Resumen Bases de Datos

PostgreSQL

IFCD0112

Programación orientada a objetos y base de datos relacionales

Juan Manuel Piñero Sánchez

Índice

1 – Modelado y definición de tablas en PostgreSQL

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:

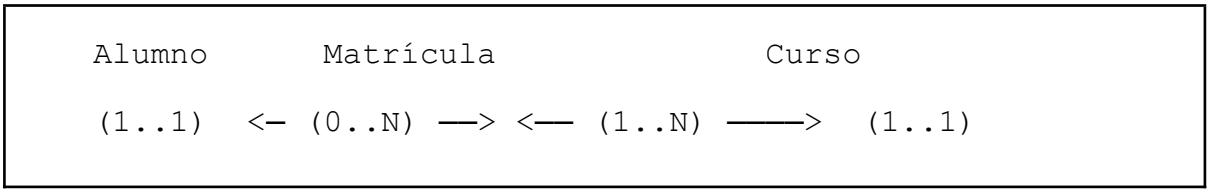
- Una entidad: se convertirá en una columna en la tabla correspondiente.
- Una relación: si la relación tiene atributos, se transforma en una tabla propia.

Representación de cardinalidades

Las cardinalidades indican cuántas veces una entidad puede intervenir en una relación.

Hay dos formas de representarlas gráficamente:

Forma 1 – Dos grupos de números (más precisa)

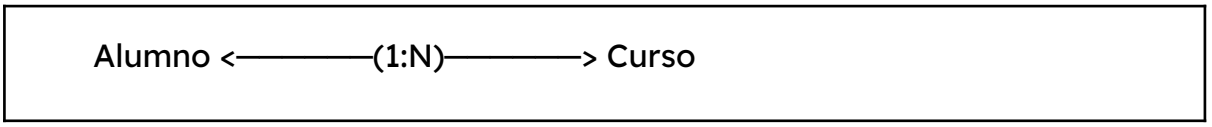


Esto significa:

- Cada alumno puede tener 0 o más matrículas.
- Cada matrícula pertenece a un único alumno.
- Cada matrícula es sobre un único curso.
- Un curso puede tener muchas matrículas.

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

Forma 2 – Un solo grupo de números en el rombo



En esta representación:

- 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.

Comparativa entre ambas formas

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

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)

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 ...
);
```

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

El tipo SERIAL

SERIAL es una forma corta de definir un campo autoincremental.

Internamente equivale a:

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

Al usar SERIAL, PostgreSQL crea una secuencia automática asociada.

Claves primarias

```
id_empleado SERIAL PRIMARY KEY
```

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

Clave primaria compuesta

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

Se utiliza cuando la combinación de varios campos debe ser única.

Muy común en relaciones N:M.

Claves foráneas

Las claves foráneas conectan una tabla con otra.

Definida dentro de la columna

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

Definida como restricción de tabla

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

ON DELETE / ON UPDATE

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

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))

Identificadores con comillas dobles

Si usas mayúsculas, espacios o caracteres especiales en nombres, debemos encerrarlos entre 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.

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
);
```

En este ejemplo:

- 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).

2 – Consultas en PostgreSQL

Consultas básicas: SELECT

- Devuelve todas las filas y columnas:

```
SELECT * FROM empleados;
```

- Puedes seleccionar columnas concretas:

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

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.

Alias de tablas

Útiles en consultas con múltiples tablas o JOINs:

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

Cláusula WHERE

Filtra los resultados según condiciones.

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

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

Ordenación y límites: ORDER BY y LIMIT

ORDER BY

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;
```

LIMIT

Restringe el número de filas devueltas:

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

ORDER BY … LIMIT 1

- 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;
```

Tipos de JOIN

Tablas de ejemplo

productos

id_producto	nombre_producto
1	Camiseta
2	Pantalón
3	Zapatos

colores

id_color	nombre_color
1	Rojo
2	Azul

INNER JOIN

- Solo coincidencias.

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

LEFT JOIN

- Muestra todos los productos, aunque no tengan color asociado.

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

LEFT JOIN + WHERE IS NULL

- Filtrar productos sin color asignado.

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

RIGHT JOIN

- Muestra todos los colores, aunque no tengan producto asociado.

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

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;
```

CROSS JOIN

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

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

FULL JOIN vs CROSS JOIN

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

Funciones de agregación

empleados

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

Ejemplos:

```
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
```

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.

Ejemplo con GROUP BY

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

departamento	AVG(salario)
Ventas	1433.33
Marketing	2000.00

HAVING vs WHERE

Cláusula	Filtra...	Cuándo se aplica
WHERE	Filas individuales	Antes del GROUP BY

HAVING	Grupos agregados	Después del GROUP BY

Ejemplo con WHERE

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

Filtra empleados individuales cuyo salario es mayor a 1500.

Ejemplo con HAVING

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

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

Subconsultas

En WHERE (subconsulta NO correlacionada)

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

Compara con el promedio general.

En SELECT

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

Agrega la media a cada fila.

En FROM

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

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.

3 – Inserción, uso y eliminación de datos

INSERT: Añadir datos a una tabla

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);
```

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.

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.

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.

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.

Uso de WHERE en UPDATE y DELETE

Tanto UPDATE como DELETE pueden filtrar filas usando WHERE.

Ejemplo en UPDATE (modificación)

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

Ejemplo en DELETE (borrado)

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

TRUNCATE: Vaciar una tabla completamente

```
TRUNCATE TABLE empleados;
```

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

4.- Índices y análisis de consultas

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.

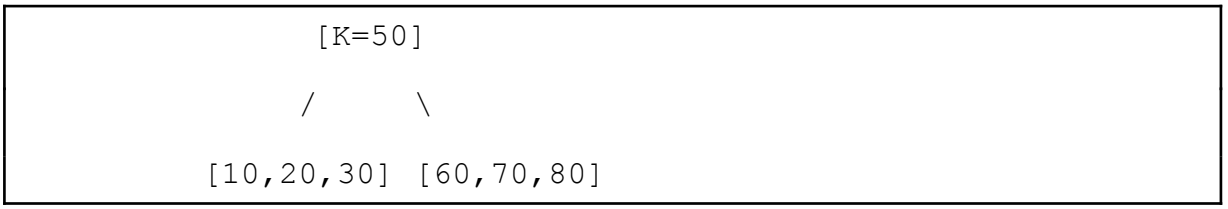
Estructura interna: el índice B-tree

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

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**, es decir, todas las hojas están a la misma profundidad.
  Esto garantiza que **el tiempo de búsqueda, inserción y borrado sea logarítmico**: O(log n).



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.

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.

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.

Buenas prácticas en el uso de índices

Usar índices en columnas que aparecen con frecuencia en:

- cláusulas WHERE
- condiciones de JOIN
- ORDER BY
- GROUP BY

No indexar todo.

Cada índice adicional ocupa espacio y ralentizan las operaciones de escritura (INSERT, UPDATE, DELETE).

Evitar índices en columnas con baja selectividad.

Una columna activo BOOLEAN con 95% de valores true no se beneficiará de un índice.

Analizar el tamaño del índice.

```
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.

Análisis de rendimiento con EXPLAIN y EXPLAIN ANALYZE

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.

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)

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.**

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
```

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 de filas:
    - “Seq Scan”
    - (el optimizador decide que leer toda la tabla es más rápido).

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.

5 – Programación con funciones y procedimientos

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

Ejemplo de llamada

Llamada a función

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

Llamada a procedimiento

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

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.

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;
```

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;
```

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

Ejemplo con OUT

```
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
```

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.

Condicionales

IF / ELSIF / ELSE

Permiten ejecutar distintas acciones según condiciones:

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

CASE

Útil cuando se desea evaluar múltiples posibles valores:

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

Bucles

LOOP con EXIT

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

WHILE

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

FOR IN SELECT

Recorre el resultado de una consulta automáticamente:

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

Variable mágica FOUND

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

Uso típico

```
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.

Cursores en PostgreSQL

Un cursor es una estructura que permite recorrer los resultados de una consulta fila por fila, lo cual es muy útil cuando necesitas procesar múltiples registros dentro de un procedimiento o función.

¿Cómo se declara un cursor?

La declaración se hace en la sección DECLARE, especificando una consulta SQL que se ejecutará cuando el cursor sea abierto:

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

Puedes usar cualquier consulta válida: con filtros, ordenaciones, funciones, joins, etc.

Ejemplo completo: recorrido ascendente (por defecto)

Este es el recorrido más común, usando FETCH NEXT (que es la dirección por defecto):

```
DO $$
DECLARE
  -- Declaración del cursor
  cur_empleados CURSOR FOR
    SELECT * FROM empleados ORDER BY id;

  -- Variable tipo RECORD para almacenar cada fila
  fila RECORD;
BEGIN
  OPEN cur_empleados;  -- Se abre el cursor

  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; -- Siempre cerrar el cursor
END;
$$;
```

Este ejemplo recorre la tabla empleados desde el primer registro hasta el último.

Ejemplo completo: recorrido descendente

También se puede recorrer una consulta desde el final hacia el principio usando FETCH PRIOR, empezando con FETCH LAST:

```
DO $$
DECLARE
  -- Declaración del cursor
  cur_empleados CURSOR FOR
    SELECT * FROM empleados ORDER BY id;

  -- Variable tipo RECORD para almacenar cada fila
  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;
$$;
```

Aquí recorremos la consulta en sentido inverso.

Resumen: direcciones de FETCH

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

Buenas prácticas

- Siempre cerrar el cursor con CLOSE al final.
- Usar ORDER BY dentro de la consulta del cursor si el orden es importante.
- Utilizar FOUND para saber si el FETCH devolvió una fila válida.

Variables tipo RECORD

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

Declaración

```
DECLARE
  empleado RECORD;
```

Uso con SELECT INTO

```
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;
```

Las variables tipo RECORD son muy útiles para consultas dinámicas o cursores.

RAISE NOTICE y RAISE EXCEPTION

RAISE NOTICE

Imprime información por pantalla sin detener la ejecución:

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

RAISE EXCEPTION

Lanza un error que detiene el procedimiento:

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

Template strings con placeholders

Tanto NOTICE cómo EXCEPTION permiten usar placeholders (%) en cadenas:

```
RAISE NOTICE 'Empleado: %, Cargo: %', emp.nombre, emp.cargo;
```

Los % se reemplazan por los valores en orden.

Bloques BEGIN … EXCEPTION

Permiten capturar errores y continuar la ejecución:

```
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

Puedes manejar errores solo de un bloque concreto:

```
BEGIN
  -- lógica general

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

  -- continúa
END;
```

SELECT INTO

SELECT INTO permite guardar el resultado de una consulta directamente en una o varias variables declaradas.

Ejemplo básico (una sola columna)

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

¿Qué ocurre?

- Si no se encuentra ninguna fila, la variable toma el valor NULL
- Si se encuentran varias filas, se toma solo la primera, sin error

SELECT INTO con múltiples columnas

Cuando se desea guardar más de una columna, hay dos opciones:

Opción A: Usar varias variables

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

Opción B: Usar una variable tipo RECORD

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

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

Esta opción es más flexible, especialmente si no se conoce la estructura completa.

SELECT INTO STRICT

Este tipo de SELECT INTO obliga a que la consulta devuelva exactamente UNA fila. Si no lo hace, lanza una excepción automáticamente.

Qué excepciones lanza:

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

Ejemplo completo con manejo de errores

```
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;
$$;
```

En este ejemplo se usa una variable RECORD para guardar múltiples campos, y se controla cualquier excepción que STRICT pueda lanzar.

Rollback implícito

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

Lanzar errores con RAISE EXCEPTION

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

Este tipo de excepción, si no se captura, revierte todos los cambios realizados dentro de la función.

6.- Triggers

Un trigger (en español, disparador) es un mecanismo automático que se ejecuta como respuesta a un evento que ocurre sobre una tabla o una vista.

El evento puede ser una operación de inserción (INSERT), modificación (UPDATE), borrado (DELETE), o incluso una operación DDL (como CREATE TABLE, ALTER, etc., aunque estas son menos comunes).

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.”

¿Para qué sirven los triggers?

Los triggers se utilizan para automatizar tareas repetitivas o críticas, como por ejemplo:

- **Mantener integridad lógica entre tablas.**

  Ej.: Si se borra un pedido, borrar automáticamente sus líneas asociadas.

- **Auditar cambios**

  Registrar en una tabla de log quién modificó un registro y cuándo.

- **Validar o corregir datos antes de guardarlos.**

  Convertir texto a mayúsculas, comprobar rangos, valores, etc.

- **Sincronizar información.**

  Actualizar totales, saldos o estadísticas cuando cambian datos base.

Tipos de triggers en PostgreSQL

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 a otra tabla
INSTEAD OF	En lugar de la operación (solo para vistas)	Personalizar comportamiento de vistas actualizables

Según el alcance

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

Estructura general de un trigger, ejemplo

En PostgreSQL, un trigger tiene dos componentes:

1. Una función (escrita normalmente en PL/pgSQL) que define la lógica a ejecutar.
2. Una declaración CREATE TRIGGER que vincula esa función a un evento concreto en una tabla.

Paso 1. Crear tabla principal

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

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
);
```

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	Variable del sistema que indica la operación (INSERT, UPDATE, DELETE).
NEW	Fila nueva (en INSERT o UPDATE).
OLD	Fila antigua (en UPDATE o DELETE).
current_user	Usuario que ejecutó la acción.

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 modificada.
EXECUTE FUNCTION	llama a la función definida antes.

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 que el programador tenga que hacerlo manualmente.

Ejemplo de trigger BEFORE (validación de datos)

Podemos usar un trigger BEFORE para validar o corregir valores antes de que se guarden:

```
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();
```

En este caso:

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

Ejemplo con INSTEAD OF (en vistas)

Los triggers INSTEAD OF se usan para hacer vistas actualizables.

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

Ahora creamos un trigger que permita insertar a través de la vista:

```
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, en realidad, se inserta en la tabla real.

Variables útiles dentro de un trigger

Variable	Descripción
TG_OP	Tipo de operación (INSERT, UPDATE, DELETE)
TG_TABLE_NAME	Nombre de la tabla sobre la que actúa
TG_WHEN	Momento del trigger (BEFORE, AFTER, INSTEAD OF)
NEW	Nueva fila (solo en INSERT y UPDATE)
OLD	Fila antigua (solo en UPDATE y DELETE)
TG_ARGV[]	Argumentos pasados al trigger (si los hay)

Buenas prácticas con triggers

Usar solo cuando sea necesario.

Los triggers pueden complicar la depuración si se abusa de ellos.

Documentar cada trigger.

Indica claramente en los comentarios qué hace y cuándo se ejecuta.

Evitar operaciones pesadas dentro de un trigger.

Deben ser rápidas, para no ralentizar la operación principal.

Usar AFTER para auditorías y logs, y BEFORE para validaciones o transformaciones.

Probar y depurar con cuidado.

Puedes usar RAISE NOTICE dentro de la función para imprimir mensajes de depuración:

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

Cómo ver y eliminar triggers

Listar triggers de una tabla

```
\d nombre_tabla
```

o

```
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();
```

7.- Casting de tipos

En PostgreSQL, puedes convertir un dato de un tipo a otro usando:

- ::tipo
- CAST(valor AS tipo)

Texto a número

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

Número a texto

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

Texto a fecha

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

TIMESTAMP a DATE (descarta la hora)

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

Alternativa con CAST

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

8.- Funciones comunes en PL/pgSQL

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
```

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'
```

Funciones matemáticas

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

Funciones condicionales

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

9.- Formatos de fecha en PostgreSQL

Formato natural (locale española)

Cuando el sistema usa la configuración regional española, el formato habitual de fecha es:

'27/09/2025 15:30:00'

Es decir: día/mes/año, seguido de la hora.

Este formato se puede aceptar en algunas configuraciones locales, pero no es estándar y puede causar errores al intercambiar datos entre sistemas.

Es recomendable usar el formato ISO 8601 para evitar posibles errores al interpretar la fecha.

También se puede usar una simplificación del formato ISO: YYYY-MM-DD (año-mes-dia).

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

Formato ISO 8601 (internacional)

ISO 8601 es un estándar internacional para representar fechas y horas.

```
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, hora y zona horaria	2023-09-27T15:30:00+02:00
Fecha, hora y zona horaria simplificada	2023-09-27T15:30:00+02
Separador entre fecha y hora	T
Zona Horaria	+02:00 / +02

PostgreSQL acepta este formato directamente en consultas e inserciones.

Ejemplo:

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

Fechas en formato ISO 8601 sin separadores

PostgreSQL admite también la variante compacta del estándar ISO 8601, es decir, sin guiones ni dos puntos:

```
YYYYMMDDTHHMMSS
```

Por ejemplo:

```
SELECT '20250927T183000'::TIMESTAMP;

-- Resultado: 2025-09-27 18:30:00
```

Y también con zona horaria:

```
SELECT '20250927T183000+0200'::TIMESTAMPTZ;

-- Resultado: 2025-09-27 18:30:00+02
```

Comparación de formatos

Formato	Cumple ISO 8601	Recomendado para interoperabilidad
`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

Cuándo usar formato compacto

- Cuando se cargan datos masivos
- Cuando se conoce el formato exacto en destino
- Cuando se desea optimizar espacio o parsing

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

UTC: Tiempo Universal Coordinado

- Es el tiempo estándar global, sin ajustes por horario de verano.
- Reemplaza al antiguo GMT (Greenwich Mean Time).
- UTC no cambia a lo largo del año.

¿Por qué se usa UTC como referencia?

- Un TIMESTAMP WITH TIME ZONE en PostgreSQL se guarda internamente en UTC.
- PostgreSQL convierte automáticamente entre UTC y tu zona local en la presentación:

```
-- 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
```

Esto permite comparar y almacenar fechas de forma precisa sin importar el lugar del mundo desde el que se acceda.

Siempre que trabajes con múltiples zonas horarias, utiliza TIMESTAMPTZ y piensa en UTC como el punto de referencia.

Tipo de dato TIMESTAMPTZ

TIMESTAMPTZ es una abreviatura de “timestamp with time zone”.

En PostgreSQL, existen dos tipos principales para manejar fechas y horas con precisión:

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

¿Cómo funciona realmente TIMESTAMPTZ?

Cuando guardas un valor en una columna TIMESTAMPTZ:

1. PostgreSQL convierte automáticamente esa fecha y hora a UTC (Tiempo Universal Coordinado).
2. Al recuperar el dato, lo muestra ajustado a la zona horaria de la sesión del usuario que lo consulta.

Ejemplo:

Supón que tu zona horaria es Europe/Madrid (UTC+2 en verano):

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

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

- PostgreSQL almacenará internamente: 2025-09-27 18:00:00+00 (UTC)

- Pero si haces:

```
SELECT fecha FROM eventos;
```

Verás: 2025-09-27 20:00:00+02

Ventajas de TIMESTAMPTZ

- Ideal para aplicaciones internacionales.
- Permite almacenar un valor estandarizado y mostrarlo en la zona local del usuario.

- Perfecto para comparar fechas reales de eventos sin preocuparte por zonas horarias.

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';
```

- En el primer caso, es una hora absoluta, sin contexto.
- En el segundo, PostgreSQL puede convertirla correctamente a UTC y a otras zonas.

Conclusión

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

10.- Instalación y Configuración de PostgreSQL + pgAdmin + JDBC

Instalación de PostgreSQL y pgAdmin en Windows

1. Descarga el instalador oficial desde:
   https://www.postgresql.org/download/windows/

2. Ejecuta el instalador de EnterpriseDB (EDB) y selecciona los componentes:
    - PostgreSQL Server
    - pgAdmin 4
    - Command Line Tools

3. Durante la instalación:
    - Define una contraseña para el usuario postgres.
    - Deja el puerto por defecto 5432.
    - Anota la carpeta de datos (normalmente C:\Program Files\PostgreSQL\16\data).

Al finalizar, PostgreSQL se inicia automáticamente como servicio de Windows.

Puedes comprobarlo en Administrador de tareas → pestaña Servicios o desde PowerShell:

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

Activar acceso por contraseña en Windows

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

Abre el archivo:

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

Busca la línea:

```
local   all             postgres                                trust
```

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.

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

Opción 1: Repositorios estándar (versión estable de Ubuntu)

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

Opción 2 (recomendada): Repositorios oficiales de PostgreSQL.org

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

Instala utilidades necesarias:

```
sudo apt install wget ca-certificates -y
```

Añade la clave GPG oficial de PostgreSQL:

```
wget -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | \
   sudo apt-key add -
```

Carga la información del sistema operativo y usa el código de versión:

```
. /etc/os-release
```

Añade el repositorio oficial:

```
echo "deb http://apt.postgresql.org/pub/repos/apt \
   ${VERSION_CODENAME}-pgdg main" | \
   sudo tee /etc/apt/sources.list.d/pgdg.list
```

Actualiza los paquetes e instala la versión más reciente:

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

(Esto instalará automáticamente la versión más nueva disponible.)

Verifica que el servicio esté activo:

```
sudo systemctl status postgresql
```

(Opcional) Instalar una versión específica:

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

Cambiar la contraseña del usuario postgres

Conéctate al intérprete de PostgreSQL:

```
sudo -u postgres psql
```

Dentro de psql, ejecuta:

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

Ejemplo:

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

Sal del intérprete con:

```
\q
```

Permitir acceso por contraseña (modo md5)

Edita el archivo de configuración de autenticación:

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

Busca esta línea:

```
local   all             postgres                                peer
```

Cámbiala por:

```
local   all             postgres                                md5
```

Y añade (si no existe) para habilitar conexiones TCP locales:

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

Guarda los cambios y reinicia PostgreSQL:

```
sudo systemctl restart postgresql
```

Métodos de autenticación más comunes:

Método	Significado
peer	confía en el usuario del sistema operativo (sin contraseña).
md5	requiere contraseña cifrada (recomendado).
trust	permite acceder sin contraseña (solo útil para pruebas locales).

Crear usuarios y bases de datos

Conéctate al sistema:

```
sudo -u postgres psql
```

Crear un usuario con contraseña

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

Crear una base de datos

```
CREATE DATABASE cursos;
```

Conceder todos los permisos al usuario

```
GRANT ALL PRIVILEGES ON DATABASE cursos TO juanma;
```

Darle permisos dentro del esquema público

```
\c cursos

GRANT ALL PRIVILEGES ON SCHEMA public TO juanma;
ALTER ROLE juanma CREATEDB;
```

Conectarse con psql y pgAdmin

Desde la terminal

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

Desde pgAdmin

- Abre pgAdmin
- Crea una Master Password local.
- En Servers → Create → Server…
    - Name: Localhost
    - Host: 127.0.0.1
    - Port: 5432
    - Username: postgres (u otro)
    - Password: la que configuraste
- Guarda la conexión y accede a la base de datos.

Instalar el driver JDBC

Manualmente

Descarga desde:

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

Archivo típico:

```
postgresql-42.7.2.jar
```

Colócalo en tu proyecto:

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

Compila y ejecuta con el classpath adecuado:

```
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 :)

Con Maven

En proyectos que usan Maven, añade esta dependencia al pom.xml:

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

Maven descargará automáticamente el driver y lo incluirá en el classpath.

Verifica con:

```
mvn dependency:tree
```

Deberías ver:

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