keep coding

BIG DATA & IA

BOOTCAMP

SQL.
Primeros pasos

🛠️ SQL

SQL significa Structured Query Language (lenguaje de consulta estructurado). Es el lenguaje estándar para trabajar con bases de datos relacionales.

Permite crear y modificar tablas, insertar o actualizar registros, consultar datos o gestionar permisos en las bases de datos.

¿Para qué se utiliza?

- Trabajar con tablas (DDL)
- Manipular registros dentro de las tablas (DML)
- Consultar datos (DQL)
- Gestionar el acceso a la base de datos y sus tablas (DCL)

🛠️ DDL

Data Definition Language

Se utiliza para definir o modificar la estructura de la base de datos.

```
CREATE TABLE

ALTER TABLE

DROP TABLE
```

🛠️ DML

Data Manipulation Language

Sirve para manipular los datos dentro de las tablas de una base de datos

```
INSERT INTO

UPDATE

DELETE FROM
```

🛠️ DQL

Data Query Language

Se utiliza para realizar consultas y recuperar información específica de las tablas de la base de datos

```
SELECT
```

🛠️ DCL

Data Control Language

Se ocupa del control de accesos y permisos en la base de datos, determinando quién puede acceder y qué operaciones puede realizar

GRANT

REVOKE

DENY

keep coding

Data Definition Language

Creación y modificación de la estructura

keep coding

🛠️ DDL - Creación de tablas

Se utiliza para definir o modificar la estructura de la base de datos.

```
CREATE TABLE customer AS(
customer_id SERIAL PRIMARY KEY,
name VARCHAR(255),
surname VARCHAR(255),
email VARCHAR(255),
phone VARCHAR(20)
);
```

🛠️ DDL - Creación de tablas

Con la siguiente consulta definimos la tabla reservas e indicamos qué tipo de dato es cada campo, cuál es la clave primaria e incremental y cuáles son claves foráneas.

```
CREATE TABLE reservation AS (
reservation_id SERIAL PRIMARY KEY,
customer_id INT,
apartment_id INT,
start_date DATE,
end_date DATE,
FOREIGN KEY (customer_id) REFERENCES
    customer(customer_id),
FOREIGN KEY (apartment_id) REFERENCES
    apartment(apartment_id)
);
```

🛠️ DDL - Creación de tablas

La siguiente consulta crea un índice en la tabla reservation sobre la fecha de comienzo de la reserva.

Un **índice** en una tabla de base de datos es una estructura que facilita listar y acceder a los registros, igual que un índice en un libro. Permite recorrer la tabla de forma más fácil y rápida. No almacena los datos, si no las referencias a dónde se encuentran.

```
CREATE INDEX idx_start_date ON reservation (start_date);
```

🛠️ DDL - Creación de tablas

En SQL el índice no aparece como una columna más. Es un objeto interno de la tabla.

	start_date	reservation_id	customer_id	apartment_id	end_date
1	2025-09-01	1	101	10	2025-09-07
2	2025-09-05	2	102	11	2025-09-12
3	2025-09-10	3	103	10	2025-09-14
4	2025-09-15	4	101	12	2025-09-18
5	2025-09-20	5	104	13	2025-09-25

🛠️ DDL - Creación de tablas

Si creas un índice en la columna start_date, la base de datos podrá buscar y recuperar las reservas ordenadas por fechas de inicio mucho más rápidamente. Sin un índice, la base de datos tendría que revisar cada fila para encontrar las fechas de inicio deseadas, lo cual podría ser lento, especialmente si hay muchas filas.

Es útil cuando tienes consultas frecuentes que seleccionan u ordenan datos basados en un campo, pero pueden ocupar espacio adicional y ralentizar operaciones de escritura. Su uso debe ser cuidadosamente planificado para evitar impactos negativos en el rendimiento.

🛠️ DDL - Creación de tablas

En la siguiente consulta, la función UNIQUE establece una restricción de unicidad en la combinación de valores de las columnas apartment_id y amenity_id. Esto significa que no puede haber dos filas en la tabla con la misma combinación de valores para apartment_id y amenity_id.

```
CREATE TABLE apt_amenity AS (
apt_amenity_id SERIAL PRIMARY KEY,
apartment_id INT,
amenity_id INT,
FOREIGN KEY (apartment_id) REFERENCES
    APARTMENT(apartment_id),
FOREIGN KEY (amenity_id) REFERENCES
    AMENITY(amenity_id),
UNIQUE (apartment_id, amenity_id)
);
```

keep coding

🛠️ DDL - Modificación de tablas

El comando ALTER TABLE modifica la tabla para hacer que la columna email sea única en los registros.

```
ALTER TABLE customer
ADD CONSTRAINT unique_email UNIQUE(email);
```

🛠️ DDL - Modificación de tablas

O en este caso, para que la columna email no puede contener valores nulos.

```
ALTER TABLE customer
ALTER COLUMN email SET NOT NULL;
```

🛠️ DDL - Modificación de tablas

Haciendo uso de la función **`ALTER TABLE`** crea las siguientes consultas sobre la tabla `reservation`

- Agregar una columna
- Quita una columna
- Modificar el tipo de dato de una columna
- Cambiar el nombre de una columna
- Añadir una clave primaria o foránea

	start_date	reservation_id	customer_id	apartment_id	end_date
1	2025-09-01	1	101	10	2025-09-07
2	2025-09-05	2	102	11	2025-09-12
3	2025-09-10	3	103	10	2025-09-14
4	2025-09-15	4	101	12	2025-09-18
5	2025-09-20	5	104	13	2025-09-25

keep coding

🛠️ DDL - Modificación de tablas

```sql
ALTER TABLE reservation
ADD COLUMN city VARCHAR(255);
```

```sql
ALTER TABLE reservation
DROP COLUMN end_date;
```

```sql
ALTER TABLE reservation
RENAME COLUMN city TO city_id;
```

```sql
ALTER TABLE reservation
ALTER COLUMN city_id TYPE INT;
```

```sql
ALTER TABLE reservation
ADD CONSTRAINT city_id FOREIGN KEY(city_id) REFERENCES city(id);
```

🛠️ DDL - Borrar tablas

La función DROP elimina la tabla y todos sus datos. Asegúrate de tener una copia de seguridad de los datos importantes antes de realizar una acción de este tipo, ya que no se puede deshacer y resultará en la pérdida permanente de la tabla y sus contenidos.

```
DROP TABLE customer;
```

Data Manipulation Language

Manipulación de datos

🛠️ DML - Insertar datos

La instrucción INSERT en SQL se utiliza para agregar nuevos registros (filas) a una tabla.

- INSERT INTO customer: Indica que estás insertando datos en la tabla llamada customer.
- (name, surname, email, phone): Especifica las columnas a las que estás insertando valores.
- VALUES ('Ana', 'González', 'ana.gonzalez@email.com', '912-222333'): Proporciona los valores que deseas insertar en esas columnas para la nueva fila.

El campo customer_id no es necesario especificarlo porque el propio DBMS lo genera (al crear la tabla pusimos SERIAL)

```
INSERT INTO customer(name, surname, email, phone)
VALUES ('Ana', 'González', 'ana.gonzalez@email.com', '912-222333');
```

🛠️ DML - Eliminar datos

La instrucción DELETE en SQL se utiliza para eliminar registros (filas) de una tabla.

- DELETE FROM customer: Indica que estás eliminando datos de la tabla llamada customer.
- WHERE email = 'ana.gonzalez@email.com': Especifica la condición que debe cumplir una fila para ser eliminada.

Ten en cuenta que la cláusula WHERE es opcional, pero sin ella, eliminarías todas las filas de la tabla. Es importante tener cuidado al usar DELETE para asegurarte de que estás eliminando las filas deseadas y de no perder información importante. Sería equivalente a usar TRUNCATE TABLE. Diferenciar entre DROP TABLE y DELETE, DROP borra datos y estructura, mientras DELETE sólo borra datos.

A veces se hacen "borrados lógicos" agregando el campo `deleted_at`.

```
DELETE FROM customer WHERE email='ana.gonzalez@email.com';
```

🛠️ DML - Actualizar datos

La instrucción UPDATE en SQL se utiliza para modificar los datos existentes en una tabla.

- UPDATE customer: Indica que estás actualizando datos en la tabla llamada customer.
- SET name = 'Anna': Especifica la columna que deseas actualizar (name en este caso) y el nuevo valor que deseas asignar.
- WHERE email = 'ana.gonzalez@email.com': Especifica la condición que debe cumplir una fila para que se actualice.

Es importante usar la cláusula WHERE para asegurarte de que estás actualizando las filas deseadas y no todas las filas en la tabla. Sin la cláusula WHERE, todos los registros en la tabla serían actualizados con el nuevo valor.

Suele existir un campo `last_modified_at` y `last_modified_by` para saber cuándo y quién ha hecho la última actualización del registro.

```
UPDATE customer SET name='Anna' WHERE email='ana.gonzalez@email.com';
```

🛠️ DML - Actualizar datos

La instrucción MERGE nos permite combinar dos tablas en una sola, realizando operaciones de inserción, actualización o eliminación de filas en función de una condición.

```
MERGE INTO customer AS tgt
USING customer_bis AS src
ON tgt.customer_id = src.customer_id
WHEN MATCHED THEN UPDATE SET
name = src.name,
surname = src.surname,
email = src.email,
phone = src.phone
WHEN NOT MATCHED THEN INSERT (
customer_id,
name,
surname,
email,
phone
) VALUES (
src.customer_id,
src.name,
src.surname,
src.email,
src.phone
);
```

🛠️ DML - Actualizar datos

- La cláusula `MERGE INTO` especifica la tabla de destino, customer, y `USING` la tabla de origen, customer_bis.
- La cláusula `ON` especifica la condición que se utilizará para comparar las filas de las dos tablas. En este caso, la condición es que las columnas `customer_id` de ambas tablas sean iguales.
- La cláusula `WHEN MATCHED` se ejecuta si la fila de la tabla de destino ya existe. En este caso, la instrucción actualiza los valores de las filas coincidentes con los valores de las filas de la tabla de origen.
- La cláusula `WHEN NOT MATCHED` se ejecuta si la fila de la tabla de destino no existe. En este caso, la instrucción inserta una nueva fila en la tabla de destino con los valores de la fila de la tabla de origen.

keep coding

Data Query Language

Consulta de datos

🛠️ DQL - Consulta de datos

La cláusula `SELECT` se utiliza para elegir qué columnas y filas quieres ver de una o varias tablas

```
SELECT
      columna1,
      columna2
FROM tabla
WHERE condición;
```

→ Datos de las columnas que quieres ver

→ Filtros aplicados a esos datos para seleccionar las filas que cumplan los criterios

🛠️ DQL - Consulta de datos

```
SELECT
      reservation_id
FROM reservations
WHERE apartment_id = 10;
```

	start_date	reservation_id	customer_id	apartment_id	end_date
1	2025-09-01	1	101	10	2025-09-07
2	2025-09-05	2	102	11	2025-09-12
3	2025-09-10	3	103	10	2025-09-14
4	2025-09-15	4	101	12	2025-09-18
5	2025-09-20	5	104	13	2025-09-25

🛠️ DQL - Consulta de datos

```
SELECT
        reservation_id
FROM reservations
WHERE apartment_id = 10;
```

	start_date	reservation_id	customer_id	apartment_id	end_date
1	2025-09-01	1	101	10	2025-09-07
2	2025-09-05	2	102	11	2025-09-12
3	2025-09-10	3	103	10	2025-09-14
4	2025-09-15	4	101	12	2025-09-18
5	2025-09-20	5	104	13	2025-09-25

🛠️ DQL - Consulta de datos

reservation_id
1
3

🛠️ DQL - Consulta de datos

```sql
SELECT *
FROM libros;
```

```sql
SELECT *
FROM libros
LIMIT 5;
```

```sql
SELECT
      nombre,
      ano_publicacion
FROM libros
LIMIT 5;
```

```sql
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE serie = 'El Cementerio de los libros olvidados';
```

```sql
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE serie = 'El Cementerio de los libros olvidados'
ORDER BY ano_publicacion ASC;
```

```sql
SELECT
      nombre,
      serie
FROM libros
WHERE ano_publicacion BETWEEN 2017 AND 2020;
```

🛠️ DQL - Combinar datos de diferentes tablas

La instrucción `JOIN` se utiliza junto con `SELECT` para combinar filas de dos o más tablas, basándose en una condición de relación entre las tablas (normalmente una clave primaria – PK en una tabla y una clave foránea – FK en la otra).

Existen diferentes tipos de `JOIN` dependiendo de la información que quieres obtener de ambas tablas

id	customer_id	apartment_id	start_date	end_date
1	10	101	2025-08-01	2025-08-05
2	13	102	2025-08-05	2025-08-08
3	10	103	2025-09-10	2025-09-15
4	13	104	2025-09-15	2025-09-20

Reservations

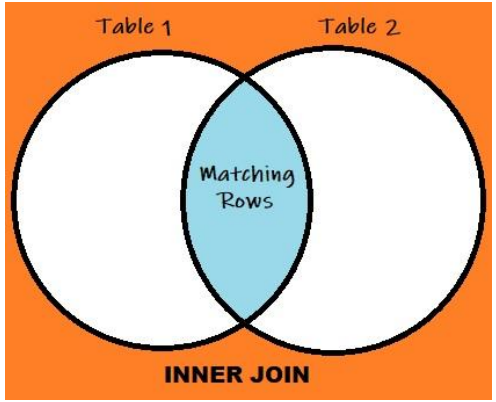
id	name	surname	email	phone
10	Ana	Pérez	ana@example.com	600111222
11	Luis	Gómez	luis@example.com	600222333
12	María	López	maria@example.com	600333444
13	Carlos	Ramírez	carlos@example.com	600444555

Customers

🛠️ DQL - INNER JOIN

`INNER JOIN` devuelve las filas para las que existen registros en ambas tablas

```sql
SELECT reservations.id,
       reservations.apartment_id,
       customers.email
FROM reservations
INNER JOIN customers
ON reservations.customer_id = customers.id;
```



🛠️ DQL - INNER JOIN

Reservations

id	customer_id	apartment_id	start_date	end_date
1	10	101	2025-08-01	2025-08-05
2	13	102	2025-08-05	2025-08-08
3	10	103	2025-09-10	2025-09-15
4	13	104	2025-09-15	2025-09-20

Customers

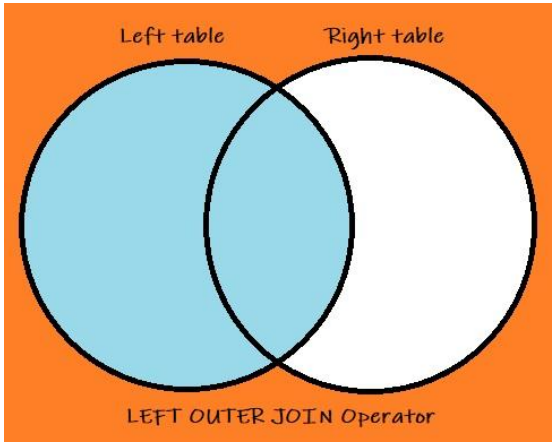
id	name	surname	email	phone
10	Ana	Pérez	ana@example.com	600111222
11	Luis	Gómez	luis@example.com	600222333
12	María	López	maria@example.com	600333444
13	Carlos	Ramírez	carlos@example.com	600444555

id	apartment_id	email
1	101	ana@example.com
3	103	ana@example.com
2	102	carlos@example.com
4	104	carlos@example.com

🛠️ DQL - LEFT JOIN

`LEFT JOIN` o `LEFT OUTER JOIN` devuelve todas las filas de la tabla de la izquierda y las coincidentes en la derecha

```
SELECT customers.name,
       customers.surname,
       customers.email,
       reservations.id as reservation_id,
       reservations.apartment_id
FROM customers
LEFT JOIN reservations
ON customers.id = reservations.customer_id;
```



🛠️ DQL - LEFT JOIN

Customers

id	name	surname	email	phone
10	Ana	Pérez	ana@example.com	600111222
11	Luis	Gómez	luis@example.com	600222333
12	María	López	maria@example.com	600333444
13	Carlos	Ramírez	carlos@example.com	600444555

Reservations

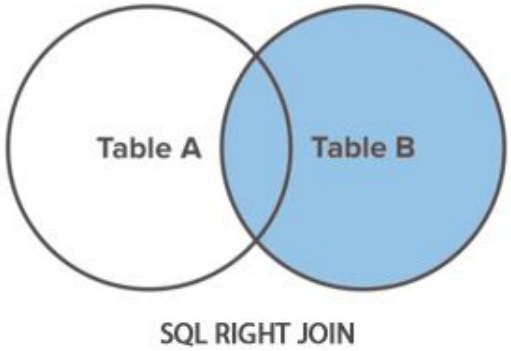
id	customer_id	apartment_id	start_date	end_date
1	10	101	2025-08-01	2025-08-05
2	13	102	2025-08-05	2025-08-08
3	10	103	2025-09-10	2025-09-15
4	13	104	2025-09-15	2025-09-20

name	surname	email	reservation_id	apartment_id
Ana	Pérez	ana@example.com	1	101
Ana	Pérez	ana@example.com	3	103
Luis	Gómez	luis@example.com		
María	López	maria@example.com		
Carlos	Ramírez	carlos@example.com	2	102
Carlos	Ramírez	carlos@example.com	4	104

🛠️ DQL - RIGHT JOIN

`RIGHT JOIN` o `RIGHT OUTER JOIN` devuelve todas las filas de la tabla de la derecha y las coincidentes en la izquierda

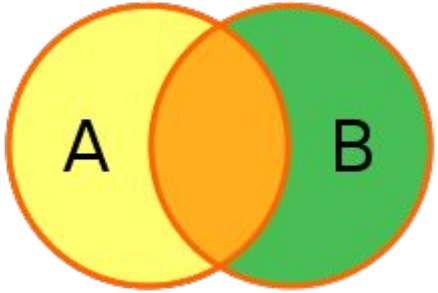
```
SELECT customers.name,
       customers.surname,
       customers.email,
       reservations.id as reservation_id,
       reservations.apartment_id
FROM reservations
RIGHT JOIN customers
ON reservations.customer_id = customers.id;
```



🛠️ DQL - FULL OUTER JOIN

`FULL OUTER JOIN` combina los datos de ambas tablas, incluso si no hay relación entre ellas

```
SELECT customers.name,
       customers.surname,
       customers.email,
       reservations.id as reservation_id,
       reservations.apartment_id
FROM reservations
FULL OUTER JOIN customers
ON reservations.customer_id = customers.id;
```



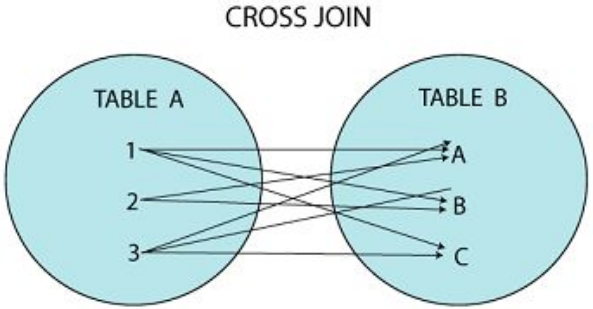
🛠️ DQL - FULL OUTER JOIN

customer_id	name	surname	email	reservation_id	apartment_id
10	Ana	Pérez	ana@example.com	1	101
10	Ana	Pérez	ana@example.com	3	103
11	Luis	Gómez	luis@example.com		
12	María	López	maria@example.com		
13	Carlos	Ramírez	carlos@example.com	2	102
13	Carlos	Ramírez	carlos@example.com	4	104
				5	105
				6	106

🛠️ DQL - CROSS JOIN

`CROSS JOIN` devuelve el producto cartesiano de dos tablas, es decir, combina cada fila de la primera tabla con cada fila de la segunda tabla. No utiliza una condición de coincidencia como los otros tipos de JOIN

```
SELECT customers.name,
       customers.surname,
       customers.email,
       reservations.id as reservation_id,
       reservations.apartment_id
FROM reservations
CROSS JOIN customers;
```



🛠️ DQL - CROSS JOIN

Cada cliente combina con todas las reservas, independientemente de que estén vinculados o no.

customer_id	name	surname	email	reservation_id	apartment_id
10	Ana	Pérez	ana@example.com	1	101
10	Ana	Pérez	ana@example.com	2	102
10	Ana	Pérez	ana@example.com	3	103
10	Ana	Pérez	ana@example.com	4	104
10	Ana	Pérez	ana@example.com	5	105
10	Ana	Pérez	ana@example.com	6	106
11	Luis	Gómez	luis@example.com	1	101
11	Luis	Gómez	luis@example.com	2	102
11	Luis	Gómez	luis@example.com	3	103
11	Luis	Gómez	luis@example.com	4	104
11	Luis	Gómez	luis@example.com	5	105
11	Luis	Gómez	luis@example.com	6	106
12	María	López	maria@example.com	1	101
12	María	López	maria@example.com	2	102

keep coding

Data Query Language

Manipulación de datos - Otras cláusulas

🛠️ WHERE

`WHERE` es una cláusula para filtrar las filas individualmente según la condición que se indique.

```
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE serie = 'El Cementerio de
los libros olvidados';
```

```
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE ano_publicacion BETWEEN
2019 AND 2024;
```

🛠️ GROUP BY

`GROUP BY` es una cláusula para agrupar filas que tienen valores iguales en una o más columnas y aplicar funciones de agregación, como SUM, AVG, COUNT, etc., a cada grupo de filas.

```sql
SELECT
      idioma,
      COUNT(*) AS total_libros_idioma
FROM libros
GROUP BY idioma
```

🛠️ ORDER BY

`ORDER BY` es una cláusula para ordenar los resultados según la columna o columnas que se indican. Se ordenan de forma `ASC` o `DESC` y hay que considerar el tipo de dato de la columna.

```
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE serie = 'El Cementerio de
los libros olvidados'
ORDER BY ano_publicacion ASC;
```

```
SELECT
      nombre,
      serie,
      autor_id
FROM libros
WHERE serie = 'El Cementerio de
los libros olvidados'
ORDER BY nombre DESC;
```

🛠️ HAVING

`HAVING` es una cláusula para filtrar resultados agrupados. A diferencia de `WHERE` que filtra individualmente las filas, `HAVING` filtra grupos agregados después de aplicar funciones de agregación como `COUNT()` `SUM()`, `AVG()`, etc.

Se utiliza junto con la cláusula `GROUP BY`

```sql
SELECT
      idioma,
      COUNT(*) AS total_libros_idioma
FROM libros
GROUP BY idioma
HAVING COUNT(*) >= 3;
```

```sql
WITH libros_idioma as (
      SELECT
            idioma,
            COUNT(*) AS total_libros_idioma
      FROM libros
      GROUP BY idioma
)
SELECT
      idioma,
      total_libros_idioma
FROM libros_idioma
WHERE total_libros_idioma >= 3;
```

🛠️ Orden de ejecución

```sql
SELECT                                  -- 1. Qué columnas mostrar
    columna1,
    columna2,
    COUNT(*),
    SUM(columna3)
FROM tabla                                  -- 2. De qué tabla
LEFT JOIN otra_tabla                    -- 3. Unir tablas
ON tabla.id = otra_tabla.fk_id          -- 4. En qué campos
WHERE condicion                         -- 5. Filtrar filas antes de agrupar
GROUP BY columna1, columna2             -- 6. Agrupar filas
HAVING COUNT(*) > 1                     -- 7. Filtrar grupos
ORDER BY columna2 DESC                  -- 8. Ordenar resultados
LIMIT 10;                               -- 9. Limitar filas
```

🛠️ CTE

CTE o Common Table Expression es una consulta temporal que se define en la parte superior de la query y se ejecuta en primer lugar. Aísla el código para hacerlo más legible, modularizarlo en partes más sencillas y reutilizar consultas en la misma query.

Comienza con la cláusula `WITH` y si le siguen más CTE, solo es necesario nombrarlas

```
WITH libros_idioma as (
      SELECT
            idioma,
            COUNT(*) AS total_libros_idioma
      FROM libros
      GROUP BY idioma
)
SELECT
      idioma,
      total_libros_idioma
FROM libros_idioma
WHERE total_libros_idioma >= 3;
```

🛠️ CTE

```
WITH libros as (
      SELECT
            nombre AS libro_nombre,
            ano_publicacion,
            autor_id
      FROM libros
),
autor as (
      SELECT
            id AS autor_id,
            nombre AS autor_nombre
      FROM autor
)
SELECT
      libros.libro_nombre,
      libros.ano_publicacion,
      autor.autor_nombre
FROM libros
INNER JOIN autor
ON libros.autor_id = autor.autor_id;
```

🛠️ SUBQUERY

Una subquery es una consulta anidada dentro de otra consulta principal. Se ejecuta una vez finalizada la primera consulta. No se le considera una buena práctica al complejizar el código y hacerlo más lento en ejecución

```
SELECT
      nombre,
      ano_publicacion
FROM libros
WHERE autor_id IN (
      SELECT id
      FROM autor
      WHERE nombre = “Gabriel García
Marquez”
)
;
```

🛠️ TRIGGER

Un trigger es un conjunto de instrucciones que se ejecutan automáticamente en respuesta a ciertos eventos en una tabla o vista. Los triggers son útiles para realizar acciones automáticas, como la actualización de datos en una tabla secundaria cuando ocurren cambios en una tabla principal.

Un trigger se utiliza para mantener la integridad de los datos, realizar un seguimiento de los cambios y aplicar reglas empresariales. Al activar los triggers, nos aseguramos la coherencia de los datos al automatizar tareas.

El proceso se invoca cuando se produce un evento especial en base de datos. Este conjunto de acciones pertenece a una clase específica de procedimientos almacenados, que se invocan automáticamente como respuesta a eventos del servidor de bases de datos.

🛠️ TRIGGER

```
CREATE OR REPLACE FUNCTION comprobar_precio()
RETURNS TRIGGER AS $$
BEGIN
IF NEW.price < 0 THEN
NEW.price = 0;
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_comprobar_precio
BEFORE INSERT ON apartment
FOR EACH ROW
EXECUTE FUNCTION comprobar_precio();
```

✅ Buenas prácticas

- Usa una indentación clara y consistente para facilitar la lectura del código.
- `SELECT *` no es una buena práctica. Es mejor seleccionar los campos que deseamos y en el orden deseado.
- Es conveniente usar alias para tablas y que estos sean relevantes.
- No usar `JOIN` implícitos.
- Modularizar usando CTE.
- Poner en mayúsculas las palabras reservadas del lenguaje.
- Evitar la cláusula `WHERE IN/NOT IN` (SUBQUERY) en su lugar usar `JOIN`.

www.keepcoding.io cursos@keepcoding.io (+34) 916 33 1779