_event
WHERE shipment_id = 1;                -- 0
ROLLBACK;
SELECT count(*) FROM shipment_event
WHERE shipment_id = 1;                -- 4
```

Transacción

Un grupo de sentencias que se aplica entero o no se aplica.

- `BEGIN`, `COMMIT`, `ROLLBACK`.
- `psql` hace commit automático de cada sentencia suelta.
- En PostgreSQL, el DDL también se deshace (en Oracle y MySQL, no).
- El resto, en las semanas 13 a 15.

Restricciones sobre una tabla

- `NOT NULL`: sin valores nulos.
- `UNIQUE (A1, ..., Ak)`: una superclave. El motor no verifica que sea mínima.
- `PRIMARY KEY` = `UNIQUE` + `NOT NULL`; una por tabla.
- `CHECK (P)`: P vale en cada fila; puede mencionar varias columnas.

Implicación

"si p, entonces q" $\equiv \neg p \vee q$

demo.sql

```
1  -- si hay compensación, fue aceptado
2  CONSTRAINT
       claim_refund_needs_accepted_ck
3    CHECK (refund IS NULL
4           OR status = 'accepted')
```

psql

```
=# INSERT INTO demo.claim (shipment_id,
     refund)
-# VALUES (1, 50);
ERROR:  new row for relation "claim"
     violates check
constraint "claim_refund_needs_accepted_ck"
```

`CHECK` rechaza solo lo *falso*

```
psql

=# SELECT NULL::numeric > 0 AS result;
 result
--------

(1 row)

=# INSERT INTO demo.claim (shipment_id, refund)
-# VALUES (1, NULL);
INSERT 0 1
=# INSERT INTO demo.claim (shipment_id, refund)
-# VALUES (1, 0);
ERROR:  new row for relation "claim" violates
check constraint "claim_refund_ck"
```

Predicado	`WHERE`	`CHECK`
verdadero	conserva	acepta
falso	descarta	rechaza
unknown	descarta	acepta

Error común

`CHECK (refund > 0)` acepta `NULL`. Si la columna es obligatoria, además lleva `NOT NULL`.

UNIQUE y NULL

psql

```
=# SELECT count(*) AS customers, count(phone) AS with_phone FROM customer;
 customers | with_phone
-----------+------------
      1800 |       1541
(1 row)

=# ALTER TABLE customer ADD CONSTRAINT customer_phone_uq UNIQUE (phone);
ALTER TABLE
=# ALTER TABLE customer DROP CONSTRAINT customer_phone_uq;
ALTER TABLE
=# ALTER TABLE customer ADD CONSTRAINT customer_phone_uq UNIQUE NULLS NOT DISTINCT (phone);
ERROR:  could not create unique index "customer_phone_uq"
DETAIL:  Key (phone)=(null) is duplicated.
```

- 259 clientes sin teléfono, y el `UNIQUE` se acepta: dos `NULL` no son iguales.
- `NULLS NOT DISTINCT` (SQL:2023, PostgreSQL 15) los trata como iguales.

Nombrar las restricciones

psql

```
=# UPDATE shipment SET delivered_at = created_at - interval '1 day' WHERE shipment_id = 1;
ERROR:  new row for relation "shipment" violates check constraint "shipment_check"
DETAIL:  Failing row contains (1, 178, 337, 9, 8, 996, 64.60, 1453.00, 88.17, delivered,
2026-01-28 19:46:00-05, 2026-01-27 19:46:00-05).
```

- `shipment_check` no dice nada: el nombre lo inventó PostgreSQL.
- Con `CONSTRAINT shipment_delivered_after_created_ck CHECK (...)`, el mensaje se explica solo.
- En `demo.sql`, todas llevan nombre y un sufijo: `_pk`, `_fk`, `_uq`, `_ck`.

SQLSTATE: el código que lee un programa

Código	Lo produce
23502	`NOT NULL`
23503	clave foránea
23505	`UNIQUE`, `PRIMARY KEY`
23514	`CHECK`, dominio

En pg: `err.code`, `err.constraint`, `err.column`.

psql

```
=# \set VERBOSITY verbose
=# INSERT INTO customer (customer_id, dni, first_name,
-#   last_name, city)
-# VALUES (9001, '61830975', 'Rosa', 'Quispe', 'Puno')
    ;
ERROR:  23505: duplicate key value violates unique
constraint "customer_dni_key"
DETAIL:  Key (dni)=(61830975) already exists.
SCHEMA NAME:  public
TABLE NAME:  customer
CONSTRAINT NAME:  customer_dni_key
LOCATION:  _bt_check_unique, nbtinsert.c:666
```

Las otras tres, sobre Chasqui

psql

```
=# INSERT INTO shipment (...) VALUES (90001, 99999, 1, 1, 6, 2.5, 30, 'registered', now());
ERROR:  insert or update on table "shipment" violates foreign key constraint "shipment_sender_id_fkey"
DETAIL:  Key (sender_id)=(99999) is not present in table "customer".

=# INSERT INTO vehicle (vehicle_id, plate, kind, capacity_kg) VALUES (99, 'MOT-001', 'moto', 150);
ERROR:  new row for relation "vehicle" violates check constraint "vehicle_kind_check"
DETAIL:  Failing row contains (99, MOT-001, moto, 150).

=# INSERT INTO shipment (...) VALUES (90001, 1, 2, 1, 6, 2.5, 'registered', now());  -- sin price
ERROR:  null value in column "price" of relation "shipment" violates not-null constraint
```

(Las listas de columnas, recortadas; las sentencias completas están en `constraints.sql`.)

Agregar una restricción a una tabla con datos

psql

```
=# ALTER TABLE demo.counter
-#   ADD CONSTRAINT counter_recent_ck CHECK (opened_on >= '2025-01-01');
ERROR:  check constraint "counter_recent_ck" of relation "counter" is violated by some row
```

- `ADD CONSTRAINT` verifica todas las filas que ya existen.
- Buscar las culpables con la negación del predicado en un `WHERE`, decidir qué hacer con ellas y volver a intentar.
- `NOT VALID` verifica solo las filas nuevas; mientras no se haga `VALIDATE CONSTRAINT`, la regla no vale para todos los datos.

SECCIÓN 05

Integridad referencial

05



Clave foránea

Integridad referencial

K clave candidata de r_1, α atributos de r_2: $\Pi_\alpha(r_2) \subseteq \Pi_K(r_1)$. α es una **clave foránea** *(foreign key)* de r_2.

El destino necesita `PRIMARY KEY` o `UNIQUE` exactamente sobre esas columnas:

psql

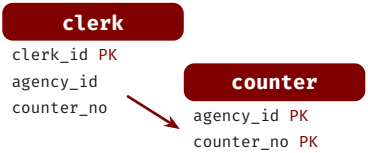
```
=# CREATE TABLE demo.counter_note (
-#   counter_no smallint REFERENCES demo.counter (counter_no), note text);
ERROR:  there is no unique constraint matching given keys for referenced table "counter"
```

La clave de `counter` es (`agency_id`, `counter_no`). Si dos agencias tienen una ventanilla 1, ¿a cuál apunta la nota? (`42830`)

Clave foránea compuesta

demo.sql

```
1  CONSTRAINT clerk_counter_fk
2    FOREIGN KEY (agency_id, counter_no)
3    REFERENCES demo.counter
4    ON DELETE SET NULL (counter_no)
5    ON UPDATE CASCADE
6    DEFERRABLE INITIALLY IMMEDIATE
```



psql

```
=# INSERT INTO demo.clerk (dni, full_name, agency_id, counter_no)
-# VALUES ('46789012', 'Pedro Salas Ugarte', 6, 3);
ERROR:  insert or update on table "clerk" violates foreign key constraint "clerk_counter_fk"
DETAIL:  Key (agency_id, counter_no)=(6, 3) is not present in table "counter".
```

Claves foráneas y `NULL`: `MATCH SIMPLE`

psql, dentro de BEGIN ... ROLLBACK

```
=# ALTER TABLE demo.clerk DROP CONSTRAINT
     clerk_agency_fk;
ALTER TABLE
=# INSERT INTO demo.clerk (dni, full_name, agency_id
     ,
-#   counter_no)
-# VALUES ('47890123', 'Elena Paz Soto', 999, NULL);
INSERT 0 1
=# SELECT full_name, agency_id, counter_no
-#  FROM demo.clerk WHERE agency_id = 999;
   full_name    | agency_id | counter_no
----------------+-----------+------------
 Elena Paz Soto |       999 |
(1 row)
```

`MATCH SIMPLE` (por defecto)

Si alguna columna de la clave foránea es `NULL`, no se verifica nada.

- La agencia 999 no existe, y pasó.
- Por eso `clerk` tiene las dos claves foráneas.
- `MATCH FULL`: toda nula o toda no nula. Aquí rechazaría a Ana (agencia sí, ventanilla no).

Qué pasa al borrar o actualizar lo referenciado

Acción	Al borrar o actualizar la fila referenciada
`NO ACTION`	error si quedan filas que la referencian; al final de la sentencia; se puede diferir. Es la acción por defecto.
`RESTRICT`	error si hay filas que la referencian; de inmediato; no se puede diferir
`CASCADE`	se borran (o actualizan) también las filas que la referencian
`SET NULL`	la clave foránea pasa a `NULL`
`SET DEFAULT`	la clave foránea toma su valor por defecto

`DELETE FROM agency WHERE agency_id = 6` falla con `23503`: la agencia tiene rutas.

Las acciones en el esquema `demo`

constraints.sql

```sql
DELETE FROM demo.counter
WHERE agency_id = 1 AND counter_no = 3;
UPDATE demo.counter SET counter_no = 7
WHERE agency_id = 1 AND counter_no = 2;
```

demo.clerk, agencia 1, después

```
     full_name      | agency_id | counter_no
--------------------+-----------+------------
 Rosa Quispe Mamani |         1 |          1
 Luis Huamán Torres |         1 |          7
 Carmen Flores Ríos |         1 |
```

Antes: ventanillas 1, 2 y 3.

- Borrar una ventanilla deja a su personal sin ventanilla: `SET NULL (counter_no)`.
- Renumerarla arrastra al personal: `ON UPDATE CASCADE`.
- Borrar una agencia borra sus ventanillas: `counter_agency_fk` es `ON DELETE CASCADE`.

En PostgreSQL

`SET NULL (columnas)` es de la versión 15. Un `SET NULL` a secas anularía también `agency_id`, que es `NOT NULL`: error `23502`.

Antes y después de las acciones

demo.clerk

ANTES

full_name	agency_id	counter_no
Rosa Quispe Mamani	1	1
Luis Huamán Torres	1	2
Carmen Flores Ríos	1	3

demo.counter

agency_id	counter_no
1	1
1	2
1	3

```
DELETE FROM demo.counter WHERE agency_id = 1 AND counter_no = 3;
```

→ ON DELETE SET NULL (counter_no): Carmen queda sin ventanilla

```
UPDATE demo.counter SET counter_no = 7 WHERE agency_id = 1 AND counter_no = 2;
```

→ ON UPDATE CASCADE: Luis pasa a la ventanilla 7

DESPUÉS

full_name	agency_id	counter_no
Rosa Quispe Mamani	1	1
Luis Huamán Torres	1	7
Carmen Flores Ríos	1	NULL

agency_id	counter_no
1	1
1	7

sin flecha: con counter_no en NULL, la clave foránea no se verifica

¿Qué acción elegir?

- Depende de qué significa la relación.
- Una fila que no tiene sentido sin la otra (una ventanilla sin su agencia): `CASCADE`.
- Una fila que es historia (un envío ya entregado): `RESTRICT` o `NO ACTION`. Que el borrado falle y alguien decida.
- Una relación opcional, y la fila sigue teniendo sentido sola: `SET NULL`.

`CASCADE` es cómodo y peligroso

Un solo `DELETE` puede borrar miles de filas en tablas que nadie nombró.

En P1 deciden la acción de cada clave foránea de Chasqui, y la justifican.

Restricciones diferidas

psql

```
=# BEGIN;
=# SET CONSTRAINTS demo.clerk_counter_fk DEFERRED;
=# INSERT INTO demo.clerk (dni, full_name,
     agency_id,
-#   counter_no)
-# VALUES ('48901234', 'Raúl Inga Paucar', 6, 4);
INSERT 0 1
=# COMMIT;
ERROR:  insert or update on table "clerk" violates
foreign key constraint "clerk_counter_fk"
DETAIL:  Key (agency_id, counter_no)=(6, 4) is not
present in table "counter".
```

El `INSERT` pasó; el error llegó en el `COMMIT` y se deshizo todo.

- `DEFERRABLE`: se puede verificar al hacer commit.
- `INITIALLY IMMEDIATE`: normal, salvo `SET CONSTRAINTS ... DEFERRED`.
  `INITIALLY DEFERRED`: diferida siempre.
- Uso clásico: ciclos de claves foráneas `NOT NULL`.

En PostgreSQL

Solo se difieren `UNIQUE`, `PRIMARY KEY`, claves foráneas y `EXCLUDE`. `NOT NULL` y `CHECK`, nunca.

En PostgreSQL: `UNIQUE` se verifica fila por fila

psql

```
=# CREATE TABLE demo.slot (n integer UNIQUE);
=# INSERT INTO demo.slot VALUES (1), (2), (3);
=# UPDATE demo.slot SET n = n + 1;
ERROR:  duplicate key value violates unique
constraint "slot_n_key"
DETAIL:  Key (n)=(2) already exists.
=# CREATE TABLE demo.slot2 (n integer UNIQUE
     DEFERRABLE);
CREATE TABLE
=# INSERT INTO demo.slot2 VALUES (1), (2), (3);
INSERT 0 3
=# UPDATE demo.slot2 SET n = n + 1;
UPDATE 3
```

- El resultado final, 2, 3, 4, no tiene duplicados.
- El estándar verifica al final de la sentencia; PostgreSQL, un `UNIQUE` no diferible, fila por fila: al pasar 1 a 2, el 2 todavía existe.
- Con `DEFERRABLE`, aunque sea `INITIALLY IMMEDIATE`, se verifica al final de la sentencia.

SECCIÓN 06

Reglas que miran otras tablas

06

Aserciones

Regla: *la compensación de un reclamo no supera el valor declarado de su envío*. Mira `claim` y `shipment`, y no es una clave foránea.

psql

```
=# CREATE ASSERTION refund_within_value CHECK (NOT EXISTS (
-#   SELECT 1 FROM demo.claim c JOIN shipment s USING (shipment_id)
-#   WHERE c.refund > s.declared_value));
ERROR:  CREATE ASSERTION is not yet implemented
=# ALTER TABLE demo.claim ADD CONSTRAINT claim_refund_within_value_ck
-#   CHECK (refund <= (SELECT declared_value FROM shipment s
-#                     WHERE s.shipment_id = claim.shipment_id));
ERROR:  cannot use subquery in check constraint
```

- El estándar tiene `CREATE ASSERTION` y subconsultas en `CHECK`. Casi ningún motor los implementa.
- El costo: habría que verificarlas tras cada cambio en cualquiera de las tablas.

El truco que no funciona: una función en el CHECK

constraints.sql

```
CREATE FUNCTION demo.declared_value_of(
    p_shipment_id integer) RETURNS numeric
  LANGUAGE sql STABLE
  AS 'SELECT declared_value FROM shipment
      WHERE shipment_id = p_shipment_id';
ALTER TABLE demo.claim
  ADD CONSTRAINT claim_refund_within_value_ck
  CHECK (refund <=
         demo.declared_value_of(shipment_id));
```

El motor lo acepta. 2000 se rechaza, 1000 entra (el envío 1 declaró 1453).

psql

```
=# UPDATE shipment SET declared_value =
     100
-#  WHERE shipment_id = 1;
UPDATE 1
=# SELECT c.refund, s.declared_value
-#  FROM demo.claim c
-#  JOIN shipment s USING (shipment_id);
 refund  | declared_value
---------+----------------
 1000.00 |         100.00
(1 row)
```

Un CHECK se verifica solo cuando cambia su fila. La regla está rota y nadie se enteró.

Entonces, ¿qué se hace?

> **Idea clave**
> Un `CHECK` no puede mirar otras tablas, ni directamente ni a través de una función.

1. **Reorganizar el esquema** para que una clave o una clave foránea exprese la regla, como la ventanilla de la misma agencia.
2. Un **trigger**: una función que el motor ejecuta ante cada cambio en cada tabla involucrada. Semana 5.
3. La **aplicación**, sabiendo que cualquier otro programa se la puede saltar.

En P1 hay una regla de este tipo: decidan cuál de las tres aplica y justifíquenlo.

SECCIÓN 07

Tipos, dominios y esquemas

07

Fechas y horas

Tipo	Qué guarda	Ejemplo
`date`	un día del calendario	`DATE '2026-07-28'`
`time`	una hora del día	`TIME '09:30'`
`timestamp`	fecha y hora "de pared", sin zona	`TIMESTAMP '2026-07-28 09:00'`
`timestamptz`	un instante	`TIMESTAMPTZ '2026-07-28 09:00-05'`
`interval`	una duración	`INTERVAL '1 day 2 hours'`

`timestamptz` no guarda una zona horaria

Guarda un instante (por dentro, en UTC) y lo **muestra** en la zona de la sesión. En Chasqui, todas las horas son `timestamptz`; la base de datos está en `America/Lima` (UTC−5, sin horario de verano).

El mismo instante, dos maneras de mostrarlo

types.sql

```
=# SELECT shipment_id, created_at FROM
      shipment
-#  WHERE shipment_id = 1;
 shipment_id |       created_at
-------------+------------------------
           1 | 2026-01-28 19:46:00-05
(1 row)

=# SET timezone = 'UTC';      -- y la misma
      consulta
 shipment_id |       created_at
-------------+------------------------
           1 | 2026-01-29 00:46:00+00
(1 row)
```

- La zona de la sesión es `America/Lima` (`SHOW timezone`).
- El mismo instante, mostrado en dos zonas.
- `AT TIME ZONE 'America/Lima'` da la hora de pared en Lima (un `timestamp` sin zona).

Error común

La *fecha* de un instante depende de la zona: en UTC, el envío 1 es del 29 de enero. Para reportes por día en hora de Lima, escriban la zona: `(created_at AT TIME ZONE 'America/Lima')::date`.

Intervalos, fechas y conversiones

types.sql

```
=# SELECT avg(arrival_at - departure_at) AS avg_duration,
-#        max(arrival_at - departure_at) AS max_duration
-#   FROM trip;
  avg_duration   | max_duration
-----------------+--------------
 08:54:31.118644 | 22:30:00
(1 row)

=# SELECT 7 / 2 AS int_division,
-#        CAST(7 AS numeric) / 2 AS cast_division;
 int_division |   cast_division
--------------+--------------------
            3 | 3.5000000000000000
(1 row)
```

- Instante − instante = `interval`. `avg` ignora los 14 viajes sin `arrival_at`.
- Fecha − fecha = días: 55.
- Entre enteros, `/` es división entera: `7 / 2` da 3; con `CAST(7 AS numeric)`, 3.5.
- `CAST(x AS t)` es estándar; `x::t`, de PostgreSQL.

Valores por defecto y columnas identity

demo.sql

```
clerk_id integer
  GENERATED ALWAYS AS IDENTITY,
...
status     demo.claim_status
           NOT NULL DEFAULT 'open',
opened_at timestamptz
           NOT NULL DEFAULT now(),
```

- `BY DEFAULT`: deja escribir el valor.
- `serial`: la forma antigua de PostgreSQL.

psql

```
=# INSERT INTO demo.clerk (clerk_id, dni, full_name,
-#   agency_id)
-# VALUES (100, '49012345', 'Marco Tito Huanca', 1);
ERROR:  cannot insert a non-DEFAULT value into
     column
"clerk_id"
DETAIL:  Column "clerk_id" is an identity column
defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

Una secuencia no retrocede con `ROLLBACK` ni con un `INSERT` fallido: hay saltos. Una clave generada identifica; no cuenta.

Dominios y enums

demo.sql

```sql
CREATE DOMAIN demo.dni AS char(8)
  CONSTRAINT dni_format_ck
  CHECK (VALUE ~ '^[0-9]{8}$');

CREATE TYPE demo.claim_status
  AS ENUM ('open', 'accepted', 'rejected'
      );
```

Un dominio: un tipo con nombre y restricciones, que se escribe una vez. `NULL` pasa su `CHECK`.

psql

```
=# INSERT INTO demo.clerk (dni, full_name,
     agency_id)
-# VALUES ('1234567X', 'Sin DNI', 1);
ERROR:  value for domain demo.dni violates
     check
constraint "dni_format_ck"
=# INSERT INTO demo.claim (shipment_id, status
     )
-# VALUES (1, 'closed');
ERROR:  invalid input value for enum
demo.claim_status: "closed"
```

El primero es `23514`; el segundo, `22P02`: `'closed'` no es un valor del tipo. Agregar un valor al enum exige `ALTER TYPE`.

Índices (anuncio) y esquemas

constraints.sql

```
1  CREATE INDEX shipment_sender_idx
2    ON shipment (sender_id);
```

- No cambia ningún resultado, solo la velocidad.
- `PRIMARY KEY` y `UNIQUE` crean su índice solos; una clave foránea, no.
- Semana 10.

psql

```
=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)

=# SELECT count(*) FROM clerk;
ERROR:  relation "clerk" does not exist
=# SET search_path = demo, public;
SET
```

Base de datos → esquema → tabla. Chasqui está en `public`; lo de esta semana, en `demo`. Nombre completo: `demo.clerk`.

SECCIÓN 08

Autorización

08

Privilegios y roles

roles.sql

```sql
CREATE ROLE counter_clerk;
CREATE ROLE courier;
CREATE ROLE analyst;
CREATE ROLE maria LOGIN;
GRANT counter_clerk TO maria;

GRANT SELECT ON customer, agency
  TO counter_clerk;
GRANT INSERT ON shipment TO counter_clerk;
GRANT SELECT ON shipment TO courier;
GRANT UPDATE (status, delivered_at)
  ON shipment TO courier;
GRANT SELECT ON customer_public TO analyst;
```

- Privilegios: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`… Algunos, por columna.
- Un usuario es un rol con `LOGIN`. Un rol miembro de otro hereda sus privilegios.
- Los roles son de todo el servidor: `roles.sql` corre dentro de `BEGIN ... ROLLBACK`.



Probar con `SET ROLE`

psql

```
=# SET ROLE maria;
=# SELECT current_user, count(*) AS customers
-#  FROM customer;
 current_user | customers
--------------+-----------
 maria        |      1800
(1 row)

=# INSERT INTO shipment (...) VALUES (12001, ...);
INSERT 0 1
=# UPDATE shipment SET price = 0
-#  WHERE shipment_id = 12001;
ERROR:  permission denied for table shipment
=# RESET ROLE;
```

- `postgres` es superusuario: ningún privilegio lo limita. Para probar, hay que cambiar de rol.
- `maria` lee clientes e inserta envíos por ser `counter_clerk`.
- No puede cambiar precios, ni leer los envíos.
- SQLSTATE `42501`, insufficient_privilege.

Privilegios por columna

psql

```
=# SET ROLE courier;
=# UPDATE shipment SET status = 'delivered', delivered_at = now() WHERE shipment_id = 46;
UPDATE 1
=# UPDATE shipment SET price = 0 WHERE shipment_id = 46;
ERROR:  permission denied for table shipment
=# RESET ROLE;
```

- `GRANT UPDATE (status, delivered_at) ON shipment TO courier`.
- El `UPDATE` necesita también `SELECT` sobre las columnas del `WHERE`.

Una vista como mecanismo de seguridad

views.sql

```sql
CREATE VIEW
    customer_public AS
SELECT customer_id,
    first_name,
    last_name, city
FROM customer;
```

psql

```
=# SET ROLE analyst;
=# SELECT * FROM customer_public
-#  WHERE customer_id <= 2;
 customer_id | first_name |    last_name    |   city
-------------+------------+-----------------+----------
           1 | Wilmer     | Chávez Ríos     | Ayacucho
           2 | Piero      | García Castillo | Puno
(2 rows)

=# SELECT dni, phone FROM customer
-#  WHERE customer_id <= 2;
ERROR:  permission denied for table customer
```

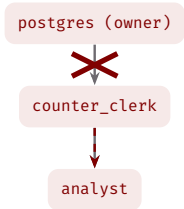
Sin DNI ni teléfono. `analyst` tiene `SELECT` sobre la vista y nada sobre `customer`.

La vista lee la tabla con los privilegios de su owner (`security_invoker = true`, desde PostgreSQL 15, hace lo contrario). Para filtrar *filas*: row-level security.

WITH GRANT OPTION y REVOKE ... CASCADE

psql

```
=# GRANT SELECT ON route TO counter_clerk
-#   WITH GRANT OPTION;
=# SET ROLE counter_clerk;
=# GRANT SELECT ON route TO analyst;
=# RESET ROLE;
=# REVOKE SELECT ON route FROM counter_clerk;
ERROR:  dependent privileges exist
HINT:  Use CASCADE to revoke them too.
=# REVOKE SELECT ON route FROM counter_clerk
     CASCADE;
REVOKE
=# SET ROLE analyst;
=# SELECT count(*) FROM route;
ERROR:  permission denied for table route
```



- Grafo de autorización: un privilegio vive mientras haya un camino desde el owner.
- El libro: en la mayoría de los sistemas, la cascada es el defecto. En PostgreSQL, el defecto es `RESTRICT`.

SECCIÓN 09

El código de la semana

09

`code/week-04` y `make test`

constraints.test.mjs

```
1  test('23503: un envío con un remitente que no
       existe',
2    async () => {
3    await inRollback(async (client) => {
4      const err = await expectError(client,
5        `INSERT INTO ${SHIPMENT_COLS} VALUES
         (90001,
6          99999, 1, 1, 6, 2.5, 30, 'registered',
         now())`);
7      assert.equal(err.code, '23503');
8      assert.equal(err.constraint,
9        'shipment_sender_id_fkey');
10   });
11 });
```

bash

```
$ make test
...
✓ 23505: un DNI repetido
✓ 23503: un envío con un remitente que
      no existe
✓ 23514: un vehículo de tipo moto
✓ 23502: un envío sin precio
...
ℹ tests 43
ℹ pass 43
ℹ fail 0
```

`inRollback`: transacción que termina en `ROLLBACK`. `expectError`: savepoint, sentencia, error.

Para la próxima clase

- Leer Silberschatz, sec. 5.1 (SQL desde un programa) y subsec. 9.8.1 (inyección SQL); para la sesión B, secs. 5.2–5.5 (funciones, triggers, consultas recursivas, agregación avanzada).
- Laboratorio: un esquema que se defiende solo. Violar cada tipo de restricción y leer el SQLSTATE; escribir su propia versión reducida de Chasqui con restricciones con nombre.
- Avanzar P1: `schema/constraints.sql` y las vistas.

Pregunta

Supongan que `shipment` tuviera `CHECK (declared_value <= 10 * price)` y que `price` no fuera `NOT NULL`. ¿Se puede insertar un envío con `declared_value` `NULL`? ¿Y con `price` `NULL`? ¿Qué agregarían para cerrar ese hueco?