SQL Primeros Pasos

KEEPCODING
Tech School

SQL

SQL, que significa "Structured Query Language" (Lenguaje de Consulta Estructurado), es un lenguaje diseñado para gestionar y manipular bases de datos relacionales. Se utiliza para realizar diversas operaciones en bases de datos, como la creación y modificación de tablas, la inserción y actualización de datos, la realización de consultas para recuperar información específica y la gestión de permisos para los diferentes usuarios.
SQL proporciona un conjunto de comandos estándar que permiten interactuar con bases de datos. Estos comandos se dividen en categorías como:

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DQL (Data Query Language)
- DCL (Data Control Language)

DDL

DDL se encarga de definir y gestionar la estructura de la base de datos, como la creación y modificación de tablas. Sus instrucciones más comunes son:

- **CREATE TABLE**: Crea una nueva tabla.
- **ALTER TABLE**: Modifica la estructura de una tabla existente.
- **DROP TABLE**: Elimina una tabla.

DML

DML se enfoca en manipular los datos dentro de las tablas, como la inserción, actualización y eliminación de registros. Sus instrucciones más comunes son:

- **INSERT INTO**: Agrega nuevos registros a una tabla.
- **UPDATE**: Modifica los datos existentes en una tabla.
- **DELETE FROM**: Elimina registros de una tabla.

DQL

DQL se utiliza para realizar consultas y recuperar información específica de la base de datos. Sus instrucciones más comunes son:

- **SELECT**: Recupera datos de una o varias tablas.

DCL

DCL se ocupa del control de accesos y permisos en la base de datos, determinando quién puede acceder y qué operaciones pueden realizar. Sus instrucciones más comunes son:

- **GRANT**: Concede privilegios o permisos a usuarios.
- **REVOKE**: Retira privilegios o permisos previamente concedidos.
- **DENY**: Niega un acceso específico a un usuario.

Creación de tablas

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

Estamos creando la tabla customer con un id entero e incremental e indicando que será su clave primaria. El resto de campos serán cadenas de caracteres de longitud variable con un máximo de 255 (20 para el teléfono).

Creación de tablas

```
CREATE TABLE reservation (
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)
);
```

Estamos creando la tabla reservation con un id entero e incremental e indicando que será su clave primaria. Los campos de fecha de inicio de la reserva y fecha final son de tipo DATE, además indicamos que los campos customer_id y apartment_id son claves foráneas y hacen referencia a otras tablas.

Creación de tablas

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

Estamos creando un índice en la tabla de reservas sobre el campo de fecha de inicio de la reserva.

Un índice en una base de datos es como un índice alfabético en un libro. Facilita encontrar información más rápidamente. En lugar de leer todo el libro, puedes ir directamente a la página que necesitas.

En el ejemplo anterior de la tabla reservation, 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.

Creación de tablas

```
CREATE TABLE apt_amenity (
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)
);
```

La línea UNIQUE (apartment_id, amenity_id) en la creación de la tabla apt_amenity 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.

Creación de tablas

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

Este comando ALTER TABLE añadirá una restricción UNIQUE a la columna email, asegurando que no puede haber dos filas en la tabla customer con el mismo valor en la columna email.

Creación de tablas

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

Este comando modificará la estructura de la tabla customer para especificar que la columna email no puede contener valores nulos.

Creación de tablas

ALTER TABLE nos puede servir para:

- Agregar y quitar columnas

```
ALTER TABLE nombre_tabla
ADD COLUMN nombre_columna tipo_dato;

ALTER TABLE nombre_table
DROP COLUMN nombre_columna;
```

- Modificar el tipo de dato de una columna

```
ALTER TABLE nombre_tabla
ALTER COLUMN nombre_columna TYPE nuevo_tipo_dato;
```

- Cambiar el nombre de una columna

```
ALTER TABLE nombre_tabla
RENAME COLUMN nombre_columna TO nuevo_nombre;
```

- Añadir una clave primaria o foránea

```
ALTER TABLE nombre_tabla
ADD CONSTRAINT nombre_clave_primaria PRIMARY KEY (columna1);

ALTER TABLE nombre_tabla
ADD CONSTRAINT nombre_clave_foranea FOREIGN KEY (columna) REFERENCES otra_tabla(otra_columna);
```

Borrar Tabla

```
DROP TABLE customer;
```

Este comando eliminará permanentemente la tabla customer 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.

Insertar Datos

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

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)

Eliminar Datos

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

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. En este caso, estás eliminando al cliente cuyo email es ana.gonzalez@email.com

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.

Actualizar Datos

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

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. En este caso, estás actualizando al cliente cuyo email es ana.gonzalez@email.com

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 quien ha hecho la última actualización del registro.

Actualizar Datos

```
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
);
```

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.

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.

Consultar Datos

```
SELECT customer_id
, name
, surname
FROM customer
WHERE customer_id > 5
ORDER BY customer_id
```

La instrucción SELECT en SQL se utiliza para recuperar datos de una o varias tablas.

- **SELECT customer_id, name, surname**: Indica las columnas que deseas ver en el resultado (que datos) .
- **FROM customer**: Especifica la tabla de la cual deseas seleccionar datos, en este caso, la tabla llamada **customer** (desde dónde obtengo los datos).
- **WHERE customer_id > 5:** Especifica la condición que debe cumplir una fila para que se muestre. En este caso, estás mostrando a los clientes cuyo id es mayor que 5. (< > = IN NOT IN IS LIKE)
- **ORDER BY customer_id:** Ordena el resultado por el campo customer_id de modo ascendente (por defecto).

Ejercicio: nombre y apellidos de los owner que tengan email nulo ordenados por apellido de modo descendente.

Combinar datos de diferentes tablas

Para combinar filas de dos o más tablas de usa la instrucción JOIN, basándose en una condición de relación entre ellas (PK y FK). Los distintitos tipos de JOIN son:

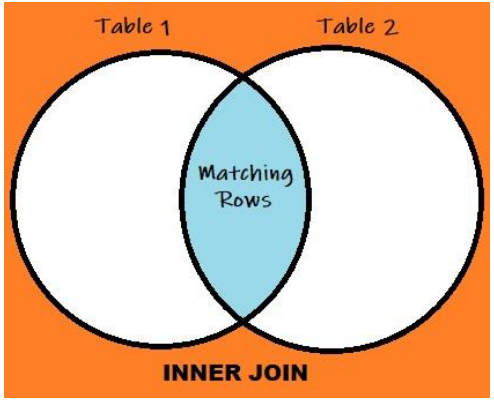
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- CROSS JOIN

INNER JOIN

El INNER JOIN devuelve las filas que tienen coincidencias en ambas tablas.

Ejemplo: que clientes a su vez están dados de alta como propietarios con el mismo email

```
SELECT *
FROM customer
INNER
JOIN owner
ON customer.email = owner.email;
```

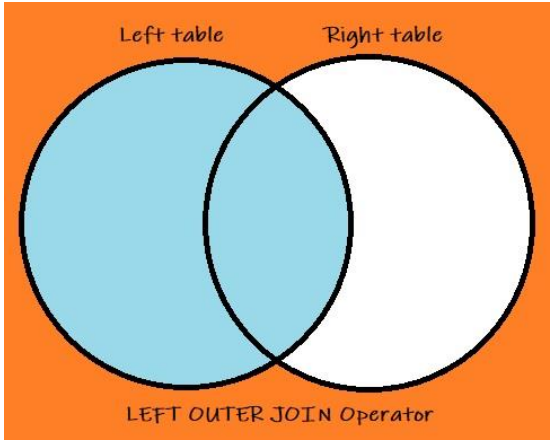


LEFT JOIN

El LEFT JOIN devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha. Si no hay coincidencias, las columnas de la tabla derecha serán NULL.

Ejemplo: datos de cliente y si es posible su nif de propietario

```
SELECT customer.*, owner.nif
FROM customer
LEFT
JOIN owner
ON customer.email = owner.email;
```

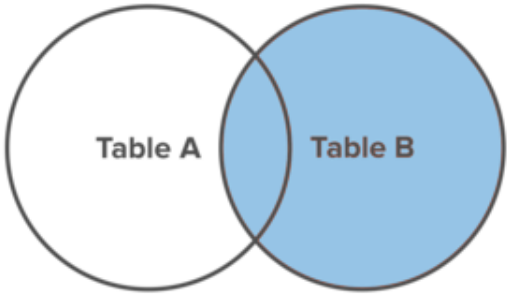


RIGHT JOIN

El RIGHT JOIN devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquieda. Si no hay coincidencias, las columnas de la tabla izquierda serán NULL.

Ejemplo: datos de cliente y si es posible su nif de propietario

```
SELECT customer.*, owner.nif
FROM owner
RIGHT
JOIN customer
ON customer.email = owner.email;
```



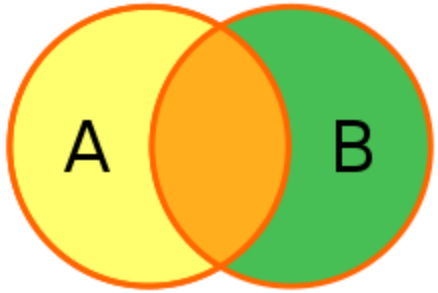
SQL RIGHT JOIN

FULL JOIN

El FULL JOIN devuelve todas las filas de la tabla derecha y todas las filas de la tabla izquieda. Si no hay coincidencias, las columnas serán NULL.

Ejemplo: datos de cliente y propietarios

```
SELECT *
FROM owner
FULL
JOIN customer
ON customer.email = owner.email
```

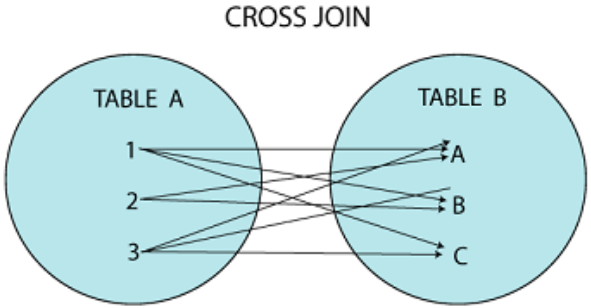


CROSS JOIN

El CROSS JOIN en SQL 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

Ejemplo: todas las comodidades en todos los apartamentos

```
SELECT *
FROM apartment
CROSS
JOIN amenity
```



GROUP BY

La cláusula GROUP BY en SQL se utiliza 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.

Ejemplo: cuántas reservas tiene cada cliente (email) y cuál es su precio medio?

```
SELECT customer.email
, COUNT(*) AS total_reservas
, AVG(apartment.price) AS precio_medio
FROM customer
INNER
JOIN reservation
ON reservation.customer_id = customer.customer_id
INNER
JOIN apartment
ON apartment.apartment_id = reservation.apartment_id
GROUP BY email
```


GROUP BY

La cláusula HAVING en SQL se utiliza en combinación con la cláusula GROUP BY y se emplea para filtrar el resultado de una consulta que involucra funciones de agregación. Mientras que la cláusula WHERE se utiliza para filtrar filas antes de que se realice la agregación, la cláusula HAVING filtra los resultados después de la agregación, basándose en los resultados de esas funciones.

Ejemplo: cuántas reservas tiene cada cliente (email) y cuál es su precio medio?

```
SELECT customer.email
, COUNT(*) AS total_reservas
, AVG(apartment.price) AS precio_medio
FROM customer
INNER
JOIN reservation
ON reservation.customer_id = customer.customer_id
INNER
JOIN apartment
ON apartment.apartment_id = reservation.apartment_id
GROUP BY email
HAVING AVG(apartment.price) > 135
ORDER BY precio_medio DESC
```

ORDEN DE EJECUCIÓN

1. **FROM:** La fase **FROM** es la primera en ejecutarse. Aquí se identifican las tablas o vistas que se utilizarán en la consulta.
2. **WHERE:** Después de la fase **FROM**, se aplica la cláusula **WHERE** para filtrar las filas según las condiciones especificadas.
3. **GROUP BY:** Si se utiliza una cláusula **GROUP BY**, se agrupan las filas según las columnas especificadas.
4. **HAVING:** La cláusula **HAVING** se aplica después de la agrupación para filtrar los grupos según las condiciones especificadas.
5. **SELECT:** La cláusula **SELECT** se ejecuta después de aplicar las condiciones de filtrado y agrupación. Aquí se seleccionan las columnas y se aplican las funciones de agregación si es necesario.
6. **ORDER BY:** La fase **ORDER BY** se aplica después de todas las demás operaciones y se utiliza para ordenar los resultados según las columnas especificadas.

SUBQUERY

Una subconsulta, también conocida como subquery, es una consulta que está anidada dentro de otra consulta principal. La subconsulta se ejecuta primero y su resultado se utiliza en la consulta principal.

Datos sobre los apartamentos que tengan 3 o más reservas:

```
SELECT apartment.apartment_id
    , apartment.price
    , apartment.owner_id
    , apartment.location
 FROM apartment
 JOIN (SELECT apartment_id
         , COUNT(*) AS numero_reservas
       FROM reservation
       GROUP BY 1
      HAVING COUNT(*) >= 3) mas_reservados
  ON apartment.apartment_id = mas_reservados.apartment_id;
```


CTE

Una CTE (Common Table Expression) te permite definir temporalmente una tabla en el ámbito de una consulta SQL. A menudo, se utiliza para mejorar la legibilidad y modularidad del código, dividiendo una consulta compleja en partes más manejables y con nombres descriptivos.

Datos sobre los apartamentos que tengan 3 o más reservas:

```
SELECT apartment.apartment_id
    , apartment.price
    , apartment.owner_id
    , apartment.location
 FROM apartment
 JOIN (SELECT apartment_id
         , COUNT(*) AS numero_reservas
       FROM reservation
      GROUP BY 1
     HAVING COUNT(*) >= 3) mas_reservados
  ON apartment.apartment_id = mas_reservados.apartment_id;
```


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.

```
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 JOINS implícitos.
- Modularizar usando CTE.
- Poner en mayúsculas las palabras reservadas del lenguaje.
- Evitar la cláusula WHERE IN/NOT IN (SUBQUERY) en su lugar usar JOINS.

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá