Contenido

[Creación de tablas 3](#_Toc207203891)

[Ejercicio 1 3](#_Toc207203892)

[Ejercicio 2 5](#_Toc207203893)

[Group by 7](#_Toc207203894)

[Ejercicio 1 7](#_Toc207203895)

[Ejercicio 2 8](#_Toc207203896)

[Ejercicio 3 8](#_Toc207203897)

[Ejercicio 4 10](#_Toc207203898)

[Having 11](#_Toc207203899)

[Ejercicio 1 11](#_Toc207203900)

[Ejercicio 2 13](#_Toc207203901)

[Ejercicio 3 14](#_Toc207203902)

[Order by 15](#_Toc207203903)

[Ejercicio 1 15](#_Toc207203904)

[Ejercicio 2 17](#_Toc207203905)

[Ejercicio 3 19](#_Toc207203906)

[Distinct 20](#_Toc207203907)

[Ejercicio 1 20](#_Toc207203908)

[Ejercicio 2 21](#_Toc207203909)

[Beetwen 24](#_Toc207203910)

[Ejercicio 1 24](#_Toc207203911)

[Ejercicio 2 25](#_Toc207203912)

[Ejercicio 3 27](#_Toc207203913)

[Like 29](#_Toc207203914)

[Ejercicio 1 29](#_Toc207203915)

[Ejercicio 2 30](#_Toc207203916)

[Is null 31](#_Toc207203917)

[Ejercicio 1 31](#_Toc207203918)

[Ejercicio 2 32](#_Toc207203919)

[Ejercicio 3 34](#_Toc207203920)

[Count() 37](#_Toc207203921)

[Ejercicio 1 37](#_Toc207203922)

[Ejercicio 2 38](#_Toc207203923)

[Ejercicio 3 39](#_Toc207203924)

[Ejercicio 4 42](#_Toc207203925)

[Ejercicio 5\* 43](#_Toc207203926)

[Union 45](#_Toc207203927)

[Ejercicio 1 45](#_Toc207203928)

[Union all 47](#_Toc207203929)

[Ejercicio 1 47](#_Toc207203930)

[Case, when 48](#_Toc207203931)

[Ejercicio 1 48](#_Toc207203932)

[Ejercicio 2 50](#_Toc207203933)

[Trabajar con fechas 53](#_Toc207203934)

[Ejercicio 1 53](#_Toc207203935)

[Ejercicio 2 53](#_Toc207203936)

[Ejercicio 3 55](#_Toc207203937)

[Inner join 57](#_Toc207203938)

[Ejercicio 1 57](#_Toc207203939)

[Ejercicio 2 59](#_Toc207203940)

[Ejercicio 3 61](#_Toc207203941)

[Left join 62](#_Toc207203942)

[Ejercicio 1 62](#_Toc207203943)

[Ejercicio 2 65](#_Toc207203944)

[Ejercicio 3 67](#_Toc207203945)

[Right join 71](#_Toc207203946)

[Ejercicio 1 71](#_Toc207203947)

[Ejercicio 2 73](#_Toc207203948)

[Ejercicio 3 75](#_Toc207203949)

[Full o cross join 77](#_Toc207203950)

[Ejercicio 1 77](#_Toc207203951)

[Ejercicio 2 80](#_Toc207203952)

[Ejercicio 3 83](#_Toc207203953)

[Self join (Unión Automática) 86](#_Toc207203954)

[Ejercicio 1 87](#_Toc207203955)

[Ejercicio 2 87](#_Toc207203956)

[Subconsultas 89](#_Toc207203957)

[Subconsultas escalares (devuelven un solo valor) 89](#_Toc207203958)

[Subconsultas que devuelven conjuntos de filas 90](#_Toc207203959)

[Subconsultas correlacionadas 90](#_Toc207203960)

[Ejercicio 1 90](#_Toc207203961)

[Ejercicio 2 92](#_Toc207203962)

[Ejercicio 3 94](#_Toc207203963)

[Procedimientos almacenados 94](#_Toc207203964)

[Ejercicio 1 95](#_Toc207203965)

[Ejercicio 2 96](#_Toc207203966)

[Ejercicio 3 97](#_Toc207203967)

[Funciones 97](#_Toc207203968)

[Ejemplo 1 98](#_Toc207203969)

[Ejemplo 2 98](#_Toc207203970)

[Ejemplo 3 98](#_Toc207203971)

[Trigger 98](#_Toc207203972)

[Eventos 99](#_Toc207203973)

[Extras 100](#_Toc207203974)

[CTEs 100](#_Toc207203975)

[Ejercicio 1 100](#_Toc207203976)

[Ejercicio 2 103](#_Toc207203977)

[Tablas derivadas 107](#_Toc207203978)

[Funciones ventana 107](#_Toc207203979)

[Group\_concat 107](#_Toc207203980)

[Rollup 108](#_Toc207203981)

[Coalesce 108](#_Toc207203982)

Prompt:

Genera un ejercicio nivel Y para practicar Z en SQL. Dame también la salida esperada con el código solución. Además, agrega la tabla o tablas (en forma de tabla directamente y en formato de código donde el código de la construcción de las tablas me lo des en una sola celda para copiar y pegar)

# Creación de tablas

## Ejercicio 1

Crea dos tablas llamadas Clientes y Pedidos.

* La tabla Clientes debe tener:
  + ClienteID (entero, clave primaria)
  + Nombre (texto de máximo 50 caracteres)
  + Ciudad (texto de máximo 50 caracteres)
* La tabla Pedidos debe tener:
  + PedidoID (entero, clave primaria)
  + ClienteID (entero, clave foránea que hace referencia a Clientes)
  + FechaPedido (tipo fecha)
  + Total (decimal con 2 decimales)

Create database empresa;

Use empresa;

Create table clientes

(Cliente\_id int primary key,

Nombre varchar(50),

Ciudad varchar(50)

);

Create table pedidos

(pedido\_id int primary key,

Cliente\_id int,

Fecha\_pedido date

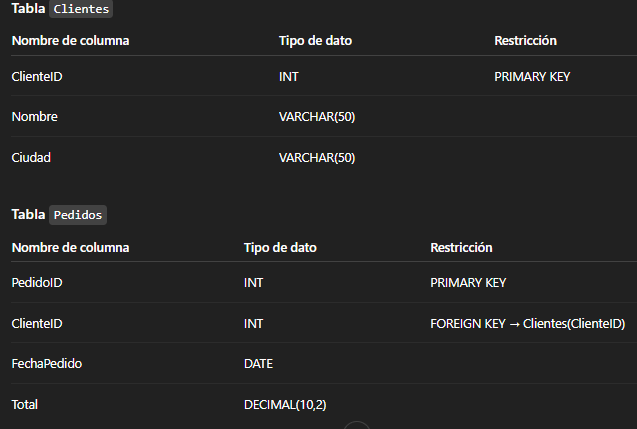
Total decimal (10,2),

Foreign key (cliente\_id) references clientes(cliente\_id)

);

FOREIGN KEY (<columna\_en\_esta\_tabla>)

REFERENCES <tabla\_referenciada>(<columna\_en\_tabla\_referenciada>)



## Ejercicio 2

Crea una tabla llamada Empleados con las siguientes condiciones:

* EmpleadoID: entero, clave primaria, debe ser mayor a 0.
* Nombre: texto de máximo 100 caracteres, no puede ser nulo.
* Edad: entero, debe estar entre 18 y 65 años.
* Salario: decimal con 2 decimales, mayor o igual a 0.
* Departamento: texto, debe estar restringido a uno de los siguientes valores: 'Ventas', 'TI', 'Recursos Humanos', 'Finanzas'.
* Además, asegúrate de que no pueda haber dos empleados con el mismo Nombre y Departamento.

Use name\_database;

CREATE TABLE Empleados (

EmpleadoID INT PRIMARY KEY CHECK (EmpleadoID > 0),

Nombre VARCHAR(100) NOT NULL,

Edad INT CHECK (Edad BETWEEN 18 AND 65),

Salario DECIMAL(10,2) CHECK (Salario >= 0),

Departamento VARCHAR(50) CHECK (Departamento IN ('Ventas', 'TI', 'Recursos Humanos', 'Finanzas')),

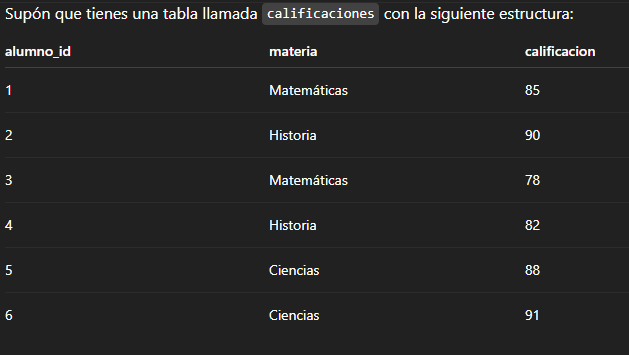
CONSTRAINT uq\_nombre\_departamento UNIQUE (Nombre, Departamento)

);



# Group by

## Ejercicio 1

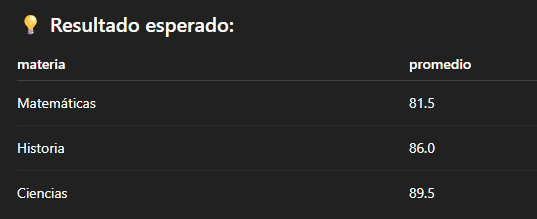


Obtener el **promedio de calificación por materia**.

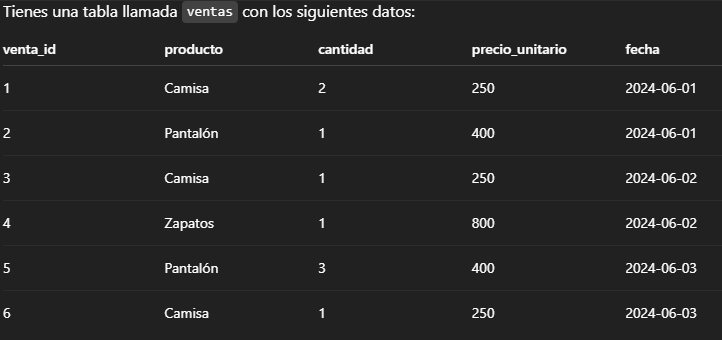
Select materia, avg(calificacion) as promedio

From calificaciones

Group by materia;



## Ejercicio 2



 Calcular el **total vendido por producto** (multiplicando cantidad \* precio\_unitario).

 Mostrar solo los productos que hayan generado **más de $1000 pesos en total de ventas**.

 Ordenar el resultado de mayor a menor.

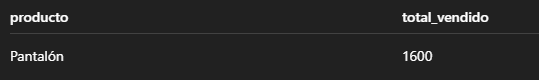
Select producto, sum(precio\_unitario\*cantidad) as total\_vendido

Form ventas

Group by productos

Having total\_vendido > 1000

Order by total\_ventas desc;



## Ejercicio 3

CREATE TABLE ventas (

id\_venta INT PRIMARY KEY,

id\_vendedor INT,

producto VARCHAR(50),

cantidad INT,

total\_venta DECIMAL(10,2),

fecha DATE

);

Tabla ventas



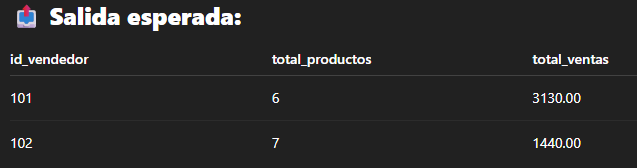
Obtener el **total de ventas** y la **cantidad total de productos vendidos por cada vendedor**, mostrando solo a aquellos **que hayan vendido más de $1000 en total**.

Select id\_vendedor, sum(cantidad) as total\_unidades, sum(total\_venta) as total\_vendido

From ventas

Group by id\_vendedor

Having sum(total\_venta) > 1000;



## Ejercicio 4

Muestra el **total de ventas y el número de compras realizadas por cada cliente**, **agrupando por nombre del cliente**. Solo incluye aquellos clientes que hayan realizado **más de 2 compras**.

CREATE TABLE clientes (

cliente\_id INT PRIMARY KEY,

nombre VARCHAR(100)

);

CREATE TABLE ventas (

venta\_id INT PRIMARY KEY,

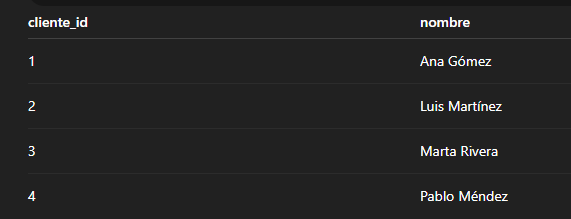
cliente\_id INT,

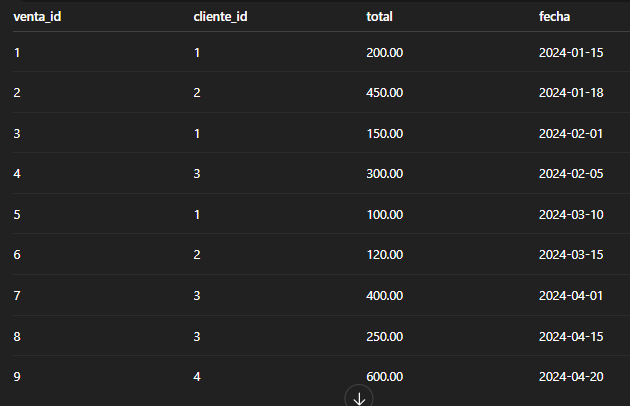
total DECIMAL(10, 2),

fecha DATE,

FOREIGN KEY (cliente\_id) REFERENCES clientes(cliente\_id)

);





Muestra el **total de ventas y el número de compras realizadas por cada cliente**, **agrupando por nombre del cliente**. Solo incluye aquellos clientes que hayan realizado **más de 2 compras**.

SELECT

c.nombre,

COUNT(v.venta\_id) AS num\_compras,

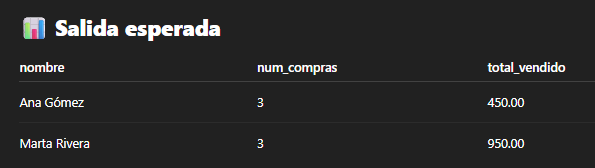
SUM(v.total) AS total\_vendido

FROM ventas v

JOIN clientes c ON v.cliente\_id = c.cliente\_id

GROUP BY c.nombre

HAVING COUNT(v.venta\_id) > 2;



# Having

## Ejercicio 1

CREATE TABLE ventas (

venta\_id INT,

vendedor\_id INT,

producto\_id INT,

fecha DATE

);

INSERT INTO ventas (venta\_id, vendedor\_id, producto\_id, fecha) VALUES

(1, 101, 1, '2024-01-01'),

(2, 101, 2, '2024-01-02'),

(3, 101, 1, '2024-01-03'),

(4, 102, 3, '2024-01-01'),

(5, 102, 4, '2024-01-01'),

(6, 102, 5, '2024-01-02'),

(7, 102, 6, '2024-01-02'),

(8, 103, 1, '2024-01-02'),

(9, 103, 2, '2024-01-02'),

(10, 103, 2, '2024-01-03');



Obtén los vendedores (vendedor\_id) que hayan vendido **más de 3 productos distintos** en total (sin contar productos repetidos). Muestra también la **cantidad de productos distintos vendidos**.

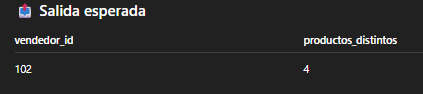
SELECT v.vendedor\_id, COUNT(DISTINCT producto\_id) AS productos\_distintos

FROM ventas v

GROUP BY vendedor\_id

HAVING

COUNT(DISTINCT producto\_id) > 3;



## Ejercicio 2

CREATE TABLE ventas (

id\_venta INT,

producto VARCHAR(50),

categoria VARCHAR(50),

cantidad INT,

precio\_unitario DECIMAL(10, 2),

fecha DATE

);

INSERT INTO ventas VALUES

(1, 'Laptop', 'Electrónica', 3, 1000, '2024-05-01'),

(2, 'Celular', 'Electrónica', 5, 500, '2024-05-02'),

(3, 'Mouse', 'Accesorios', 10, 25, '2024-05-03'),

(4, 'Teclado', 'Accesorios', 7, 30, '2024-05-04'),

(5, 'Monitor', 'Electrónica', 2, 300, '2024-05-05'),

(6, 'Impresora', 'Oficina', 1, 200, '2024-05-06'),

(7, 'Escritorio', 'Oficina', 2, 150, '2024-05-06'),

(8, 'Mouse', 'Accesorios', 15, 20, '2024-05-07');



Calcula el total de ingresos (cantidad \* precio\_unitario) por cada categoría de producto. Muestra solo aquellas categorías cuyo total de ingresos sea mayor a **$1,000**.

Select categoría, sum(v.cantidad \* v.precio\_unitario) as total\_ingresos

From ventas v

Group by categoría

Having sum(v.cantidad \* v.precio\_unitario) > 1000;

## Ejercicio 3

CREATE TABLE ordenes (

id\_orden INT,

cliente VARCHAR(50),

producto VARCHAR(50),

categoria VARCHAR(50),

cantidad INT,

precio\_unitario DECIMAL(10, 2),

fecha DATE

);

INSERT INTO ordenes VALUES

(1, 'Ana', 'Laptop', 'Electrónica', 1, 1200, '2024-06-01'),

(2, 'Luis', 'Celular', 'Electrónica', 2, 600, '2024-06-01'),

(3, 'Ana', 'Teclado', 'Accesorios', 3, 40, '2024-06-02'),

(4, 'María', 'Mouse', 'Accesorios', 4, 25, '2024-06-03'),

(5, 'Luis', 'Monitor', 'Electrónica', 1, 350, '2024-06-03'),

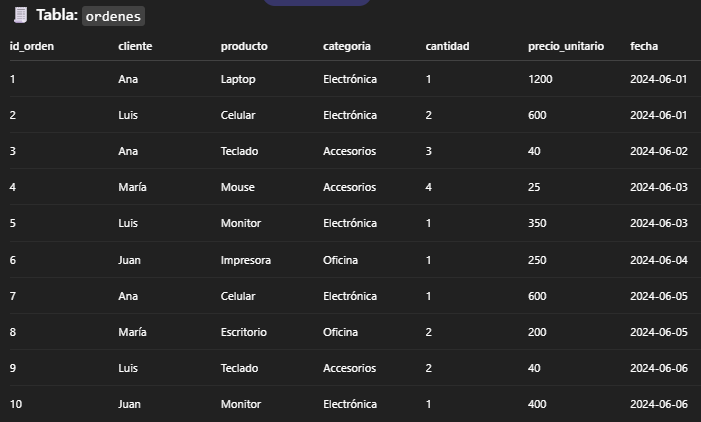
(6, 'Juan', 'Impresora', 'Oficina', 1, 250, '2024-06-04'),

(7, 'Ana', 'Celular', 'Electrónica', 1, 600, '2024-06-05'),

(8, 'María', 'Escritorio', 'Oficina', 2, 200, '2024-06-05'),

(9, 'Luis', 'Teclado', 'Accesorios', 2, 40, '2024-06-06'),

(10, 'Juan', 'Monitor', 'Electrónica', 1, 400, '2024-06-06');



Muestra los clientes que hayan comprado **más de 3 productos en total**, y cuyo **gasto total** haya sido **mayor a $1000**, **pero solo considerando productos de la categoría "Electrónica"**.

Select clientes, sum(cantidad) as unidades\_vendidas, sum(precio\_unitario \* cantidad) as gasto\_total

From ordenes

Where catergoria == “Electronica”

Group by clientes

Having sum(precio\_unitario \* cantidad) > 1000 and sum(cantidad) > 3;

# Order by

## Ejercicio 1

CREATE TABLE clientes (

cliente\_id INT,

nombre VARCHAR(50),

puntos INT

);

INSERT INTO clientes (cliente\_id, nombre, puntos) VALUES

(1, 'Ana', 120),

(2, 'Luis', 250),

(3, 'Sofía', 180),

(4, 'Carlos', 90),

(5, 'Lucía', 300);



Muestra todos los clientes ordenados de **mayor a menor** según los puntos acumulados.

Select \*

From clientes

Order by c.puntos desc;



## Ejercicio 2

CREATE TABLE ventas (

id\_venta INT,

cliente VARCHAR(50),

producto VARCHAR(50),

cantidad INT,

precio\_unitario DECIMAL(6,2),

fecha DATE

);

INSERT INTO ventas VALUES

(1, 'Ana', 'Laptop', 1, 1200.00, '2024-07-01'),

(2, 'Luis', 'Mouse', 3, 20.00, '2024-07-03'),

(3, 'Carlos', 'Teclado', 2, 45.00, '2024-07-02'),

(4, 'Ana', 'Monitor', 1, 250.00, '2024-07-01'),

(5, 'Luis', 'Laptop', 1, 1150.00, '2024-07-04'),

(6, 'Carlos', 'Mouse', 2, 18.00, '2024-07-02'),

(7, ' Ana', 'Teclado', 1, 47.00, '2024-07-05');



Consulta las ventas realizadas y ordénalas de mayor a menor según el valor total de la venta (cantidad × precio\_unitario). En caso de empate, usa la fecha más reciente primero. Muestra las columnas: cliente, producto, cantidad, precio\_unitario, total\_venta y fecha.

Select cliente, producto, cantidad, precio\_unitario, total\_venta, fecha, cantidad\*precio\_unitario as valor\_total

From ventas

Order by cantidad\*precio\_unitario desc, fecha desc;

En una cláusula ORDER BY con **varias columnas**, **el orden sí importa** y **la primera columna tiene prioridad** (en este caso cantidad\*precio\_unitario);

 El **orden de las columnas en ORDER BY determina la jerarquía** de ordenamiento.

 Se aplica de **izquierda a derecha**.

 Muy útil para ordenar por métricas principales y luego deshacer empates con otros criterios (como fecha, ID, etc.).

Esto quiere decir que la segunda condición del order by (la fecha) solo se ejecuta en caso de que la primera condición (cantidad\*precio\_unitario) tenga un valor repetido y de esa forma entra a “desempatar” la condición de fecha



## Ejercicio 3

CREATE TABLE ventas\_empleados (

empleado\_id INT,

nombre VARCHAR(50),

region VARCHAR(20),

anio INT,

total\_ventas DECIMAL(10,2)

);



Escribe una consulta SQL que muestre la lista de todos los empleados con sus ventas, ordenada de la siguiente forma:

1. Primero por anio **descendente**.
2. Luego por region **alfabéticamente ascendente**.
3. Después por total\_ventas **descendente**, asegurando que los NULL vayan al **final**.
4. Finalmente, si hay empates en ventas, ordena por nombre **ascendente**.

Select \*

From empleados\_ventas

Order by anio desc, región asc, total\_ventas desc null last, nombre asc;



# Distinct

## Ejercicio 1

DISTINCT se utiliza en SQL para **eliminar duplicados** y devolver solo **valores únicos** de una columna o combinación de columnas en los resultados de una consulta.

CREATE TABLE ventas (

id INT,

producto VARCHAR(50),

cliente VARCHAR(50)

);

INSERT INTO ventas (id, producto, cliente) VALUES

(1, 'Camisa', 'Ana'),

(2, 'Pantalón', 'Luis'),

(3, 'Camisa', 'Luis'),

(4, 'Zapatos', 'Ana'),

(5, 'Camisa', 'Ana');



Obtén la lista de **productos únicos** que se han vendido.

Select distinct producto

From ventas;



## Ejercicio 2

CREATE TABLE compras\_clientes (

compra\_id INT,

cliente\_id INT,

nombre\_cliente VARCHAR(50),

producto VARCHAR(50),

anio INT

);



Muestra una lista de combinaciones **únicas** de cliente, producto y año, es decir, ignora las compras duplicadas del mismo producto por el mismo cliente en el mismo año.

Ordena la salida por nombre\_cliente y luego por anio.

SELECT DISTINCT cliente\_id, nombre\_cliente, producto, anio

FROM compras\_clientes

ORDER BY nombre\_cliente, anio;



# Beetwen

## Ejercicio 1

CREATE TABLE productos (

producto\_id INT,

nombre VARCHAR(50),

precio DECIMAL(6,2)

);

INSERT INTO productos (producto\_id, nombre, precio) VALUES

(1, 'Teclado', 150.00),

(2, 'Mouse', 80.00),

(3, 'Monitor', 220.00),

(4, 'USB', 50.00),

(5, 'Auriculares', 120.00),

(6, 'Webcam', 180.00);



Muestra los productos cuyo **precio esté entre 100 y 200** (inclusive).

Select p.nombre, p.precio

From productos p

Where p.precio between 100 and 200;



En este caso between es similar a WHERE precio >= 100 AND precio <= 200

En el ejercicio La cláusula betwein incluye siempre un intervalo cerrado. Entonces si queremos obtener un intervalo abierto tenemos que usar necesariamente los condiciones lógicos de comparación:

SELECT \* FROM productos

WHERE precio > 100 AND precio < 200;

## Ejercicio 2

CREATE TABLE clientes (

id INT PRIMARY KEY,

nombre VARCHAR(50),

apellido VARCHAR(50),

edad INT,

pais VARCHAR(50)

);

INSERT INTO clientes (id, nombre, apellido, edad, pais) VALUES

(1, 'Carlos', 'Ramírez', 28, 'México'),

(2, 'Ana', 'Gómez', 34, 'Colombia'),

(3, 'Luis', 'Pérez', 41, 'Argentina'),

(4, 'Marta', 'Sánchez', 22, 'Chile'),

(5, 'Diego', 'Torres', 45, 'México'),

(6, 'Laura', 'Martínez', 38, 'España'),

(7, 'Jorge', 'Ruiz', 50, 'Perú');



Tienes una base de datos con información de clientes. Se te pide obtener los clientes cuya edad esté **entre 30 y 45 años inclusive**. La consulta debe mostrar su nombre completo (nombre + apellido), edad y país de residencia.

Select \*

From clientes

Where edad between 30 and 45;

La consulta anterior está bien, pero en el caso de que queramos crear una columna temporal para mostrar los datos en una sola columna:

SELECT

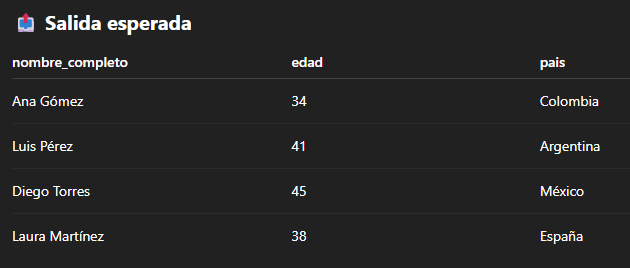
nombre CONCAT(nombre, ' ', apellido) AS nombre\_completo,

edad,

pais

FROM clientes

WHERE edad BETWEEN 30 AND 45;



## Ejercicio 3

CREATE TABLE pedidos (

id\_pedido INT PRIMARY KEY,

cliente VARCHAR(50),

fecha\_pedido DATE,

total DECIMAL(10, 2)

);

INSERT INTO pedidos (id\_pedido, cliente, fecha\_pedido, total) VALUES

(101, 'Carlos Ramírez', '2024-02-28', 150.00),

(102, 'Ana Gómez', '2024-03-15', 300.00),

(103, 'Luis Pérez', '2024-04-10', 250.50),

(104, 'Marta Sánchez', '2024-05-05', 100.75),

(105, 'Diego Torres', '2024-06-01', 500.00),

(106, 'Laura Martínez', '2024-03-01', 80.00),

(107, 'Jorge Ruiz', '2024-05-31', 120.00);



Tienes una tabla de pedidos. Se te pide obtener todos los pedidos realizados **entre el 1 de marzo de 2024 y el 31 de mayo de 2024**, inclusive. La consulta debe mostrar el id\_pedido, el cliente, la fecha\_pedido y el total.

Select \*

From pedidos

WHERE fecha\_pedido BETWEEN '2024-03-01' AND '2024-05-31';



# Like

## Ejercicio 1

CREATE TABLE clientes (

id INT,

nombre VARCHAR(50),

ciudad VARCHAR(50)

);

INSERT INTO clientes (id, nombre, ciudad) VALUES

(1, 'Carlos Pérez', 'Monterrey'),

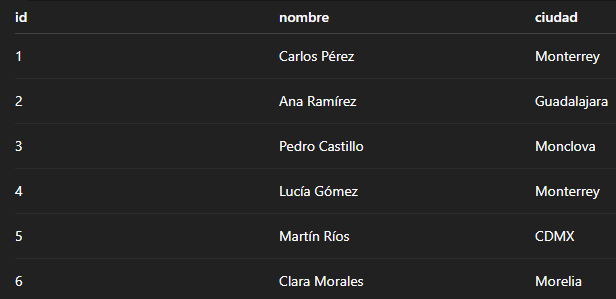
(2, 'Ana Ramírez', 'Guadalajara'),

(3, 'Pedro Castillo', 'Monclova'),

(4, 'Lucía Gómez', 'Monterrey'),

(5, 'Martín Ríos', 'CDMX'),

(6, 'Clara Morales', 'Morelia');

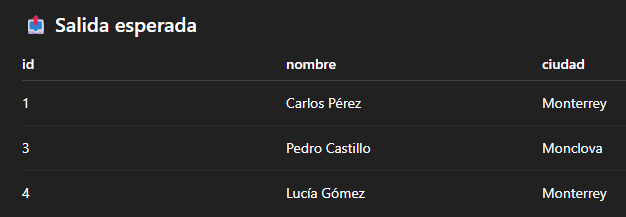


Escribe una consulta que muestre todos los registros de clientes cuya ciudad **comience con 'Mon'**.

SELECT \*

FROM clientes

WHERE ciudad LIKE 'Mon%';



## Ejercicio 2

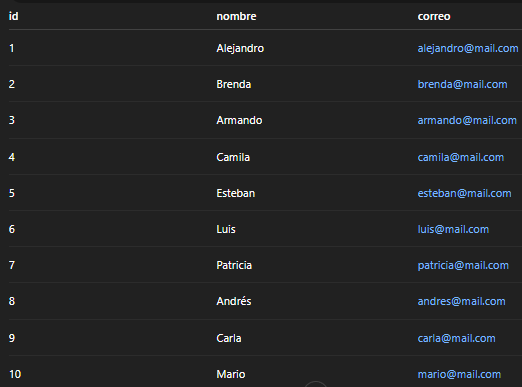
CREATE TABLE Clientes (

id INT,

nombre VARCHAR(50),

correo VARCHAR(100)

);



Tienes una tabla llamada Clientes que almacena información de los clientes de una tienda en línea. Necesitas obtener una lista de los clientes cuyo **nombre** cumpla con **todas** las siguientes condiciones:

1. Contiene la letra 'a' en cualquier posición.
2. Comienza con una consonante.
3. Termina en una vocal.
4. Tiene al menos 6 caracteres.

SELECT \*

FROM Clientes

WHERE

nombre LIKE '%a%' AND -- Contiene la letra 'a'

nombre ~\* '^[^aeiou]' AND -- Empieza con consonante (usando POSIX regex para PostgreSQL)

nombre ~\* '[aeiou]$' AND -- Termina en vocal

LENGTH(nombre) >= 6; -- Mínimo 6 caracteres



# Is null

## Ejercicio 1

CREATE TABLE ordenes (

id INT,

cliente VARCHAR(50),

producto VARCHAR(50),

fecha\_entrega DATE

);

INSERT INTO ordenes (id, cliente, producto, fecha\_entrega) VALUES

(1, 'Carlos Pérez', 'Laptop', '2025-06-15'),

(2, 'Ana Ramírez', 'Tablet', NULL),

(3, 'Pedro Castillo', 'Smartphone', '2025-06-20'),

(4, 'Lucía Gómez', 'Monitor', NULL),

(5, 'Clara Morales', 'Teclado', '2025-06-25');

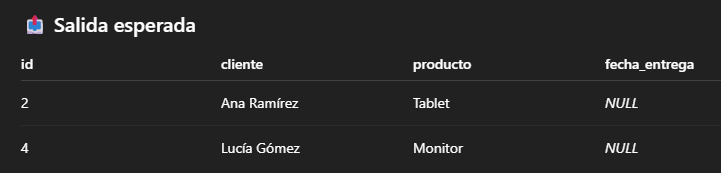


Escribe una consulta que muestre todos los pedidos que **aún no tienen una fecha de entrega asignada**.

Select \*

From ordenes

Where fecha\_entrega is null;



## Ejercicio 2

CREATE TABLE Empleados (

id\_empleado INT,

nombre VARCHAR(100),

departamento\_id INT

);

INSERT INTO Empleados (id\_empleado, nombre, departamento\_id) VALUES

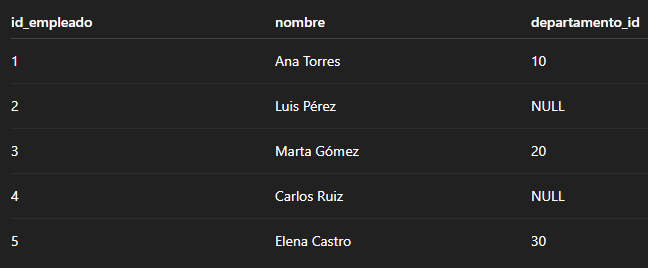
(1, 'Ana Torres', 10),

(2, 'Luis Pérez', NULL),

(3, 'Marta Gómez', 20),

(4, 'Carlos Ruiz', NULL),

(5, 'Elena Castro', 30);



Tienes una tabla llamada Empleados con información sobre los trabajadores de una empresa. Algunos empleados aún no tienen asignado un departamento.

**Objetivo**:

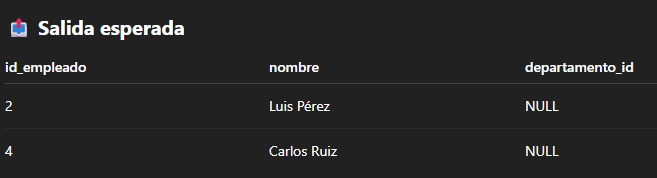
1. Obtén una lista de los empleados que **no tienen asignado un departamento**.
2. Muestra las columnas: id\_empleado, nombre, y departamento\_id.
3. Ordena los resultados por id\_empleado.

Select e.id\_empleado, e.nombre, e.departamento\_id

From empleados e

Where e.departamento\_id is null

Order by e.id\_empleado;



## Ejercicio 3

CREATE TABLE Empleados (

id\_empleado INT,

nombre VARCHAR(100),

departamento\_id INT

);

INSERT INTO Empleados (id\_empleado, nombre, departamento\_id) VALUES

(1, 'Ana Torres', 10),

(2, 'Luis Pérez', NULL),

(3, 'Marta Gómez', 20),

(4, 'Carlos Ruiz', 99), -- Departamento no existente

(5, 'Elena Castro', 30),

(6, 'José García', NULL),

(7, 'Laura Navas', 50); -- Departamento no existente

CREATE TABLE Departamentos (

id\_departamento INT,

nombre\_departamento VARCHAR(100)

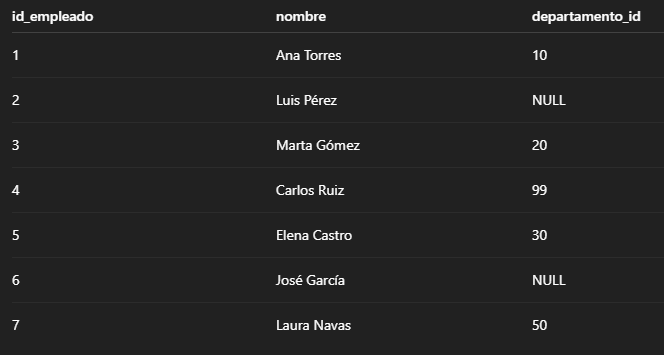
);

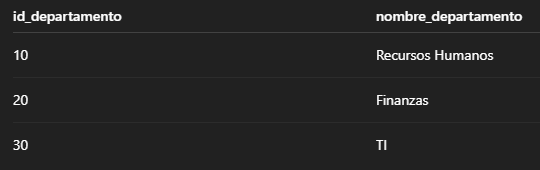
INSERT INTO Departamentos (id\_departamento, nombre\_departamento) VALUES

(10, 'Recursos Humanos'),

(20, 'Finanzas'),

(30, 'TI');





Tienes dos tablas:

1. Empleados: contiene a todos los empleados de la empresa.
2. Departamentos: contiene información de los departamentos existentes.

**Objetivo**:

1. Encuentra los empleados que **no están asignados a ningún departamento válido** (es decir, su departamento\_id no existe en la tabla Departamentos).
2. Muestra: id\_empleado, nombre, departamento\_id.
3. Ordena por id\_empleado.

Select e.id\_empleado, e.nombre, e.departamento\_id

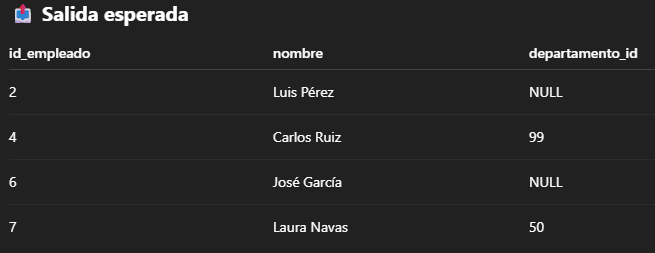
From empleados e

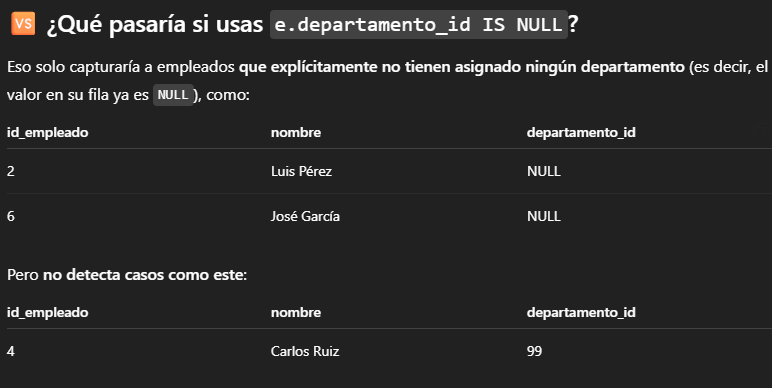
Left join departamentos d

On e.departamento\_id = d.id\_departamento

WHERE d.id\_departamento IS NULL

Order by e.id\_empleado;





# Count()

## Ejercicio 1

CREATE TABLE ventas (

id INT,

producto VARCHAR(50),

cantidad INT

);

INSERT INTO ventas (id, producto, cantidad) VALUES

(1, 'Laptop', 2),

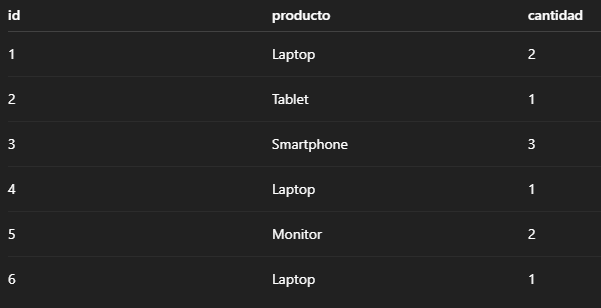
(2, 'Tablet', 1),

(3, 'Smartphone', 3),

(4, 'Laptop', 1),

(5, 'Monitor', 2),

(6, 'Laptop', 1);



Cuenta cuántas ventas se han realizado del producto **"Laptop"**.

SELECT COUNT(\*) AS total laptops

FROM ventas

WHERE producto = 'Laptop';



Si queremos mostrar las ventas por cada producto:

Select producto, count(\*) as venta\_producto

From ventas

Group by producto;

## Ejercicio 2

La función COUNT() se usa para contar filas en una consulta.  
Puede contar todas las filas (COUNT(\*)) o sólo las no nulas de una columna específica (COUNT(columna)).  
Es útil para obtener totales, frecuencias o verificar cuántos valores cumplen cierta condición.  
Se usa comúnmente con GROUP BY o en filtros.

CREATE TABLE empleados (

id INT,

nombre VARCHAR(50),

departamento VARCHAR(50)

);

INSERT INTO empleados (id, nombre, departamento) VALUES

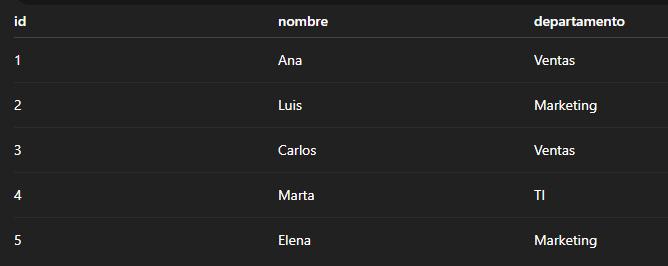
(1, 'Ana', 'Ventas'),

(2, 'Luis', 'Marketing'),

(3, 'Carlos', 'Ventas'),

(4, 'Marta', 'TI'),

(5, 'Elena', 'Marketing');



Dada una tabla empleados, cuenta cuántos empleados hay registrados en total.  
Luego, cuenta cuántos empleados hay por cada departamento.

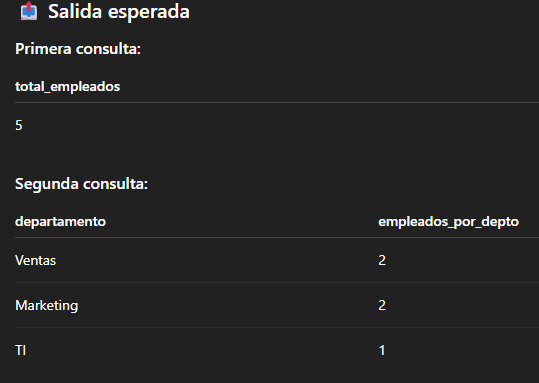
Select count(\*) as total\_empleados

From empleados;

Select departamento, count(\*) as total\_empleados

From empleados

Group by departamento;



## Ejercicio 3

CREATE TABLE clientes (

id INT,

nombre VARCHAR(50)

);

INSERT INTO clientes (id, nombre) VALUES

(1, 'Ana'),

(2, 'Luis'),

(3, 'Carlos'),

(4, 'Marta');

CREATE TABLE pedidos (

id INT,

cliente\_id INT,

producto VARCHAR(50)

);

INSERT INTO pedidos (id, cliente\_id, producto) VALUES

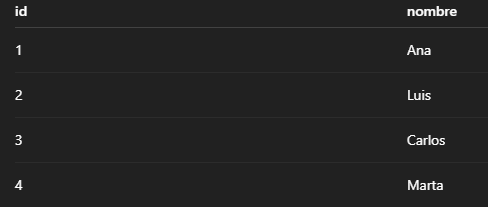
(1, 1, 'Laptop'),

(2, 2, 'Teléfono'),

(3, 1, 'Mouse'),

(4, 2, 'Monitor');

clientes



pedidos



1. ¿Cuántos pedidos ha hecho cada cliente?
2. ¿Cuántos clientes no han hecho ningún pedido?

-- 1. Pedidos por cliente

SELECT c.nombre, COUNT(p.id) AS total\_pedidos

FROM clientes c

LEFT JOIN pedidos p

ON c.id = p.cliente\_id

GROUP BY c.nombre;

-- 2. Clientes sin pedidos

SELECT COUNT(\*) AS clientes\_sin\_pedidos

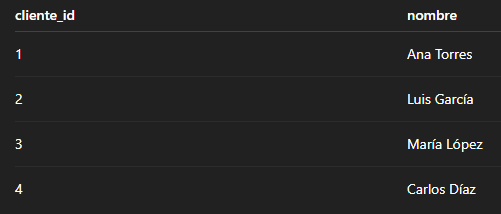
FROM clientes c

LEFT JOIN pedidos p ON c.id = p.cliente\_id

WHERE p.id IS NULL;



## Ejercicio 4





Tienes dos tablas: clientes y pedidos. Cada pedido está asociado a un cliente mediante el campo cliente\_id. Queremos saber **cuántos pedidos ha hecho cada cliente** (incluso si no ha hecho ninguno), y mostrar su nombre junto con ese número.

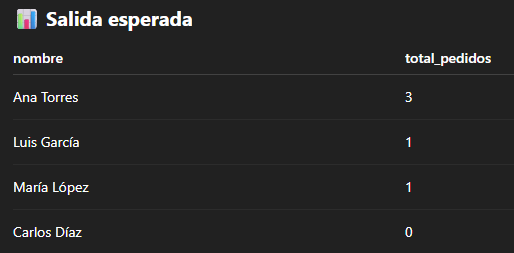
Select c.nombre, count(p.pedido\_id) as total\_pedidos

From clientes c

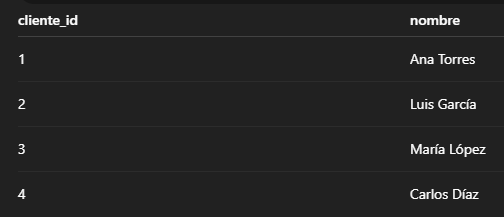
Left join pedidos p

On c.cliente\_id = p.cliente\_id

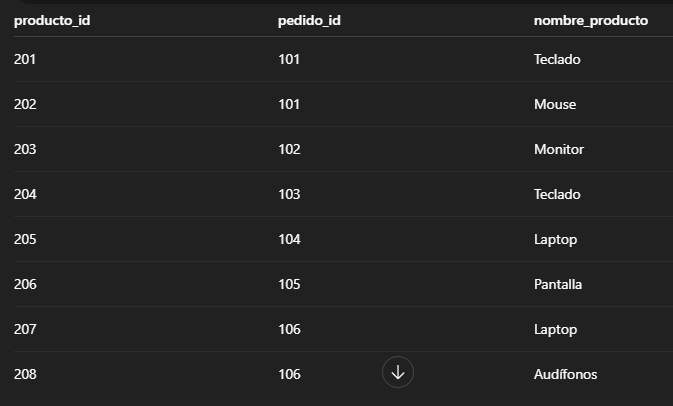
Group by c.nombre;



## Ejercicio 5\*







Tienes tres tablas: clientes, pedidos, y productos\_pedidos. Queremos obtener una lista de **clientes que han realizado al menos 2 pedidos distintos que incluyeron productos diferentes**. Muestra el nombre del cliente y el número de pedidos distintos que ha hecho.

SELECT

c.nombre,

COUNT(DISTINCT p.pedido\_id) AS num\_pedidos

FROM

clientes c

JOIN

pedidos p ON c.cliente\_id = p.cliente\_id

JOIN

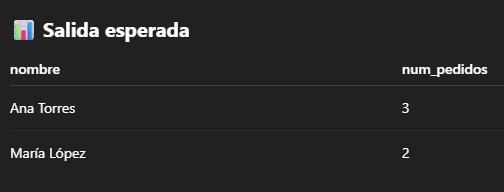
productos\_pedidos pp ON p.pedido\_id = pp.pedido\_id

GROUP BY

c.nombre

HAVING

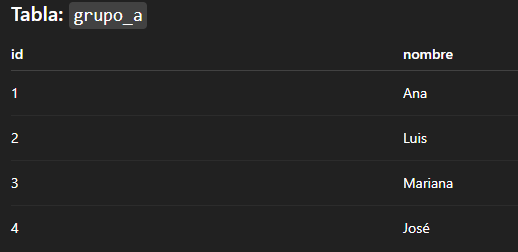
COUNT(DISTINCT p.pedido\_id) >= 2;

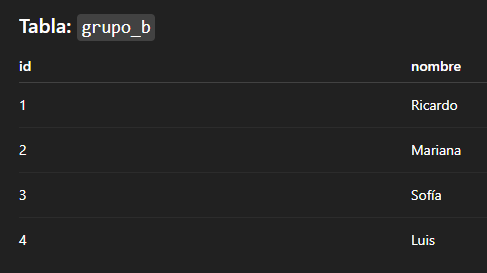


# Union

## Ejercicio 1

Supón que tienes dos tablas con nombres de estudiantes de **dos diferentes grupos**. Crea una consulta que **muestre todos los nombres de los estudiantes sin repetir nombres**, es decir, una **lista única** combinando ambas tablas.

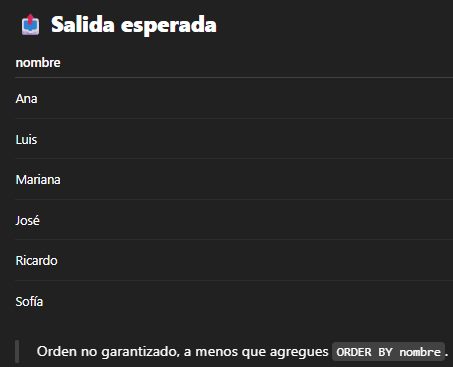




SELECT nombre FROM grupo\_a

UNION

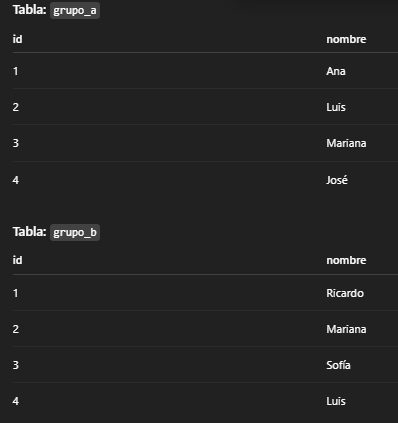
SELECT nombre FROM grupo\_b;



# Union all

## Ejercicio 1

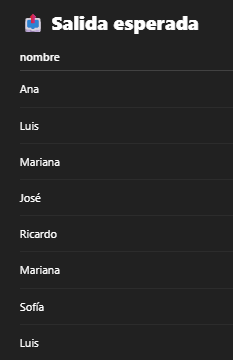
muestra una lista de **todos los nombres de estudiantes**, incluyendo **los nombres repetidos**.



SELECT nombre FROM grupo\_a

UNION ALL

SELECT nombre FROM grupo\_b;



Como puedes ver, aparecen dos veces “Mariana” y dos veces “Luis” porque están en ambas tablas, y UNION ALL **no los elimina**.

# Case, when

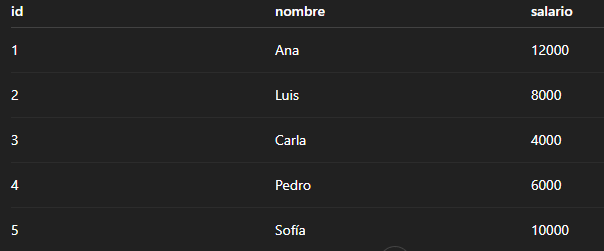
## Ejercicio 1

CASE WHEN en SQL permite realizar **evaluaciones condicionales** dentro de las consultas. Es útil para crear columnas derivadas que dependen de ciertas condiciones, como clasificar datos, reemplazar valores o aplicar lógicas "if-else". Se puede usar en SELECT, ORDER BY, WHERE, entre otros.

Clasifica a los empleados según su salario en tres categorías:

* "Alto" si el salario es mayor o igual a 10000
* "Medio" si está entre 6000 y 9999
* "Bajo" si es menor a 6000

Usa CASE WHEN para agregar una columna llamada nivel\_salarial.



SELECT

nombre,

salario,

CASE

WHEN salario >= 10000 THEN 'Alto'

WHEN salario >= 6000 AND salario < 10000 THEN 'Medio'

ELSE 'Bajo'

END AS nivel\_salarial

FROM empleados;



## Ejercicio 2

CREATE TABLE empleados\_ventas (

id\_empleado INT,

nombre VARCHAR(50),

region VARCHAR(20),

ventas\_mensuales INT

);

INSERT INTO empleados\_ventas (id\_empleado, nombre, region, ventas\_mensuales)

VALUES

(1, 'Laura', 'Norte', 12000),

(2, 'Carlos', 'Sur', 8000),

(3, 'Ana', 'Norte', 16000),

(4, 'Pedro', 'Centro', 4000),

(5, 'Lucía', 'Sur', 10000),

(6, 'Miguel', 'Centro', 2000),

(7, 'Sofía', 'Norte', 6000);



Escribe una consulta SQL que devuelva las columnas nombre, region, ventas\_mensuales y una columna adicional llamada clasificacion de acuerdo al siguiente criterio:

* Si ventas\_mensuales > 12,000 → 'Excelente'
* Si está entre 8,000 y 12,000 inclusive → 'Bueno'
* Si está entre 5,000 y 7,999 → 'Regular'
* Si es menor a 5,000 → 'Deficiente'

Select nombre, región, ventas\_mensuales,

Case

When ventas\_mensuales > 12000 then ‘Excelente’

When ventas\_mensuales >= 8000 and ventas\_mensuales <= 12000 then ‘Bueno’

When ventas\_mensuales > 5000 and ventas\_mensuales < 8000 then ‘Regular’

Else ‘Deficiente’

End as calificación

From empleados;

Si lo hacemos con between

SELECT

nombre,

region,

ventas\_mensuales,

CASE

WHEN ventas\_mensuales > 12000 THEN 'Excelente'

WHEN ventas\_mensuales BETWEEN 8000 AND 12000 THEN 'Bueno'

WHEN ventas\_mensuales BETWEEN 5000 AND 7999 THEN 'Regular'

ELSE 'Deficiente'

END AS clasificacion

FROM empleados\_ventas;



# Trabajar con fechas

## Ejercicio 1

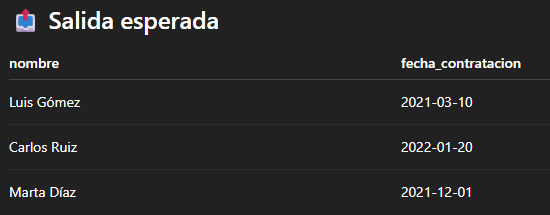
Dada una tabla de empleados con su nombre, fecha de contratación y salario, escribe una consulta para obtener los nombres de los empleados que fueron contratados **después del 1 de enero de 2021**.



SELECT nombre, fecha\_contratacion

FROM empleados

WHERE fecha\_contratacion > '2021-01-01';



## Ejercicio 2



Dada una tabla de órdenes de compra con el nombre del cliente, la fecha de la orden y el total pagado, escribe una consulta que:

* Muestre el **mes** y el **año** de cada orden,
* El **total de ventas por mes**,
* Sólo incluya las ventas hechas en el **año 2022**,
* Ordene los resultados por mes de forma cronológica.

SELECT

EXTRACT(MONTH FROM fecha\_orden) AS mes,

EXTRACT(YEAR FROM fecha\_orden) AS anio,

SUM(total) AS total\_mensual

FROM ordenes

WHERE EXTRACT(YEAR FROM fecha\_orden) = 2022

GROUP BY EXTRACT(YEAR FROM fecha\_orden), EXTRACT(MONTH FROM fecha\_orden)

ORDER BY anio, mes;

Note que en este caso es lo mismo utilizar en where y el having (correctamente) ya que tendríamos el mismo resultado



## Ejercicio 3



Calcula el tiempo que cada empleado ha trabajado en la empresa hasta el **1 de julio de 2025**, expresado en **años completos**. Luego, **filtra** sólo a los empleados que han trabajado **más de 3 años completos**.

Devuelve el nombre, fecha de contratación, años trabajados y salario.

Select nombre, fecha\_contratacion, salario

From contrataciones

Where extract(year from fecha\_contratacion) = 2022 and extract(month from fecha\_contratacion) = 7 and extract(day from fecha\_contratacion) = 1;

La consulta previa solo filtra aquellos que fueron contratados el 1 de julio de 2025

SELECT

nombre,

fecha\_contratacion,

FLOOR(DATEDIFF('2025-07-01', fecha\_contratacion) / 365) AS anios\_trabajados,

salario

FROM empleados

WHERE FLOOR(DATEDIFF('2025-07-01', fecha\_contratacion) / 365) > 3;

FLOOR(DATEDIFF('2025-07-01', fecha\_contratacion) / 365) AS anios\_trabajados Esta expresión calcula **cuántos años completos** han pasado desde la fecha de contratación hasta el **1 de julio de 2025**.

1. DATEDIFF('2025-07-01', fecha\_contratacion)

* Esta función calcula la **diferencia en días** entre dos fechas.
* Por ejemplo, si fecha\_contratacion = '2020-01-15', entonces:

DATEDIFF('2025-07-01', '2020-01-15') = número total de días entre esas fechas ≈ 1994 días

2. DATEDIFF(...) / 365

* Se convierte esa cantidad de días en **años decimales**, dividiendo entre 365 días por año.

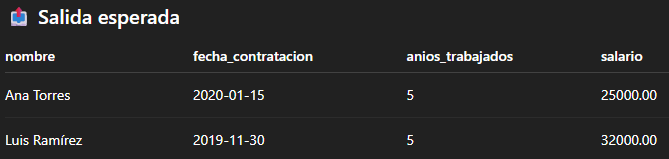
1994 / 365 ≈ 5.46 años

3. FLOOR(...)

* La función FLOOR() redondea **hacia abajo** al entero más cercano.

FLOOR(5.46) = 5

Es decir, **no se cuentan los años incompletos**, sólo los **años terminados**.



# Inner join

## Ejercicio 1

-- Tabla de clientes

CREATE TABLE Clientes (

id\_cliente INT,

nombre VARCHAR(50)

);

-- Tabla de pedidos

CREATE TABLE Pedidos (

id\_pedido INT,

id\_cliente INT,

producto VARCHAR(50)

);

-- Datos para Clientes

INSERT INTO Clientes VALUES (1, 'Ana'), (2, 'Luis'), (3, 'María');

-- Datos para Pedidos

INSERT INTO Pedidos VALUES (101, 1, 'Laptop'),

(102, 2, 'Teclado'),

(103, 1, 'Mouse');



Haz una consulta que muestre todos los pedidos junto con el nombre del cliente que los hizo. Solo deben mostrarse los pedidos que tengan un cliente registrado.

**La tabla que contiene la llave foránea (FK)** es la **tabla dependiente** o **tabla hija**

Select p.id\_pedido, c.nombre, p.producto

From clientes c

Inner join pedidos p

On c.id\_cliente = p.id\_cliente;

En este caso particular hubiera sido completamente equivalente primero poner pedidos y luego clientes



Nota:

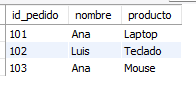
Note que si cambiamos adecuadamente la tabla A y B y hacemos un left join obtenemos lo mismo:

Select p.id\_pedido, c.nombre, p.producto

From pedidos p

left join clientes c

On c.id\_cliente = p.id\_cliente;



## Ejercicio 2

CREATE TABLE Clientes (

cliente\_id INT PRIMARY KEY,

nombre VARCHAR(100)

);

CREATE TABLE Pedidos (

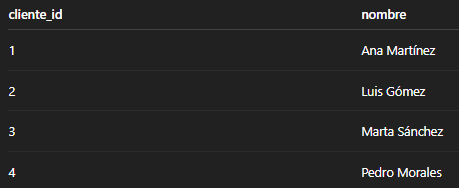
pedido\_id INT PRIMARY KEY,

cliente\_id INT,

fecha DATE,

FOREIGN KEY (cliente\_id) REFERENCES Clientes(cliente\_id)

);





Tienes dos tablas: Clientes y Pedidos. Queremos obtener un listado con los **nombres de los clientes** que **han hecho al menos un pedido**, junto con el **ID y la fecha de su pedido**.

Select c.nombre, p.pedido\_id, p.fecha

From pedidos p

Inner join clientes c

On c.cliente\_id = p.cliente\_id;

Note que en este caso (y en general para los inner jooin es indiferente poner primero una tabla u otra ya que es una intersección)

SELECT c.nombre, p.pedido\_id, p.fecha

FROM Clientes c

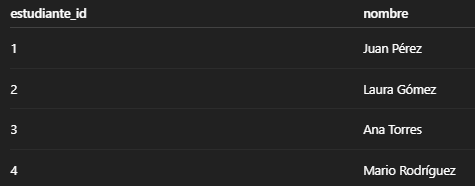
INNER JOIN Pedidos p

ON c.cliente\_id = p.cliente\_id;



## Ejercicio 3

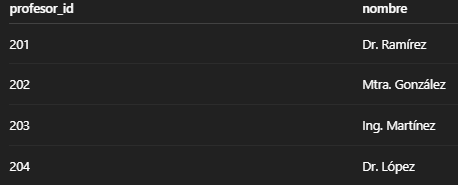
estudiantes



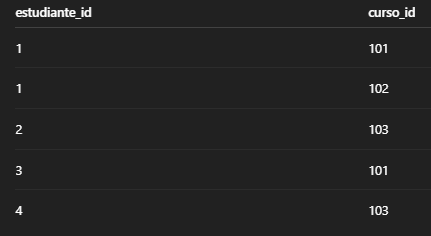
cursos



profesores



inscripciones



Tienes tres tablas: Estudiantes, Cursos, y Profesores. Cada estudiante puede inscribirse en varios cursos. Cada curso es impartido por un profesor. Queremos obtener un listado con:

* Nombre del estudiante
* Nombre del curso
* Nombre del profesor que imparte ese curso

Solo se deben mostrar los estudiantes **que están inscritos en algún curso** y sus respectivos profesores.

* Cursos y profesores: profesores es la tabla hija de cursos, con lo que la tabla cursos es la tabla padre
* Estudiantes, inscripciones y cursos: inscripciones es la tabla hija de las tablas cursos y estudiantes. La tabla inscripciones actúa como una tabla intermedia (tabla puente o tabla de relación)

SELECT

e.nombre AS nombre\_estudiante,

c.nombre AS nombre\_curso,

p.nombre AS nombre\_profesor

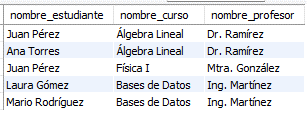
FROM inscripciones i

JOIN estudiantes e ON i.estudiante\_id = e.estudiante\_id

JOIN cursos c ON i.curso\_id = c.curso\_id

JOIN profesores p ON c.profesor\_id = p.profesor\_id;

Salida:



las consultas con múltiples JOIN pueden ser difíciles de leer porque son **dependientes del estado anterior**. Cada JOIN transforma el resultado de la consulta y hay que entenderlas como una secuencia lógica, no como algo que ocurre "todo al mismo tiempo". Pero con práctica, técnicas de descomposición, y uso de alias o CTEs, se vuelve mucho más manejable.

# Left join

## Ejercicio 1

CREATE TABLE empleados (

id\_empleado INT PRIMARY KEY,

nombre VARCHAR(50)

);

INSERT INTO empleados VALUES

(1, 'Ana'),

(2, 'Luis'),

(3, 'Marta'),

(4, 'Jorge');

CREATE TABLE proyectos (

id\_proyecto INT PRIMARY KEY,

id\_empleado INT,

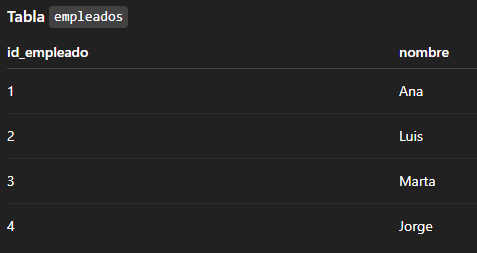
nombre\_proyecto VARCHAR(50)

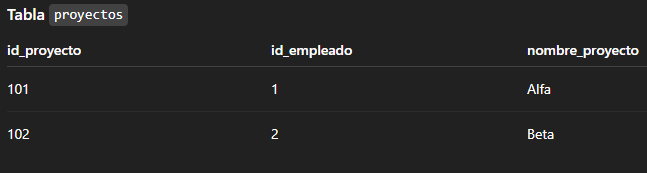
);

INSERT INTO proyectos VALUES

(101, 1, 'Alfa'),

(102, 2, 'Beta');





Genera una consulta que muestre **todos los empleados**, incluso aquellos que **no están asignados a ningún proyecto**, y muestra el nombre del proyecto si lo tienen.

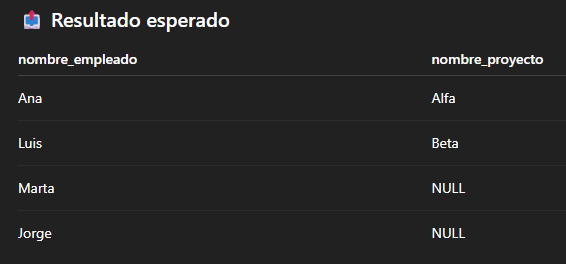
La tabla padre es empleados (A) y la tabla hija (B) es la de proyectos

Select e.nombre as nombre\_empleado, p.nombre\_proyecto

From empleados e

Left join proyectos p

On e.id\_empleado = p.id\_empleado;



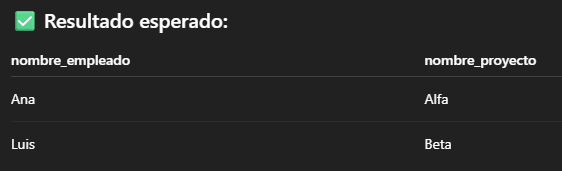
Si el resultado hubiera sido:

Select e.nombre as nombre\_empleado, p.nombre\_proyecto

From proyectos p

Left join empelados e

On e.id\_empleado = p.id\_empleado;



## Ejercicio 2

-- Crear tablas

CREATE TABLE clientes (

cliente\_id INT PRIMARY KEY,

nombre VARCHAR(100),

ciudad VARCHAR(100)

);

CREATE TABLE pedidos (

pedido\_id INT PRIMARY KEY,

cliente\_id INT,

fecha DATE,

total DECIMAL(10, 2),

FOREIGN KEY (cliente\_id) REFERENCES clientes(cliente\_id)

);

-- Insertar datos

INSERT INTO clientes (cliente\_id, nombre, ciudad) VALUES

(1, 'Ana Torres', 'Madrid'),

(2, 'Luis García', 'Barcelona'),

(3, 'Marta López', 'Sevilla'),

(4, 'Pablo Ruiz', 'Valencia');

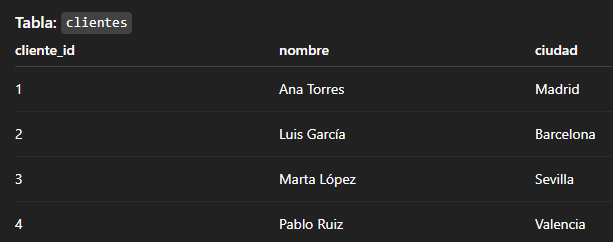
INSERT INTO pedidos (pedido\_id, cliente\_id, fecha, total) VALUES

(101, 1, '2023-01-10', 150),

(102, 2, '2023-01-12', 200),

(103, 1, '2023-01-15', 300),

(104, 2, '2023-01-18', 120);





Tienes dos tablas: clientes y pedidos. Queremos listar todos los **clientes**, junto con la información de sus pedidos (si los tienen). Si un cliente no ha hecho ningún pedido, igualmente debe aparecer en la lista con los campos de pedido en NULL.

Select c.cliente\_id, c.nombre, c.ciudad, p.fecha, p.total

From clientes c

Left join pedidos p

On c.cliente\_id = p.cliente\_id;



## Ejercicio 3

-- Crear tablas

CREATE TABLE empleados (

empleado\_id INT PRIMARY KEY,

nombre VARCHAR(100),

departamento VARCHAR(100)

);

CREATE TABLE proyectos (

proyecto\_id INT PRIMARY KEY,

nombre\_proyecto VARCHAR(100),

empleado\_id INT,

FOREIGN KEY (empleado\_id) REFERENCES empleados(empleado\_id)

);

CREATE TABLE horas\_trabajadas (

proyecto\_id INT,

empleado\_id INT,

fecha DATE,

horas INT,

FOREIGN KEY (proyecto\_id) REFERENCES proyectos(proyecto\_id),

FOREIGN KEY (empleado\_id) REFERENCES empleados(empleado\_id)

);

-- Insertar datos

INSERT INTO empleados (empleado\_id, nombre, departamento) VALUES

(1, 'Laura Méndez', 'Finanzas'),

(2, 'Carlos Ríos', 'Tecnología'),

(3, 'Elisa Blanco', 'Marketing'),

(4, 'Andrés Salgado', 'Finanzas');

INSERT INTO proyectos (proyecto\_id, nombre\_proyecto, empleado\_id) VALUES

(101, 'Reporte Anual', 1),

(102, 'Plataforma Web', 2),

(103, 'Campaña Verano', 3),

(104, 'Auditoría Interna', 1);

INSERT INTO horas\_trabajadas (proyecto\_id, empleado\_id, fecha, horas) VALUES

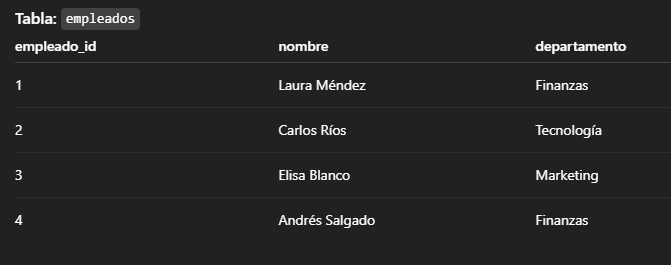
(101, 1, '2023-01-10', 5),

(101, 1, '2023-01-11', 4),

(102, 2, '2023-01-12', 6),

(103, 3, '2023-01-13', 7),

(104, 1, '2023-01-14', 3);







Tienes tres tablas:

* empleados: lista de empleados.
* proyectos: lista de proyectos asignados a empleados.
* horas\_trabajadas: número de horas que cada empleado ha trabajado en un proyecto en un día determinado.

Queremos saber **cuántas horas ha trabajado cada empleado en total, por proyecto**, incluyendo a aquellos empleados **que no tienen proyectos asignados** o que **aún no han registrado horas**.

Empleados con proyectos será un left join

Y después empleados con horas con left join

SELECT

e.empleado\_id,

e.nombre AS nombre\_empleado,

p.proyecto\_id,

p.nombre\_proyecto,

SUM(ht.horas) AS total\_horas

FROM

empleados e

LEFT JOIN proyectos p

ON e.empleado\_id = p.empleado\_id

LEFT JOIN horas\_trabajadas ht

ON e.empleado\_id = ht.empleado\_id AND p.proyecto\_id = ht.proyecto\_id

GROUP BY

e.empleado\_id, e.nombre, p.proyecto\_id, p.nombre\_proyecto

ORDER BY

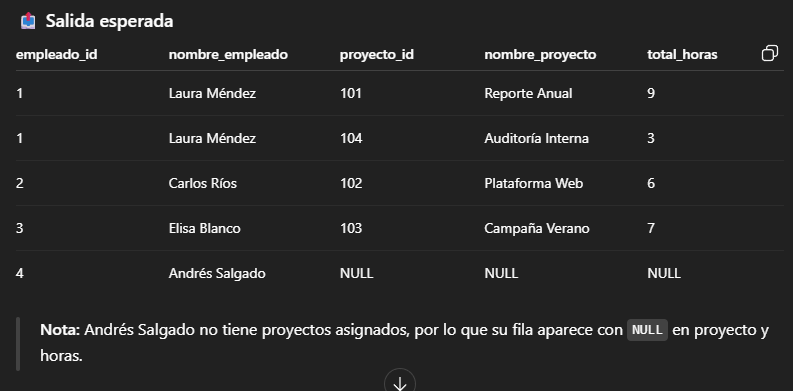
e.empleado\_id, p.proyecto\_id;

✅ **Regla general:**

**Sí**, cuando estás uniendo múltiples tablas relacionadas entre sí mediante llaves foráneas, **debes usar en la cláusula ON todas las condiciones necesarias** que aseguren que cada fila unida sea **la correcta**.

Si omites alguna condición de relación entre las tablas, puedes obtener:

* **Filas duplicadas**
* **Datos incorrectos** (por ejemplo, sumar horas del proyecto incorrecto)
* **Combinaciones inválidas** (un "producto cartesiano parcial")



# Right join

## Ejercicio 1

-- Tabla de Empleados

CREATE TABLE empleados (

id\_empleado INT,

nombre VARCHAR(50)

);

INSERT INTO empleados VALUES

(1, 'Ana'),

(2, 'Luis'),

(3, 'Carla'),

(4, 'Pedro');

-- Tabla de Proyectos

CREATE TABLE proyectos (

id\_proyecto INT,

nombre\_proyecto VARCHAR(50),

id\_empleado INT

);

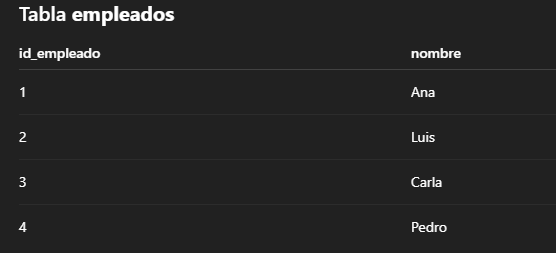
INSERT INTO proyectos VALUES

(101, 'Proyecto A', 2),

(102, 'Proyecto B', 4),

(103, 'Proyecto C', NULL),

(104, 'Proyecto D', 5);





Obtener el nombre del proyecto y el nombre del empleado asignado, mostrando **todos los proyectos** aunque no tengan empleado asignado.

Tabla padre A: empleados

Tabla hija B: proyectos

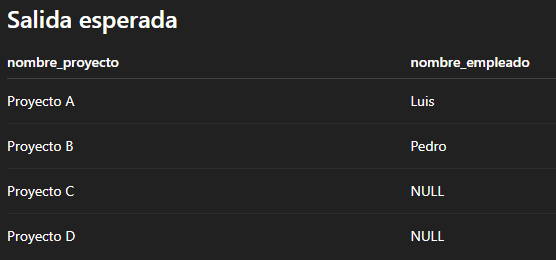
Select p.nombre\_proyecto, e.nombre as nombre\_empleado

From empleados e

Right join proyectos p

On p.id\_empleado=e.id\_empleado;

E n este caso, la tabla A es la que está a la derecha de la unión (right join)



## Ejercicio 2

-- Crear tabla empleados

CREATE TABLE empleados (

empleado\_id INT PRIMARY KEY,

nombre VARCHAR(100)

);

-- Crear tabla proyectos

CREATE TABLE proyectos (

proyecto\_id INT PRIMARY KEY,

nombre\_proyecto VARCHAR(100),

empleado\_id INT

);

-- Insertar datos en empleados

INSERT INTO empleados (empleado\_id, nombre) VALUES

(1, 'Ana García'),

(2, 'Luis Torres'),

(3, 'Marta Pérez');

-- Insertar datos en proyectos

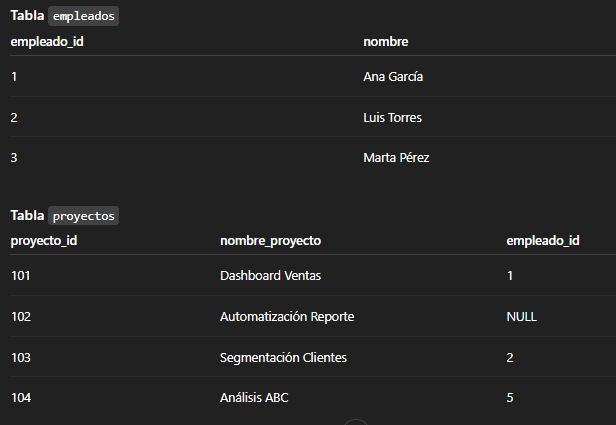
INSERT INTO proyectos (proyecto\_id, nombre\_proyecto, empleado\_id) VALUES

(101, 'Dashboard Ventas', 1),

(102, 'Automatización Reporte', NULL),

(103, 'Segmentación Clientes', 2),

(104, 'Análisis ABC', 5);



Escribe una consulta que muestre **todos los proyectos**, incluso si no tienen un empleado asignado, junto con el nombre del empleado (si existe)

SELECT

p.proyecto\_id,

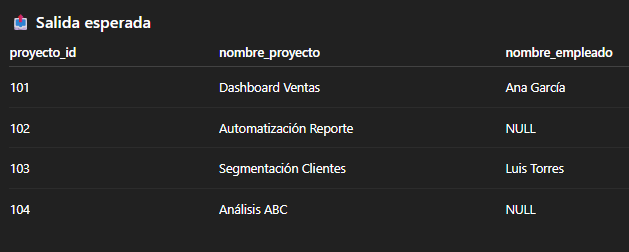
p.nombre\_proyecto,

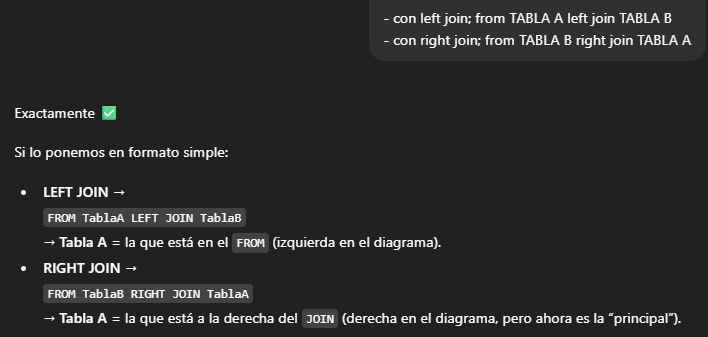
e.nombre AS nombre\_empleado

FROM empleados e

RIGHT JOIN proyectos p

ON e.empleado\_id = p.empleado\_id;





**En esencia, LEFT JOIN y RIGHT JOIN hacen lo mismo pero desde perspectivas opuestas**:

## Ejercicio 3

CREATE TABLE empleados (

empleado\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

CREATE TABLE proyectos (

proyecto\_id INT PRIMARY KEY,

nombre\_proyecto VARCHAR(50),

empleado\_id INT

);

INSERT INTO empleados (empleado\_id, nombre) VALUES

(1, 'Ana'),

(2, 'Luis'),

(3, 'Marta'),

(4, 'Carlos');

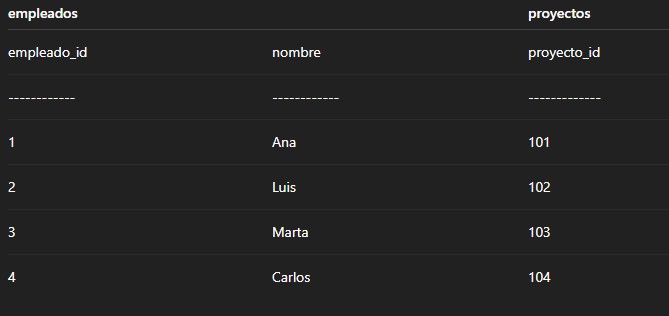
INSERT INTO proyectos (proyecto\_id, nombre\_proyecto, empleado\_id) VALUES

(101, 'Proyecto Alfa', 1),

(102, 'Proyecto Beta', NULL),

(103, 'Proyecto Gamma', 3),

(104, 'Proyecto Delta', NULL);



Queremos obtener una lista de todos los proyectos y, para cada proyecto, el nombre del empleado que está asignado.  
Pero no todos los proyectos tienen un empleado asignado aún. Por eso, queremos mostrar **todos los proyectos**, aunque no tengan empleado asignado (es decir, proyectos sin asignación deben aparecer con NULL en el nombre del empleado).

Select e.nomnbre as nombre\_empleado, p.proyecto\_id, p.proyecto\_id, p.nombre\_proyecto

From empleados e

Right join proyectos p

On e.empleado\_id = p.empleado\_id;



Misma consulta con left join:

Select e.nomnbre as nombre\_empleado, p.proyecto\_id, p.proyecto\_id, p.nombre\_proyecto

From proyectos p

Left join empleados e

On e.empleado\_id = p.empleado\_id;

# Full o cross join

## Ejercicio 1

FULL JOIN devuelve **todos los registros** de ambas tablas, combinando los que tienen coincidencias en la clave indicada y dejando NULL en las columnas faltantes donde no hay coincidencia. Es útil para analizar datos incompletos o diferencias entre conjuntos.

CREATE TABLE Empleados (

ID INT,

Nombre VARCHAR(50)

);

CREATE TABLE Proyectos (

ID INT,

Proyecto VARCHAR(50)

);



Escribe una consulta para mostrar todos los empleados y proyectos, aunque no todos coincidan en ID.

Select e.nombre, p.proyectos

From empleados e

Full join proyectos p

On e.id=p.id;



Nota:

En un INNER JOIN, **el orden de las tablas no afecta el resultado final**, porque solo se devuelven las filas que tienen coincidencia en ambas tablas

En un FULL JOIN, **también es indistinto el orden de las tablas**

Casos donde el orden **SÍ importa**:

✅ En LEFT JOIN o RIGHT JOIN, porque la tabla que pongas primero es la que conserva todos sus registros.

Pero… Si inviertes el orden de las tablas, puedes obtener el mismo resultado.



Left join:

SELECT c.nombre\_cliente, o.producto

FROM Clientes c

LEFT JOIN Ordenes o

ON c.id\_cliente = o.id\_cliente;

Consulta equivalente con right join (esta consulta equivale a un left join, pero ejecutada con tablas invertidas en el right join):

SELECT c.nombre\_cliente, o.producto

FROM Ordenes o

RIGHT JOIN Clientes c

ON c.id\_cliente = o.id\_cliente;

Solo asegúrate de ajustar el **orden de las tablas y los campos en el SELECT** para que se vea igual.

Conclusión: Sí, usando LEFT JOIN o RIGHT JOIN puedes lograr consultas equivalentes, **siempre que se inviertan adecuadamente las tablas**.

## Ejercicio 2

-- Crear tabla Clientes

CREATE TABLE Clientes (

cliente\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

-- Crear tabla Pedidos

CREATE TABLE Pedidos (

pedido\_id INT PRIMARY KEY,

cliente\_id INT,

producto VARCHAR(50)

);

-- Insertar datos en Clientes

INSERT INTO Clientes (cliente\_id, nombre) VALUES

(1, 'Ana López'),

(2, 'Juan Pérez'),

(3, 'María Ruiz'),

(4, 'Luis Gómez');

-- Insertar datos en Pedidos

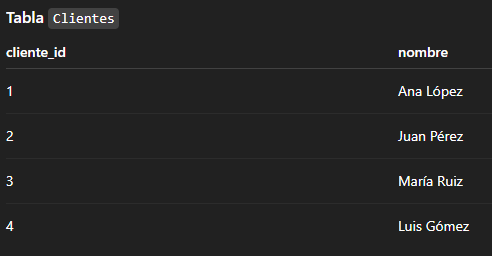
INSERT INTO Pedidos (pedido\_id, cliente\_id, producto) VALUES

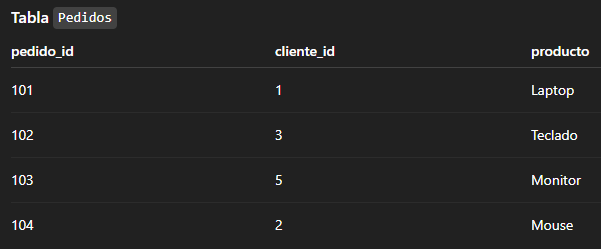
(101, 1, 'Laptop'),

(102, 3, 'Teclado'),

(103, 5, 'Monitor'),

(104, 2, 'Mouse');





Usando FULL JOIN, obtener una lista que muestre **todos los clientes y todos los pedidos**, incluyendo:

* cliente\_id, nombre del cliente (si existe),
* pedido\_id y producto (si existe).

Select c.nombre, c.cliente\_id, p.pedido p.pedido\_id

From clienetes c

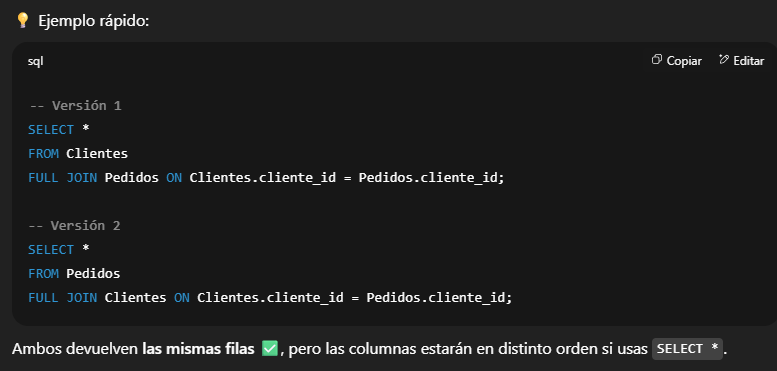
From pedidos p

On e.cliente\_id = p.productos\_id

Order by c.cliente\_id;

en un FULL JOIN **no importa cuál tabla pongas a la izquierda o a la derecha**





## Ejercicio 3

-- Crear tabla Clientes

CREATE TABLE Clientes (

cliente\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

-- Crear tabla Pedidos

CREATE TABLE Pedidos (

pedido\_id INT PRIMARY KEY,

cliente\_id INT,

producto VARCHAR(50)

);

-- Crear tabla Pagos

CREATE TABLE Pagos (

pago\_id INT PRIMARY KEY,

pedido\_id INT,

monto DECIMAL(10,2)

);

-- Insertar datos

INSERT INTO Clientes (cliente\_id, nombre) VALUES

(1, 'Ana López'),

(2, 'Juan Pérez'),

(3, 'María Ruiz'),

(4, 'Luis Gómez');

INSERT INTO Pedidos (pedido\_id, cliente\_id, producto) VALUES

(101, 1, 'Laptop'),

(102, 3, 'Teclado'),

(103, 5, 'Monitor'),

(104, 2, 'Mouse');

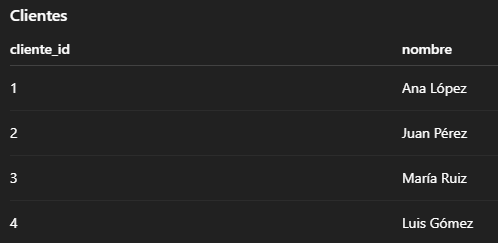
INSERT INTO Pagos (pago\_id, pedido\_id, monto) VALUES

(201, 101, 1500),

(202, 102, 200),

(203, 106, 400),

(204, 104, 50);







Mediante **FULL JOIN**, obtén una tabla que muestre:

* cliente\_id
* nombre del cliente (o "Cliente desconocido" si no existe)
* pedido\_id y producto (o "Sin pedido" si no hay)
* pago\_id y monto (o 0 si no hay pago)

**Ordena** por cliente\_id ascendente y luego por pedido\_id.

Select c.cliente\_id, c.nombre as nombre\_cliente, p.producto\_id, p.producto as nombre\_producto, pa.pago\_id, pa.monto

From clientes c

Full join pedidos p

On c.cliente\_id = p.cliente\_id

Full join pagos pa

On p.pedido\_id = pa.pedido\_id

Order by c.cliente\_id, p.pedido\_id;

SELECT

COALESCE(C.cliente\_id, P.cliente\_id) AS cliente\_id,

COALESCE(C.nombre, 'Cliente desconocido') AS nombre,

COALESCE(P.pedido\_id, Pg.pedido\_id) AS pedido\_id,

COALESCE(P.producto, 'Sin pedido') AS producto,

COALESCE(Pg.pago\_id, 0) AS pago\_id,

COALESCE(Pg.monto, 0) AS monto

FROM Clientes C

FULL JOIN Pedidos P

ON C.cliente\_id = P.cliente\_id

FULL JOIN Pagos Pg

ON P.pedido\_id = Pg.pedido\_id

ORDER BY cliente\_id, pedido\_id;



# Self join (Unión Automática)

 Un **self join** es un **join de una tabla consigo misma**.

 Se usa cuando necesitas comparar filas de la misma tabla.

 Generalmente se da un **alias** diferente a cada "versión" de la tabla.

## Ejercicio 1



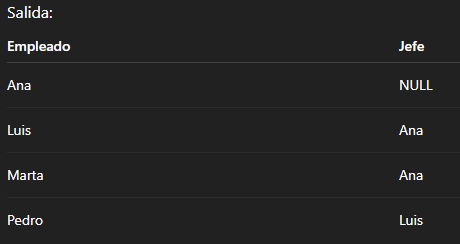
quién es jefe de quién

SELECT e.nombre AS Empleado, j.nombre AS Jefe

FROM Empleados e

LEFT JOIN Empleados j

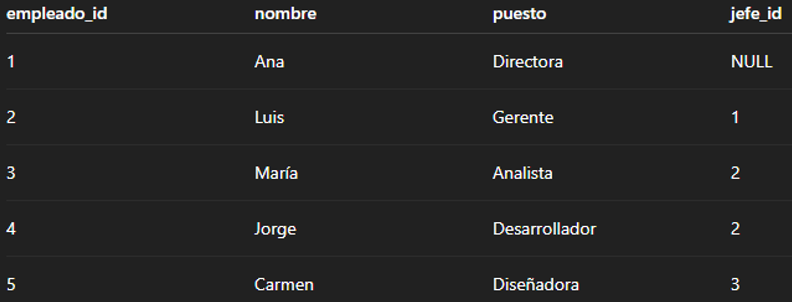
ON e.id\_jefe = j.id;



## Ejercicio 2

Tienes una tabla llamada **Empleados** que contiene información de los trabajadores de una empresa y su **jefe inmediato**.  
Necesitas obtener el **nombre de cada empleado junto con el nombre de su jefe**.



A la siguiente tabla le daremos el alias “j”

SELECT

e.nombre AS Empleado,

e.puesto AS Puesto,

j.nombre AS Jefe

FROM Empleados e

LEFT JOIN Empleados j

ON e.jefe\_id = j.empleado\_id;



# Subconsultas

Una **subconsulta** es una consulta dentro de otra consulta.  
Se utiliza para devolver un valor o un conjunto de valores que luego son usados por la consulta principal.

## Subconsultas escalares (devuelven un solo valor)

Aquí lo **más común** es usar funciones de agregación (AVG, MAX, MIN, COUNT, SUM).

SELECT nombre, salario

FROM Empleados

WHERE salario > (

SELECT AVG(salario)

FROM Empleados

);

## Subconsultas que devuelven conjuntos de filas

Aquí sí se **llama directamente a tablas** sin necesidad de agregación.

SELECT nombre

FROM Empleados

WHERE departamento\_id IN (

SELECT id

FROM Departamentos

WHERE region = 'Norte'

);

 La subconsulta devuelve una **lista de valores** (los id de departamentos).

 No usamos agregación, sino que simplemente consultamos una tabla.

## Subconsultas correlacionadas

Se usan cuando la subconsulta depende de la fila externa. Pueden o no usar agregación.

SELECT e.nombre, e.salario

FROM Empleados e

WHERE e.salario > (

SELECT AVG(salario)

FROM Empleados

WHERE departamento\_id = e.departamento\_id

);

## Ejercicio 1

CREATE TABLE Empleados (

id\_empleado INT PRIMARY KEY,

nombre VARCHAR(50),

departamento VARCHAR(50),

salario INT

);

INSERT INTO Empleados (id\_empleado, nombre, departamento, salario) VALUES

(1, 'Ana', 'Ventas', 1200),

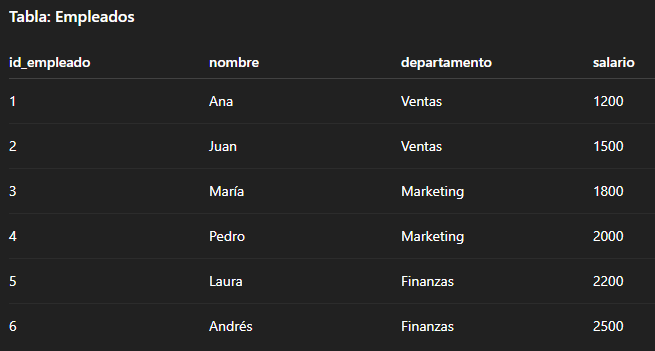
(2, 'Juan', 'Ventas', 1500),

(3, 'María', 'Marketing', 1800),

(4, 'Pedro', 'Marketing', 2000),

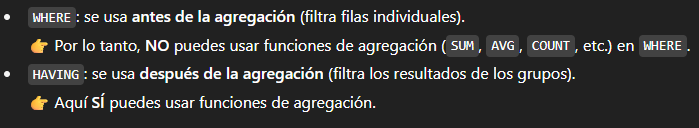
(5, 'Laura', 'Finanzas', 2200),

(6, 'Andrés', 'Finanzas', 2500);



Obtén los **empleados cuyo salario es mayor que el salario promedio de todos los empleados** usando una **subconsulta**.

Las funciones de agregación no se pueden usar en el where, en el having si:



SELECT nombre, departamento, salario

FROM Empleados

WHERE salario > (

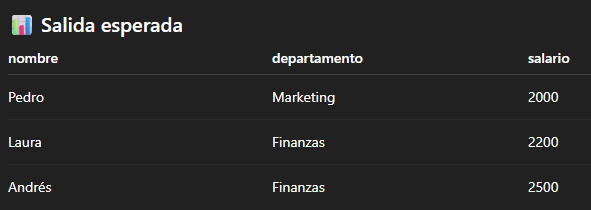
SELECT AVG(salario)

FROM Empleados

);

 **Lógicamente** devuelve **un único valor** (el promedio de todos los salarios).

 **Técnicamente en SQL**, toda consulta devuelve un **resultado en forma de tabla**.



La subconsulta en este caso la podemos entender como una “función agregada”

## Ejercicio 2

CREATE TABLE Empleados (

id\_empleado INT PRIMARY KEY,

nombre VARCHAR(50),

departamento VARCHAR(50),

salario INT

);

INSERT INTO Empleados (id\_empleado, nombre, departamento, salario) VALUES

(1, 'Ana', 'Ventas', 1200),

(2, 'Juan', 'Ventas', 1500),

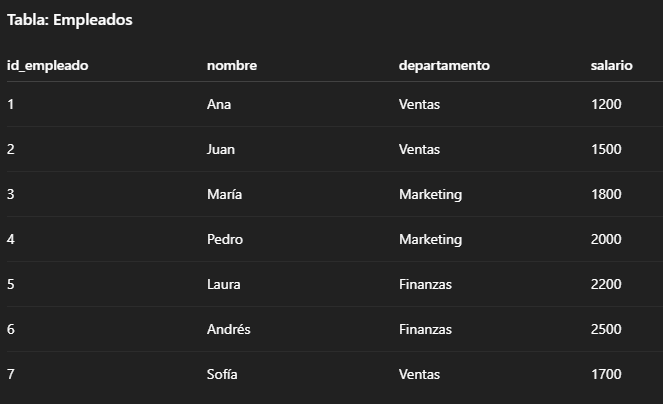
(3, 'María', 'Marketing', 1800),

(4, 'Pedro', 'Marketing', 2000),

(5, 'Laura', 'Finanzas', 2200),

(6, 'Andrés', 'Finanzas', 2500),

(7, 'Sofía', 'Ventas', 1700);



Obtén los **empleados que tienen el salario más alto dentro de su departamento** (uno por departamento), usando una **subconsulta correlacionada**.

SELECT nombre, departamento, salario

FROM Empleados e1

WHERE salario = (

SELECT MAX(salario)

FROM Empleados e2

WHERE e1.departamento = e2.departamento

);



 La consulta externa toma cada empleado e1.

 La subconsulta interna busca el **máximo salario en el mismo departamento** (WHERE e1.departamento = e2.departamento).

 Si el salario de e1 coincide con ese máximo → el empleado se incluye.

## Ejercicio 3

# Procedimientos almacenados

Un **procedimiento almacenado** (stored procedure) en SQL es un conjunto de instrucciones SQL que se almacenan en la base de datos y que se pueden ejecutar de forma repetida. Sirve para:

✅ ¿Para qué sirve un procedimiento almacenado?

1. **Reutilización de código**: Evita repetir las mismas consultas SQL.
2. **Modularización**: Permite encapsular lógica compleja y dividirla en bloques reutilizables.
3. **Mejora el rendimiento**: El código se compila una vez y se puede ejecutar múltiples veces.
4. **Seguridad**: Puedes restringir el acceso a los datos y permitir solo la ejecución del procedimiento.
5. **Mantenimiento centralizado**: Si necesitas hacer cambios, los haces solo en el procedimiento.

Ejemplo sencillo

Select \* from limpieza; -- se quiere evitar escribirlo repetidamente

-- ----- crear el procedimiento para consultar sin escribir toda la consulta

DELIMITER //

CREATE PROCEDURE limp()

BEGIN

SELECT \* FROM limpieza;

END //

DELIMITER ;

-- ejecutar el procedimiento

CALL limp;

## Ejercicio 1

CREATE TABLE Empleados (

id INT PRIMARY KEY,

nombre VARCHAR(100),

departamento VARCHAR(50),

salario DECIMAL(10,2)

);

**Calcular el salario promedio por departamento**

Crea un procedimiento almacenado que reciba como parámetro el nombre de un departamento y devuelva el **salario promedio** de los empleados que pertenecen a él.

DELIMITER $$

CREATE PROCEDURE sp\_SalarioPromedioDepartamento(IN p\_Depto VARCHAR(50))

BEGIN

SELECT

departamento,

AVG(salario) AS SalarioPromedio

FROM Empleados

WHERE departamento = p\_Depto

GROUP BY departamento;

END $$

DELIMITER ;

CALL sp\_SalarioPromedioDepartamento('Ventas');

## Ejercicio 2

**Insertar un nuevo registro con validación**

Crea un procedimiento que permita insertar un nuevo empleado en la tabla Empleados, pero con la condición de que **no se puede repetir el nombre del empleado**.

-- Crear procedimiento almacenado

CREATE PROCEDURE sp\_InsertarEmpleado

@Nombre VARCHAR(100),

@Departamento VARCHAR(50),

@Salario DECIMAL(10,2)

AS

BEGIN

-- Verificar si el empleado ya existe

IF EXISTS (SELECT 1 FROM Empleados WHERE nombre = @Nombre)

BEGIN

PRINT 'Error: Ya existe un empleado con ese nombre.';

RETURN;

END

-- Insertar el nuevo empleado

INSERT INTO Empleados (nombre, departamento, salario)

VALUES (@Nombre, @Departamento, @Salario);

PRINT 'Empleado insertado correctamente.';

END;

## Ejercicio 3

# Funciones

Se parecen a los procedimientos almacenados, pero son distintos

Una **función** en SQL:

1. **Devuelve un valor** (obligatoriamente).
2. **Se puede usar en una consulta** como si fuera una columna (a diferencia de un procedimiento).
3. **Sirve para encapsular lógica que devuelve un resultado concreto**: cálculos, transformaciones, validaciones, etc.
4. **No puede modificar datos** directamente (no puedes hacer INSERT, UPDATE o DELETE dentro de ella en la mayoría de los RDBMS como SQL Server, PostgreSQL o MySQL).

Ejemplo sencillo

Supongamos que queremos una función que calcule el nuevo salario de un empleado después de aplicar un aumento porcentual, **pero sin modificar nada en la base de datos**, solo devolver el resultado.

CREATE TABLE Empleados (

empleado\_id INT PRIMARY KEY,

nombre VARCHAR(100),

salario DECIMAL(10, 2),

departamento VARCHAR(50)

);

Creamos la función:

DELIMITER $$

CREATE FUNCTION CalcularNuevoSalario(

p\_salario DECIMAL(10,2),

p\_porcentaje DECIMAL(5,2)

)

RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

DECLARE nuevo\_salario DECIMAL(10,2);

SET nuevo\_salario = p\_salario + (p\_salario \* p\_porcentaje / 100);

RETURN nuevo\_salario;

END $$

DELIMITER ;

La forma de uso:

Puedes usar esta función dentro de una consulta SELECT:

SELECT

nombre,

salario,

CalcularNuevoSalario(salario, 10) AS salario\_con\_aumento

FROM Empleados;

Esto te muestra el salario actual y el nuevo salario calculado con un 10% de aumento, **sin modificar los datos reales**.

## Ejemplo 1

## Ejemplo 2

## Ejemplo 3

# Trigger

Un **trigger** (o disparador) en SQL es un tipo de objeto de base de datos que se **ejecuta automáticamente en respuesta a ciertos eventos** en una tabla o vista, como INSERT, UPDATE o DELETE.

🔹 ¿Para qué sirve un trigger?

* Auditar cambios (por ejemplo, guardar quién y cuándo modificó un registro).
* Validar o forzar reglas de negocio.
* Sincronizar datos entre tablas.
* Generar automáticamente valores calculados.

Supongamos que tienes una tabla de Usuarios y quieres guardar en otra tabla cuándo se insertó un nuevo usuario.

CREATE TABLE Usuarios (

id INT PRIMARY KEY,

nombre VARCHAR(50)

);

CREATE TABLE LogInsert (

id INT IDENTITY(1,1) PRIMARY KEY,

usuario\_id INT,

fecha\_insert DATETIME

);

Crear el trigger:

CREATE TRIGGER trg\_AuditoriaInsert

ON Usuarios

AFTER INSERT

AS

BEGIN

INSERT INTO LogInsert (usuario\_id, fecha\_insert)

SELECT id, GETDATE()

FROM INSERTED;

END;

🧠 ¿Qué hace este trigger?

* Se activa **automáticamente después** (AFTER) de un INSERT en la tabla Usuarios.
* Inserta en LogInsert el id del nuevo usuario y la fecha actual (GETDATE()).
* Usa la tabla especial INSERTED, que contiene los datos recién insertados.

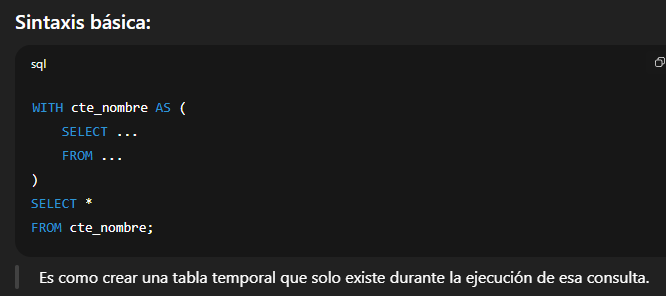
# Eventos

## Extras

# CTEs

Las **CTEs** (Common Table Expressions) en SQL son **subconsultas temporales** que se definen al inicio de una consulta con la palabra clave WITH. Sirven para:

* Hacer consultas más **legibles** y organizadas.
* **Evitar repetir** subconsultas complejas.
* Permitir operaciones **recursivas**.



## Ejercicio 1

CREATE TABLE Estudiantes (

estudiante\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

CREATE TABLE Cursos (

curso\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

CREATE TABLE Inscripciones (

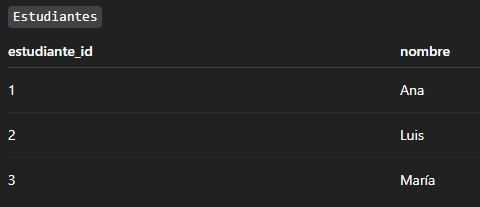
estudiante\_id INT,

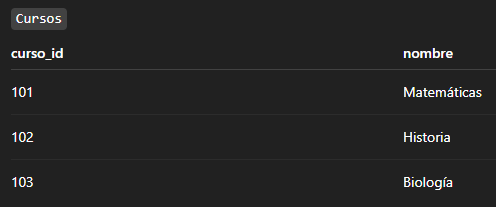
curso\_id INT,

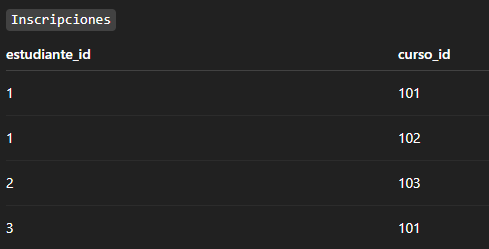
FOREIGN KEY (estudiante\_id) REFERENCES Estudiantes(estudiante\_id),

FOREIGN KEY (curso\_id) REFERENCES Cursos(curso\_id)

);







Utilizando CTEs, muestra una lista con los nombres de los estudiantes y los nombres de los cursos en los que están inscritos. Usa un CTE para unir primero la tabla Inscripciones con Estudiantes, y luego realiza una segunda unión con la tabla Cursos.

WITH EstudiantesConInscripciones AS (

SELECT

i.estudiante\_id,

e.nombre AS nombre\_estudiante,

i.curso\_id

FROM Inscripciones i

JOIN Estudiantes e ON i.estudiante\_id = e.estudiante\_id

)

SELECT

eci.nombre\_estudiante,

c.nombre AS nombre\_curso

FROM EstudiantesConInscripciones eci

JOIN Cursos c ON eci.curso\_id = c.curso\_id;

Y si queremos ver el paso a paso, tenemos que ir haciendo from en cada join:

WITH EstudiantesConInscripciones AS (

SELECT

i.estudiante\_id,

e.nombre AS nombre\_estudiante,

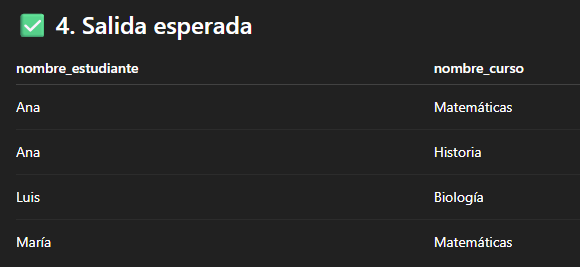
i.curso\_id

FROM Inscripciones i

JOIN Estudiantes e ON i.estudiante\_id = e.estudiante\_id

)

SELECT \* FROM EstudiantesConInscripciones;



## Ejercicio 2

-- Crear tablas

CREATE TABLE Estudiantes (

estudiante\_id INT PRIMARY KEY,

nombre VARCHAR(50)

);

CREATE TABLE Cursos (

curso\_id INT PRIMARY KEY,

nombre\_curso VARCHAR(50)

);

CREATE TABLE Inscripciones (

estudiante\_id INT,

curso\_id INT,

calificacion INT,

FOREIGN KEY (estudiante\_id) REFERENCES Estudiantes(estudiante\_id),

FOREIGN KEY (curso\_id) REFERENCES Cursos(curso\_id)

);

-- Insertar datos

INSERT INTO Estudiantes VALUES

(1, 'Ana'),

(2, 'Luis'),

(3, 'Beatriz'),

(4, 'Carlos');

INSERT INTO Cursos VALUES

(101, 'Matemáticas'),

(102, 'Historia'),

(103, 'Física');

INSERT INTO Inscripciones VALUES

(1, 101, 95),

(2, 101, 95),

(3, 101, 91),

(4, 101, 89),

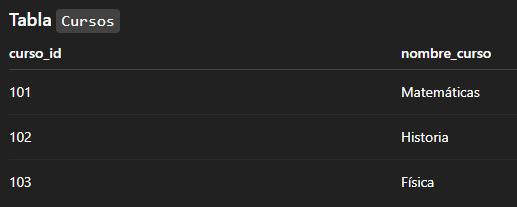
(2, 102, 78),

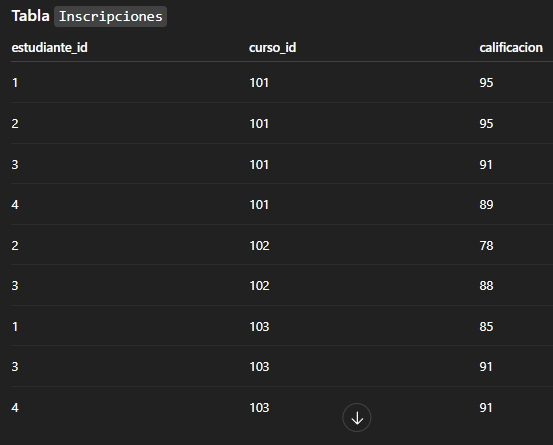
(3, 102, 88),

(1, 103, 85),

(3, 103, 91),

(4, 103, 91);



Dada una base de datos con información de estudiantes, cursos e inscripciones con calificaciones, **utiliza una CTE para encontrar el estudiante con la calificación más alta por cada curso**. Muestra:

* nombre del curso
* nombre del estudiante
* calificación

**Desempata por orden alfabético si hay más de un estudiante con la misma calificación máxima.**

WITH MaximasCalificaciones AS (

SELECT

i.curso\_id,

MAX(i.calificacion) AS max\_calificacion

FROM Inscripciones i

GROUP BY i.curso\_id

),

EstudiantesDestacados AS (

SELECT

i.curso\_id,

i.estudiante\_id,

i.calificacion

FROM Inscripciones i

JOIN MaximasCalificaciones m

ON i.curso\_id = m.curso\_id AND i.calificacion = m.max\_calificacion

)

SELECT

c.nombre\_curso,

e.nombre AS nombre\_estudiante,

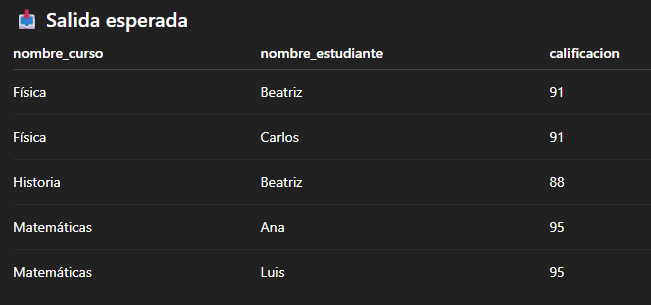
ed.calificacion

FROM EstudiantesDestacados ed

JOIN Estudiantes e ON ed.estudiante\_id = e.estudiante\_id

JOIN Cursos c ON ed.curso\_id = c.curso\_id

ORDER BY c.nombre\_curso, e.nombre



Note que la solución más práctica para este ejercicio es simplemente filtrar por calificación concierto valor y estaría lista la consulta. La ventaja que tiene cetes es que nos permite ver el flujo de cada unión

# Coalesce

# Tablas derivadas

# Funciones ventana

# Group\_concat

# Rollup