MF0226_3 – Programación de Bases de Datos Relacionales

IFCD0112 - Programación Orientada a Objetos y Bases de Datos Relacionales

Juan Manuel Piñero Sánchez

2025

Table of contents

1. Índice

Este módulo aborda el diseño, creación y administración de bases de datos relacionales en **PostgreSQL**,
incluyendo la definición de tablas, las operaciones SQL básicas, la normalización, el uso de claves,
consultas complejas, funciones, vistas y la conexión desde aplicaciones Java mediante JDBC.

1.1 Contenidos

Nota: Cada capítulo incluye ejemplos prácticos en SQL completamente funcionales para PostgreSQL.

Se recomienda disponer de una base de datos local (`postgres`) para ejecutar las sentencias de prueba.

2. 1 – Modelado y definición de tablas en PostgreSQL

2.1 Elementos gráficos del Diagrama Entidad-Relación (ER)

Elemento	Forma en el diagrama	Representa
Entidad	Rectángulo	Una tabla
Relación	Rombo	Asociación entre dos o más entidades
Atributo	Óvalo	Propiedad de una entidad o una relación
Flecha	Flecha →	Une entidad (rectángulo) a relación (rombo)

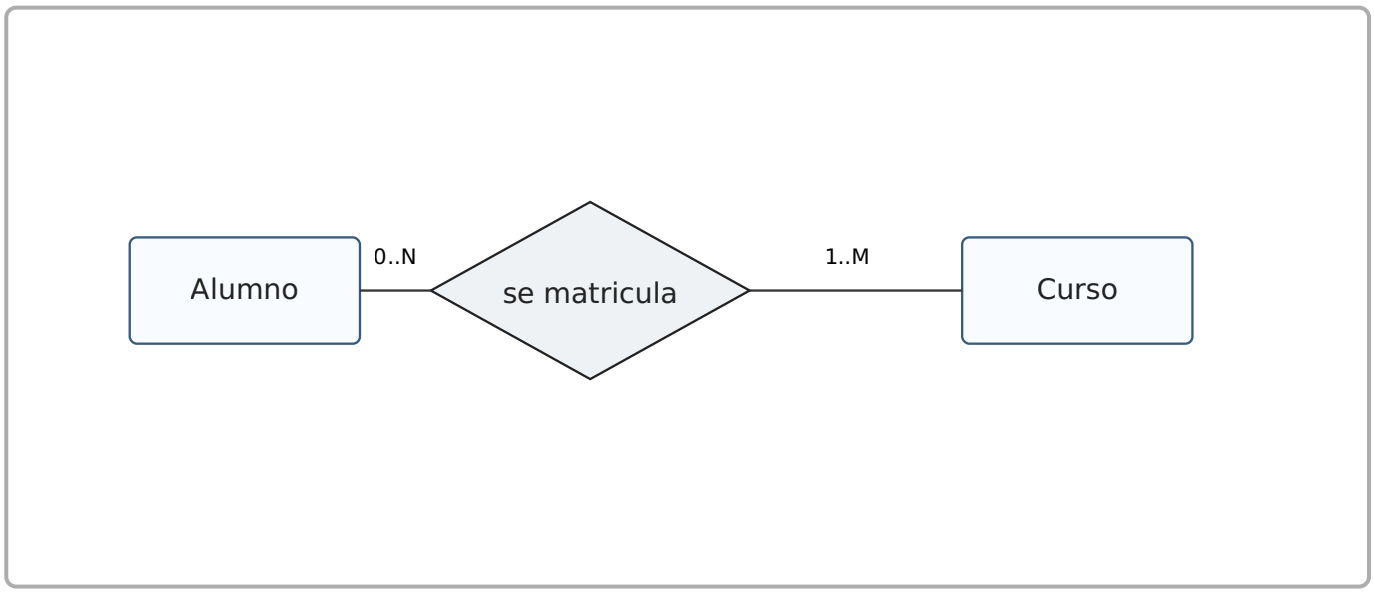
Las flechas siempre van desde una entidad hacia una relación, nunca al revés. Un atributo puede estar conectado a:

- Una entidad: se convertirá en una columna en la tabla correspondiente.
- Una relación: si la relación tiene atributos, se transforma en una tabla propia.

2.2 Representación de cardinalidades

Las cardinalidades indican cuántas veces una entidad puede intervenir en una relación. Hay dos formas de representarlas gráficamente:

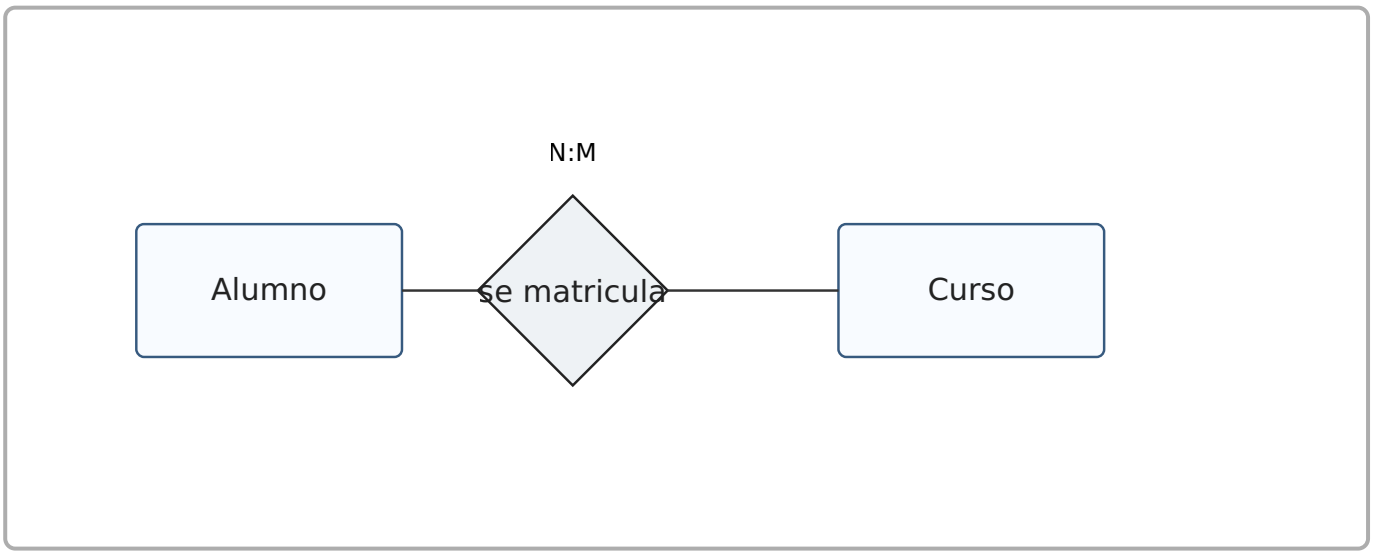
2.2.1 Forma 1 – Dos grupos de números (más precisa)



- Cada alumno puede estar matriculado de 1 o mas cursos.
- Cada curso puede tener matriculados 0 o más alumnos.

Esta forma es muy explícita, ya que indica claramente el mínimo y máximo de participación de cada entidad en la relación.

2.2.2 Forma 2 – Un solo grupo de números en el rombo



- Se entiende que un alumno puede estar inscrito en varios cursos.
- Un curso tiene varios alumnos inscritos.
- No se distingue entre 0..N y 1..N, ni entre 1..1 y 0..1. Es más compacta pero menos precisa.

Aspecto	Forma 1: dos lados	Forma 2: en el rombo
Precisión	Muy alta	Media (general)
Muestra mínimos	Sí (0..N, 1..1, etc.)	No
Claridad visual	Detallada	Más simple visualmente
Ambigüedad	Ninguna	Puede generar confusión
Uso común	Formal, académico	Bocetos, esquemas rápidos

2.3 Traducción del modelo ER al modelo relacional (SQL)

Elemento ER	Traducción SQL
Entidad	Tabla con CREATE TABLE
Atributo	Columna
Relación 1:N	Clave foránea en el lado N
Relación N:M	Nueva tabla intermedia con 2 claves foráneas
Relación con atributos propios	Tabla adicional con claves foráneas y columnas extra
Atributo clave primaria	PRIMARY KEY
Atributo clave foránea	FOREIGN KEY
Atributo obligatorio	NOT NULL
Atributo único	UNIQUE
Atributo con condición	CHECK (condición)

2.4 Sintaxis general de CREATE TABLE

```
CREATE TABLE IF NOT EXISTS nombre_tabla (
    columna1 tipo [restricciones],
    columna2 tipo [restricciones],
    ...
    CONSTRAINT nombre_restriccion CHECK (...),
    PRIMARY KEY (...),
    FOREIGN KEY (...) REFERENCES otra_tabla(columna)
        ON DELETE ... ON UPDATE ...
);
```

2.5 Tipos de datos más usados en PostgreSQL

Tipo	Descripción
INTEGER	Números enteros
TEXT	Texto de longitud variable
BOOLEAN	TRUE o FALSE
DATE	Fecha (YYYY-MM-DD)
TIMESTAMP	Fecha y hora combinadas
NUMERIC	Números decimales de precisión fija
SERIAL	Entero autoincremental

2.6 El tipo SERIAL

```
INTEGER NOT NULL DEFAULT nextval('nombre_secuencia')
```

Al usar SERIAL, PostgreSQL crea una secuencia automática asociada.

2.7 Claves primarias

```
id_empleado SERIAL PRIMARY KEY
```

Garantiza unicidad y no permite valores nulos. No necesita NOT NULL ni UNIQUE porque ya están implícitos.

```
PRIMARY KEY (id_alumno, id_curso)
```

Se utiliza cuando la combinación de varios campos debe ser única. Muy común en relaciones N:M.

2.8 Claves foráneas

```
id_departamento INTEGER REFERENCES departamentos(id_departamento)
```

```
FOREIGN KEY (id_departamento)
    REFERENCES departamentos(id_departamento)
    ON DELETE CASCADE
    ON UPDATE RESTRICT
```

Acción	Efecto
CASCADE	Borra/modifica también las filas relacionadas
SET NULL	Asigna NULL al campo en la tabla dependiente
SET DEFAULT	Asigna el valor por defecto
RESTRICT	Impide borrar/modificar si hay filas relacionadas
NO ACTION	Igual que RESTRICT pero verificado al final de la transacción

---

2.9 Restricciones adicionales

Restricción	¿Dónde se aplica?	Significado
NOT NULL	Columna	El valor no puede ser nulo
UNIQUE	Columna o grupo	No puede haber valores repetidos
CHECK	Columna o tabla	Condición que debe cumplirse (CHECK (edad > 0))

2.10 Identificadores con comillas dobles

```
CREATE TABLE "Empleados Activos" (
    "ID Usuario" SERIAL,
    "Nombre Completo" TEXT
);
```

```
SELECT "Nombre Completo" FROM "Empleados Activos";
```

Una vez creado así, debes usar comillas dobles siempre para referenciar ese campo o tabla.

2.11 Ejemplo final con todos los elementos aplicados

```
CREATE TABLE IF NOT EXISTS empleados (
    id_empleado SERIAL PRIMARY KEY,
    nombre TEXT NOT NULL,
    dni TEXT UNIQUE,
    edad INTEGER CHECK (edad > 17),
    id_departamento INTEGER REFERENCES departamentos(id_departamento)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    fecha_ingreso DATE DEFAULT CURRENT_DATE
);
```

- id_empleado es autoincremental y clave primaria.
- dni debe ser único.
- edad debe ser mayor de 17.
- id_departamento es una clave foránea con ON DELETE SET NULL.
- fecha_ingreso tiene un valor por defecto (la fecha actual).

3. 2 – Consultas en PostgreSQL

3.1 Consultas básicas: SELECT

Devuelve todas las filas y columnas:

```
SELECT * FROM empleados;
```

Puedes seleccionar columnas concretas:

```
SELECT nombre, edad FROM empleados;
```

3.2 Alias de columnas

Los alias permiten renombrar columnas en la salida:

```
SELECT
    nombre AS "Nombre completo",
    salario * 1.1 AS "Salario con subida"
FROM empleados;
```

Se puede usar AS, aunque es opcional. Si el alias tiene espacios o mayúsculas, hay que usar comillas dobles.

3.3 Alias de tablas

Útiles en consultas con múltiples tablas o JOINs:

```sql
SELECT
    e.nombre,
    d.nombre_departamento
FROM
    empleados AS e
JOIN departamentos AS d
    ON e.id_departamento = d.id_departamento;
```

3.4 Cláusula WHERE

Filtra los resultados según condiciones.

```sql
SELECT nombre FROM empleados WHERE edad > 30;
```

3.5 Operadores habituales

Operador	Uso
=	Igual a
<> o !=	Distinto de
<, >, <=, >=	Comparaciones numéricas
BETWEEN	Rango de valores
IN	Coincidencia en una lista
LIKE	Patrón textual (%, _)
IS NULL	Es nulo
IS NOT NULL	No es nulo

3.6 Ordenación y límites: ORDER BY y LIMIT

Ordena el resultado:

```
SELECT * FROM empleados ORDER BY edad;
```

Por defecto es ascendente (ASC).
Descendente:

```
ORDER BY edad DESC
```

También puedes ordenar por el número de columna:

```
SELECT
    nombre,
    salario
FROM
    empleados
ORDER BY
    2 DESC;
```

Restringe el número de filas devueltas:

```
SELECT
    *
FROM
    empleados
ORDER BY
    salario DESC
LIMIT 3;
```

Empleado con salario más alto:

```
SELECT
    nombre,
    salario
FROM
    empleados
ORDER BY
    salario DESC
LIMIT 1;
```

Empleado más joven:

```
SELECT
    nombre,
    edad
FROM
    empleados
ORDER BY
    edad ASC
LIMIT 1;
```

3.7 Tipos de JOIN

3.7.1 Tablas de ejemplo

id_producto	nombre_producto
1	Camiseta
2	Pantalón
3	Zapatos

id_color	nombre_color
1	Rojo
2	Azul

3.7.2 INNER JOIN

Solo coincidencias.

```
SELECT
    *
FROM
    productos
    INNER JOIN colores
        ON productos.id_producto = colores.id_color;
```

3.7.3 LEFT JOIN

Muestra todos los productos, aunque no tengan color asociado.

```
SELECT
    *
FROM
    productos
    LEFT JOIN colores
        ON productos.id_producto = colores.id_color;
```

Filtrar productos sin color asignado:

```
SELECT
    *
FROM
    productos
    LEFT JOIN colores
        ON productos.id_producto = colores.id_color
WHERE
    colores.id_color IS NULL;
```

3.7.4 RIGHT JOIN

Muestra todos los colores, aunque no tengan producto asociado.

```
SELECT
    *
FROM
    productos
    RIGHT JOIN colores
        ON productos.id_producto = colores.id_color;
```

3.7.5 FULL JOIN

Devuelve todas las filas de ambas tablas, coincidan o no.

```
SELECT
    *
FROM
    productos
    FULL JOIN colores
        ON productos.id_producto = colores.id_color;
```

3.7.6 CROSS JOIN

Producto cartesiano: todas las combinaciones posibles (3 × 2 = 6 filas).

```
SELECT
    *
FROM
    productos
    CROSS JOIN colores;
```

JOIN	¿Qué hace?
FULL JOIN	Une filas coincidentes y muestra también las no relacionadas
CROSS JOIN	Todas las combinaciones posibles (producto cartesiano)

3.8 Funciones de agregación

id	nombre	edad	salario	departamento
1	Ana	25	1200	Ventas
2	Luis	45	1800	Marketing
3	Marta	30	1500	Ventas
4	Pedro	50	2200	Marketing
5	Clara	29	1600	Ventas

```
SELECT COUNT(*) FROM empleados;              -- 5
SELECT SUM(salario) FROM empleados;          -- 8300
SELECT AVG(salario) FROM empleados;          -- 1660
SELECT MIN(edad), MAX(edad) FROM empleados; -- 25 y 50
```

3.9 Agrupación: GROUP BY

```
SELECT
    departamento,
    COUNT(*)
FROM
    empleados
GROUP BY
    departamento;
```

Regla: todos los campos que están en GROUP BY deben aparecer en el SELECT, salvo funciones de agregación.

```
SELECT
    departamento,
    AVG(salario)
FROM
    empleados
GROUP BY
    departamento;
```

departamento	AVG(salario)
Ventas	1433.33
Marketing	2000.00

3.10 HAVING vs WHERE

Cláusula	Filtra...	Cuándo se aplica
WHERE	Filas individuales	Antes del GROUP BY
HAVING	Grupos agregados	Después del GROUP BY

```
SELECT
    nombre,
    salario
FROM
    empleados
WHERE
    salario > 1500;
```

Filtra empleados individuales cuyo salario es mayor a 1500.

```
SELECT
    departamento,
    COUNT(*) AS empleados
FROM
    empleados
GROUP BY
    departamento
HAVING COUNT(*) > 2;
```

Filtra grupos (departamentos) que tienen más de 2 empleados.

3.11 Subconsultas

3.11.1 En WHERE (subconsulta NO correlacionada)

```
SELECT
    nombre
FROM
    empleados
WHERE
    salario > (
        SELECT AVG(salario) FROM empleados
);
```

Compara con el promedio general.

3.11.2 En SELECT

```
SELECT
    nombre,
    (
        SELECT
            AVG(edad)
        FROM
            empleados
    ) AS edad_media
FROM
    empleados;
```

Agrega la media a cada fila.

3.11.3 En FROM

```
SELECT
    departamento,
    total
FROM
  (
      SELECT
          departamento,
          COUNT(*) AS total
      FROM
          empleados
      GROUP BY
          departamento
  ) AS resumen
WHERE
    total > 2;
```

3.11.4 Subconsulta correlacionada vs no correlacionada

No correlacionada:

```
SELECT
    nombre
FROM
    empleados
WHERE
    salario > (SELECT AVG(salario) FROM empleados);
```

No depende de la fila externa.

Correlacionada:

```
SELECT
    e.nombre
FROM
    empleados e
WHERE
    salario > (
        SELECT
            AVG(salario)
        FROM
            empleados
        WHERE
            departamento = e.departamento
);
```

Compara con el promedio del mismo departamento.

4. 3 – Inserción, uso y eliminación de datos

4.1 INSERT: Añadir datos a una tabla

4.1.1 Forma básica con VALUES

```
INSERT INTO empleados (nombre, edad, salario)
VALUES ('Ana', 30, 1500);
```

Se insertan los valores en el orden de los campos indicados. Puedes omitir los campos si insertas en todos y en orden:

```
INSERT INTO empleados
VALUES (1, 'Luis', 40, 1800);
```

4.1.2 Insertar múltiples filas con VALUES

También es posible insertar varias filas a la vez:

```
INSERT INTO empleados (nombre, edad, salario)
VALUES
  ('Carlos', 25, 1400),
  ('Lucía', 28, 1550),
  ('Mario', 35, 1750);
```

Esto es más eficiente que hacer varios INSERT individuales. Todas las filas deben tener el mismo número de valores.

4.1.3 INSERT con SELECT

Permite copiar datos desde otra tabla o subconsulta:

```
INSERT INTO
   empleados_archivados (nombre, edad, salario)
SELECT
   nombre, edad, salario
FROM
   empleados
WHERE
   edad > 60;
```

Los campos del SELECT deben coincidir en número y tipo con los del INSERT.

4.2 UPDATE: Modificar registros existentes

```
UPDATE
   empleados
SET
   salario = salario * 1.05
WHERE
   departamento = 'Ventas';
```

Aplica una modificación filtrada con WHERE. Si hay triggers definidos para UPDATE, se ejecutan automáticamente.

4.3 DELETE: Eliminar registros

```
DELETE FROM empleados
WHERE edad > 65;
```

Elimina filas que cumplan la condición WHERE. También puede disparar triggers de tipo AFTER DELETE o BEFORE DELETE.

4.4 Uso de WHERE en UPDATE y DELETE

Tanto UPDATE como DELETE pueden filtrar filas usando WHERE.

4.4.1 Ejemplo en UPDATE (modificación)

```
UPDATE
   productos
SET
   stock = stock - 1
WHERE
   id_producto = 5;
```

4.4.2 Ejemplo en DELETE (borrado)

```
DELETE FROM usuarios
WHERE fecha_registro < '2023-01-01';
```

4.5 TRUNCATE: Vaciar una tabla completamente

```
TRUNCATE TABLE empleados;
```

4.6 Diferencias entre DELETE y TRUNCATE

Característica	DELETE	TRUNCATE
Usa WHERE	✅ Sí	❌ No
Dispara triggers	✅ Sí	⚠️ No (en muchos casos)
Puede deshacerse	✅ Sí (si hay transacción activa)	✅ Sí (si hay transacción activa)
Velocidad	Más lento (registro por registro)	Muy rápido
Recuperación	Posible con transacciones	Posible si se usa en transacción

5. 4 – Índices y análisis de consultas

5.1 Concepto de índice

Un índice es una estructura auxiliar que acelera la búsqueda y el acceso a los registros de una tabla. Su función es reducir el número de filas que el motor de la base de datos necesita leer para localizar los datos que cumplen una condición. Sin índice, PostgreSQL tiene que realizar un sequential scan (escaneo secuencial), revisando una por una todas las filas de la tabla. Con un índice, puede saltar directamente a la ubicación donde se encuentran los datos relevantes, de forma análoga a cómo usamos el índice de un libro para encontrar rápidamente una palabra o tema.

5.2 Estructura interna: el índice B-tree

El B-tree (Balanced Tree) es el tipo de índice más común en PostgreSQL (y en la mayoría de los SGBD).

5.2.1 Cómo funciona un B-tree

Imagina un árbol ordenado: Los nodos intermedios contienen claves que dividen el espacio de búsqueda. Los nodos hoja contienen punteros a las filas reales en la tabla. El árbol se mantiene equilibrado: todas las hojas están a la misma profundidad. Esto garantiza que el tiempo de búsqueda, inserción y borrado sea logarítmico: O(log n).

```
-
            [K=50]
           /     \
    [10,20,30] [60,70,80]
-
```

Cuando buscamos el valor 70, el árbol compara: 70 > 50 → va a la rama derecha. Busca en [60,70,80] → encuentra la posición exacta → devuelve la fila correspondiente. El acceso es mucho más rápido que recorrer todos los registros secuencialmente.

5.3 Índices implícitos

PostgreSQL crea automáticamente algunos índices sin que el usuario los pida:

Clave primaria (PRIMARY KEY):

```
CREATE TABLE usuarios (
    id SERIAL PRIMARY KEY,
    nombre TEXT
);
```

Crea automáticamente un índice B-tree sobre id.

Restricciones UNIQUE:

```
CREATE TABLE productos (
    codigo TEXT UNIQUE
);
```

Crea un índice implícito que garantiza la unicidad de codigo. Estos índices automáticos son esenciales para mantener la integridad referencial y acelerar las búsquedas por claves.

5.4 Creación de índices manuales

Índice básico:

```
CREATE INDEX idx_usuarios_nombre ON usuarios (nombre);
```

Índice en orden descendente:

```
CREATE INDEX idx_pedidos_fecha_desc ON pedidos (fecha DESC);
```

Índice múltiple (o compuesto):

```
CREATE INDEX idx_clientes_apellido_ciudad ON clientes (apellido,
ciudad);
```

Importante: Los índices múltiples se aprovechan por su primera columna y las subsecuentes, en ese orden.

Por ejemplo:

```
WHERE apellido = 'García'
    -- ✅ usa el índice.

WHERE apellido = 'García' AND ciudad = 'Sevilla'
    -- ✅ usa el índice.

WHERE ciudad = 'Sevilla'
    -- ❌ no usa el índice
    -- (la primera columna apellido no se filtra).
```

Este comportamiento se conoce como regla del prefijo del índice.

Índice parcial:

```
CREATE INDEX idx_activos ON empleados (dni) WHERE activo = true;
```

Ideal para grandes tablas con muchas filas inactivas.

Índice único:

```
CREATE UNIQUE INDEX idx_email_unico ON usuarios (email);
```

Garantiza que no se repitan valores en esa columna.

5.5 Buenas prácticas en el uso de índices

- Usar índices en columnas que aparecen con frecuencia en cláusulas WHERE, condiciones de JOIN, ORDER BY o GROUP BY.
- No indexar todo: cada índice ocupa espacio y ralentiza INSERT, UPDATE y DELETE.
- Evitar índices en columnas con baja selectividad (por ejemplo, un campo BOOLEAN con casi todos los valores iguales).
- Analizar el tamaño de los índices.

```
SELECT relname, pg_size_pretty(pg_total_relation_size(indexrelid))
FROM pg_stat_user_indexes
WHERE schemaname = 'public';
```

Actualizar estadísticas regularmente:

```
VACUUM ANALYZE;
```

Esto ayuda al optimizador a tomar decisiones más acertadas.

5.6 Análisis de rendimiento con EXPLAIN y EXPLAIN ANALYZE

5.6.1 EXPLAIN

Muestra el plan estimado de ejecución:

```
EXPLAIN SELECT * FROM usuarios WHERE nombre = 'Juan';
```

Salida:

```
Index Scan using idx_usuarios_nombre on usuarios  (cost=0.15..8.17
rows=1 width=48)
```

Componente	Significado
Index Scan	Está usando el índice.
cost	Estimación del optimizador.

5.6.2 EXPLAIN ANALYZE

Ejecuta realmente la consulta y muestra los tiempos reales:

```
EXPLAIN ANALYZE SELECT * FROM usuarios WHERE nombre = 'Juan';
```

Ejemplo de salida:

```
Index Scan using idx_usuarios_nombre on usuarios  (cost=0.15..8.17
rows=1 width=48)
(actual time=0.030..0.035 rows=1 loops=1)
```

Componente	Significado
cost	Coste estimado (cuanto menor, mejor).
actual time	Tiempo real de ejecución.
rows	Número de filas encontradas.
loops	Veces que se repitió el plan (en subconsultas o bucles).

5.7 Comparativa de rendimiento: sin índice y con índice

Supón que tenemos 1 millón de filas en usuarios:

```
SELECT * FROM usuarios WHERE nombre = 'Juan';
```

Sin índice:

```
Seq Scan on usuarios (cost=0.00..25000.00 rows=1 width=48)
(actual time=50.000..50.001 rows=1 loops=1)
```

Con índice:

```
Index Scan using idx_usuarios_nombre on usuarios (cost=0.15..8.17
rows=1 width=48)
(actual time=0.035..0.036 rows=1 loops=1)
```

El acceso pasa de 50 ms a 0.03 ms, un ahorro enorme.

5.8 Mantenimiento de índices

Comprobar el uso de índices:

```
SELECT
   relname AS tabla,
   indexrelname AS indice,
   idx_scan AS veces_usado,
   idx_tup_read AS tuplas_leidas,
   idx_tup_fetch AS tuplas_devueltas
FROM
   pg_stat_user_indexes
WHERE
   schemaname = 'public';
```

Esta vista muestra cuántas veces se ha utilizado cada índice desde el último reinicio de estadísticas.

Reindexar un índice específico:

```
REINDEX INDEX idx_usuarios_nombre;
```

Reindexar una tabla completa:

```
REINDEX TABLE usuarios;
```

Ver todos los índices existentes:

```
\di
```

5.9 Ejemplo completo

Supongamos una tabla de pedidos:

```
CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INT,
    fecha DATE,
    total DECIMAL
);
```

Creamos un índice para acelerar búsquedas por fecha:

```
CREATE INDEX idx_pedidos_fecha ON pedidos (fecha);
```

Consultamos:

```
EXPLAIN ANALYZE
SELECT * FROM pedidos
WHERE fecha BETWEEN '2025-01-01' AND '2025-03-31';
```

Si el rango abarca pocas filas: “Index Scan”.
Si abarca la mayoría: “Seq Scan” (el optimizador decide que leer toda la tabla es más rápido).

5.10 Resumen final

- Los índices mejoran la velocidad de lectura pero empeoran ligeramente la escritura.
- Los B-trees son los índices por defecto y más versátiles.
- Los índices compuestos funcionan en orden de las columnas, empezando por la primera.
- Usa EXPLAIN ANALYZE para comprobar si se usan correctamente.
- Revisa estadísticas periódicamente y evita sobreindexar.

6. 5 – Programación con funciones y procedimientos

6.1 Diferencias entre función y procedimiento

Elemento	Función	Procedimiento
Devuelve un valor	✅ Sí, con RETURN	❌ No
Llamada	En una SELECT o expresión	Con el comando CALL
Se puede usar en SQL	✅ Sí	❌ No directamente
Tiene OUT	❌ Generalmente no	✅ Sí

6.2 Ejemplo de llamada

Llamada a función:

```
SELECT mi_funcion(5);
```

Llamada a procedimiento:

```
CALL mi_procedimiento('dato');
```

6.3 Esqueleto de una función o procedimiento

Ambos comienzan con CREATE OR REPLACE y pueden tener:

- Sección DECLARE: para definir variables internas.
- Bloque BEGIN ... END: cuerpo principal.
- Sección EXCEPTION: captura de errores.

6.3.1 Función

```
CREATE OR REPLACE FUNCTION suma(a INT, b INT)
RETURNS INT AS $$
DECLARE
  resultado INT;
BEGIN
  resultado := a + b;
  RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

6.3.2 Procedimiento

```
CREATE OR REPLACE PROCEDURE imprimir_suma(a INT, b INT)
AS $$
DECLARE
  resultado INT;
BEGIN
  resultado := a + b;
  RAISE NOTICE 'La suma es: %', resultado;
END;
$$ LANGUAGE plpgsql;
```

6.4 Parámetros IN, OUT e INOUT

Tipo	Descripción
IN	Valor de entrada (por defecto)
OUT	Parámetro que devuelve un valor
INOUT	Sirve como entrada y salida

```
CREATE OR REPLACE PROCEDURE dame_doble(IN entrada INT, OUT salida
INT)
AS $$
BEGIN
  salida := entrada * 2;
END;
$$ LANGUAGE plpgsql;
```

Llamada desde consola: CALL dame_doble(5, x); -- x = 10

6.5 Nota importante sobre excepciones

Cada procedimiento o función en PostgreSQL se ejecuta automáticamente dentro de una transacción. Por eso NO es necesario iniciar manualmente una transacción con BEGIN. Sin embargo, si se lanza una excepción (RAISE EXCEPTION) y no se captura con EXCEPTION, se produce un rollback implícito de toda la función o procedimiento.

6.6 Condicionales

6.6.1 IF / ELSIF / ELSE

```
IF total > 100 THEN
  RAISE NOTICE 'Total alto';
ELSIF total = 100 THEN
  RAISE NOTICE 'Total exacto';
ELSE
  RAISE NOTICE 'Total bajo';
END IF;
```

6.6.2 CASE

```
CASE tipo_producto
  WHEN 'A' THEN
    precio := 10;
  WHEN 'B' THEN
    precio := 15;
  ELSE
    precio := 5;
END CASE;
```

6.7 Bucles

6.7.1 LOOP con EXIT

```
LOOP
  total := total + 1;
  EXIT WHEN total >= 5;
END LOOP;
```

6.7.2 WHILE

```
WHILE stock > 0 LOOP
  stock := stock - 1;
END LOOP;
```

6.7.3 FOR IN SELECT

```
FOR fila IN SELECT * FROM productos LOOP
  RAISE NOTICE 'Producto: %', fila.nombre;
END LOOP;
```

6.8 Variable mágica FOUND

FOUND indica si la última operación de tipo SELECT INTO, FETCH, UPDATE, DELETE, etc. encontró al menos una fila.

```
LOOP
  FETCH mi_cursor INTO fila;
  EXIT WHEN NOT FOUND;
  -- procesar fila
END LOOP;
```

También útil para salir de bucles si ya no hay más resultados.

6.9 Cursores en PostgreSQL

Un cursor permite recorrer los resultados de una consulta fila por fila.

6.9.1 ¿Cómo se declara un cursor?

```
DECLARE
  cur_empleados CURSOR FOR
    SELECT * FROM empleados ORDER BY id;
```

6.9.2 Ejemplo completo: recorrido ascendente

```
DO $$
DECLARE
  cur_empleados CURSOR FOR
    SELECT * FROM empleados ORDER BY id;
  fila RECORD;
BEGIN
  OPEN cur_empleados;
  FETCH NEXT FROM cur_empleados INTO fila;

  WHILE FOUND LOOP
    RAISE NOTICE 'Empleado: % (% años)', fila.nombre, fila.edad;
    FETCH NEXT FROM cur_empleados INTO fila;
  END LOOP;

  CLOSE cur_empleados;
END;
$$;
```

6.9.3 Ejemplo completo: recorrido descendente

```
DO $$
DECLARE
  cur_empleados CURSOR FOR
    SELECT * FROM empleados ORDER BY id;
  fila RECORD;
BEGIN
  OPEN cur_empleados;
  FETCH LAST FROM cur_empleados INTO fila;

  WHILE FOUND LOOP
    RAISE NOTICE 'Empleado: % (% años)', fila.nombre, fila.edad;
    FETCH PRIOR FROM cur_empleados INTO fila;
  END LOOP;

  CLOSE cur_empleados;
END;
$$;
```

6.9.4 Resumen: direcciones de FETCH

Dirección	Descripción
NEXT	Fila siguiente (por defecto)
PRIOR	Fila anterior
FIRST	Primera fila
LAST	Última fila

- Siempre cerrar el cursor con CLOSE al final.
- Usar ORDER BY si el orden es importante.
- Utilizar FOUND para saber si FETCH devolvió fila.

6.10 Variables tipo RECORD

Las variables RECORD permiten guardar varios campos sin declarar uno por uno.

```
DECLARE
  empleado RECORD;

SELECT
  * INTO empleado
FROM
   empleados
WHERE id = 1;

RAISE NOTICE 'Nombre: %, Salario: %', empleado.nombre,
empleado.salario;
```

Uso con FETCH:

```
FETCH cur_empleados INTO empleado;
RAISE NOTICE 'ID: %, Nombre: %', empleado.id, empleado.nombre;
```

6.11 RAISE NOTICE y RAISE EXCEPTION

```
RAISE NOTICE 'ID: %, Nombre: %', id, nombre;
```

```
IF salario < 0 THEN
  RAISE EXCEPTION 'Salario negativo: %', salario;
END IF;
```

Los placeholders (%) se sustituyen por los valores en orden.

6.12 Bloques BEGIN … EXCEPTION

```
BEGIN
  SELECT * INTO empleado FROM empleados WHERE id = 999;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE NOTICE 'No se encontró el empleado';
  WHEN TOO_MANY_ROWS THEN
    RAISE NOTICE 'Demasiados resultados';
  WHEN OTHERS THEN
    RAISE NOTICE 'Error inesperado';
END;
```

Captura localizada con bloques anidados:

```
BEGIN
  BEGIN
    SELECT * INTO emp FROM empleados WHERE activo = TRUE;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE NOTICE 'Error solo aquí';
  END;
END;
```

6.13 SELECT INTO

6.13.1 Ejemplo básico (una sola columna)

```
DECLARE
  salario NUMERIC;
BEGIN
  SELECT sueldo INTO salario
  FROM empleados
  WHERE id = 5;
END;
```

Si no se encuentra ninguna fila, la variable toma el valor NULL.
Si se encuentran varias, se toma solo la primera.

6.13.2 SELECT INTO con múltiples columnas

```
DECLARE
  nombre TEXT;
  edad INT;
BEGIN
  SELECT nombre, edad INTO nombre, edad
  FROM empleados
  WHERE id = 1;
END;
```

```
DECLARE
  emp RECORD;
BEGIN
  SELECT id, nombre, edad INTO emp
  FROM empleados
  WHERE id = 1;

  RAISE NOTICE 'Nombre: %, Edad: %', emp.nombre, emp.edad;
END;
```

6.13.3 SELECT INTO STRICT

Situación	Excepción lanzada
Ninguna fila encontrada	NO_DATA_FOUND
Más de una fila encontrada	TOO_MANY_ROWS

```
DO $$
DECLARE
  emp RECORD;
BEGIN
  BEGIN
    SELECT id, nombre, salario INTO STRICT emp
    FROM empleados
    WHERE activo = TRUE;

    RAISE NOTICE 'Empleado activo: % con salario %', emp.nombre,
emp.salario;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE NOTICE 'No hay empleados activos';
    WHEN TOO_MANY_ROWS THEN
      RAISE NOTICE 'Hay más de un empleado activo';
  END;
END;
$$;
```

6.14 Rollback implícito

Cuando se produce una excepción no capturada, PostgreSQL hace un rollback automático de todo el procedimiento o función.

```
IF saldo < 0 THEN
  RAISE EXCEPTION 'Saldo negativo: %', saldo;
END IF;
```

Si no se captura, revierte todos los cambios realizados dentro de la función.

7. 6 – Triggers

Un trigger (disparador) es un mecanismo automático que se ejecuta como respuesta a un evento sobre una tabla o vista. Puede activarse con operaciones INSERT, UPDATE, DELETE o incluso DDL (como CREATE TABLE o ALTER). En resumen:

Un trigger actúa como un "reaccionador" dentro de la base de datos: “Cuando ocurra X en la tabla Y, ejecuta automáticamente el código Z.”

7.1 ¿Para qué sirven los triggers?

- Mantener integridad lógica entre tablas (por ejemplo, borrar líneas de pedido al borrar un pedido).
- Auditar cambios: registrar en una tabla quién modificó un registro y cuándo.
- Validar o corregir datos antes de guardarlos (convertir texto, comprobar rangos, etc.).
- Sincronizar información: actualizar totales o estadísticas al cambiar datos base.

7.2 Tipos de triggers en PostgreSQL

7.2.1 Según el momento

Tipo	Se ejecuta…	Uso típico
BEFORE	Antes de la operación	Validar, modificar o impedir cambios
AFTER	Después de la operación	Registrar cambios, propagar datos
INSTEAD OF	En lugar de la operación (solo vistas)	Personalizar vistas actualizables

7.2.2 Según el alcance

Tipo	Actúa sobre...	Ejemplo
FOR EACH ROW	Cada fila afectada	Registrar log por cada registro insertado
FOR EACH STATEMENT	Una vez por sentencia	Calcular totales tras un UPDATE masivo

7.3 Estructura general de un trigger

Un trigger tiene dos componentes:

1- Una función PL/pgSQL que define la lógica.

2- Una declaración CREATE TRIGGER que la asocia a una tabla y evento.

7.3.1 Paso 1. Crear tabla principal

```
CREATE TABLE empleados (
    id SERIAL PRIMARY KEY,
    nombre TEXT,
    salario NUMERIC(10,2)
);
```

7.3.2 Paso 2. Crear tabla de auditoría

```
CREATE TABLE log_empleados (
    id SERIAL PRIMARY KEY,
    empleado_id INT,
    accion TEXT,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    usuario TEXT
);
```

7.3.3 Paso 3. Crear la función trigger

```
CREATE OR REPLACE FUNCTION registrar_cambio_empleado()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO log_empleados (empleado_id, accion, usuario)
        VALUES (NEW.id, 'INSERT', current_user);

    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO log_empleados (empleado_id, accion, usuario)
        VALUES (NEW.id, 'UPDATE', current_user);

    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO log_empleados (empleado_id, accion, usuario)
        VALUES (OLD.id, 'DELETE', current_user);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Elemento	Explicación
TG_OP	Operación (INSERT, UPDATE, DELETE)
NEW	Fila nueva (INSERT/UPDATE)
OLD	Fila anterior (UPDATE/DELETE)
current_user	Usuario que ejecutó la acción

7.3.4 Paso 4. Asociar el trigger a la tabla

```
CREATE TRIGGER tr_log_empleados
AFTER INSERT OR UPDATE OR DELETE
ON empleados
FOR EACH ROW
EXECUTE FUNCTION registrar_cambio_empleado();
```

Elemento	Significado
AFTER	Se ejecuta después de la acción.
FOR EACH ROW	Actúa por cada fila.
EXECUTE FUNCTION	Llama a la función PL/pgSQL definida.

7.3.5 Ejemplo de funcionamiento

```
INSERT INTO empleados (nombre, salario)
VALUES ('Lucía', 2500.00);
```

PostgreSQL ejecuta automáticamente:

```
INSERT INTO log_empleados (empleado_id, accion, usuario)
VALUES (1, 'INSERT', 'juanma');
```

El log se rellena sin intervención manual del programador.

7.4 Ejemplo de trigger BEFORE (validación de datos)

```
CREATE OR REPLACE FUNCTION validar_salario()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.salario < 1000 THEN
        RAISE EXCEPTION 'El salario mínimo debe ser 1000 euros';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tr_validar_salario
BEFORE INSERT OR UPDATE ON empleados
FOR EACH ROW
EXECUTE FUNCTION validar_salario();
```

Si se intenta insertar un salario menor a 1000, la operación se cancela. El trigger BEFORE actúa antes de escribir los datos.

7.5 Ejemplo con INSTEAD OF (vistas actualizables)

```
CREATE VIEW vista_empleados AS
SELECT id, nombre, salario FROM empleados WHERE salario > 2000;
```

```
CREATE OR REPLACE FUNCTION insertar_en_vista()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO empleados (nombre, salario) VALUES (NEW.nombre,
NEW.salario);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tr_insertar_vista
INSTEAD OF INSERT ON vista_empleados
FOR EACH ROW
EXECUTE FUNCTION insertar_en_vista();
```

Permite ejecutar INSERT sobre una vista que, internamente, inserta en la tabla real.

7.6 Variables útiles dentro de un trigger

Variable	Descripción
TG_OP	Tipo de operación (INSERT, UPDATE, DELETE)
TG_TABLE_NAME	Nombre de la tabla afectada
TG_WHEN	Momento del trigger (BEFORE, AFTER, INSTEAD OF)
NEW	Fila nueva (solo INSERT y UPDATE)
OLD	Fila antigua (solo UPDATE y DELETE)
TG_ARGV[]	Argumentos pasados al trigger

7.7 Buenas prácticas con triggers

- Usar solo cuando sea necesario, ya que pueden complicar la depuración.
- Documentar claramente cada trigger y su función.
- Evitar operaciones pesadas dentro del trigger.
- Usar AFTER para auditorías y BEFORE para validaciones.
- Probar y depurar con cuidado, usando RAISE NOTICE para depuración.

```
RAISE NOTICE 'Se ha insertado el empleado %', NEW.nombre;
```

7.8 Cómo ver y eliminar triggers

Listar triggers de una tabla:

```
\d nombre_tabla
```

```
SELECT tgname, tgtype::regtype, tgfoid::regproc
FROM pg_trigger
WHERE tgrelid = 'empleados'::regclass;
```

Eliminar un trigger:

```
DROP TRIGGER tr_log_empleados ON empleados;
```

Eliminar la función asociada:

```
DROP FUNCTION registrar_cambio_empleado();
```

8. 7 – Casting de tipos

En PostgreSQL puedes convertir un dato de un tipo a otro usando el operador de conversión (::) o la función CAST().

8.1 Sintaxis general

```
valor::tipo
```

```
CAST(valor AS tipo)
```

8.2 Ejemplos de conversiones comunes

8.2.1 Texto a número

```
SELECT '123'::INTEGER;
```

8.2.2 Número a texto

```
SELECT 99::TEXT;
```

8.2.3 Texto a fecha

```
SELECT '2025-09-27'::DATE;
```

8.2.4 TIMESTAMP a DATE (descarta la hora)

```
SELECT CURRENT_TIMESTAMP::DATE;
```

8.2.5 Alternativa con CAST

```
SELECT CAST('123' AS INTEGER);
```

9. 8 – Funciones comunes en PL/pgSQL

9.1 Funciones de fecha

```
SELECT CURRENT_DATE;                -- Solo la fecha
SELECT CURRENT_TIME;                -- Solo la hora
SELECT NOW();                       -- Fecha y hora con zona horaria
SELECT DATE_TRUNC('month', NOW()); -- Recorta al mes
SELECT EXTRACT(DAY FROM NOW());     -- Día actual
```

9.2 Funciones de texto

```
SELECT UPPER('hola');               -- HOLA
SELECT LOWER('TEXTO');              -- texto
SELECT LENGTH('hola mundo');        -- 11
SELECT TRIM('  hola  ');            -- 'hola'
SELECT CONCAT('Hola', ' mundo');    -- 'Hola mundo'
```

9.3 Funciones matemáticas

```
SELECT ABS(-8);             -- 8
SELECT GREATEST(10, 3, 5);  -- 10
SELECT LEAST(10, 3, 5);     -- 3
```

9.4 Funciones condicionales

```
SELECT COALESCE(NULL, 'valor'); -- devuelve 'valor'
```

10. 9 – Formatos de fecha en PostgreSQL

10.1 Formato natural (locale española)

Cuando el sistema usa la configuración regional española, el formato habitual de fecha es: '27/09/2025 15:30:00' (día/mes/año + hora). Este formato es legible, pero no estándar y puede causar errores al intercambiar datos entre sistemas.

Es recomendable usar el formato ISO 8601 (internacional) para garantizar interoperabilidad y evitar ambigüedades. También se acepta la versión simplificada YYYY-MM-DD.

Tipo	Ejemplo
Solo fecha	2023-09-27
Fecha y hora	2023-09-27 15:30:00
Fecha, hora y zona	2023-09-27 15:30:00+02

10.2 Formato ISO 8601 (internacional)

ISO 8601 es un estándar para representar fechas y horas con formato claro y compatible. Estructura general: AAAA-MM-DDTHH:MM:SS.sss+ZZ:ZZ

Elemento	Ejemplo / Significado
Solo fecha	2023-09-27
Fecha y hora	2023-09-27T15:30:00
Fecha y hora con milisegundos	2023-09-27T15:30:00.000
Fecha y hora con zona horaria	2023-09-27T15:30:00+02:00
Separador entre fecha y hora	T
Zona horaria	+02:00 / +02

PostgreSQL acepta este formato directamente en consultas e inserciones.

```
SELECT '2025-09-27T18:45:00.000+02:00'::TIMESTAMPTZ;
```

10.3 Fechas en formato ISO 8601 sin separadores

PostgreSQL admite también la variante compacta del formato ISO, sin guiones ni dos puntos:

```
YYYYMMDDTHHMMSS
```

```
SELECT '20250927T183000'::TIMESTAMP;
-- Resultado: 2025-09-27 18:30:00
```

```
SELECT '20250927T183000+0200'::TIMESTAMPTZ;
-- Resultado: 2025-09-27 18:30:00+02
```

10.4 Comparación de formatos

Formato	Cumple ISO 8601	Recomendado
2025-09-27T18:30:00	✅ ISO extendido	✅ Sí
20250927T183000	✅ ISO básico	⚠️ Aceptable, menos legible
2025-09-27 18:30:00	❌ No ISO estricto	⚠️ Legible, pero informal

10.4.1 Cuándo usar formato compacto

- Cuando se cargan datos masivos.
- Cuando se conoce el formato exacto de destino.

• Cuando se desea optimizar espacio o parsing.

Usa el formato extendido (YYYY-MM-DDTHH:MM:SS) en informes, APIs o documentación formal.

10.5 UTC: Tiempo Universal Coordinado

UTC es el estándar global de tiempo sin ajustes de horario de verano, sustituyendo al antiguo GMT. Un TIMESTAMP WITH TIME ZONE se guarda internamente en UTC. PostgreSQL convierte automáticamente entre UTC y la zona local al mostrarlo.

```
-- Mostrado en hora local:
SELECT NOW();  -- 2025-09-27 17:15:00+02

-- Mostrado como UTC:
SELECT NOW() AT TIME ZONE 'UTC';  -- 2025-09-27 15:15:00+00
```

De esta forma, las comparaciones y almacenamiento de fechas son consistentes globalmente. Siempre que trabajes con múltiples zonas horarias, usa TIMESTAMPTZ.

10.6 Tipo de dato TIMESTAMPTZ

Tipo de dato	Significado
TIMESTAMP	Fecha y hora sin zona horaria
TIMESTAMPTZ	Fecha y hora con zona horaria

10.6.1 Cómo funciona realmente TIMESTAMPTZ

PostgreSQL almacena internamente el valor en UTC y lo muestra ajustado a la zona horaria del cliente.

```
CREATE TABLE eventos (
  id SERIAL PRIMARY KEY,
  nombre TEXT,
  fecha TIMESTAMPTZ
);

INSERT INTO eventos (nombre, fecha)
VALUES ('Concierto', '2025-09-27 20:00:00');
```

Internamente se guarda como: 2025-09-27 18:00:00+00 (UTC)
Pero al consultar desde Europe/Madrid (UTC+2) se muestra: 2025-09-27 20:00:00+02

10.6.2 Ventajas de TIMESTAMPTZ

- Ideal para aplicaciones internacionales.
- Permite almacenar un valor estandarizado y mostrarlo localmente.
- Facilita comparaciones precisas entre zonas horarias.

10.6.3 Comparación práctica

```
-- Timestamp sin zona horaria
SELECT TIMESTAMP '2025-09-27 20:00:00';

-- Timestamp con zona horaria
SELECT TIMESTAMPTZ '2025-09-27 20:00:00+02';
```

El primero es una hora absoluta sin contexto.
El segundo permite conversión automática entre zonas.

10.7 Conclusión

Siempre que trabajes con usuarios en distintas regiones o necesites conservar la hora real de un evento global, usa TIMESTAMPTZ.

11. 10 – Instalación y Configuración de PostgreSQL + pgAdmin + JDBC

11.1 Instalación de PostgreSQL y pgAdmin en Windows

Descarga el instalador oficial desde:

```
https://www.postgresql.org/download/windows/
```

1. Ejecuta el instalador de EnterpriseDB (EDB) y selecciona los componentes:
   - PostgreSQL Server
   - pgAdmin 4
   - Command Line Tools
2. Define una contraseña para el usuario postgres.
3. Deja el puerto por defecto 5432.
4. Anota la carpeta de datos (por ejemplo C:\Program Files\PostgreSQL\16\data).

Al finalizar, PostgreSQL se inicia automáticamente como servicio de Windows. Puedes comprobarlo en el Administrador de tareas (pestaña Servicios) o con PowerShell:

```
net start postgresql-x64-16
```

11.2 Activar acceso por contraseña en Windows

Por defecto, el usuario postgres puede acceder localmente sin contraseña. Para requerir contraseña (necesario para pgAdmin, DBeaver o JDBC):

1.- Abre el archivo:

```
C:\Program Files\PostgreSQL\16\data\pg_hba.conf
```

2.- Busca la línea:

```
local   all             postgres                                trust
```

3.- Cámbiala por:

```
local   all             postgres                                md5
```

Opcional: permitir acceso local TCP/IP:

```
host    all             all             127.0.0.1/32            md5
```

Guarda los cambios y reinicia el servicio:

```
net stop postgresql-x64-16
net start postgresql-x64-16
```

Verifica el acceso:

```
psql -U postgres -h localhost
```

Debería pedir la contraseña configurada durante la instalación.

11.3 Instalación de PostgreSQL y pgAdmin en Linux (Ubuntu, Mint, Pop!_OS, etc.)

11.3.1 Opción 1 – Repositorios estándar (versión estable de Ubuntu)

```
sudo apt update
sudo apt install postgresql postgresql-contrib pgadmin4 -y
```

11.3.2 Opción 2 (recomendada) – Repositorios oficiales de PostgreSQL.org

Permite instalar versiones más recientes (por ejemplo PostgreSQL 16 o 17).

```
sudo apt install wget ca-certificates -y
wget -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | \
   sudo apt-key add -
. /etc/os-release
echo "deb http://apt.postgresql.org/pub/repos/apt $
{VERSION_CODENAME}-pgdg main" | \
   sudo tee /etc/apt/sources.list.d/pgdg.list
sudo apt update
sudo apt install postgresql postgresql-contrib -y
```

Verifica que el servicio esté activo:

```
sudo systemctl status postgresql
```

(Opcional) Instalar una versión específica:

```
sudo apt install postgresql-16 postgresql-contrib -y
```

11.4 Cambiar la contraseña del usuario postgres

```
sudo -u postgres psql
```

```
ALTER USER postgres PASSWORD 'nueva_contraseña';
```

Ejemplo:

```
ALTER USER postgres PASSWORD 'admin123';
```

```
\q
```

11.5 Permitir acceso por contraseña (modo md5)

Edita el archivo de configuración de autenticación:

```
sudo nano /etc/postgresql/16/main/pg_hba.conf
```

Reemplaza:

```
local   all             postgres                                peer
```

Por:

```
local   all             postgres                                md5
```

Y añade:

```
host    all             all             127.0.0.1/32            md5
```

```
sudo systemctl restart postgresql
```

Método	Significado
peer	Usa el usuario del sistema operativo (sin contraseña)
md5	Requiere contraseña cifrada (recomendado)
trust	Permite acceso sin contraseña (solo para pruebas locales)

11.6 Crear usuarios y bases de datos

```
sudo -u postgres psql
```

```
CREATE USER juanma WITH PASSWORD '12345';

CREATE DATABASE cursos;

GRANT ALL PRIVILEGES ON DATABASE cursos TO juanma;

\c cursos

GRANT ALL PRIVILEGES ON SCHEMA public TO juanma;

ALTER ROLE juanma CREATEDB;
```

11.7 Conectarse con psql y pgAdmin

11.7.1 Desde la terminal

```
psql -U juanma -d cursos -h localhost -W
```

11.7.2 Desde pgAdmin

1. Abre pgAdmin y crea una Master Password local.
2. En Servers → Create → Server:
   - Name: Localhost
   - Host: 127.0.0.1
   - Port: 5432
   - Username: postgres
   - Password: la configurada
3. Guarda la conexión y accede a la base de datos.

11.8 Instalar el driver JDBC

11.8.1 Instalación manual

Descarga desde:

```
https://jdbc.postgresql.org/download.html
```

Archivo típico:

```
postgresql-42.7.2.jar
```

Estructura de proyecto:

```
ProyectoJava/
├─ src/
└─ lib/postgresql-42.7.2.jar
```

Compila y ejecuta con:

```
javac -cp ".:lib/postgresql-42.7.2.jar" Main.java
java  -cp ".:lib/postgresql-42.7.2.jar" Main
```

(En Windows usa ; en lugar de :)

11.8.2 Instalación con Maven

```
<dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.2</version>
    </dependency>
</dependencies>
```

```
mvn dependency:tree
```

Deberías ver:

```
org.postgresql:postgresql:jar:42.7.2:compile
```