

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Datos semiestructurados: JSON en PostgreSQL

Diseño de bases de datos · Repaso y examen parcial

Gerson Garrido

Semana 8

Hoy

Sesión A: JSON en PostgreSQL · Repaso

1. Datos semiestructurados; `json` y `jsonb`
2. Operadores: `->`, `->>`, `#>`, `@>`, `?`
3. Caminos SQL/JSON; de JSON a filas y de filas a JSON
4. Índices GIN (adelanto); cuándo sí y cuándo no
5. Repaso de las partes 1 y 2

Sesión B: examen parcial

6. Semanas 1 a 8

Lectura

Silberschatz, sec. 8.1 (*Semi-structured Data*), la parte de JSON. PostgreSQL: *JSON Types* (*jsonb Containment and Existence*, *jsonb Indexing*) y *JSON Functions and Operators* (*Processing and Creating JSON Data*).

SECCIÓN 01

Datos semiestructurados

01

Cuando el esquema no alcanza

Chasqui quiere guardar características del paquete que cambian de un cliente a otro:

- **muchas y escasas**: cada envío trae unas pocas
- **cambian**: aparece una nueva cada mes
- **anidadas o multivaluadas**: medidas, etiquetas, seguro

Cuando el esquema no alcanza

Chasqui quiere guardar características del paquete que cambian de un cliente a otro:

- **muchas y escasas**: cada envío trae unas pocas
- **cambian**: aparece una nueva cada mes
- **anidadas o multivaluadas**: medidas, etiquetas, seguro

¿Veinte columnas casi siempre en `NULL`?
¿Una tabla por atributo?

Cuando el esquema no alcanza

Chasqui quiere guardar características del paquete que cambian de un cliente a otro:

- **muchas y escasas**: cada envío trae unas pocas
- **cambian**: aparece una nueva cada mes
- **anidadas o multivaluadas**: medidas, etiquetas, seguro

¿Veinte columnas casi siempre en `NULL`? ¿Una tabla por atributo? ¿O un documento por envío, en una sola columna?

details de dos envíos

```
{"fragile": true,
 "dimensions_cm":
   {"l": 32, "w": 27, "h": 22},
 "insurance":
   {"insurer": "Seguros Inti",
    "policy": "POL-12"}}

{"tags": ["alimentos", "perecible"],
 "weight_declared": 11}
```

Ninguna clave en común.

Qué es un dato semiestructurado

Dato semiestructurado

Su estructura viaja con él: cada valor dice qué atributos tiene, y dos valores del mismo conjunto pueden tener atributos distintos (**esquema flexible**). Admite atributos **multivaluados** (listas) y **anidados** (un valor con atributos).

Qué es un dato semiestructurado

Dato semiestructurado

Su estructura viaja con él: cada valor dice qué atributos tiene, y dos valores del mismo conjunto pueden tener atributos distintos (**esquema flexible**). Admite atributos **multivaluados** (listas) y **anidados** (un valor con atributos).

- El libro (sec. 8.1): arreglos, conjuntos, mapas clave-valor; JSON y XML.
- Nos quedamos con JSON: el formato de casi todas las APIs web.

Qué es un dato semiestructurado

Dato semiestructurado

Su estructura viaja con él: cada valor dice qué atributos tiene, y dos valores del mismo conjunto pueden tener atributos distintos (**esquema flexible**). Admite atributos **multivaluados** (listas) y **anidados** (un valor con atributos).

- El libro (sec. 8.1): arreglos, conjuntos, mapas clave-valor; JSON y XML.
- Nos quedamos con JSON: el formato de casi todas las APIs web.

Y la 1NF de la semana 7

Una columna `jsonb` rompe los dominios atómicos **a propósito**. Dentro del documento no hay tipos, `NOT NULL`, claves foráneas ni dependencias funcionales que el motor haga cumplir.

SECCIÓN 02

JSON, el formato

02

Seis tipos (RFC 8259)

Tipo	Ejemplo	Nota
objeto	`{"l": 32, "w": 27}`	claves: textos con comillas dobles
arreglo	`["ropa", "regalo"]`	en orden; tipos mezclados
string	`"Seguros Inti"`	comillas dobles, Unicode
number	`32`, `-1.5`, `2.5e3`	un solo tipo; sin `NaN`
boolean	`true`, `false`	minúsculas
null	`null`	la ausencia de valor, como valor

Seis tipos (RFC 8259)

Tipo	Ejemplo	Nota
objeto	`{"l": 32, "w": 27}`	claves: textos con comillas dobles
arreglo	`["ropa", "regalo"]`	en orden; tipos mezclados
string	`"Seguros Inti"`	comillas dobles, Unicode
number	`32`, `-1.5`, `2.5e3`	un solo tipo; sin `NaN`
boolean	`true`, `false`	minúsculas
null	`null`	la ausencia de valor, como valor

Sin comentarios, sin comillas simples, sin fechas: una fecha es un string ISO 8601, `"2026-02-15T18:18:00-05:00"`.

Seis tipos (RFC 8259)

Tipo	Ejemplo	Nota
objeto	`{"l": 32, "w": 27}`	claves: textos con comillas dobles
arreglo	`["ropa", "regalo"]`	en orden; tipos mezclados
string	`"Seguros Inti"`	comillas dobles, Unicode
number	`32`, `-1.5`, `2.5e3`	un solo tipo; sin `NaN`
boolean	`true`, `false`	minúsculas
null	`null`	la ausencia de valor, como valor

Sin comentarios, sin comillas simples, sin fechas: una fecha es un string ISO 8601, `"2026-02-15T18:18:00-05:00"`.

Tres cosas distintas

La clave tiene el valor `null` de JSON · la clave no está · la columna es `NULL` de SQL.

Un documento de seguimiento

el envío 72, como lo mandaría una API

```
{"shipment": 72, "status": "returned",
 "from": "Arequipa - Cerro Colorado", "to": "Lima - La Victoria",
 "events": [
   {"seq": 1, "at": "2026-02-15T18:18:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "received"},
   {"seq": 2, "at": "2026-02-16T12:00:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "departed"},
   {"seq": 3, "at": "2026-02-17T06:30:00-05:00", "agency": "Lima - La Victoria", "kind": "arrived"},
   {"seq": 4, "at": "2026-02-17T19:30:00-05:00", "agency": "Lima - La Victoria", "kind": "returned"}]}
```

- En tablas: `shipment`, `shipment_event`, `agency` y dos joins.

Un documento de seguimiento

el envío 72, como lo mandaría una API

```
{"shipment": 72, "status": "returned",
 "from": "Arequipa - Cerro Colorado", "to": "Lima - La Victoria",
 "events": [
   {"seq": 1, "at": "2026-02-15T18:18:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "received"},
   {"seq": 2, "at": "2026-02-16T12:00:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "departed"},
   {"seq": 3, "at": "2026-02-17T06:30:00-05:00", "agency": "Lima - La Victoria", "kind": "arrived"},
   {"seq": 4, "at": "2026-02-17T19:30:00-05:00", "agency": "Lima - La Victoria", "kind": "returned"}]}
```

- En tablas: `shipment`, `shipment_event`, `agency` y dos joins.
- En el documento: todo junto, y los nombres de las agencias copiados.

Un documento de seguimiento

el envío 72, como lo mandaría una API

```
{"shipment": 72, "status": "returned",
 "from": "Arequipa - Cerro Colorado", "to": "Lima - La Victoria",
 "events": [
   {"seq": 1, "at": "2026-02-15T18:18:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "received"},
   {"seq": 2, "at": "2026-02-16T12:00:00-05:00", "agency": "Arequipa - Cerro Colorado", "kind": "departed"},
   {"seq": 3, "at": "2026-02-17T06:30:00-05:00", "agency": "Lima - La Victoria", "kind": "arrived"},
   {"seq": 4, "at": "2026-02-17T19:30:00-05:00", "agency": "Lima - La Victoria", "kind": "returned"}]}
```

- En tablas: `shipment`, `shipment_event`, `agency` y dos joins.
- En el documento: todo junto, y los nombres de las agencias copiados.
- Buena **salida** de la base de datos. Como forma de **guardar** esos datos, no: es la redundancia que la normalización quitó.

En tablas y en un documento

SHIPMENT

shipment_id	status	origin_agency_id	dest_agency_id
72	returned	5	1

SHIPMENT_EVENT

shipment_id	seq	at	agency_id	kind
72	1	02-15 18:18	5	received
72	2	02-16 12:00	5	departed
72	3	02-17 06:30	1	arrived
72	4	02-17 19:30	1	returned

AGENCY

agency_id	name
1	Lima - La Victoria
5	Arequipa - Cerro Colorado

at: fechas de 2026, en hora de Lima

DOC DE SHIPMENT_TRACKING (SIN "AT")

```
{"shipment": 72,
 "status": "returned",
 "from": "Arequipa - Cerro Colorado",
 "to": "Lima - La Victoria",
 "events": [
  {"seq": 1, "kind": "received",
   "agency": "Arequipa - Cerro …"},
  {"seq": 2, "kind": "departed",
   "agency": "Arequipa - Cerro …"},
  {"seq": 3, "kind": "arrived",
   "agency": "Lima - La Victoria"},
  {"seq": 4, "kind": "returned",
   "agency": "Lima - La Victoria"}
 ]}
```

(los nombres largos se recortan aquí por espacio)

→ jsonb_build_object + jsonb_agg (vista shipment_tracking)

← jsonb_to_recordset (de events a filas)

en las tablas, cada nombre de agencia está una sola vez; en el documento, copiado:
si la agencia cambia de nombre, un documento guardado (tracking_export) conserva el viejo

SECCIÓN 03

json y jsonb

03



Dos tipos

- `json`: el texto tal cual (espacios, orden, claves repetidas). Se vuelve a analizar en cada consulta.
- `jsonb`: analizado y binario. Sin claves repetidas (gana la última), sin orden de claves. Igualdad, contención e índices.

En el curso: `jsonb`.

examples.sql

```
SELECT '{"b": 1,   "a": 2, "a": 3}'::json
        AS as_json,
       '{"b": 1,   "a": 2, "a": 3}'::jsonb
        AS as_jsonb;
```

Dos tipos

- `json`: el texto tal cual (espacios, orden, claves repetidas). Se vuelve a analizar en cada consulta.
- `jsonb`: analizado y binario. Sin claves repetidas (gana la última), sin orden de claves. Igualdad, contención e índices.

En el curso: `jsonb`.

examples.sql

```
SELECT '{"b": 1,   "a": 2, "a": 3}'::json
        AS as_json,
       '{"b": 1,   "a": 2, "a": 3}'::jsonb
        AS as_jsonb;
```

psql

```
          as_json           |     as_jsonb
----------------------------+------------------
 {"b": 1,   "a": 2, "a": 3} | {"a": 3, "b": 1}
```

Igualdad: solo `jsonb`

examples.sql

```sql
SELECT '{"a": 1, "b": [1, 2]}'::jsonb = '{"b": [1, 2], "a": 1}'::jsonb AS same_object,
       '[1, 2]'::jsonb = '[2, 1]'::jsonb AS same_array,
       '1.50'::jsonb = '1.5'::jsonb AS same_number;
```

psql

```
 same_object | same_array | same_number
-------------+------------+-------------
 t           | f          | t
```

Igualdad: solo `jsonb`

examples.sql

```
SELECT '{"a": 1, "b": [1, 2]}'::jsonb = '{"b": [1, 2], "a": 1}'::jsonb AS same_object,
       '[1, 2]'::jsonb = '[2, 1]'::jsonb AS same_array,
       '1.50'::jsonb = '1.5'::jsonb AS same_number;
```

psql

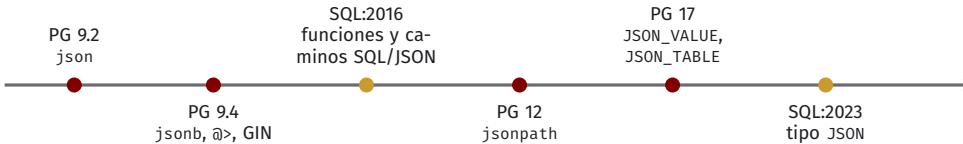
```
 same_object | same_array | same_number
-------------+------------+-------------
 t           | f          | t
```

psql

```
ERROR:  operator does not exist: json = json
```

- Objetos: mismo contenido, cualquier orden. Arreglos: el orden importa.
- `json` tampoco sirve en `GROUP BY`, `DISTINCT` ni `UNIQUE`.

PostgreSQL y el estándar



En PostgreSQL

`->`, `-»`, `#>`, `@>` y `?` son propios de PostgreSQL: llegaron antes que el estándar. El lenguaje de caminos y `JSON_VALUE`, `JSON_QUERY`, `JSON_EXISTS`, `JSON_TABLE` son del estándar.

SECCIÓN 04

El ejemplo de la semana

04

`details` en `shipment` (hipotético)

json-setup.sql y examples.sql

```sql
ALTER TABLE shipment ADD COLUMN details jsonb;

SELECT shipment_id, details FROM shipment WHERE shipment_id IN (1, 2, 5) ORDER BY shipment_id;
```

psql

```
 shipment_id |                                        details
-------------+-------------------------------------------------------------------------------------
           1 | {"instructions": "llamar antes de entregar"}
           2 | ∅
           5 | {"tags": ["alimentos", "perecible"], "instructions": null, "weight_declared": 11}
```

`make` carga Chasqui en `chasqui_w08` y corre `json-setup.sql`: la columna (llena con valores que se deducen del número de envío), la vista `shipment_tracking` y la tabla `tracking_export` con 12 000 documentos.

Qué claves aparecen

examples.sql

```sql
SELECT k AS key, count(*) AS shipments
FROM shipment,
     jsonb_object_keys(details) AS k
GROUP BY k
ORDER BY shipments DESC, key;
```

psql

```
       key       | shipments
-----------------+-----------
 tags            |      6000
 fragile         |      4000
 insurance       |      3571
 dimensions_cm   |      3000
 instructions    |      3000
 weight_declared |      2400
```

- `fragile`: `true`, `false` o ausente
- `dimensions_cm`: objeto `{l, w, h}`
- `tags`: arreglo de strings
- `insurance`: objeto `{insurer, policy}`
- `weight_declared`: number, el peso que dijo el cliente
- `instructions`: string o `null`

1427 envíos tienen `details` en `NULL`.

SECCIÓN 05

Navegar un documento

05

-> devuelve `jsonb`; -» devuelve `text`

examples.sql

```sql
SELECT details -> 'dimensions_cm'          AS dims,
       details -> 'dimensions_cm' -> 'l'   AS l_jsonb,
       details #> '{dimensions_cm,l}'      AS l_path,
       details #>> '{dimensions_cm,l}'     AS l_text,
       details ->> 'fragile'               AS fragile_text,
       details -> 'tags'                   AS tags
FROM shipment WHERE shipment_id = 12;
```

psql

```
            dims             | l_jsonb | l_path | l_text | fragile_text | tags
-----------------------------+---------+--------+--------+--------------+------
 {"h": 22, "l": 32, "w": 27} | 32      | 32     | 32     | true         | ∅
```

-> devuelve `jsonb`; -» devuelve `text`

examples.sql

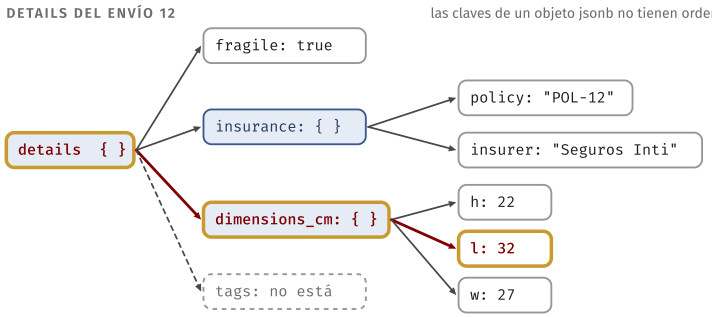
```sql
SELECT details -> 'dimensions_cm'          AS dims,
       details -> 'dimensions_cm' -> 'l'   AS l_jsonb,
       details #> '{dimensions_cm,l}'      AS l_path,
       details #>> '{dimensions_cm,l}'     AS l_text,
       details ->> 'fragile'               AS fragile_text,
       details -> 'tags'                   AS tags
FROM shipment WHERE shipment_id = 12;
```

psql

```
            dims             | l_jsonb | l_path | l_text | fragile_text | tags
-----------------------------+---------+--------+--------+--------------+------
 {"h": 22, "l": 32, "w": 27} | 32      | 32     | 32     | true         | ∅
```

- `->` en los pasos intermedios; `-»` en el último, para comparar o convertir.
- `#>`, `#»`: un camino. Una clave que no está: `NULL`, sin error.

Un documento es un árbol



`details -> 'dimensions_cm' -> 'l'`	`32`	jsonb: el número JSON
`details #> '{dimensions_cm,l}'`	`32`	jsonb: el mismo camino de una vez
`details #>> '{dimensions_cm,l}'`	`'32'`	text: hay que convertirlo para comparar
`details -> 'tags'`	`NULL`	NULL de SQL: la clave no está

Arreglos: desde 0

examples.sql

```sql
SELECT details -> 'tags' -> 0  AS first_tag,
       details -> 'tags' ->> 1 AS second_tag,
       details -> 'tags' -> -1 AS last_tag,
       details -> 'tags' -> 5  AS no_tag
FROM shipment WHERE shipment_id = 5;
```

psql

```
  first_tag   | second_tag |   last_tag    | no_tag
--------------+------------+---------------+--------
 "alimentos" | perecible  | "perecible" | ∅
```

Arreglos: desde 0

examples.sql

```sql
SELECT details -> 'tags' -> 0  AS first_tag,
       details -> 'tags' ->> 1 AS second_tag,
       details -> 'tags' -> -1 AS last_tag,
       details -> 'tags' -> 5  AS no_tag
FROM shipment WHERE shipment_id = 5;
```

psql

```
  first_tag   | second_tag |  last_tag   | no_tag
--------------+------------+-------------+--------
 "alimentos" | perecible  | "perecible" | ∅
```

Error común: contar desde 1. Los arreglos de SQL (`text[]`) empiezan en 1; los de JSON, en **0**, como en JavaScript.

`null` de JSON y `NULL` de SQL

examples.sql

```
SELECT shipment_id, details -> 'instructions' AS arrow, details ->> 'instructions' AS double_arrow,
       (details -> 'instructions') IS NULL AS arrow_is_null, details ? 'instructions' AS has_key
FROM shipment WHERE shipment_id IN (1, 2, 4, 5) ORDER BY shipment_id;
```

psql

```
 shipment_id |           arrow            |       double_arrow       | arrow_is_null | has_key
-------------+----------------------------+--------------------------+---------------+---------
           1 | "llamar antes de entregar" | llamar antes de entregar | f             | t
           2 | ∅                          | ∅                        | t             | ∅
           4 | ∅                          | ∅                        | t             | f
           5 | null                       | ∅                        | f             | t
```

`null` de JSON y `NULL` de SQL

examples.sql

```sql
SELECT shipment_id, details -> 'instructions' AS arrow, details ->> 'instructions' AS double_arrow,
       (details -> 'instructions') IS NULL AS arrow_is_null, details ? 'instructions' AS has_key
FROM shipment WHERE shipment_id IN (1, 2, 4, 5) ORDER BY shipment_id;
```

psql

```
 shipment_id |           arrow            |       double_arrow       | arrow_is_null | has_key
-------------+----------------------------+--------------------------+---------------+---------
           1 | "llamar antes de entregar" | llamar antes de entregar | f             | t
           2 | ∅                          | ∅                        | t             | ∅
           4 | ∅                          | ∅                        | t             | f
           5 | null                       | ∅                        | f             | t
```

- **2**: `details` es `NULL`; hasta `?` da unknown. **4**: sin la clave: `->` da `NULL`, `?` da `f`.

`null` de JSON y `NULL` de SQL

examples.sql

```sql
SELECT shipment_id, details -> 'instructions' AS arrow, details ->> 'instructions' AS double_arrow,
       (details -> 'instructions') IS NULL AS arrow_is_null, details ? 'instructions' AS has_key
FROM shipment WHERE shipment_id IN (1, 2, 4, 5) ORDER BY shipment_id;
```

psql

```
 shipment_id |           arrow            |       double_arrow       | arrow_is_null | has_key
-------------+----------------------------+--------------------------+---------------+---------
           1 | "llamar antes de entregar" | llamar antes de entregar | f             | t
           2 | Ø                          | Ø                        | t             | Ø
           4 | Ø                          | Ø                        | t             | f
           5 | null                       | Ø                        | f             | t
```

- **2**: `details` es `NULL`; hasta `?` da unknown. **4**: sin la clave: `->` da `NULL`, `?` da `f`.
- **5**: la clave vale `null` de JSON, que no es `NULL` de SQL; `->` sí lo convierte en `NULL`.

El error del texto

examples.sql

```sql
SELECT count(*) AS heavy
FROM shipment
WHERE details ->> 'weight_declared' > '9';
```

psql

```
 heavy
-------
    78
```

El error del texto

examples.sql

```
SELECT count(*) AS heavy
FROM shipment
WHERE details ->> 'weight_declared' > '9';
```

psql

```
 heavy
-------
    78
```

examples.sql

```
SELECT count(*) AS heavy
FROM shipment
WHERE (details ->> 'weight_declared')::numeric
      > 9;
```

psql

```
 heavy
-------
  1595
```

El error del texto

examples.sql

```sql
SELECT count(*) AS heavy
FROM shipment
WHERE details ->> 'weight_declared' > '9';
```

psql

```
 heavy
-------
    78
```

examples.sql

```sql
SELECT count(*) AS heavy
FROM shipment
WHERE (details ->> 'weight_declared')::numeric
      > 9;
```

psql

```
 heavy
-------
  1595
```

- `->` da `text`: `'10' > '9'` es falso, `'95' > '9'` es verdadero. Los 78 declararon entre 90 y 99 kilos.
- Sin error, sin aviso: faltan 1517 filas.

Comparar números

examples.sql

```sql
SELECT '10' > '9' AS text_cmp, 10 > 9 AS num_cmp, '10'::jsonb > '9'::jsonb AS jsonb_cmp;
```

psql

```
 text_cmp | num_cmp | jsonb_cmp
----------+---------+-----------
 f        | t       | t
```

- El problema es el texto: dos números `jsonb` se comparan como números. `details -> 'weight_declared' > '9'` también da 1595.
- Igual conviene convertir con `::numeric`: entre valores `jsonb` de tipos distintos rige un orden fijo entre tipos, casi nunca el que se quiere.

Una conversión que falla es un error

psql

```
=> SELECT ('{"l": "cuarenta"}'::jsonb ->> 'l')::integer;
ERROR:  invalid input syntax for type integer: "cuarenta"
```

- No es `NULL`: falla la consulta entera.
- `details -» 'fragile' = true` tampoco: `operator does not exist: text = boolean`. Sí: `= 'true'` o `(details -> 'fragile')::boolean`.

El costo del esquema flexible

Con una columna `integer`, ese valor nunca entraba. Con `jsonb`, entra, y el error aparece meses después, en la consulta de otra persona.

SECCIÓN 06

Contención y existencia

06

@>: un documento dentro de otro

a @> b

Todo lo que dice b también lo dice a, recursivamente:

- **objeto**: cada clave de b está en a, con un valor que contiene al de b
- **arreglo**: cada elemento de b está contenido en alguno de a (sin orden ni repeticiones)
- **escalar**: igual

examples.sql

```
SELECT count(*) AS fragile
FROM shipment
WHERE details @> '{"fragile": true}';
```

psql

```
 fragile
---------
    2000
```

La contención respeta la estructura

examples.sql

```
SELECT '{"tags": ["ropa", "regalo"], "fragile": true}'::jsonb
       @> '{"tags": ["regalo"]}'                          AS sub_array,
      '[1, 2, 3]'::jsonb @> '[3, 1, 1]'                    AS any_order,
      '{"dimensions_cm": {"l": 40, "w": 30}}'::jsonb
       @> '{"dimensions_cm": {"l": 40}}'                  AS nested,
      '{"dimensions_cm": {"l": 40, "w": 30}}'::jsonb
       @> '{"l": 40}'                                     AS not_top_level,
      '{"fragile": true}'::jsonb @> '{}'                   AS empty;
```

psql

```
 sub_array | any_order | nested | not_top_level | empty
-----------+-----------+--------+---------------+-------
 t         | t         | t      | f             | t
```

La contención respeta la estructura

examples.sql

```
SELECT '{"tags": ["ropa", "regalo"], "fragile": true}'::jsonb
       @> '{"tags": ["regalo"]}'                          AS sub_array,
      '[1, 2, 3]'::jsonb @> '[3, 1, 1]'                    AS any_order,
      '{"dimensions_cm": {"l": 40, "w": 30}}'::jsonb
       @> '{"dimensions_cm": {"l": 40}}'                  AS nested,
      '{"dimensions_cm": {"l": 40, "w": 30}}'::jsonb
       @> '{"l": 40}'                                     AS not_top_level,
      '{"fragile": true}'::jsonb @> '{}'                   AS empty;
```

psql

```
 sub_array | any_order | nested | not_top_level | empty
-----------+-----------+--------+---------------+-------
 t         | t         | t      | f             | t
```

`not_top_level`: `l` está *dentro* de `dimensions_cm`. Excepción de PostgreSQL: `'["a", "b"]' @> '"a"'`.

Existencia: ?, ?| y ?&

examples.sql

```sql
SELECT count(*) FILTER (WHERE details ? 'insurance')                        AS insured,
       count(*) FILTER (WHERE details ?| ARRAY['insurance', 'fragile'])     AS insured_or_flag,
       count(*) FILTER (WHERE details ?& ARRAY['insurance', 'dimensions_cm']) AS insured_and_dims
FROM shipment;
```

psql

```
 insured | insured_or_flag | insured_and_dims
---------+-----------------+------------------
    3571 |            6386 |              907
```

- `?`: la clave está en el primer nivel, con cualquier valor (también `null`).
- Sobre un arreglo: `details -> 'tags' ? 'perecible'`.

¿@> o -»? Y los tres valores

`@> '{"fragile": true}'` y `-» 'fragile' = 'true'` dan 2000.

- `@>` opera sobre el documento: un índice GIN lo responde.
- `-»` saca un texto y lo compara: el GIN no sirve.

Para filtrar por igualdad, `@>`.

¿@> o -»? Y los tres valores

`@> '{"fragile": true}'` y `-» 'fragile' = 'true'` dan 2000.

- `@>` opera sobre el documento: un índice GIN lo responde.
- `-»` saca un texto y lo compara: el GIN no sirve.

Para filtrar por igualdad, `@>`.

examples.sql

```sql
SELECT details ->> 'fragile' AS fragile,
       count(*) AS shipments
FROM shipment
GROUP BY details ->> 'fragile'
ORDER BY fragile;
```

psql

```
 fragile | shipments
---------+-----------
 false   |      2000
 true    |      2000
 ∅       |      8000
```

¿@> o -»? Y los tres valores

`@> '{"fragile": true}'` y `-» 'fragile' = 'true'` dan 2000.

- `@>` opera sobre el documento: un índice GIN lo responde.
- `-»` saca un texto y lo compara: el GIN no sirve.

Para filtrar por igualdad, `@>`.

"No frágil" no es `NOT details @> '{"fragile": true}'`: eso incluye a los que no dijeron nada.

examples.sql

```sql
SELECT details ->> 'fragile' AS fragile,
       count(*) AS shipments
FROM shipment
GROUP BY details ->> 'fragile'
ORDER BY fragile;
```

psql

```
 fragile | shipments
---------+-----------
 false   |      2000
 true    |      2000
 ∅       |      8000
```

SECCIÓN 07

De JSON a filas

07

`jsonb_array_elements_text`: una fila por elemento

examples.sql

```sql
SELECT tag, count(*) AS shipments
FROM shipment
  CROSS JOIN LATERAL
    jsonb_array_elements_text(details -> 'tags')
      AS t(tag)
GROUP BY tag
ORDER BY shipments DESC, tag;
```

- Sin `tags`: cero filas, el `CROSS JOIN` lo elimina.
- Es la tabla `shipment_tag` de la semana 6, calculada en cada consulta.

psql

```
     tag      | shipments
--------------+-----------
 regalo       |      2400
 electrónica  |      2352
 alimentos    |      1200
 documentos   |      1200
 perecible    |      1200
 ropa         |      1200
 medicamentos |        48
 urgente      |        48
```

`jsonb_to_recordset`: un arreglo de objetos es una tabla

examples.sql

```sql
SELECT ev.*
FROM tracking_export t,
     jsonb_to_recordset(t.doc -> 'events') AS ev(seq integer, at timestamptz, agency text, kind text)
WHERE (t.doc ->> 'shipment')::integer = 72
ORDER BY ev.seq;
```

psql

```
 seq |           at           |          agency           |   kind
-----+------------------------+---------------------------+----------
   1 | 2026-02-15 18:18:00-05 | Arequipa - Cerro Colorado | received
   2 | 2026-02-16 12:00:00-05 | Arequipa - Cerro Colorado | departed
   3 | 2026-02-17 06:30:00-05 | Lima - La Victoria        | arrived
   4 | 2026-02-17 19:30:00-05 | Lima - La Victoria        | returned
```

El string de `at` salió como `timestamptz`: así se declaró.

SECCIÓN 08

SQL/JSON path

08



Un lenguaje de caminos

Pieza	Significado	Ejemplo
`$`	el documento entero	`$`
`.clave`	el valor de una clave	`$.dimensions_cm.l`
`[n]`, `[*]`	un elemento (desde 0), o todos	`$.events[*]`
`? (condición)`	filtro	`$.events[*] ? (@.kind == "returned")`
`@`	el valor que se prueba	`@.l > 50`
`$nombre`	una variable, que va aparte	`@.l > $min`

- Comparaciones como en JavaScript: `==`, `!=`, `<`, `>=`.
- Strings con comillas dobles: el camino va entre comillas simples de SQL.
- Es parte del estándar (SQL:2016).

Funciones y operadores

examples.sql

```sql
WHERE jsonb_path_exists(details, '$.dimensions_cm ? (@.l > 50)')                -- función
WHERE details @? '$.dimensions_cm ? (@.l > 50)'                                 -- ¿encuentra algo?
WHERE details @@ '$.dimensions_cm.l > 50'                                       -- ¿es verdadero?
WHERE jsonb_path_exists(details, '$.dimensions_cm ? (@.l > $min)', '{"min": 50}')  -- con variable
```

- Las cuatro cuentan 1474, como `(details #» '{dimensions_cm,l}')::integer > 50`.
- Dentro del camino, `@.l > 50` compara números: no hay conversión que olvidar.
- Las variables van aparte, en un objeto `jsonb`: así se pasa un valor desde la aplicación.
- `jsonb_path_query(details, '$.tags[*]')`: una fila por valor encontrado.

Filtrar dentro de un arreglo

examples.sql

```sql
SELECT doc ->> 'shipment' AS shipment,
       jsonb_path_query_first(doc, '$.events[*] ? (@.kind == "returned").agency') AS returned_at
FROM tracking_export
WHERE doc @? '$.events[*] ? (@.kind == "returned")'
ORDER BY (doc ->> 'shipment')::integer LIMIT 5;
```

psql

```
 shipment |     returned_at
----------+---------------------
 63       | "Lima - Ate"
 72       | "Lima - La Victoria"
 134      | "Juliaca - Centro"
 144      | "Huánuco - Centro"
 146      | "Lima - La Victoria"
```

- Todos los eventos → los de `kind` igual a `"returned"` → su `agency`.
- Sin `LIMIT`: 319, los envíos `returned`.

lax y strict

examples.sql

```
SELECT jsonb_path_query_array('{"tags": "ropa"}', 'lax $.tags[*]')     AS lax_array,
       jsonb_path_query_array('{"a": 1}', 'lax $.tags')                AS lax_missing,
       jsonb_path_exists('{"a": 1}', 'strict $.tags', silent => true)  AS strict_silent;
```

psql

```
 lax_array | lax_missing | strict_silent
-----------+-------------+---------------
 ["ropa"]  | []          | ∅
```

psql

```
=> SELECT jsonb_path_query('{"a": 1}', 'strict $.tags');
ERROR:  JSON object does not contain key "tags"
```

`lax` (por defecto): una clave que falta no encuentra nada. `strict`: es error.

Las funciones del estándar (PostgreSQL 17+)

examples.sql

```
SELECT JSON_VALUE(details, '$.weight_declared' RETURNING integer) AS weight,
       JSON_QUERY(details, '$.tags')                              AS tags,
       JSON_EXISTS(details, '$.insurance')                        AS insured
FROM shipment WHERE shipment_id = 5;
```

psql

```
 weight |            tags            | insured
--------+----------------------------+---------
     11 | ["alimentos", "perecible"] | f
```

`JSON_VALUE ... RETURNING integer` es la versión estándar de `(details -» 'weight_declared')::integer`.

JSON_TABLE

examples.sql

```sql
SELECT jt.*
FROM tracking_export t,
     JSON_TABLE(t.doc, '$.events[*]'
       COLUMNS (seq integer PATH '$.seq', agency text PATH '$.agency', kind text PATH '$.kind')) AS jt
WHERE (t.doc ->> 'shipment')::integer = 72
ORDER BY jt.seq;
```

psql

```
 seq |          agency           |   kind
-----+---------------------------+----------
   1 | Arequipa - Cerro Colorado | received
   2 | Arequipa - Cerro Colorado | departed
   3 | Lima - La Victoria        | arrived
   4 | Lima - La Victoria        | returned
```

Lo que hace `jsonb_to_recordset`, con un camino por columna.

SECCIÓN 09

De filas a JSON

09

Una fila, un objeto

examples.sql

```sql
SELECT row_to_json(a) AS as_json, to_jsonb(a) AS as_jsonb FROM agency a WHERE agency_id = 13;
```

psql, con \x

```
-[ RECORD 1 ]-------------------------------------------------------------------------
as_json  | {"agency_id":13,"name":"Ica - Centro","city":"Ica","region":"Ica"}
as_jsonb | {"city": "Ica", "name": "Ica - Centro", "region": "Ica", "agency_id": 13}
```

- `a` es el alias de la tabla usado como valor: la fila entera.
- Tipos: números a number, `text` a string, `NULL` a `null`, `timestamptz` a un string ISO 8601.

`jsonb_build_object`: elegir las claves

examples.sql

```sql
SELECT json_build_object('name', name, 'city', city, 'agency_id', agency_id)  AS j,
       jsonb_build_object('name', name, 'city', city, 'agency_id', agency_id) AS jb
FROM agency WHERE agency_id = 13;
```

psql, con \x

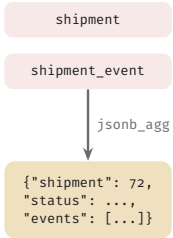
```
-[ RECORD 1 ]-------------------------------------------------
j  | {"name" : "Ica - Centro", "city" : "Ica", "agency_id" : 13}
jb | {"city": "Ica", "name": "Ica - Centro", "agency_id": 13}
```

- `json_build_object` conserva el orden de los pares; `jsonb` no tiene orden de claves.
- Agregados: `jsonb_agg` (un arreglo por grupo) y `jsonb_object_agg(clave, valor)`.

El seguimiento como un documento

examples.sql

```sql
SELECT jsonb_build_object(
         'shipment', s.shipment_id,
         'status', s.status,
         'events', jsonb_agg(
            jsonb_build_object('at', e.at,
                               'kind', e.kind)
            ORDER BY e.seq)) AS doc
FROM shipment s
  JOIN shipment_event e
    ON e.shipment_id = s.shipment_id
WHERE s.shipment_id = 72
GROUP BY s.shipment_id, s.status;
```



- `jsonb_agg` es un agregado; acepta `ORDER BY`.
- Fechas en hora de Lima (la zona de la sesión).
- La vista `shipment_tracking` lo hace para todos.

`jsonb_agg` sobre un `LEFT JOIN`

examples.sql

```sql
SELECT a.name, jsonb_agg(d.name ORDER BY d.name) AS destinations
FROM agency a
  LEFT JOIN route r ON r.origin_agency_id = a.agency_id
  LEFT JOIN agency d ON d.agency_id = r.dest_agency_id
WHERE a.agency_id IN (10, 29)
GROUP BY a.agency_id, a.name ORDER BY a.name;
```

sin FILTER

```
        name       |     destinations
-------------------+----------------------
 Iquitos - Centro | [null]
 Tumbes - Centro  | ["Piura - Castilla"]
```

jsonb_agg sobre un LEFT JOIN

examples.sql

```sql
SELECT a.name, jsonb_agg(d.name ORDER BY d.name) AS destinations
FROM agency a
  LEFT JOIN route r ON r.origin_agency_id = a.agency_id
  LEFT JOIN agency d ON d.agency_id = r.dest_agency_id
WHERE a.agency_id IN (10, 29)
GROUP BY a.agency_id, a.name ORDER BY a.name;
```

sin FILTER

```
       name       |     destinations
------------------+----------------------
 Iquitos - Centro | [null]
 Tumbes - Centro  | ["Piura - Castilla"]
```

con FILTER y coalesce

```
       name       |     destinations
------------------+----------------------
 Iquitos - Centro | []
 Tumbes - Centro  | ["Piura - Castilla"]
```

`jsonb_agg` no ignora los `NULL`; sobre cero filas da `NULL`. Corrección: `coalesce(jsonb_agg(d.name ORDER BY d.name) FILTER (WHERE d.agency_id IS NOT NULL), '[]')`.

SECCIÓN 10

Modificar documentos

10

Un documento nuevo, con `UPDATE`

- `details || '{"fragile": false}'`: une dos objetos; gana la derecha.
- `details - 'insurance'`: quita una clave.
- `details #- '{dimensions_cm,h}'`: quita un camino.
- `jsonb_set(details, '{dimensions_cm,l}', '45')`: reemplaza en un camino.

El `UPDATE` reescribe el documento entero.

Un documento nuevo, con UPDATE

- `details || '{"fragile": false}'`: une dos objetos; gana la derecha.
- `details - 'insurance'`: quita una clave.
- `details #- '{dimensions_cm,h}'`: quita un camino.
- `jsonb_set(details, '{dimensions_cm,l}', '45')`: reemplaza en un camino.

El UPDATE reescribe el documento entero.

modify.sql

```
UPDATE shipment
SET details = details || '{"fragile": true}'
WHERE shipment_id = 2
RETURNING shipment_id, details;
```

psql

```
 shipment_id | details
-------------+---------
           2 | ∅
```

Un documento nuevo, con UPDATE

- `details || '{"fragile": false}'`: une dos objetos; gana la derecha.
- `details - 'insurance'`: quita una clave.
- `details #- '{dimensions_cm,h}'`: quita un camino.
- `jsonb_set(details, '{dimensions_cm,l}', '45')`: reemplaza en un camino.

El `UPDATE` reescribe el documento entero.

modify.sql

```
UPDATE shipment
SET details = details || '{"fragile": true}'
WHERE shipment_id = 2
RETURNING shipment_id, details;
```

psql

```
 shipment_id | details
-------------+---------
           2 | ∅
```

`NULL || x` es `NULL`. Con `coalesce(details, '{}') || ...` queda `{"fragile": true}`.

SECCIÓN 11

JSON desde Node.js

11



pg entrega objetos

tracking.mjs

```
export async function trackingDoc(db, shipmentId) {
  const { rows } = await db.query(
    'SELECT doc FROM shipment_tracking
     WHERE shipment_id = $1',
    [shipmentId],
  );
  return rows.length === 0 ? null : rows[0].doc;
}
```

Sin `JSON.parse`: `doc.events[3].kind` es `'returned'`.

pg entrega objetos

tracking.mjs

```
export async function trackingDoc(db, shipmentId) {
  const { rows } = await db.query(
    'SELECT doc FROM shipment_tracking
     WHERE shipment_id = $1',
    [shipmentId],
  );
  return rows.length === 0 ? null : rows[0].doc;
}
```

Sin `JSON.parse`: `doc.events[3].kind` es `'returned'`.

	columna	en `jsonb`
`numeric`	`string`	`number`
`timestamptz`	`Date`	`string`
`integer`	`number`	`number`

Un número `jsonb` de 20 dígitos pierde precisión en el `number` de JavaScript.

Mandar JSON como parámetro

tracking.mjs

```
export async function shipmentsMatching(db, pattern) {
  const { rows } = await db.query(
    'SELECT shipment_id FROM shipment WHERE details @> $1::jsonb ORDER BY shipment_id',
    [JSON.stringify(pattern)],
  );
  return rows.map((r) => r.shipment_id);
}
```

- `shipmentsMatching(pool, { tags: ['medicamentos'] })`: los 48 envíos.

Mandar JSON como parámetro

tracking.mjs

```
export async function shipmentsMatching(db, pattern) {
  const { rows } = await db.query(
    'SELECT shipment_id FROM shipment WHERE details @> $1::jsonb ORDER BY shipment_id',
    [JSON.stringify(pattern)],
  );
  return rows.map((r) => r.shipment_id);
}
```

- `shipmentsMatching(pool, { tags: ['medicamentos'] })`: los 48 envíos.
- Un **objeto** como parámetro, `pg` lo manda como JSON.
- Un **arreglo**, como arreglo de PostgreSQL (`{"ropa","regalo"}`): `invalid input syntax for type json`.
- Regla: siempre `JSON.stringify(...)` y `$1::jsonb`. Sigue siendo un valor: nada de inyección.

SECCIÓN 12

Índices sobre `jsonb` (adelanto)

12



GIN: un índice invertido

- Cada clave y cada valor que aparece, con la lista de filas donde aparece.
- Como el índice alfabético de un libro.
- `@>` busca los elementos del patrón y cruza sus listas.

Cómo funciona por dentro: parte 3 del curso.

El plan cambia

sin índice

```
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate
   ->  Seq Scan on shipment
         Filter: (details @> '{"tags": ["medicamentos"]}'::jsonb)
```

El plan cambia

sin índice

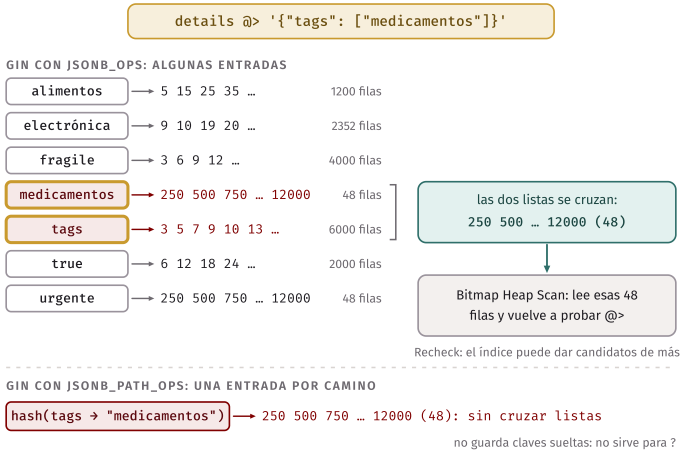
```
                          QUERY PLAN
-------------------------------------------------------------
 Aggregate
   ->  Seq Scan on shipment
         Filter: (details @> '{"tags": ["medicamentos"]}'::jsonb)
```

después de `CREATE INDEX shipment_details_gin ON shipment USING gin (details)`

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate
   ->  Bitmap Heap Scan on shipment
         Recheck Cond: (details @> '{"tags": ["medicamentos"]}'::jsonb)
         ->  Bitmap Index Scan on shipment_details_gin
               Index Cond: (details @> '{"tags": ["medicamentos"]}'::jsonb)
```

El índice da las filas candidatas; `Recheck Cond` vuelve a probar cada una.

Lo que hace el índice con @>



El GIN no sirve para -»

```
con el GIN: details -» 'fragile' = 'true'

                          QUERY PLAN
-------------------------------------------------------------
 Aggregate
   ->  Seq Scan on shipment
         Filter: ((details ->> 'fragile'::text) = 'true'::text)
```

- El índice entiende operadores sobre `details`: `@>`, `?`, `@?`, `@@`. Con `details ? 'insurance'`, el plan usa el mismo índice.
- `details -» 'fragile' = 'true'` compara dos textos, y el primero se calcula fila por fila.
- Es la razón práctica para filtrar con `@>`.

Tres índices posibles

Índice	Sirve para	No sirve	Tamaño
`gin (details)`, `jsonb_ops`	`@>`, `?`, `?|`, `?&`, `@?`, `@@`	`-»`	440 kB
`gin (details jsonb_path_ops)`	`@>`, `@?`, `@@`	`?`, `-»`	304 kB
B-tree sobre `((details #» '{insurance,insurer}'))`	esa expresión: `=`, `<`, `ORDER BY`	lo demás	

En PostgreSQL

Con pocas filas, o si la condición la cumplen muchas, el optimizador puede preferir el `Seq Scan`. Sus estadísticas de una columna `jsonb` son de documentos enteros, no de cada clave: estima a grandes rasgos. Semana 12.

SECCIÓN 13

Cuándo sí y cuándo no

13

“Para no hacer joins”

Un compañero propone guardar remitente, destinatario y agencia de destino en `details`.

modify.sql

```
UPDATE shipment SET details = details || '{"dest_agency_id": 999}' WHERE shipment_id = 12
RETURNING details ->> 'dest_agency_id' AS dest_agency_id,
          EXISTS (SELECT 1 FROM agency WHERE agency_id = 999) AS agency_exists;
```

psql

```
 dest_agency_id | agency_exists
----------------+---------------
 999            | f
```

“Para no hacer joins”

Un compañero propone guardar remitente, destinatario y agencia de destino en `details`.

modify.sql

```sql
UPDATE shipment SET details = details || '{"dest_agency_id": 999}' WHERE shipment_id = 12
RETURNING details ->> 'dest_agency_id' AS dest_agency_id,
          EXISTS (SELECT 1 FROM agency WHERE agency_id = 999) AS agency_exists;
```

psql

```
 dest_agency_id | agency_exists
----------------+---------------
 999            | f
```

- **Integridad referencial**: no hay clave foránea dentro de un documento.
- **Tipos y `NOT NULL`**: cada documento trae lo que quiere.
- **Dependencias funcionales**: `dni` $\rightarrow$ nombre, repetido en cada envío.
- **Optimización**: sin estadísticas por clave, joins a mano.

Lo que se recupera con CHECK

modify.sql

```sql
ALTER TABLE shipment
  ADD CONSTRAINT shipment_details_fragile_ck
  CHECK (NOT details ? 'fragile' OR jsonb_typeof(details -> 'fragile') = 'boolean');
```

psql, con "fragile": "sí"

```
ERROR:  new row for relation "shipment" violates check constraint "shipment_details_fragile_ck"
```

Lo que se recupera con CHECK

modify.sql

```
ALTER TABLE shipment
  ADD CONSTRAINT shipment_details_fragile_ck
  CHECK (NOT details ? 'fragile' OR jsonb_typeof(details -> 'fragile') = 'boolean');
```

psql, con "fragile": "sí"

```
ERROR:  new row for relation "shipment" violates check constraint "shipment_details_fragile_ck"
```

- Un CHECK mira una fila: tipos y claves obligatorias, sí.
- Clave foránea o UNIQUE sobre una clave del documento: no (un índice único de expresión, sí).
- Si la regla necesita otras filas u otras tablas, el dato pide su columna.

Una guía para decidir

`jsonb` es razonable	Mejor columnas o tablas
atributos opcionales, escasos y variables	lo que tiene todo envío
no participan en joins ni en restricciones	lo que se referencia
documentos de otro sistema, guardados tal cual	datos que la aplicación crea y cambia
se leen enteros, se filtran por pocas claves	se agregan, se suman, se ordenan
la estructura cambia más rápido que el esquema	la estructura es estable

Una guía para decidir

`jsonb` es razonable	Mejor columnas o tablas
atributos opcionales, escasos y variables	lo que tiene todo envío
no participan en joins ni en restricciones	lo que se referencia
documentos de otro sistema, guardados tal cual	datos que la aplicación crea y cambia
se leen enteros, se filtran por pocas claves	se agregan, se suman, se ordenan
la estructura cambia más rápido que el esquema	la estructura es estable

Error común: usarlo para no diseñar

"Guardemos todo en `data jsonb` y después vemos." El esquema existe igual, en la aplicación, sin que nadie lo verifique. `jsonb` es para lo que queda *después* del E-R.

SECCIÓN 14

Repaso de las partes 1 y 2

14

Lo que entra en el parcial

Sem.	Tema	Se pide
1	Modelo relacional	superclave, clave candidata, primaria, foránea
2	Álgebra y SQL básico	σ, Π, ρ, $\times$, $\bowtie$, $\cup$, $-$; de álgebra a SQL
3	Conjuntos, `NULL`, agregación	tres valores; `NOT IN`; `HAVING`; `EXISTS`; γ y $\div$
4	Joins, vistas, restricciones	joins externos; `CHECK`, claves foráneas
5	SQL desde un programa y avanzado	`pg`, inyección, transacciones; triggers, recursivas, ventanas
6	Modelo E-R	diagrama, cardinalidades, entidades débiles, tablas
7	Normalización	clausura, claves, cubierta canónica, BCNF, 3NF
8	JSON	solo leer `->`, `->>`, `@>`

Repaso 1: envíos por agencia, incluidas las que no tuvieron

review.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
WHERE s.created_at >= '2026-06-20'
GROUP BY a.agency_id, a.name
ORDER BY shipments, a.name LIMIT 3;
```

psql

```
         name          | shipments
-----------------------+-----------
 Huancavelica - Centro |         1
 Pucallpa - Centro     |         2
 Abancay - Centro      |         3
```

Repaso 1: envíos por agencia, incluidas las que no tuvieron

review.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
WHERE s.created_at >= '2026-06-20'
GROUP BY a.agency_id, a.name
ORDER BY shipments, a.name LIMIT 3;
```

psql

```
         name          | shipments
-----------------------+-----------
 Huancavelica - Centro |         1
 Pucallpa - Centro     |         2
 Abancay - Centro      |         3
```

review.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
   AND s.created_at >= '2026-06-20'
GROUP BY a.agency_id, a.name
ORDER BY shipments, a.name LIMIT 3;
```

psql

```
         name          | shipments
-----------------------+-----------
 Iquitos - Centro      |         0
 Huancavelica - Centro |         1
 Pucallpa - Centro     |         2
```

Sin `LIMIT`: 28 filas contra 29. Con `count(*)`, Iquitos sale con 1.

Repaso 2: refutar una dependencia funcional

review.sql

```sql
SELECT count(*) AS counterexamples
FROM (SELECT origin_agency_id, dest_agency_id, weight_kg
      FROM shipment
      GROUP BY origin_agency_id, dest_agency_id, weight_kg
      HAVING count(DISTINCT price) > 1) AS t;
```

psql

```
 counterexamples
-----------------
            1083
```

- origen, destino, peso $\rightarrow$ `price` es falsa: 1083 contraejemplos.

Repaso 2: refutar una dependencia funcional

review.sql

```sql
SELECT count(*) AS counterexamples
FROM (SELECT origin_agency_id, dest_agency_id, weight_kg
      FROM shipment
      GROUP BY origin_agency_id, dest_agency_id, weight_kg
      HAVING count(DISTINCT price) > 1) AS t;
```

psql

```
 counterexamples
-----------------
            1083
```

- origen, destino, peso $\rightarrow$ `price` es falsa: 1083 contraejemplos.
- Con `declared_value` también a la izquierda: 0. ¿Se cumple? Solo sabemos que los datos de hoy no la contradicen. Lo decide la regla de la tarifa, no los datos.

Repaso 3: avg y los NULL

review.sql

```sql
SELECT count(*) AS shipments, count(declared_value) AS declared,
       round(avg(declared_value), 2) AS avg_declared,
       round(sum(declared_value) / count(*), 2) AS sum_over_all,
       round(avg(coalesce(declared_value, 0)), 2) AS avg_zero
FROM shipment;
```

psql

```
 shipments | declared | avg_declared | sum_over_all | avg_zero
-----------+----------+--------------+--------------+----------
     12000 |     7136 |      1512.33 |       899.33 |   899.33
```

- `avg` ignora los `NULL`. Las otras dos responden *otra* pregunta.
- Sobre cero filas: `count` da 0, `sum` y `avg` dan `NULL`; con `GROUP BY`, cero filas.

Errores frecuentes

1. `NOT IN` con un `NULL` en la subconsulta
2. El filtro de la derecha de un `LEFT JOIN` en el `WHERE`
3. `count(*)` con un join externo
4. `= NULL` en vez de `IS NULL`
5. `sum` sobre cero filas es `NULL`
6. Una columna fuera del `GROUP BY`
7. Natural join con un `name` compartido
8. Filtrar por una window function en su `WHERE`
9. Concatenar SQL; `pool.query('BEGIN')`; `release()` fuera del `finally`
10. Atributos de un muchos a muchos en una entidad
11. Deducir dependencias de los datos
12. "Está en BCNF" sin revisar cada dependencia

Para practicar

`exams/parcial-practica.md`: resuélvanlo en papel, en dos horas; después, `psql`.

SECCIÓN 15

Sesión B: examen parcial

15

Examen parcial

- Semanas 1 a 8: modelo relacional y SQL; diseño de bases de datos.
- Formato (`exams/README.md`):
  - escribir consultas en SQL y en álgebra relacional, sobre un esquema dado
  - leer consultas: `NULL`, duplicados, agregados sobre conjuntos vacíos; leer código con `pg`
  - diseño: E-R y su traducción; dependencias, claves, BCNF y 3NF
  - preguntas cortas
- En SQL importa la idea: un error menor de sintaxis no la invalida.

Para la próxima clase

- P2 (Diseño y aplicación) se entrega la semana 9.
- Leer Silberschatz, secs. 12.1 a 12.4 y 12.6 (almacenamiento físico), para la semana 9.
- Correr `make test` en `code/week-08` y leer los apuntes de la semana.

Pregunta

Chasqui quiere guardar en `details` la placa del vehículo que llevó el envío, "porque a veces cambia". ¿Qué perderían respecto de `trip.vehicle_id`? ¿Qué pregunta de la semana 7 deberían hacerse antes de decidir?