

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Sistemas de bases de datos y el modelo relacional

Modelo relacional y SQL

Gerson Garrido

Semana 1

Hoy

Sesión A

1. El curso: qué vamos a aprender y qué vamos a construir
2. ¿Por qué un DBMS? Chasqui con archivos
3. Niveles de abstracción, esquema e instancia
4. DDL, DML y el motor por dentro

Sesión B

5. Relaciones, tuplas, atributos y dominios
6. Superclave, clave candidata, clave primaria
7. Claves foráneas e integridad referencial
8. El diagrama de esquema de Chasqui

> **Lectura**
>
> Silberschatz, secs. 1.1–1.6 (1.7–1.8 rápido, 1.9 opcional) y 2.1–2.4. PostgreSQL: *What Is PostgreSQL?* y *Architectural Fundamentals*. CMU 15-445, notas de la Lecture #01 hasta las claves.

SECCIÓN 01

El curso

01

Cuatro partes

Modelo relacional y SQL

claves, álgebra, SQL, Node.js

sem. 1–5

Cuatro partes

Modelo relacional y SQL

claves, álgebra, SQL, Node.js

sem. 1–5

Diseño

E-R, dependencias funcionales, formas normales

sem. 6–8

Cuatro partes

Modelo relacional y SQL

claves, álgebra, SQL, Node.js

sem. 1–5

Diseño

E-R, dependencias funcionales, formas normales

sem. 6–8

Almacenamiento y consultas

páginas, B+ trees, joins, optimización

sem. 9–12

Cuatro partes

Modelo relacional y SQL
claves, álgebra, SQL, Node.js
sem. 1–5

Diseño
E-R, dependencias funcionales, formas normales
sem. 6–8

Almacenamiento y consultas
páginas, B+ trees, joins, optimización
sem. 9–12

Transacciones
ACID, locks, MVCC, recuperación, distribuidos
sem. 13–16

La primera mitad usa una base de datos; la segunda abre el motor por dentro.

Cómo funciona el curso

- **Teoría** (4 h): dos sesiones por semana.
- **Laboratorio** (2 h): práctica guiada.
- **Proyecto**: Chasqui, un solo sistema que crece todo el semestre.
- **Lectura**: Silberschatz, *Database System Concepts* (7.ª ed.), y cada semana al menos una lectura gratuita en línea.
- **Herramientas**: PostgreSQL 18 y Node.js 26.

Actividad	Cuándo
Proyecto (P0–P5)	todo el semestre
Laboratorios	cada semana
Examen parcial	semana 8
Examen final	después de la 16

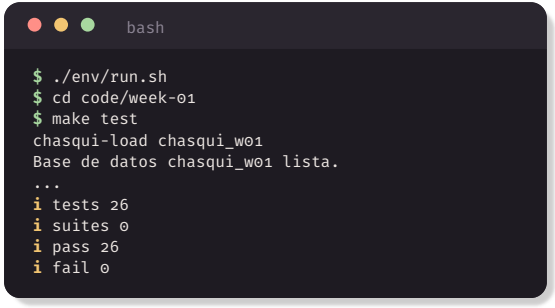
Chasqui: lo que van a construir

	Entrega	Lo que agrega
P0	Calentamiento	Consultas SQL sobre Chasqui; un primer script en Node.js
P1	Esquema y consultas	Restricciones, joins, agregación, subconsultas, vistas
P2	Diseño y aplicación	Modelo E-R normalizado, migración, aplicación con `pg` y transacciones
P3	Motor: almacenamiento	`quipu`: páginas con slots, heap file, buffer pool, B+ tree
P4	Motor: consultas	Operadores con iteradores, joins, ordenamiento externo
P5	Motor: transacciones	Lock manager con 2PL estricto, deadlocks, log de recuperación

Una empresa de encomiendas entre agencias de todo el Perú. Cada entrega debe pasar sus pruebas y las de las anteriores.

El entorno: el mismo para todos

- Un contenedor **Docker** con PostgreSQL 18, `psql` y Node.js 26.
- Igual en Linux, Windows (WSL2) y macOS.
- Los datos quedan en un volumen: no se pierden al salir.
- Se instala en el laboratorio de esta semana.



Salida recortada.

SECCIÓN 02

¿Por qué un DBMS?

02



Base de datos y DBMS

Base de datos *(database)*

Una colección de datos relacionados entre sí, que describen algo del mundo: los envíos de una empresa, las matrículas de una universidad.

Sistema de gestión de bases de datos *(DBMS)*

El software que guarda esa colección y da acceso a ella de forma conveniente y eficiente.

- Detrás de la app del banco, la matrícula, la tienda en línea, los vuelos…
- En el curso: **PostgreSQL**, relacional y de código abierto, nacido en Berkeley.

Chasqui con archivos

Cada agencia lleva un CSV con los envíos que salen de ella. Los datos del remitente se copian en cada fila:

agency-01.csv

```
shipment_id,created_at,sender_dni,sender_name,sender_phone,dest_city,weight_kg,price,status
11,2026-01-26,44165965,Patricia Salazar Cárdenas,981313291,Arequipa,10.20,54.68,delivered
17,2026-05-04,14414736,Camila Medina Cáceres,993060280,Arequipa,17.30,78.35,delivered
23,2026-04-11,15094236,Yolanda Espinoza Ramos,923940626,Trujillo,74.50,103.76,delivered
```

- `files.mjs` arma un archivo por agencia de origen y los usa como un programa hecho a mano.

Chasqui con archivos

Cada agencia lleva un CSV con los envíos que salen de ella. Los datos del remitente se copian en cada fila:

agency-01.csv

```
shipment_id,created_at,sender_dni,sender_name,sender_phone,dest_city,weight_kg,price,status
11,2026-01-26,44165965,Patricia Salazar Cárdenas,981313291,Arequipa,10.20,54.68,delivered
17,2026-05-04,14414736,Camila Medina Cáceres,993060280,Arequipa,17.30,78.35,delivered
23,2026-04-11,15094236,Yolanda Espinoza Ramos,923940626,Trujillo,74.50,103.76,delivered
```

- `files.mjs` arma un archivo por agencia de origen y los usa como un programa hecho a mano.
- ¿Qué podría salir mal?

Lo que sale mal

bash

```
$ node files.mjs
28 archivos, 12000 filas en total
DNI 48479833: 64 filas en 25 archivos, teléfonos 972898189
se corrigieron 9 filas de agency-01.csv; teléfonos ahora: 999000111 972898189
en la base de datos: UPDATE 1
corte de luz reescribiendo agency-02.csv: tenía 974 filas, quedan 487
fila con peso -5.00 agregada sin quejas a agency-03.csv
```

Lo que sale mal

bash

```
$ node files.mjs
28 archivos, 12000 filas en total
DNI 48479833: 64 filas en 25 archivos, teléfonos 972898189
se corrigieron 9 filas de agency-01.csv; teléfonos ahora: 999000111 972898189
en la base de datos: UPDATE 1
corte de luz reescribiendo agency-02.csv: tenía 974 filas, quedan 487
fila con peso -5.00 agregada sin quejas a agency-03.csv
```

- Línea 2: **redundancia**. Línea 3: **inconsistencia**.

Lo que sale mal

bash

```
$ node files.mjs
28 archivos, 12000 filas en total
DNI 48479833: 64 filas en 25 archivos, teléfonos 972898189
se corrigieron 9 filas de agency-01.csv; teléfonos ahora: 999000111 972898189
en la base de datos: UPDATE 1
corte de luz reescribiendo agency-02.csv: tenía 974 filas, quedan 487
fila con peso -5.00 agregada sin quejas a agency-03.csv
```

- Línea 2: **redundancia**. Línea 3: **inconsistencia**.
- Línea 5: falta de **atomicidad**.

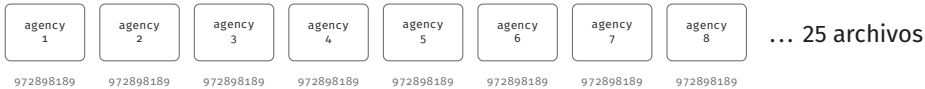
Lo que sale mal

bash

```
$ node files.mjs
28 archivos, 12000 filas en total
DNI 48479833: 64 filas en 25 archivos, teléfonos 972898189
se corrigieron 9 filas de agency-01.csv; teléfonos ahora: 999000111 972898189
en la base de datos: UPDATE 1
corte de luz reescribiendo agency-02.csv: tenía 974 filas, quedan 487
fila con peso -5.00 agregada sin quejas a agency-03.csv
```

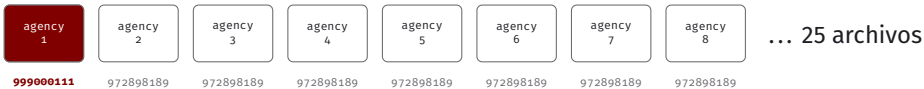
- Línea 2: **redundancia**. Línea 3: **inconsistencia**.
- Línea 5: falta de **atomicidad**.
- Línea 6: **problemas de integridad**.

Redundancia e inconsistencia



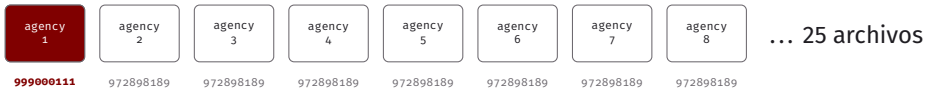
- La misma clienta, 64 envíos desde 25 agencias: su teléfono está copiado 64 veces.

Redundancia e inconsistencia



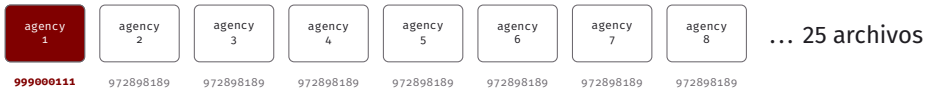
- La misma clienta, 64 envíos desde 25 agencias: su teléfono está copiado 64 veces.
- Avisa en La Victoria que cambió de número. Se corrige `agency-01.csv`; los otros 24 no.

Redundancia e inconsistencia



- La misma clienta, 64 envíos desde 25 agencias: su teléfono está copiado 64 veces.
- Avisa en La Victoria que cambió de número. Se corrige `agency-01.csv`; los otros 24 no.
- Ahora hay dos teléfonos para la misma persona. ¿Cuál es el bueno?

Redundancia e inconsistencia



- La misma clienta, 64 envíos desde 25 agencias: su teléfono está copiado 64 veces.
- Avisa en La Victoria que cambió de número. Se corrige `agency-01.csv`; los otros 24 no.
- Ahora hay dos teléfonos para la misma persona. ¿Cuál es el bueno?
- En la base de datos el teléfono está **una vez**, en `customer`: `UPDATE 1`.

Cada pregunta nueva, un programa nuevo

files.mjs

```mjs
export function countByStatus(dir) {
  const counts = {};
  for (const file of listFiles(dir)) {
    for (const r of readCsv(file))
      counts[r.status] =
        (counts[r.status] ?? 0) + 1;
  }
  return counts;
}
```

examples.sql

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

psql

```
   status   | count
------------+-------
 delivered  | 11115
 in_transit |    66
 registered |   500
 returned   |   319
(4 rows)
```

Las pruebas comprueban que las dos dan lo mismo. La diferencia es cuánto costó cada una.

Integridad: las reglas, en el esquema

- El CSV es texto: aceptó un envío de -5 kg sin quejarse.
- Las reglas (peso positivo, DNI de 8 dígitos, cuatro estados) quedan escondidas en cada programa.

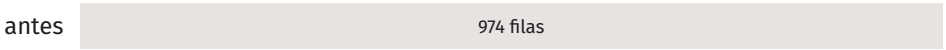
Integridad: las reglas, en el esquema

- El CSV es texto: aceptó un envío de -5 kg sin quejarse.
- Las reglas (peso positivo, DNI de 8 dígitos, cuatro estados) quedan escondidas en cada programa.
- Un DBMS las declara **una vez** y las verifica en **cada** cambio:

psql

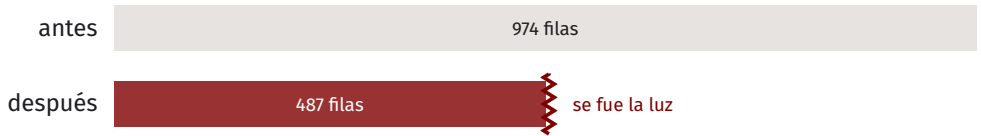
```
chasqui=# INSERT INTO shipment (shipment_id, sender_id, recipient_id, origin_agency_id,
                    dest_agency_id, weight_kg, price, status, created_at)
VALUES (12001, 178, 337, 9, 8, -5, 15.00, 'registered', now());
ERROR:  new row for relation "shipment" violates check constraint "shipment_weight_kg_check"
```

Atomicidad: el corte de luz



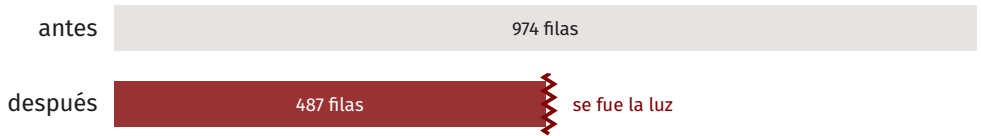
- El programa abre el archivo vacío y reescribe las filas una por una.

Atomicidad: el corte de luz



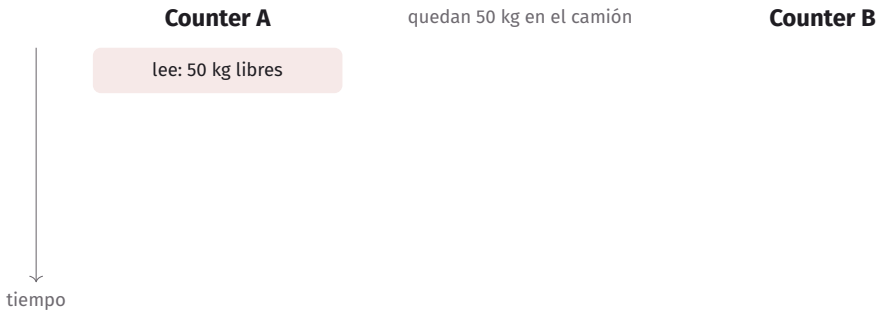
- El programa abre el archivo vacío y reescribe las filas una por una.
- Se necesitaba que el cambio ocurra completo o no ocurra: **atomicidad**.

Atomicidad: el corte de luz

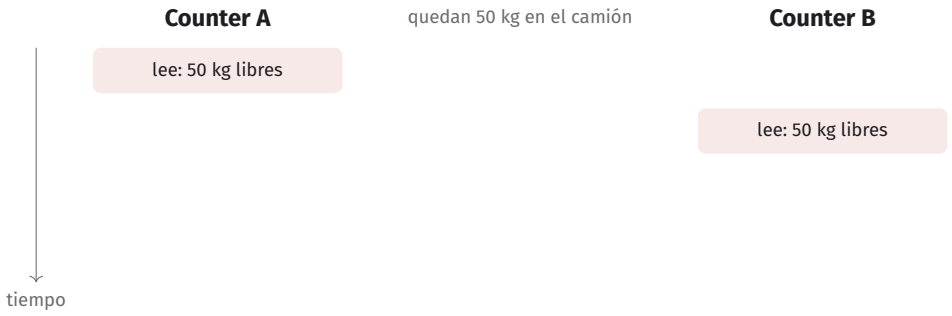


- El programa abre el archivo vacío y reescribe las filas una por una.
- Se necesitaba que el cambio ocurra completo o no ocurra: **atomicidad**.
- En Chasqui: registrar un envío *y* su cobro. No puede quedar uno sin el otro.
- Un DBMS agrupa los cambios en **transacciones** (semanas 13–15).

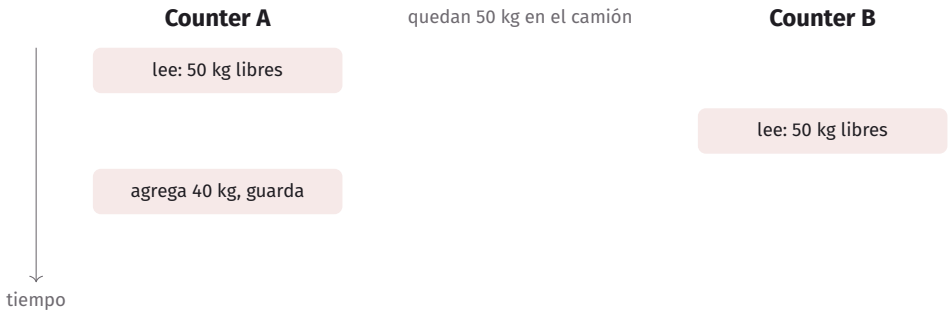
Acceso concurrente: dos counters, un camión



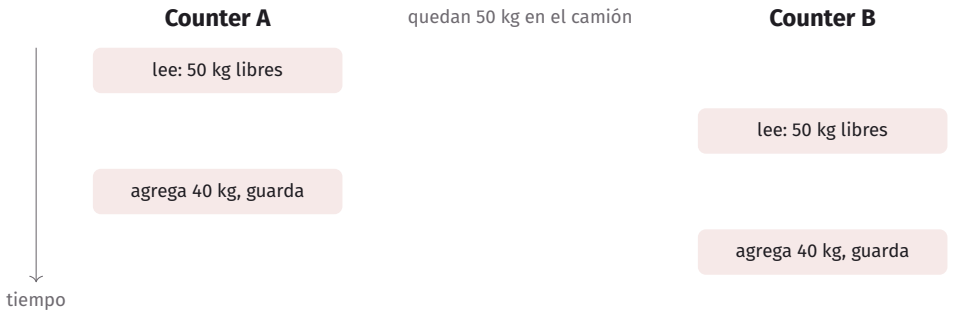
Acceso concurrente: dos counters, un camión



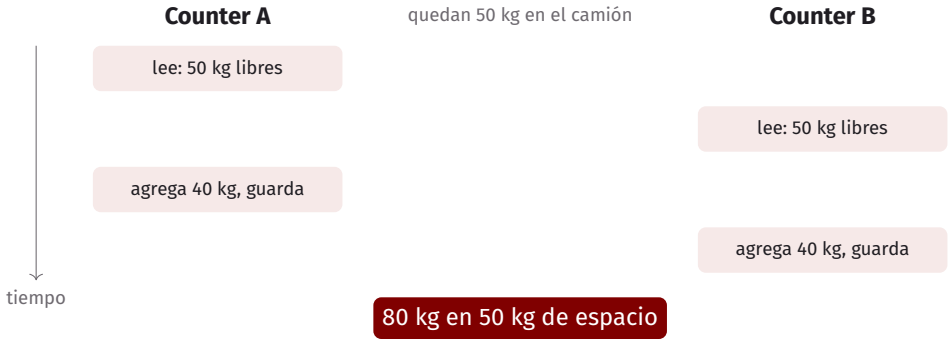
Acceso concurrente: dos counters, un camión



Acceso concurrente: dos counters, un camión



Acceso concurrente: dos counters, un camión



Un DBMS ofrece **control de concurrencia**: las transacciones se comportan como si se hubieran ejecutado una detrás de otra (semanas 13 y 14).

Seguridad, y lo que hace un DBMS

- Quien lee el CSV lo lee todo: DNI, teléfono, valor declarado.
- Un DBMS da **permisos** por tabla, columna y operación, y **vistas** con una parte de los datos (semana 4).

Un DBMS

- guarda los datos **una vez**, con un esquema;
- responde preguntas nuevas con **consultas**;
- verifica **restricciones**;
- ejecuta **transacciones** atómicas;
- controla la **concurrencia**;
- controla **quién ve qué**.

Seguridad, y lo que hace un DBMS

- Quien lee el CSV lo lee todo: DNI, teléfono, valor declarado.
- Un DBMS da **permisos** por tabla, columna y operación, y **vistas** con una parte de los datos (semana 4).
- Todo se puede resolver a mano con archivos. Pero habría que resolverlo todo, en cada programa, y bien.

Un DBMS

- guarda los datos **una vez**, con un esquema;
- responde preguntas nuevas con **consultas**;
- verifica **restricciones**;
- ejecuta **transacciones** atómicas;
- controla la **concurrencia**;
- controla **quién ve qué**.

SECCIÓN 03

La vista de los datos

03

Modelos de datos

Modelo de datos *(data model)*

Herramientas conceptuales para describir los datos, sus relaciones, su significado y sus restricciones.

- **Relacional**: tablas. El de casi todos los DBMS y el de este curso.
- **Entidad-relación (E-R)**: entidades y relaciones entre ellas; para diseñar (semana 6).
- **Semiestructurado**: cada dato con sus propios atributos; JSON (semana 8).
- **Basado en objetos**: herencia e identidad de objetos llevadas a las tablas.

Ojo con *relación*

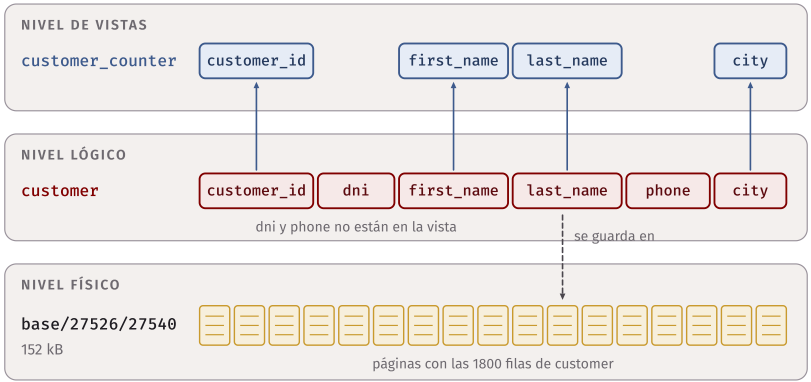
En el modelo relacional es una **tabla** *(relation)*. En E-R es un **vínculo** *(relationship)*.

Tres niveles de abstracción



- **Físico**: cómo están guardados los bytes. El más bajo y el más complicado.
- **Lógico**: qué datos hay y cómo se relacionan. Aquí se escriben las consultas.
- **De vistas**: lo que ve cada usuario, una parte de la base de datos.

Los tres niveles de `customer`



El nivel físico y el de vistas en PostgreSQL

examples.sql

```sql
SELECT pg_relation_filepath('customer')
    AS file,
      pg_size_pretty(pg_relation_size('
    customer')) AS size;
```

psql

```
      file       |  size
-----------------+--------
 base/27526/27540 | 152 kB
(1 row)
```

Los números cambian en cada máquina.

examples.sql

```sql
CREATE VIEW customer_counter AS
  SELECT customer_id, first_name,
      last_name, city
  FROM customer;
SELECT * FROM customer_counter ORDER BY
      customer_id LIMIT 3;
```

psql

```
 customer_id | first_name |    last_name    |   city
-------------+------------+-----------------+----------
           1 | Wilmer     | Chávez Ríos     | Ayacucho
           2 | Piero      | García Castillo | Puno
           3 | Milagros   | Ramos Herrera   | Lima
(3 rows)
```

Esquema e instancia

Esquema *(schema)*

El diseño: qué tablas, qué columnas, qué tipos, qué reglas. Cambia poco.

Instancia *(instance)*

El contenido en un momento dado: las filas de ahora. Cambia siempre.

- Como el tipo de una variable frente a su valor.
- Esquema **físico**, esquema **lógico** y **subesquemas** (uno por vista).
- En Chasqui: el esquema es `schema.sql`; la instancia, 29 agencias, 1800 clientes, 12 000 envíos…

Esquema e instancia

Esquema *(schema)*

El diseño: qué tablas, qué columnas, qué tipos, qué reglas. Cambia poco.

Instancia *(instance)*

El contenido en un momento dado: las filas de ahora. Cambia siempre.

- Como el tipo de una variable frente a su valor.
- Esquema **físico**, esquema **lógico** y **subesquemas** (uno por vista).
- En Chasqui: el esquema es `schema.sql`; la instancia, 29 agencias, 1800 clientes, 12 000 envíos…
- En PostgreSQL, *schema* es además un namespace: las tablas de Chasqui están en `public`.

Independencia física de los datos

examples.sql

```sql
BEGIN;
SELECT count(*) FROM customer WHERE city = 'Arequipa';
CREATE INDEX customer_city_idx ON customer (city);
SELECT count(*) FROM customer WHERE city = 'Arequipa';
ROLLBACK;
```

EXPLAIN antes

```
Aggregate
  ->  Seq Scan on customer
        Filter: (city = 'Arequipa'::text)
```

- Las dos consultas devuelven 89.

`EXPLAIN (COSTS OFF)`; los planes se leen en la semana 12.

Independencia física de los datos

examples.sql

```sql
BEGIN;
SELECT count(*) FROM customer WHERE city = '
    Arequipa';
CREATE INDEX customer_city_idx ON customer (
    city);
SELECT count(*) FROM customer WHERE city = '
    Arequipa';
ROLLBACK;
```

- Las dos consultas devuelven 89.
- El índice es un cambio **físico**; la consulta no se enteró.

EXPLAIN antes

```
Aggregate
  ->  Seq Scan on customer
        Filter: (city = 'Arequipa'::text)
```

EXPLAIN después

```
Aggregate
  ->  Index Only Scan using customer_city_idx on customer
        Index Cond: (city = 'Arequipa'::text)
```

`EXPLAIN (COSTS OFF)`; los planes se leen en la semana 12.

Independencia física de los datos

examples.sql

```sql
BEGIN;
SELECT count(*) FROM customer WHERE city = '
    Arequipa';
CREATE INDEX customer_city_idx ON customer (
    city);
SELECT count(*) FROM customer WHERE city = '
    Arequipa';
ROLLBACK;
```

- Las dos consultas devuelven 89.
- El índice es un cambio **físico**; la consulta no se enteró.
- **Independencia física**: cambiar el esquema físico sin reescribir los programas.

EXPLAIN antes

```
Aggregate
  ->  Seq Scan on customer
        Filter: (city = 'Arequipa'::text)
```

EXPLAIN después

```
Aggregate
  ->  Index Only Scan using customer_city_idx on customer
        Index Cond: (city = 'Arequipa'::text)
```

`EXPLAIN (COSTS OFF)`; los planes se leen en la semana 12.

SECCIÓN 04

Lenguajes y motor

04

DDL: definir el esquema

schema.sql

```sql
CREATE TABLE customer (
  customer_id integer PRIMARY KEY,
  dni  char(8) NOT NULL UNIQUE
       CHECK (dni ~ '^[0-9]{8}$'),
  first_name  text NOT NULL,
  last_name   text NOT NULL,
  phone       text
       CHECK (phone ~ '^9[0-9]{8}$'),
  city        text NOT NULL
);
```

- El **DDL** *(data-definition language)* describe el esquema.

DDL: definir el esquema

schema.sql

```sql
CREATE TABLE customer (
  customer_id integer PRIMARY KEY,
  dni  char(8) NOT NULL UNIQUE
       CHECK (dni ~ '^[0-9]{8}$'),
  first_name  text NOT NULL,
  last_name   text NOT NULL,
  phone       text
       CHECK (phone ~ '^9[0-9]{8}$'),
  city        text NOT NULL
);
```

- El **DDL** *(data-definition language)* describe el esquema.
- Y sus **restricciones de integridad**: de dominio, referencial, autorización.

DDL: definir el esquema

schema.sql

```sql
CREATE TABLE customer (
  customer_id integer PRIMARY KEY,
  dni  char(8) NOT NULL UNIQUE
       CHECK (dni ~ '^[0-9]{8}$'),
  first_name  text NOT NULL,
  last_name   text NOT NULL,
  phone       text
       CHECK (phone ~ '^9[0-9]{8}$'),
  city        text NOT NULL
);
```

- El **DDL** *(data-definition language)* describe el esquema.
- Y sus **restricciones de integridad**: de dominio, referencial, autorización.
- El DBMS guarda el esquema en el **diccionario de datos** (catálogo): metadatos.
- `\d customer` consulta el catálogo.

DML: consultar y modificar

- El **DML** *(data-manipulation language)* lee, inserta, borra y modifica la instancia.
- La parte que solo lee es el **lenguaje de consultas** *(query language)*.

Procedural

Dice qué datos quiere y cómo obtenerlos, paso a paso.

`countByStatus`: abrir cada archivo, recorrer, sumar.

Declarativo

Dice qué datos quiere; el motor decide cómo.

`GROUP BY status`.

DML: consultar y modificar

- El **DML** *(data-manipulation language)* lee, inserta, borra y modifica la instancia.
- La parte que solo lee es el **lenguaje de consultas** *(query language)*.

Procedural

Dice qué datos quiere y cómo obtenerlos, paso a paso.

`countByStatus`: abrir cada archivo, recorrer, sumar.

Declarativo

Dice qué datos quiere; el motor decide cómo.

`GROUP BY status`.

SQL es declarativo

La misma consulta se ejecutó con y sin índice: lo eligió el **query optimizer**. Eso hace posible la independencia física.

SQL desde un programa

Node.js

```
1  import pg from 'pg';
2
3  const pool = new pg.Pool();
4  const { rows } = await pool.query(
5    'SELECT count(*) AS n FROM shipment WHERE
         status = $1',
6    ['delivered'],
7  );
8  console.log(rows[0], typeof rows[0].n);
9  await pool.end();
```

bash

```
{ n: '11115' } string
```

- El **driver** `pg` habla el protocolo de PostgreSQL.

SQL desde un programa

Node.js

```
import pg from 'pg';

const pool = new pg.Pool();
const { rows } = await pool.query(
  'SELECT count(*) AS n FROM shipment WHERE
      status = $1',
  ['delivered'],
);
console.log(rows[0], typeof rows[0].n);
await pool.end();
```

bash

```
{ n: '11115' } string
```

- El **driver** `pg` habla el protocolo de PostgreSQL.
- `$1` es un parámetro: el valor viaja aparte. Siempre así (semana 5).

SQL desde un programa

Node.js

```
1  import pg from 'pg';
2
3  const pool = new pg.Pool();
4  const { rows } = await pool.query(
5    'SELECT count(*) AS n FROM shipment WHERE
         status = $1',
6    ['delivered'],
7  );
8  console.log(rows[0], typeof rows[0].n);
9  await pool.end();
```

bash

```
{ n: '11115' } string
```

- El **driver** `pg` habla el protocolo de PostgreSQL.
- `$1` es un parámetro: el valor viaja aparte. Siempre así (semana 5).
- `count(*)` es `bigint`, 64 bits: llega como texto. Un `Number` no los representa todos.

El motor por dentro: el mapa del curso

Procesador de consultas

DDL interpreter, DML compiler, query evaluation engine

sem. 11–12

El motor por dentro: el mapa del curso

Procesador de consultas

DDL interpreter, DML compiler, query evaluation engine

sem. 11–12

Gestor de almacenamiento

file manager, buffer manager, índices, diccionario

sem. 9–10

El motor por dentro: el mapa del curso

Procesador de consultas

DDL interpreter, DML compiler, query evaluation engine

sem. 11–12

Gestor de almacenamiento

file manager, buffer manager, índices, diccionario

sem. 9–10

Gestor de transacciones

atomicidad y durabilidad con el log; control de concurrencia

sem. 13–15

disco: archivos de datos, índices, diccionario de datos, log

Desde la semana 9 construyen versiones propias de estas piezas en `quipu`.

Cliente-servidor

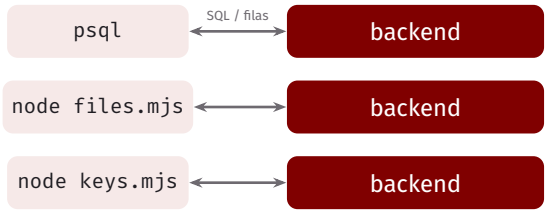
```
psql
```

```
node files.mjs
```

```
node keys.mjs
```

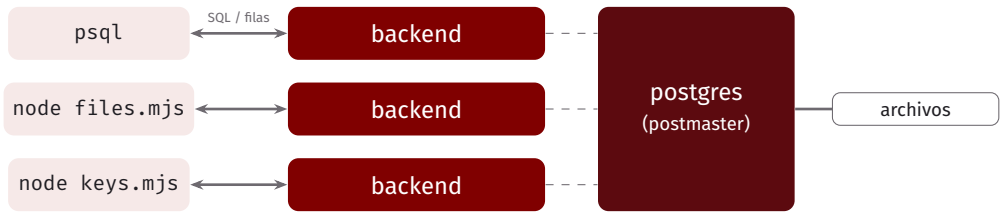
- El **servidor** administra los archivos y ejecuta el SQL. Los **clientes** le envían SQL por una conexión.

Cliente-servidor



- El **servidor** administra los archivos y ejecuta el SQL. Los **clientes** le envían SQL por una conexión.
- PostgreSQL crea un proceso servidor por conexión (un *backend*).

Cliente-servidor



- El **servidor** administra los archivos y ejecuta el SQL. Los **clientes** le envían SQL por una conexión.
- PostgreSQL crea un proceso servidor por conexión (un *backend*).

Los procesos del servidor

bash

```
$ ps -o pid,cmd -u postgres
  PID CMD
   19 /usr/lib/postgresql/18/bin/postgres -D /var/lib/postgresql/18/docker
   20 postgres: io worker 0
   21 postgres: io worker 1
   22 postgres: io worker 2
   23 postgres: checkpointer
   24 postgres: background writer
   26 postgres: walwriter
   27 postgres: autovacuum launcher
   28 postgres: logical replication launcher
 1039 postgres: postgres chasqui_w01 [local] SELECT
```

- 19: el servidor principal. 20–28: procesos auxiliares (E/S asíncrona, escritura a disco, log).

Los procesos del servidor

bash

```
$ ps -o pid,cmd -u postgres
  PID CMD
   19 /usr/lib/postgresql/18/bin/postgres -D /var/lib/postgresql/18/docker
   20 postgres: io worker 0
   21 postgres: io worker 1
   22 postgres: io worker 2
   23 postgres: checkpointer
   24 postgres: background writer
   26 postgres: walwriter
   27 postgres: autovacuum launcher
   28 postgres: logical replication launcher
 1039 postgres: postgres chasqui_w01 [local] SELECT
```

- 19: el servidor principal. 20–28: procesos auxiliares (E/S asíncrona, escritura a disco, log).
- 1039: el backend de un `psql` conectado a `chasqui_w01`, ejecutando un `SELECT`.

Arquitecturas y usuarios

Arquitectura de la aplicación

- **Dos capas**: el programa se conecta directo al DBMS. Los scripts del curso.
- **Tres capas**: el usuario habla con un servidor de aplicaciones, y este con el DBMS. Las aplicaciones web.

Usuarios

- **Finales**: usan una aplicación (el counter).
- **Programadores de aplicaciones**: ustedes, en P2.
- **Avanzados**: escriben SQL directamente.
- **DBA**: define el esquema y la organización física, da permisos, hace backups.

Historia (sec. 1.9): Codd propone el modelo relacional en IBM en 1970; de Ingres sale POSTGRES (Stonebraker, Berkeley), que en 1996 se vuelve PostgreSQL.

SECCIÓN 05

El modelo relacional

05

Relación, tupla, atributo

customer_id	dni	first_name	last_name	phone	city
1	61830975	Wilmer	Chávez Ríos	909975104	Ayacucho
2	17232205	Piero	García Castillo	946039842	Puno
3	36948162	Milagros	Ramos Herrera		Lima

- Tabla = **relación** *(relation)*; fila = **tupla** *(tuple)*; columna = **atributo** *(attribute)*.

Relación, tupla, atributo

customer_id	dni	first_name	last_name	phone	city
1	61830975	Wilmer	Chávez Ríos	909975104	Ayacucho
2	17232205	Piero	García Castillo	946039842	Puno
3	36948162	Milagros	Ramos Herrera		Lima

tupla

- Tabla = **relación** *(relation)*; fila = **tupla** *(tuple)*; columna = **atributo** *(attribute)*.

Relación, tupla, atributo

customer_id	dni	first_name	last_name	phone	city
1	61830975	Wilmer	Chávez Ríos	909975104	Ayacucho
2	17232205	Piero	García Castillo	946039842	Puno
3	36948162	Milagros	Ramos Herrera		Lima

tupla

atributo

- Tabla = **relación** *(relation)*; fila = **tupla** *(tuple)*; columna = **atributo** *(attribute)*.
- Cada atributo tiene un **dominio** *(domain)*: los valores permitidos. `dni`: cadenas de 8 dígitos.
- El teléfono del cliente 3 está vacío: es `NULL`.

La definición formal

Relación

Sean $D_1, \ldots, D_n$ los dominios de los atributos $A_1, \ldots, A_n$. Una relación r sobre ellos es un subconjunto del producto cartesiano:

$$r \subseteq D_1 \times D_2 \times \cdots \times D_n$$

- Cada tupla es una lista $(v_1, \ldots, v_n)$ con $v_i \in D_i$.
- $t[A]$ es el valor de la tupla t en el atributo A: $t[\texttt{city}] = \texttt{Ayacucho}$.

La definición formal

Relación

Sean $D_1, \ldots, D_n$ los dominios de los atributos $A_1, \ldots, A_n$. Una relación r sobre ellos es un subconjunto del producto cartesiano:

$$r \subseteq D_1 \times D_2 \times \cdots \times D_n$$

- Cada tupla es una lista $(v_1, \ldots, v_n)$ con $v_i \in D_i$.
- $t[A]$ es el valor de la tupla t en el atributo A: $t[\texttt{city}] = \texttt{Ayacucho}$.
- **Esquema de relación**: `customer(customer_id, dni, first_name, last_name, phone, city)`.
- **Instancia de relación**: las 1800 tuplas de hoy. Se escribe $r(R)$: la relación r con esquema R.

El esquema de una relación en `psql`

psql

```
chasqui=# \d customer
                    Table "public.customer"
   Column    |     Type      | Collation | Nullable | Default
-------------+---------------+-----------+----------+---------
 customer_id | integer       |           | not null |
 dni         | character(8)  |           | not null |
 first_name  | text          |           | not null |
 last_name   | text          |           | not null |
 phone       | text          |           |          |
 city        | text          |           | not null |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
    "customer_dni_key" UNIQUE CONSTRAINT, btree (dni)
Check constraints:
    "customer_dni_check" CHECK (dni ~ '^[0-9]{8}$'::text)
    "customer_phone_check" CHECK (phone ~ '^9[0-9]{8}$'::text)
Referenced by:
    TABLE "shipment" CONSTRAINT "shipment_recipient_id_fkey" FOREIGN KEY (recipient_id) REFERENCES customer(customer_id)
    TABLE "shipment" CONSTRAINT "shipment_sender_id_fkey" FOREIGN KEY (sender_id) REFERENCES customer(customer_id)
```

Columnas y tipos: el esquema de la relación. Lo demás: restricciones y detalles físicos.

Dominios atómicos

- Un atributo guarda un valor, que el modelo trata como indivisible.

customer_id	phone
1	909975104
1	946039842

`customer_phone`, hipotética

En la semana 7 esto se llama *primera forma normal*.

Dominios atómicos

- Un atributo guarda un valor, que el modelo trata como indivisible.
- Nada de `'909975104,946039842'` en `phone`.
- Dos teléfonos: otra relación `customer_phone(customer_id, phone)`.

customer_id	phone
1	909975104
1	946039842

`customer_phone`, hipotética

En la semana 7 esto se llama *primera forma normal*.

Dominios atómicos

- Un atributo guarda un valor, que el modelo trata como indivisible.
- Nada de `'909975104,946039842'` en `phone`.
- Dos teléfonos: otra relación `customer_phone(customer_id, phone)`.
- *Atómico* depende del **uso**: Chasqui nunca consulta el tercer dígito de un DNI.

customer_id	phone
1	909975104
1	946039842

`customer_phone`, hipotética

En la semana 7 esto se llama *primera forma normal*.

Una relación es un conjunto

- **Sin orden** entre las tuplas. Sin `ORDER BY`, el orden de las filas no está garantizado.
- **Sin tuplas repetidas**.

Una relación es un conjunto

- **Sin orden** entre las tuplas. Sin `ORDER BY`, el orden de las filas no está garantizado.
- **Sin tuplas repetidas**.
- Las tablas de SQL son multiconjuntos *(bags)*: el resultado de una consulta puede repetir filas.

examples.sql

```sql
SELECT count(*) AS rows, count(DISTINCT
    city) AS distinct_cities
FROM customer;
```

psql

```
 rows | distinct_cities
------+-----------------
 1800 |              26
(1 row)
```

Una relación es un conjunto

- **Sin orden** entre las tuplas. Sin `ORDER BY`, el orden de las filas no está garantizado.
- **Sin tuplas repetidas**.
- Las tablas de SQL son multiconjuntos *(bags)*: el resultado de una consulta puede repetir filas.
- `SELECT city FROM customer` da 1800 filas; en el modelo relacional serían 26 tuplas. Semana 2.

examples.sql

```sql
SELECT count(*) AS rows, count(DISTINCT
    city) AS distinct_cities
FROM customer;
```

psql

```
 rows | distinct_cities
------+-----------------
 1800 |              26
(1 row)
```

`NULL`, primera mirada

- `NULL`: el valor es desconocido o no aplica.
- `phone`: no conocemos el teléfono.
- `delivered_at`: el envío todavía no se entregó.

psql

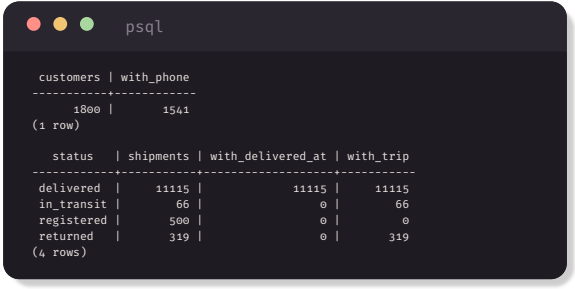
```
 customers | with_phone
-----------+------------
      1800 |       1541
(1 row)

   status   | shipments | with_delivered_at | with_trip
------------+-----------+-------------------+-----------
 delivered  |     11115 |             11115 |     11115
 in_transit |        66 |                 0 |        66
 registered |       500 |                 0 |         0
 returned   |       319 |                 0 |       319
(4 rows)
```

`count(columna)` no cuenta los `NULL`.

`NULL`, primera mirada

- `NULL`: el valor es desconocido o no aplica.
- `phone`: no conocemos el teléfono.
- `delivered_at`: el envío todavía no se entregó.
- No es cero, ni ”, ni falso. Se pregunta con `IS NULL`. Semana 3.



`count(columna)` no cuenta los `NULL`.

El esquema de Chasqui

- agency(agency_id, name, city, region)
- customer(customer_id, dni, first_name, last_name, phone, city)
- vehicle(vehicle_id, plate, kind, capacity_kg)
- route(route_id, origin_agency_id, dest_agency_id, distance_km, base_fare)
- trip(trip_id, route_id, vehicle_id, departure_at, arrival_at)
- shipment(shipment_id, sender_id, recipient_id, origin_agency_id, dest_agency_id, trip_id, weight_kg, declared_value, price, status, created_at, delivered_at)
- shipment_event(shipment_id, seq, at, agency_id, kind)

En palabras

Una **ruta** une dos agencias; un **viaje** es una salida de un **vehículo** por una ruta; un **envío** tiene remitente y destinatario (clientes), origen y destino (agencias), y quizá un viaje; cada envío acumula **eventos**.

SECCIÓN 06

Claves

06

Superclave

Superclave *(superkey)*

$K \subseteq R$ es superclave de R si, en toda instancia válida r de R,

$$t_1, t_2 \in r,\ t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$$

- En `customer`: `{customer_id}`, `{dni}`, `{customer_id, city}`, ...

Superclave

Superclave *(superkey)*

$K \subseteq R$ es superclave de R si, en toda instancia válida r de R,

$$t_1, t_2 \in r,\ t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$$

- En `customer`: `{customer_id}`, `{dni}`, `{customer_id, city}`, ...
- Todo **superconjunto** de una superclave es superclave.

Superclave

Superclave *(superkey)*

$K \subseteq R$ es superclave de R si, en toda instancia válida r de R,

$$t_1, t_2 \in r,\ t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$$

- En `customer`: `{customer_id}`, `{dni}`, `{customer_id, city}`, ...
- Todo **superconjunto** de una superclave es superclave.
- El conjunto de **todos** los atributos es superclave: no hay tuplas repetidas.

Superclave

Superclave *(superkey)*

$K \subseteq R$ es superclave de R si, en toda instancia válida r de R,

$$t_1, t_2 \in r,\ t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$$

- En `customer`: `{customer_id}`, `{dni}`, `{customer_id, city}`, ...
- Todo **superconjunto** de una superclave es superclave.
- El conjunto de **todos** los atributos es superclave: no hay tuplas repetidas.
- `{first_name, last_name}` no: dos clientes pueden llamarse igual.

Superclave

Superclave *(superkey)*

$K \subseteq R$ es superclave de R si, en toda instancia válida r de R,

$$t_1, t_2 \in r,\ t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$$

- En `customer`: `{customer_id}`, `{dni}`, `{customer_id, city}`, ...
- Todo **superconjunto** de una superclave es superclave.
- El conjunto de **todos** los atributos es superclave: no hay tuplas repetidas.
- `{first_name, last_name}` no: dos clientes pueden llamarse igual.

Toda instancia válida: una superclave es una **regla** del negocio, no una observación sobre los datos de hoy.

Clave candidata

Clave candidata *(candidate key)*
Una superclave mínima: ninguno de sus subconjuntos propios es superclave.

En `customer`	¿Superclave?	¿Candidata?
`{customer_id}`	sí	sí
`{dni}`	sí	sí

Clave candidata

Clave candidata *(candidate key)*
Una superclave mínima: ninguno de sus subconjuntos propios es superclave.

En `customer`	¿Superclave?	¿Candidata?
`{customer_id}`	sí	sí
`{dni}`	sí	sí
`{customer_id, city}`	sí	no: sobra `city`
`{dni, first_name}`	sí	no: sobra `first_name`

Clave candidata

Clave candidata *(candidate key)*
Una superclave mínima: ninguno de sus subconjuntos propios es superclave.

En `customer`	¿Superclave?	¿Candidata?
`{customer_id}`	sí	sí
`{dni}`	sí	sí
`{customer_id, city}`	sí	no: sobra `city`
`{dni, first_name}`	sí	no: sobra `first_name`
`{first_name, last_name}`	no	no

Mínima no es *la de menos atributos*: una relación puede tener varias candidatas, de tamaños distintos.

`route`: dos claves candidatas

- `{route_id}`.
- `{origin_agency_id, dest_agency_id}`: a lo más una ruta por par ordenado.

bash

```
$ node keys.mjs
...
route: PK {route_id}  UNIQUE {origin_agency_id, dest_agency_id}
  dest_agency_id -> agency
  origin_agency_id -> agency
...
```

psql

```
 route_id | origin_agency_id | dest_agency_id | distance_km | base_fare
----------+------------------+----------------+-------------+-----------
        1 |                1 |              7 |         555 |     33.00
        2 |                7 |              1 |         555 |     33.00
        3 |                1 |             11 |         368 |     24.50
(3 rows)
```

`route`: dos claves candidatas

- `{route_id}`.
- `{origin_agency_id, dest_agency_id}`: a lo más una ruta por par ordenado.
- Ninguno de los dos atributos basta solo: de La Victoria (agencia 1) salen 14 rutas.

bash

```
$ node keys.mjs
...
route: PK {route_id}  UNIQUE {origin_agency_id, dest_agency_id}
  dest_agency_id -> agency
  origin_agency_id -> agency
...
```

psql

```
 route_id | origin_agency_id | dest_agency_id | distance_km | base_fare
----------+------------------+----------------+-------------+-----------
        1 |                1 |              7 |         555 |     33.00
        2 |                7 |              1 |         555 |     33.00
        3 |                1 |             11 |         368 |     24.50
(3 rows)
```

`route`: dos claves candidatas

- `{route_id}`.
- `{origin_agency_id, dest_agency_id}`: a lo más una ruta por par ordenado.
- Ninguno de los dos atributos basta solo: de La Victoria (agencia 1) salen 14 rutas.
- Las rutas 1 y 2 son distintas: el par es ordenado.

bash

```
$ node keys.mjs
...
route: PK {route_id}  UNIQUE {origin_agency_id, dest_agency_id}
  dest_agency_id -> agency
  origin_agency_id -> agency
...
```

psql

```
 route_id | origin_agency_id | dest_agency_id | distance_km | base_fare
----------+------------------+----------------+-------------+-----------
        1 |                1 |              7 |         555 |     33.00
        2 |                7 |              1 |         555 |     33.00
        3 |                1 |             11 |         368 |     24.50
(3 rows)
```

Superclave, clave candidata y clave primaria

Concepto	Qué cumple	En `customer`
Superclave	Identifica a cada tupla	`{customer_id, dni}`
Clave candidata	Superclave mínima	`{customer_id}`, `{dni}`
Clave primaria	La candidata que elegimos	`customer_id`

Superclave, clave candidata y clave primaria

Concepto	Qué cumple	En `customer`
Superclave	Identifica a cada tupla	`{customer_id, dni}`
Clave candidata	Superclave mínima	`{customer_id}`, `{dni}`
Clave primaria	La candidata que elegimos	`customer_id`

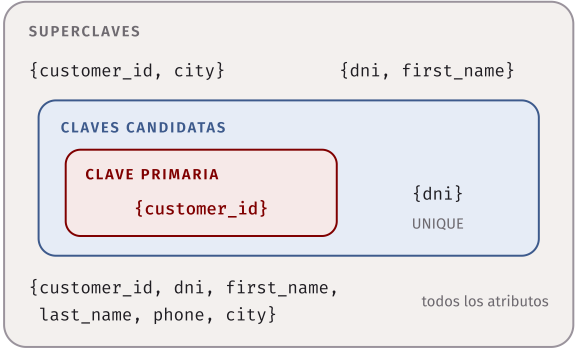
- ¿Por qué `customer_id` y no `dni`? La clave primaria debe ser un valor que no cambie nunca, o casi nunca. Un DNI mal registrado se corrige; si mañana se aceptan extranjeros, `dni` tendrá que cambiar.

Superclave, clave candidata y clave primaria

Concepto	Qué cumple	En `customer`
Superclave	Identifica a cada tupla	`{customer_id, dni}`
Clave candidata	Superclave mínima	`{customer_id}`, `{dni}`
Clave primaria	La candidata que elegimos	`customer_id`

- ¿Por qué `customer_id` y no `dni`? La clave primaria debe ser un valor que no cambie nunca, o casi nunca. Un DNI mal registrado se corrige; si mañana se aceptan extranjeros, `dni` tendrá que cambiar.
- `dni` no se pierde: se declara `UNIQUE` y el motor también la hace cumplir.
- Los atributos de la clave primaria se subrayan en el esquema.

Las claves de `customer`



NO SON SUPERCLAVES

`{first_name, last_name}`

hay dos Christian Aguilar Vásquez

`{phone}`

dos clientes pueden
compartir un teléfono

Clave primaria compuesta: `shipment_event`

psql

```
 shipment_id | seq |          at           | agency_id |   kind
-------------+-----+-----------------------+-----------+-----------
           1 |   1 | 2026-01-28 19:46:00-05 |         9 | received
           1 |   2 | 2026-02-02 12:00:00-05 |         9 | departed
           1 |   3 | 2026-02-02 16:30:00-05 |         8 | arrived
           1 |   4 | 2026-02-04 18:30:00-05 |         8 | delivered
(4 rows)
```

psql

```
 events | shipments | distinct_pairs
--------+-----------+----------------
  46421 |     12000 |          46421
(1 row)
```

- Clave primaria (`shipment_id`, `seq`): **compuesta**.

Clave primaria compuesta: `shipment_event`

psql

```
 shipment_id | seq |           at           | agency_id |   kind
-------------+-----+------------------------+-----------+-----------
           1 |   1 | 2026-01-28 19:46:00-05 |         9 | received
           1 |   2 | 2026-02-02 12:00:00-05 |         9 | departed
           1 |   3 | 2026-02-02 16:30:00-05 |         8 | arrived
           1 |   4 | 2026-02-04 18:30:00-05 |         8 | delivered
(4 rows)
```

psql

```
 events | shipments | distinct_pairs
--------+-----------+---------------
  46421 |     12000 |          46421
(1 row)
```

- Clave primaria (`shipment_id`, `seq`): **compuesta**.
- `shipment_id` solo no basta: 46 421 eventos de 12 000 envíos.
- `seq` solo tampoco: cada envío tiene su evento 1.

Clave primaria compuesta: `shipment_event`

psql

```
 shipment_id | seq |          at           | agency_id |   kind
-------------+-----+-----------------------+-----------+-----------
           1 |   1 | 2026-01-28 19:46:00-05 |         9 | received
           1 |   2 | 2026-02-02 12:00:00-05 |         9 | departed
           1 |   3 | 2026-02-02 16:30:00-05 |         8 | arrived
           1 |   4 | 2026-02-04 18:30:00-05 |         8 | delivered
(4 rows)
```

psql

```
 events | shipments | distinct_pairs
--------+-----------+---------------
  46421 |     12000 |          46421
(1 row)
```

- Clave primaria (`shipment_id`, `seq`): **compuesta**.
- `shipment_id` solo no basta: 46 421 eventos de 12 000 envíos.
- `seq` solo tampoco: cada envío tiene su evento 1.

En PostgreSQL

`PRIMARY KEY` = `UNIQUE` + `NOT NULL`, una por tabla. Para hacerla cumplir, el motor crea un índice B+ tree (`btree`).

Comprobar una clave con los datos

examples.sql

```sql
SELECT dni, count(*)
FROM customer
GROUP BY dni
HAVING count(*) > 1;
```

psql

```
 dni | count
-----+-------
(0 rows)
```

Agrupar por los atributos y buscar grupos de más de una fila (`GROUP BY` y `HAVING`: semana 3).

examples.sql

```sql
SELECT first_name, last_name, count(*)
FROM customer
GROUP BY first_name, last_name
HAVING count(*) > 1
ORDER BY last_name, first_name LIMIT 5;
```

psql

```
 first_name |    last_name    | count
------------+-----------------+-------
 Christian  | Aguilar Vásquez |     2
 Piero      | Apaza Huamán    |     2
 Julia      | Ccahua Sánchez  |     2
 Christian  | Ccori Ccahua    |     2
 Camila     | Condori Cruz    |     2
(5 rows)
```

Refutar no es demostrar

`{first_name, last_name}`

La consulta encontró dos tuplas distintas que coinciden. Un contraejemplo refuta la superclave.

`{dni}`

La consulta no encontró contraejemplos *hoy*. Eso no demuestra nada sobre las demás instancias.

Refutar no es demostrar

`{first_name, last_name}`

La consulta encontró dos tuplas distintas que coinciden. Un contraejemplo refuta la superclave.

`{dni}`

La consulta no encontró contraejemplos *hoy*. Eso no demuestra nada sobre las demás instancias.

Idea clave

Que `{dni}` es clave lo sabemos por el **significado** del DNI, y lo garantiza la restricción `UNIQUE` del esquema, no la consulta.

Error común: deducir una clave de los datos de hoy

examples.sql

```sql
SELECT phone, count(*)
FROM customer
WHERE phone IS NOT NULL
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
(0 rows)
```

- Ningún teléfono se repite hoy.

Error común: deducir una clave de los datos de hoy

examples.sql

```sql
SELECT phone, count(*)
FROM customer
WHERE phone IS NOT NULL
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
(0 rows)
```

- Ningún teléfono se repite hoy.
- ¿Es {phone} una clave? No: una madre y su hija, o una empresa con un solo número, pueden compartirlo.

Error común: deducir una clave de los datos de hoy

examples.sql

```sql
SELECT phone, count(*)
FROM customer
WHERE phone IS NOT NULL
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
(0 rows)
```

- Ningún teléfono se repite hoy.
- ¿Es {phone} una clave? No: una madre y su hija, o una empresa con un solo número, pueden compartirlo.
- El esquema no lo prohíbe. Lo comprobamos en la siguiente diapositiva.

Una instancia válida con un teléfono compartido

examples.sql

```sql
BEGIN;
INSERT INTO customer (customer_id, dni, first_name, last_name
      , phone, city)
VALUES (1801, '00000001', 'Rosa', 'Chávez Ríos', '909975104',
       'Ayacucho');
SELECT customer_id, dni, first_name, last_name, phone
FROM customer WHERE phone = '909975104' ORDER BY customer_id;
ROLLBACK;
```

psql

```
 customer_id |   dni    | first_name |  last_name  |   phone
-------------+----------+------------+-------------+-----------
           1 | 61830975 | Wilmer     | Chávez Ríos | 909975104
        1801 | 00000001 | Rosa       | Chávez Ríos | 909975104
(2 rows)
```

Una clave es una regla

Una afirmación sobre **todas** las instancias válidas, no sobre la actual.

El ROLLBACK deja la base de datos como estaba.

Cuidado con `NULL` al buscar repetidos

examples.sql

```sql
SELECT phone, count(*)
FROM customer
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
       |   259
(1 row)
```

- Sin el `WHERE phone IS NOT NULL`, `GROUP BY` junta los 259 `NULL` en un grupo.

Cuidado con `NULL` al buscar repetidos

examples.sql

```sql
SELECT phone, count(*)
FROM customer
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
       |   259
(1 row)
```

- Sin el `WHERE phone IS NOT NULL`, `GROUP BY` junta los 259 `NULL` en un grupo.
- No es un teléfono repetido: no sabemos esos teléfonos.

Cuidado con `NULL` al buscar repetidos

examples.sql

```sql
SELECT phone, count(*)
FROM customer
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
       |   259
(1 row)
```

- Sin el `WHERE phone IS NOT NULL`, `GROUP BY` junta los 259 `NULL` en un grupo.
- No es un teléfono repetido: no sabemos esos teléfonos.
- Una clave primaria no admite `NULL`: un valor desconocido no identifica nada.

Cuidado con `NULL` al buscar repetidos

examples.sql

```sql
SELECT phone, count(*)
FROM customer
GROUP BY phone
HAVING count(*) > 1;
```

psql

```
 phone | count
-------+-------
       |   259
(1 row)
```

- Sin el `WHERE phone IS NOT NULL`, `GROUP BY` junta los 259 `NULL` en un grupo.
- No es un teléfono repetido: no sabemos esos teléfonos.
- Una clave primaria no admite `NULL`: un valor desconocido no identifica nada.

En PostgreSQL

Por defecto, `UNIQUE` considera **distintos** a dos `NULL`. Desde la versión 15 existe `UNIQUE NULLS NOT DISTINCT`.

`keys.mjs`: la misma prueba, automatizada

keys.mjs

```mjs
// Los nombres no van como $1: se verifican contra
// el catálogo y se escriben con escapeIdentifier.
const t = pg.escapeIdentifier(table);
`SELECT ${list}, count(*) AS n
 FROM ${t}
 WHERE ${noNull}
 GROUP BY ${list}
 HAVING count(*) > 1`
```

- `checkSuperkey` cuenta aparte las filas con `NULL` y los grupos repetidos.

bash

```
customer, sobre los datos de hoy:
  {dni}: 0 filas con NULL, 0 grupos repetidos
  {first_name, last_name}: 0 filas con NULL, 27 grupos repetidos
  {phone}: 259 filas con NULL, 0 grupos repetidos
```

`keys.mjs`: la misma prueba, automatizada

keys.mjs

```mjs
// Los nombres no van como $1: se verifican contra
// el catálogo y se escriben con escapeIdentifier.
const t = pg.escapeIdentifier(table);
`SELECT ${list}, count(*) AS n
 FROM ${t}
 WHERE ${noNull}
 GROUP BY ${list}
 HAVING count(*) > 1`
```

- `checkSuperkey` cuenta aparte las filas con `NULL` y los grupos repetidos.
- No refuta `{dni}`; refuta `{first_name, last_name}`.

bash

```
customer, sobre los datos de hoy:
  {dni}: 0 filas con NULL, 0 grupos repetidos
  {first_name, last_name}: 0 filas con NULL, 27 grupos repetidos
  {phone}: 259 filas con NULL, 0 grupos repetidos
```

`keys.mjs`: la misma prueba, automatizada

keys.mjs

```mjs
// Los nombres no van como $1: se verifican contra
// el catálogo y se escriben con escapeIdentifier.
const t = pg.escapeIdentifier(table);
`SELECT ${list}, count(*) AS n
 FROM ${t}
 WHERE ${noNull}
 GROUP BY ${list}
 HAVING count(*) > 1`
```

- `checkSuperkey` cuenta aparte las filas con `NULL` y los grupos repetidos.
- No refuta `{dni}`; refuta `{first_name, last_name}`.
- No refuta `{phone}`, y aun así no es clave.

bash

```
customer, sobre los datos de hoy:
  {dni}: 0 filas con NULL, 0 grupos repetidos
  {first_name, last_name}: 0 filas con NULL, 27 grupos repetidos
  {phone}: 259 filas con NULL, 0 grupos repetidos
```

Claves foráneas

Restricción de clave foránea *(foreign-key constraint)*

De los atributos A de r_1 hacia la clave primaria B de r_2: en toda instancia válida, para cada tupla $t_1 \in r_1$ existe $t_2 \in r_2$ con $t_1[A] = t_2[B]$.

- r_1 es la relación **que referencia**; r_2, la **referenciada**.
- `shipment.sender_id` referencia a `customer`: cada envío apunta a su remitente.

Claves foráneas

Restricción de clave foránea *(foreign-key constraint)*

De los atributos A de r_1 hacia la clave primaria B de r_2: en toda instancia válida, para cada tupla $t_1 \in r_1$ existe $t_2 \in r_2$ con $t_1[A] = t_2[B]$.

- r_1 es la relación **que referencia**; r_2, la **referenciada**.
- `shipment.sender_id` referencia a `customer`: cada envío apunta a su remitente.
- Una clave foránea no tiene por qué ser una clave de su relación: `sender_id` no lo es (Yolanda Torres envió 64 paquetes).

Claves foráneas

Restricción de clave foránea *(foreign-key constraint)*

De los atributos A de r_1 hacia la clave primaria B de r_2: en toda instancia válida, para cada tupla $t_1 \in r_1$ existe $t_2 \in r_2$ con $t_1[A] = t_2[B]$.

- r_1 es la relación **que referencia**; r_2, la **referenciada**.
- `shipment.sender_id` referencia a `customer`: cada envío apunta a su remitente.
- Una clave foránea no tiene por qué ser una clave de su relación: `sender_id` no lo es (Yolanda Torres envió 64 paquetes).
- **Integridad referencial**, más general: los valores deben aparecer en ciertos atributos de la otra relación, aunque no sean su clave primaria.

Un envío apunta a dos agencias

shipment

shipment_id PK

origin_agency_id

dest_agency_id

agency

agency_id PK

name, city, region

Un envío apunta a dos agencias



Un envío apunta a dos agencias



Cada clave foránea *(foreign key)* debe referirse a una agencia que existe.

Un envío apunta a dos agencias



Cada clave foránea *(foreign key)* debe referirse a una agencia que existe.

Dos papeles distintos, dos claves foráneas. Igual en `route` (origen y destino) y en `shipment` (remitente y destinatario).

Integridad referencial en acción

psql

```
chasqui=# INSERT INTO shipment (shipment_id, sender_id, recipient_id, origin_agency_id,
                    dest_agency_id, weight_kg, price, status, created_at)
VALUES (12001, 99999, 337, 9, 8, 2.5, 15.00, '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".
```

Integridad referencial en acción

psql

```
chasqui=# INSERT INTO shipment (shipment_id, sender_id, recipient_id, origin_agency_id,
                    dest_agency_id, weight_kg, price, status, created_at)
VALUES (12001, 99999, 337, 9, 8, 2.5, 15.00, '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".
```

psql

```
chasqui=# DELETE FROM agency WHERE agency_id = 1;
ERROR:  update or delete on table "agency" violates foreign key constraint "route_origin_agency_id_fkey" on table "route"
DETAIL:  Key (agency_id)=(1) is still referenced from table "route".
```

Por los dos lados: ni referencias al vacío, ni borrar lo referenciado. Otras acciones (`ON DELETE`): semana 4.

Claves foráneas: `NULL` y a qué pueden apuntar

- `shipment.trip_id` admite `NULL`: los 500 envíos registrados aún no tienen viaje.

Claves foráneas: `NULL` y a qué pueden apuntar

- `shipment.trip_id` admite `NULL`: los 500 envíos registrados aún no tienen viaje.
- Una clave foránea `NULL` no se verifica: *si* hay valor, debe existir en `trip`.

Claves foráneas: `NULL` y a qué pueden apuntar

- `shipment.trip_id` admite `NULL`: los 500 envíos registrados aún no tienen viaje.
- Una clave foránea `NULL` no se verifica: *si* hay valor, debe existir en `trip`.
- Silberschatz: hacia la clave **primaria**. SQL y PostgreSQL: primaria o `UNIQUE`.

Claves foráneas: `NULL` y a qué pueden apuntar

- `shipment.trip_id` admite `NULL`: los 500 envíos registrados aún no tienen viaje.
- Una clave foránea `NULL` no se verifica: *si* hay valor, debe existir en `trip`.
- Silberschatz: hacia la clave **primaria**. SQL y PostgreSQL: primaria o `UNIQUE`.

psql

```
chasqui=# CREATE TABLE complaint (
  complaint_id integer PRIMARY KEY,
  city         text REFERENCES customer (city)
);
ERROR:  there is no unique constraint matching given keys for referenced table "customer"
```

Claves foráneas: `NULL` y a qué pueden apuntar

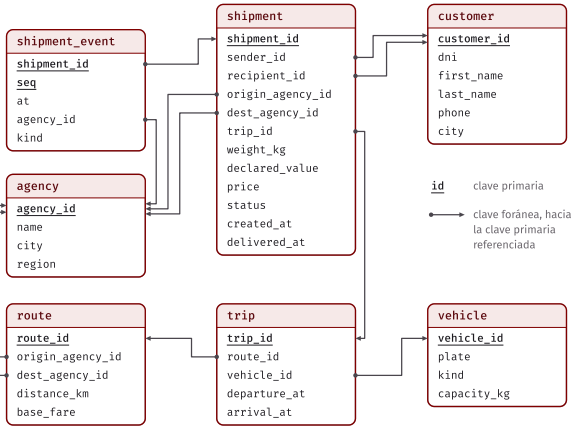
- `shipment.trip_id` admite `NULL`: los 500 envíos registrados aún no tienen viaje.
- Una clave foránea `NULL` no se verifica: *si* hay valor, debe existir en `trip`.
- Silberschatz: hacia la clave **primaria**. SQL y PostgreSQL: primaria o `UNIQUE`.

psql

```
chasqui=# CREATE TABLE complaint (
  complaint_id integer PRIMARY KEY,
  city         text REFERENCES customer (city)
);
ERROR:  there is no unique constraint matching given keys for referenced table "customer"
```

Si dos clientes viven en Lima, ¿a cuál apuntaría `city = 'Lima'`? Cada valor no nulo apunta a exactamente una tupla: lo referenciado tiene que ser una clave.

Diagrama de esquema de Chasqui



Lo que muestra el diagrama, y lo que no

- **Once** flechas: once claves foráneas.
- Cada flecha llega a la clave primaria de la relación referenciada.
- En `shipment_event`, `shipment_id` es parte de la clave y clave foránea: un evento solo existe como evento *de un envío*.

Lo que muestra el diagrama, y lo que no

- **Once** flechas: once claves foráneas.
- Cada flecha llega a la clave primaria de la relación referenciada.
- En `shipment_event`, `shipment_id` es parte de la clave y clave foránea: un evento solo existe como evento *de un envío*.
- No muestra la otra candidata de `route` ni los `CHECK`: solo claves primarias y foráneas.

Lo que muestra el diagrama, y lo que no

- **Once** flechas: once claves foráneas.
- Cada flecha llega a la clave primaria de la relación referenciada.
- En `shipment_event`, `shipment_id` es parte de la clave y clave foránea: un evento solo existe como evento *de un envío*.
- No muestra la otra candidata de `route` ni los `CHECK`: solo claves primarias y foráneas.

Idea clave

En el CSV, cada envío copiaba el nombre y el teléfono del remitente. Aquí, el envío guarda solo `sender_id` y el cliente está **una vez**. La clave foránea separa los datos sin perder la conexión, y la integridad referencial garantiza que no apunte al vacío.

SECCIÓN 07

El código de la semana

07

`code/week-01` y `make test`

- `examples.sql`: las consultas de los apuntes.
- `constraints.sql`: los errores a propósito (`\i constraints.sql`).
- `files.mjs`: Chasqui con archivos.
- `keys.mjs`: claves del catálogo y `checkSuperkey`.
- `*.test.mjs`: 26 pruebas con `node:test`.

Los errores se comparan por SQLSTATE y por nombre de restricción.

bash

```
$ make test
...
✓ countByStatus (un programa nuevo) coincide con GROUP BY (una consulta)
✓ redundancia: el teléfono de un cliente se copia en cada envío
✓ inconsistencia: corregir un solo archivo deja dos teléfonos
...
✓ peso negativo: 23514 en shipment_weight_kg_check
✓ una clave foránea debe apuntar a una clave: 42830
...
ℹ tests 26
ℹ suites 0
ℹ pass 26
ℹ fail 0
```

Salida recortada.

Resumen

- Un **DBMS** evita redundancia e inconsistencia, responde preguntas con **consultas**, verifica **restricciones**, y da **atomicidad**, **control de concurrencia** y **seguridad**.
- Niveles **físico**, **lógico** y **de vistas**. **Independencia física**: cambiar el almacenamiento sin reescribir consultas.
- **Esquema** (el diseño) frente a **instancia** (los datos de ahora). **DDL** define; **DML** consulta y modifica. SQL es **declarativo**.
- Motor: almacenamiento, consultas, transacciones. PostgreSQL: cliente-servidor, un backend por conexión.
- Una **relación** es un **conjunto** de tuplas sobre dominios atómicos; las tablas de SQL, multiconjuntos.
- **Superclave**: identifica en toda instancia válida. **Candidata**: mínima. **Primaria**: la elegida, sin `NULL`.
- Los datos de hoy pueden refutar una clave, nunca demostrarla.
- **Clave foránea**: cada valor apunta a una tupla que existe (**integridad referencial**); si es `NULL`, no se verifica.

Para la próxima clase

- Leer Silberschatz, secciones 2.5 y 2.6 (lenguajes de consulta y álgebra relacional) y la *Note 6: Relational Algebra* de Berkeley CS186.
- Laboratorio: el entorno, `psql`, la base de datos `chasqui` y un primer script con `pg`.
- Correr `make test` en `code/week-01` y leer los errores de `constraints.sql`.
- Leer el enunciado de **P0**: entrega en la semana 3.

Pregunta

En `trip`, ¿debería `{vehicle_id, departure_at}` ser una clave candidata? ¿Cómo comprobarían con una consulta si los datos de hoy la refutan, y qué concluirían si la refutan?