Semana 3.1 Clase 4 - SQL

SQL (Structured Query Language) es un lenguaje de programación utilizado para interactuar con bases de datos relacionales. Se utiliza para realizar tareas como consultar, actualizar y administrar datos almacenados en una base de datos.

Las bases de datos relacionales se organizan en tablas, cada una de las cuales tiene una serie de columnas y filas. Las columnas definen los tipos de datos que se pueden almacenar en una tabla, mientras que las filas contienen los datos reales. Esto permite a los usuarios acceder a estos datos de manera eficiente y efectiva.

Este lenguaje de programación se divide en tres partes principales: DDL (Data Definition Language), DML (Data Manipulation Language) y DCL (Data Control Language). La DDL se utiliza para definir y modificar la estructura de la base de datos, como crear y eliminar tablas y definir restricciones de integridad de datos. La DML se utiliza para manipular los datos de la base de datos, como insertar, actualizar y eliminar registros. La DCL se utiliza para controlar el acceso a la base de datos, como otorgar y revocar permisos de usuario.

Las consultas SELECT son una parte fundamental de SQL. Se utilizan para recuperar datos de una o más tablas en la base de datos. Las consultas SELECT pueden incluir filtros para restringir los resultados a un conjunto específico de datos. También se pueden utilizar para ordenar y agrupar datos. Las consultas SELECT pueden ser complejas, pero una vez que se comprende su sintaxis, se pueden realizar consultas muy poderosas y útiles.

SQL también incluye funciones integradas, como SUM, AVG, COUNT y MAX, que se pueden utilizar para realizar cálculos en los datos recuperados de la base de datos. También se pueden utilizar expresiones, como JOIN, para combinar conjuntos de resultados de varias consultas.

Otras características útiles de SQL incluyen transacciones, que permiten realizar varias operaciones en la base de datos como una sola unidad atómica, y vistas, que permiten crear tablas virtuales que se derivan de tablas existentes. Las vistas se pueden utilizar para simplificar la lógica de las consultas y para proporcionar una capa adicional de seguridad para los datos sensibles.

SQL es un lenguaje muy poderoso y popular utilizado en una amplia variedad de aplicaciones, desde pequeñas bases de datos personales hasta grandes sistemas empresariales. Aprender SQL puede abrir muchas oportunidades profesionales y es una habilidad valiosa para cualquier persona que trabaje con datos. Hay muchas herramientas y recursos disponibles para aprender SQL, incluyendo cursos en línea, tutoriales y libros de texto.

DDL (Data Definition Language)
La DDL se utiliza para crear y modificar la estructura de la base de datos. Aquí hay algunos ejemplos:

- Crear una tabla nueva:

```
CREATE TABLE customers (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100),
    created_at DATE
);
 DATETIME YYYY-MM-DD 00:00:00.000
 DATE YYYY-MM-DD
```

Este código crea una nueva tabla llamada "customers" con cuatro columnas: "id", "name", "email" y "created_at".

- Modificar una tabla existente:

```
ALTER TABLE customers ADD COLUMN phone_number VARCHAR(20);
```

Este código agrega una nueva columna llamada "phone_number" a la tabla "customers".

DML (Data Manipulation Language)

La DML se utiliza para manipular los datos en la base de datos.

- Insertar datos en una tabla:

```
INSERT INTO customers (id, name, email, created_at)
```

```
VALUES (1, 'Juan', 'juan@example.com', '2021-01-01');
```

Este código inserta un nuevo registro en la tabla "customers" con los valores especificados.

- Actualizar datos en una tabla:

```
UPDATE customers
SET email = 'juan@example.net'
WHERE id = 1;
```

Este código actualiza el valor de la columna "email" para el registro con "id" igual a 1.

- Eliminar datos de una tabla:

```
DELETE FROM customers
WHERE created_at < '2022-01-01';
```

Este código elimina todos los registros de la tabla "customers" donde la fecha de creación es anterior al 1 de enero de 2022.

DCL (Data Control Language)

La DCL se utiliza para controlar el acceso a la base de datos.

- Conceder permisos de usuario:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON customers TO user1;
```

Este código otorga al usuario "user1" los permisos para seleccionar, insertar, actualizar y eliminar datos de la tabla "customers".

- Revocar permisos de usuario:

```
REVOKE INSERT, UPDATE ON customers FROM user1;
```

Este código revoca los permisos del usuario "user1" para insertar y actualizar datos en la tabla "customers".

- Crear roles de usuario:

```
CREATE ROLE accountant;
```

Este código crea un nuevo rol de usuario llamado "accountant".

Estos son solo algunos ejemplos de cómo se utilizan las diferentes partes de SQL. Con una comprensión sólida de DDL, DML y DCL, puedes crear y administrar bases de datos de manera efectiva y segura.

Con una comprensión sólida de DDL, DML y DCL, puedes crear y administrar bases de datos de manera efectiva y segura.

Supongamos que tenemos una tabla llamada `productos` con los campos `id`, `nombre` y `precio`. Para insertar un nuevo producto en la tabla, podemos utilizar la siguiente sentencia:

```
INSERT INTO productos (nombre, precio)
VALUES ('camisa', 25.99);
```

Esta sentencia insertará un nuevo registro en la tabla `productos` con el nombre "camisa" y un precio de 25.99. Si queremos insertar más de un registro a la vez, podemos hacerlo de la siguiente manera:

```
INSERT INTO productos (nombre, precio)
VALUES ('camisa', 25.99),
       ('pantalón', 39.99),
       ('zapatos', 49.99);
```

En este caso, estamos insertando tres nuevos registros en la tabla `productos`: una camisa, un pantalón y unos zapatos, cada uno con su respectivo precio.

Es importante tener en cuenta que si la tabla tiene campos obligatorios (como puede ser el caso de una columna que requiere un valor único), debemos asegurarnos de proporcionar un valor para ese campo en la sentencia `INSERT INTO`. De lo contrario, la inserción fallará y recibiremos un error.

archivo.sql

DROP TABLE IF EXISTS dbname.tbname;

CREATE TABLE IF NOT EXISTS dbname.tbname(

id INT PK

name VARCHAR(20)

last_name VARCHAR(20)

);

INSERT INTO dbname.tbname(id, name, last_name)

ON (1, juan, campias);

Resumen:

SELECT	SELECT **Col1**, **Col2**, Col3 as columna	Nos brinda las columnas que deseamos
FROM	FROM dbnombre.tbnombre tabla	Nos brinda la tabla en la cual existen las columnas
LIMIT	LIMIT **10**	Limita la base de las filas que se devuelven
ORDER BY	ORDER BY col1	Ordena la tabla en base a una columna. Usado por defecto en **DESC y ASC**
WHERE	WHERE **Col > 5**	Una declaración condicional para filtrar los resultados
LIKE	WHERE **Col LIKE '%me%'**	Solo trae las columnas que tienen 'me' en el texto.
IN	WHERE **Col IN ('Y', 'N')**	A filter for only rows with column of 'Y' or 'N'
NOT	WHERE **Col NOT IN ('Y', 'N')**	**NOT** is frequently used with **LIKE** and **IN**
AND	WHERE **Col1 > 5 AND**	Filter rows where two or more conditions must

	Col2 < 3	be true
OR	WHERE **Col1 > 5 OR Col2 < 3**	Filter rows where at least one condition must be true
BETWEEN	WHERE **Col BETWEEN 3 AND 5**	Often easier syntax than using an **AND**

No es CASE SENSITIVE*

SELECT: La cláusula SELECT se utiliza para seleccionar las columnas que se quieren mostrar en los resultados de la consulta. Se utiliza para recuperar datos de una o varias tablas y puede incluir funciones de agregación, operaciones matemáticas y de cadenas, alias, entre otros.

Ejemplo:

```
SELECT name, age FROM students;
```

Este ejemplo recuperará los nombres y edades de todos los estudiantes en la tabla "students".

FROM: La cláusula FROM se utiliza para especificar la tabla o tablas de las cuales se recuperarán los datos.

Ejemplo:

```
SELECT o.* FROM db.orders o;
```

Este ejemplo recuperará todos los datos de la tabla "orders".

ORDER BY: La cláusula ORDER BY se utiliza para ordenar los resultados de la consulta por una o varias columnas.

Ejemplo:

```
SELECT name, age FROM students ORDER BY age DESC; ASC
```

Este ejemplo recuperará los nombres y edades de todos los estudiantes en la tabla "students" ordenados por edad de forma descendente.

GROUP BY: La cláusula GROUP BY se utiliza para agrupar los resultados de la consulta por una o varias columnas.

Ejemplo:

```
SELECT gender, AVG(age) as avg_age
FROM students
GROUP BY gender;
```

Este ejemplo recuperará el género y el promedio de edad de los estudiantes agrupados por género.

HAVING: La cláusula HAVING se utiliza para filtrar los resultados de la consulta después de que se hayan agrupado por la cláusula GROUP BY.

Ejemplo:

```
SELECT gender, AVG(age)
FROM students
GROUP BY gender
HAVING AVG(age) > 25;
```

Este ejemplo recuperará el género y el promedio de edad de los estudiantes agrupados por género, pero solo aquellos cuyo promedio de edad sea mayor a 25.

Operaciones de agregación: Las operaciones de agregación (por ejemplo, SUM, AVG, COUNT) se utilizan para realizar cálculos sobre un conjunto de datos.

Ejemplo:

```
SELECT COUNT(gender), *
FROM students;
```

Este ejemplo recuperará el número total de estudiantes en la tabla "students".

La cláusula "DISTINCT" se utiliza en SQL para seleccionar solamente los valores únicos de una columna en una consulta. Por ejemplo, si queremos obtener una lista de todos los países únicos en una tabla de datos de clientes, podemos usar la siguiente consulta SQL:

Ejemplo:

```
SELECT DISTINCT gender FROM students;
```

Este ejemplo recuperará los géneros únicos de los estudiantes en la tabla "students".

LIMIT: La cláusula LIMIT se utiliza para limitar el número de filas que se recuperan de una consulta.

Ejemplo:

```
SELECT s.sexo as edad, s.* FROM db.students s
GROUP BY sexo
ORDER BY edad DESC
LIMIT 10;
```

Este ejemplo recuperará los primeros 10 estudiantes en la tabla "students".

WHERE: La cláusula WHERE se utiliza para filtrar los resultados de la consulta según una o varias condiciones.

Ejemplo:

```
SELECT * FROM students WHERE age > 20;

SELECT *
FROM personas
WHERE estado_civil NOT IN ('Casado', 'Casada');
```

Este ejemplo recuperará todos los estudiantes en la tabla "students" cuya edad sea mayor a 20.

JOIN:

En SQL, los joins son una técnica que permite combinar datos de dos o más tablas relacionadas entre sí. Hay varios tipos de joins que se pueden utilizar, y cada uno de ellos se ajusta a diferentes necesidades en función de la relación entre las tablas. A continuación, describiremos los tipos de joins más comunes en SQL:

1. INNER JOIN: también conocido como join, es el tipo de join más común en SQL. Este tipo de join devuelve únicamente las filas que tienen coincidencias en ambas tablas. En otras palabras, el resultado solo muestra los datos que tienen una correspondencia exacta en ambas tablas.
2. LEFT JOIN: este tipo de join devuelve todas las filas de la tabla izquierda y las coincidencias de la tabla derecha. En otras palabras, las filas de la tabla izquierda se mantienen, incluso si no hay coincidencias en la tabla derecha. Si no hay coincidencias en la tabla derecha, los valores para esas filas aparecerán como NULL en el resultado.
3. RIGHT JOIN: al contrario que el left join, este tipo de join devuelve todas las filas de la tabla derecha y las coincidencias de la tabla izquierda. Las filas de la tabla derecha se mantienen, incluso si no hay coincidencias en la tabla izquierda. Si no hay coincidencias en la tabla izquierda, los valores para esas filas aparecerán como NULL en el resultado.
4. FULL OUTER JOIN: este tipo de join devuelve todas las filas de ambas tablas, incluso si no hay coincidencias en alguna de ellas. Si no hay coincidencias, los valores para esas filas aparecerán como NULL en el resultado.
5. CROSS JOIN: este tipo de join devuelve todas las combinaciones posibles de filas de ambas tablas. En otras palabras, no hay ninguna condición de unión en este tipo de join.
6. SELF JOIN: este tipo de join se utiliza cuando se necesita combinar datos de una misma tabla. Se utiliza una tabla temporal para crear una relación entre los datos de la misma tabla.

Cada tipo de join es útil en diferentes situaciones, por lo que es importante elegir el tipo de join adecuado para obtener los resultados deseados. Conociendo las características

de cada tipo de join, se puede seleccionar el más apropiado para cada situación en particular. Es importante tener en cuenta que los joins pueden tener un impacto significativo en el rendimiento de las consultas SQL, especialmente en bases de datos grandes, por lo que se deben utilizar con precaución y optimizarlos adecuadamente.

1. INNER JOIN: Este tipo de JOIN devuelve solo las filas que tienen coincidencias en ambas tablas.

Supongamos que tenemos dos tablas: "Empleados" y "Departamentos". Para mostrar solo los empleados que pertenecen al departamento "Ventas", podemos utilizar INNER JOIN de la siguiente manera:

```
SELECT Empleados.nombre, Departamentos.nombre
FROM Empleados
JOIN Departamentos
ON Empleados.departamento_id = Departamentos.id
WHERE Departamentos.nombre = 'Ventas';
```

1. LEFT JOIN: Este tipo de JOIN devuelve todas las filas de la tabla de la izquierda (la primera tabla mencionada en la consulta) y las filas coincidentes de la tabla de la derecha.

Supongamos que tenemos dos tablas: "Clientes" y "Pedidos". Para mostrar todos los clientes y sus pedidos, incluso si algunos clientes aún no han realizado ningún pedido, podemos utilizar LEFT JOIN de la siguiente manera:

```
SELECT Clientes.nombre, Pedidos.numero
FROM Clientes
LEFT JOIN Pedidos
ON Clientes.id = Pedidos.cliente_id;
```

1. RIGHT JOIN: Este tipo de JOIN devuelve todas las filas de la tabla de la derecha (la segunda tabla mencionada en la consulta) y las filas coincidentes de la tabla de la izquierda.

Supongamos que tenemos dos tablas: "Productos" y "Compras". Para mostrar todos los productos, incluso aquellos que aún no se han comprado, podemos utilizar RIGHT

JOIN de la siguiente manera:

```
SELECT Productos.nombre, Compras.fecha
FROM Productos
RIGHT JOIN Compras
ON Productos.id = Compras.producto_id;
```

1. FULL OUTER JOIN: Este tipo de JOIN devuelve todas las filas de ambas tablas. Si no hay coincidencias, se muestran valores NULL.

Supongamos que tenemos dos tablas: "Clientes" y "Pagos". Para mostrar todas las relaciones entre clientes y pagos, podemos utilizar FULL OUTER JOIN de la siguiente manera:

```
SELECT Clientes.nombre, Pagos.monto
FROM Clientes
FULL OUTER JOIN Pagos
ON Clientes.id = Pagos.cliente_id;
```

1. CROSS JOIN: Este tipo de JOIN devuelve todas las combinaciones posibles de las filas de ambas tablas.

Supongamos que tenemos dos tablas: "Alumnos" y "Materias". Para mostrar todas las combinaciones posibles de alumnos y materias, podemos utilizar CROSS JOIN de la siguiente manera:

```
SELECT Alumnos.nombre, Materias.nombre
FROM Alumnos
CROSS JOIN Materias;
```

Actividad 1:

"Quiero obtener una lista de los 10 productos más vendidos en mi tienda en el último mes, junto con su cantidad total vendida y su precio promedio. Solo quiero incluir los productos que se vendieron en más de una transacción y que tengan un precio

promedio mayor a $20. Además, quiero que la lista esté ordenada de mayor a menor cantidad vendida y que se muestren solamente los productos cuyo precio promedio es mayor a $20".

BONUS:

```
import psycopg2
```

```
conn = psycopg2.connect(
host="localhost",
database="mydatabase",
user="myusername",
password="mypassword",
port="5432"
)
```

Los parámetros necesarios para esta conexión son los siguientes:

- `host`: la dirección del host donde se encuentra la base de datos. Si estás trabajando en tu máquina local, usualmente puedes usar "localhost" o "127.0.0.1".
- `database`: el nombre de la base de datos a la que te quieres conectar.
- `user`: el nombre del usuario que tiene acceso a la base de datos.
- `password`: la contraseña del usuario.
- `port`: el número del puerto en el que se encuentra la base de datos. El valor por defecto para PostgreSQL es el puerto 5432.

3.2 Clase 5 y 6 - SQL