SQL

Data Definition Language (DDL)

CertiDevs

Índice de contenidos

1. Tipos de datos en SQL

En SQL, cada **columna** de una **tabla** tiene un **tipo de dato** asociado que define el tipo de información que puede almacenar.

Los tipos de datos en SQL varían según el sistema de gestión de bases de datos (DBMS) que se utilice, como MySQL, PostgreSQL, Oracle o SQL Server.

Sin embargo, existen algunos tipos de datos comunes en todos los DBMS que conforman el estándar SQL. A continuación, se describen los principales tipos de datos en SQL.

1.1. Tipos de datos numéricos

Los tipos de datos numéricos se utilizan para almacenar números enteros y decimales.

- `INTEGER`: número entero. El rango de valores varía según el DBMS, pero en general, el rango es de -2^31 a 2^31-1.
- `SMALLINT`: número entero pequeño. El rango de valores es menor que el de INTEGER, generalmente de -2^15 a 2^15-1.
- `BIGINT`: número entero grande. El rango de valores es mayor que el de INTEGER, generalmente de -2^63 a 2^63-1.
- `DECIMAL(p, s)`: número decimal con precisión fija. "p" es la precisión total de dígitos y "s" es la cantidad de dígitos decimales.
- `NUMERIC(p, s)`: similar a DECIMAL, pero con una precisión y escala exactas.
- `REAL`: número decimal con precisión de coma flotante de precisión simple.
- `DOUBLE PRECISION`: número decimal con precisión de coma flotante de precisión doble.

1.2. Tipos de datos de cadena

Los tipos de datos de cadena se utilizan para almacenar texto.

- `CHAR(n)`: cadena de caracteres de longitud fija. Almacena una cadena de "n" caracteres, donde "n" es un número entero.
- `VARCHAR(n)`: cadena de caracteres de longitud variable. Almacena una cadena de hasta "n" caracteres, donde "n" es un número entero.
- `TEXT`: cadena de caracteres de longitud variable sin límite de tamaño. En algunos DBMS, el tamaño máximo puede ser muy grande (por ejemplo, 4 GB en PostgreSQL).

1.3. Tipos de datos de fecha y hora

Los tipos de datos de fecha y hora se utilizan para almacenar información sobre fechas y horas.

- `DATE`: fecha sin información de hora. Almacena una fecha en formato "YYYY-MM-DD".
- `TIME`: hora sin información de fecha. Almacena una hora en formato "HH:MI:SS".

- `TIMESTAMP`: fecha y hora. Almacena una fecha y hora en formato "YYYY-MM-DD HH:MI:SS".
- `INTERVAL`: intervalo de tiempo. Almacena una cantidad de tiempo en días, horas, minutos y segundos.

1.4. Tipos de datos binarios

Los tipos de datos binarios se utilizan para almacenar datos en formato binario, como imágenes o archivos.

- `BINARY(n)`: datos binarios de longitud fija. Almacena datos binarios de "n" bytes, donde "n" es un número entero.
- `VARBINARY(n)`: datos binarios de longitud variable. Almacena datos binarios de hasta "n" bytes, donde "n" es un número entero.
- `BLOB`: datos binarios de longitud variable sin límite de tamaño.
- `BYTEA` (en PostgreSQL) o BINARY LARGE OBJECT (en otros DBMS): similar a BLOB, se utiliza para almacenar datos binarios de gran tamaño, como imágenes o archivos.

Existen otros tipos de datos en SQL que son específicos de ciertos sistemas de gestión de bases de datos.

- `ENUM`: tipo de dato enumerado que permite almacenar un valor de un conjunto predefinido de valores. Por ejemplo, un tipo ENUM para almacenar estados de ánimo podría tener valores como 'feliz', 'triste' o 'enojado'.
- `UUID`: tipo de dato que almacena identificadores únicos universales (UUID), que son números de 128 bits utilizados para identificar recursos de manera única en un sistema distribuido.
- `JSON`: tipo de dato que almacena información en formato JSON. Algunos sistemas de gestión de bases de datos, como PostgreSQL y MySQL, proporcionan funciones específicas para manipular y consultar datos JSON.
- `ARRAY`: tipo de dato que almacena arreglos unidimensionales de un tipo de dato específico. Este tipo de dato es más común en PostgreSQL y no está presente en todos los sistemas de gestión de bases de datos.

Es importante tener en cuenta que los tipos de datos disponibles y sus características pueden variar según el sistema de gestión de bases de datos que se utilice. Por lo tanto, es fundamental consultar la documentación específica del DBMS para obtener información detallada sobre los tipos de datos y sus características.

2. Creación y modificación de estructuras de datos (DDL)

DDL (Data Definition Language) es un subconjunto de SQL que se utiliza para crear, modificar y eliminar estructuras de datos en una base de datos, como tablas, índices, vistas y esquemas. Los comandos DDL principales incluyen `CREATE`, `ALTER` y `DROP`.

los comandos DDL en SQL permiten crear, modificar y eliminar estructuras de datos en una base de datos relacional.

Estos comandos son fundamentales para diseñar y mantener el esquema de una base de datos y garantizar la integridad y el rendimiento de las consultas y operaciones de datos.

2.1. CREATE TABLE

El comando `CREATE TABLE` se utiliza para crear una nueva tabla en la base de datos. La sintaxis básica es la siguiente:

```
CREATE TABLE table_name (
    column1 data_type constraints,
    column2 data_type constraints,
    ...
);
```

Por ejemplo, para crear una tabla de empleados con los campos id, nombre, apellido, fecha de nacimiento y salario, se utilizaría el siguiente código:

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    birth_date DATE,
    salary DECIMAL(10, 2)
);
```

2.2. ALTER TABLE

El comando `ALTER TABLE` se utiliza para modificar una tabla existente, como agregar o eliminar columnas, cambiar el tipo de datos de una columna o modificar restricciones.

Agregar una columna:

```
ALTER TABLE table_name
ADD COLUMN column_name data_type constraints;
```

Por ejemplo, para agregar una columna email a la tabla empleados:

```
ALTER TABLE employees
ADD COLUMN email VARCHAR(255);
```

Eliminar una columna:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Por ejemplo, para eliminar la columna email de la tabla empleados:

```
ALTER TABLE employees
DROP COLUMN email;
```

Cambiar el tipo de datos de una columna:

```
ALTER TABLE table_name
ALTER COLUMN column_name TYPE new_data_type;
```

Por ejemplo, para cambiar el tipo de datos de la columna salario de DECIMAL a INTEGER en la tabla empleados:

```
ALTER TABLE employees
ALTER COLUMN salary TYPE INTEGER;
```

2.3. DROP TABLE

El comando `DROP TABLE` se utiliza para eliminar una tabla existente y todos sus datos.

```
DROP TABLE table_name;
```

Por ejemplo, para eliminar la tabla empleados:

```
DROP TABLE employees;
```

2.4. CREATE INDEX

El comando `CREATE INDEX` se utiliza para crear un índice en una columna de una tabla para mejorar la velocidad de las consultas.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Por ejemplo, para crear un **índice** en la columna `last_name` de la tabla empleados:

```
CREATE INDEX idx_last_name
```

```
ON employees (last_name);
```

2.5. CREATE VIEW

El comando `CREATE VIEW` se utiliza para crear una vista, que es una representación virtual de una consulta almacenada en la base de datos.

```
CREATE VIEW view_name AS
SELECT ...
FROM ...
WHERE ...;
```

Por ejemplo, para crear una vista que muestre todos los empleados con un salario superior a 50000:

```
CREATE VIEW high_salary_employees AS
SELECT *
FROM employees
WHERE salary > 50000;

-- una vez creada la vista se puede invocar:

select * from high_salary_employees;
```

2.6. CREATE SCHEMA

El comando `CREATE SCHEMA` se utiliza para crear un esquema en la base de datos, que es un contenedor de objetos (como tablas, vistas e índices) que permite organizarlos de manera lógica.

```
CREATE SCHEMA schema_name;
```

Por ejemplo, para crear un **esquema** llamado 'hr' para almacenar objetos relacionados con los recursos humanos:

```
CREATE SCHEMA hr;
```

Después de crear un esquema, puedes crear objetos dentro de él usando la siguiente sintaxis:

```
CREATE TABLE schema_name.table_name (
...
);
```

Por ejemplo, para crear una tabla de empleados dentro del esquema 'hr':

```
CREATE TABLE hr.employees (
    id INTEGER PRIMARY KEY,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    birth_date DATE,
    salary DECIMAL(10, 2)
);
```

2.7. DROP INDEX

Borrar un índice:

```
DROP INDEX index_name;
```

Por ejemplo, para eliminar el índice 'idx_last_name' creado anteriormente en la columna last_name de la tabla empleados:

```
DROP INDEX idx_last_name;
```

2.8. DROP VIEW

El comando `DROP VIEW` se utiliza para eliminar una vista existente.

```
DROP VIEW view_name;
```

Por ejemplo, para eliminar la vista 'high_salary_employees' creada anteriormente:

```
DROP VIEW high_salary_employees;
```

2.9. DROP SCHEMA

El comando `DROP SCHEMA` se utiliza para eliminar un esquema existente y todos sus objetos.

```
DROP SCHEMA schema_name CASCADE;
```

Por ejemplo, para eliminar el esquema 'hr' creado anteriormente junto con todos sus objetos:

```
DROP SCHEMA hr CASCADE;
```

3. Restricciones (Constraints)

3.1. Restricción PRIMARY KEY

Las **claves primarias** son un concepto fundamental en las bases de datos relacionales. Una clave primaria es un campo o conjunto de campos que identifican de manera única cada registro en una tabla.

En MySQL y PostgreSQL, puedes crear claves primarias utilizando la misma sintaxis SQL.

3.1.1. Creación de claves primarias al crear una tabla

Puedes definir una **clave primaria** al crear una tabla utilizando la sintaxis siguiente:

```
CREATE TABLE table_name (
    column1 data_type PRIMARY KEY,
    column2 data_type,
    column3 data_type,
    ...
);
```

Ejemplo: Creación de una tabla employees con una clave primaria en MySQL y PostgreSQL:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INT
);
```

3.1.2. Creación de claves primarias compuestas al crear una tabla

En algunos casos, es posible que necesites utilizar **más de un campo** para identificar de manera única cada registro en una tabla. En ese caso, puedes crear una clave primaria compuesta utilizando la siguiente sintaxis:

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    column3 data_type,
    ...,
    PRIMARY KEY (column1, column2)
);
```

Ejemplo: Creación de una tabla employee_skills con una clave primaria compuesta en MySQL y

PostgreSQL:

```
CREATE TABLE employee_skills (
    employee_id INT,
    skill_id INT,
    proficiency_level INT,
    PRIMARY KEY (employee_id, skill_id)
);
```

3.1.3. Agregar una clave primaria a una tabla existente

Si deseas agregar una clave primaria a una tabla existente, puedes utilizar la sintaxis ALTER TABLE:

```
ALTER TABLE table_name
ADD PRIMARY KEY (column_name);
```

Ejemplo: Agregar una clave primaria a la columna employee_id en la tabla employees existente:

```
ALTER TABLE employees
ADD PRIMARY KEY (employee_id);
```

3.2. Restricción FOREIGN KEY

Las **claves foráneas** son otro concepto fundamental en las bases de datos relacionales. Una clave foránea es un campo o conjunto de campos en una tabla que hace referencia a la clave primaria de otra tabla.

Las **claves foráneas** establecen relaciones entre las tablas y ayudan a mantener la **integridad referencial** de los datos.

3.2.1. Creación de claves foráneas al crear una tabla

Puedes definir una clave foránea al crear una tabla utilizando la siguiente sintaxis:

```
CREATE TABLE table_name (
column1 data_type,
column2 data_type,
column3 data_type,
...,
FOREIGN KEY (column_name) REFERENCES referenced_table (referenced_column)
);
```

Ejemplo: Creación de una tabla orders con una clave foránea en MySQL y PostgreSQL:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
);
```

3.2.2. Creación de claves foráneas compuestas al crear una tabla

Si necesitas utilizar más de un campo para establecer una relación entre dos tablas, puedes crear una clave foránea compuesta utilizando la siguiente sintaxis:

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    column3 data_type,
    ...,
    FOREIGN KEY (column1, column2) REFERENCES referenced_table (referenced_column1,
referenced_column2)
);
```

Ejemplo: Creación de una tabla project_members con una clave foránea compuesta en MySQL y PostgreSQL:

```
CREATE TABLE project_members (
    employee_id INT,
    project_id INT,
    role VARCHAR(50),
    FOREIGN KEY (employee_id, project_id) REFERENCES employee_projects (employee_id,
project_id)
);
```

3.2.3. Agregar una clave foránea a una tabla existente

Si deseas agregar una clave foránea a una tabla existente, puedes utilizar la sintaxis ALTER TABLE:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
FOREIGN KEY (column_name) REFERENCES referenced_table (referenced_column);
```

Ejemplo: Agregar una clave foránea a la columna department_id en la tabla employees existente:

```
ALTER TABLE employees
ADD CONSTRAINT fk_department_id
```

```
FOREIGN KEY (department_id) REFERENCES departments (department_id);
```

3.3. Restricción UNIQUE

`UNIQUE`: La restricción UNIQUE garantiza que todos los valores en una columna sean distintos, es decir, no se repitan.

Esto es útil cuando se desea asegurar que una columna contenga valores únicos sin convertirla en clave primaria. Una tabla puede tener varias restricciones UNIQUE.

Ejemplo de uso de UNIQUE:

```
CREATE TABLE users (
user_id INT PRIMARY KEY,
username VARCHAR(50) UNIQUE, -- Aplicar restricción UNIQUE a la columna 'username'
email VARCHAR(100) UNIQUE -- Aplicar restricción UNIQUE a la columna 'email'
);
```

3.4. Restricción CHECK

`CHECK`: La restricción CHECK permite establecer una condición específica que los valores en una columna deben cumplir.

Si un valor no cumple con la condición establecida, no se permite insertar o actualizar el registro. Esto es útil para garantizar que los datos cumplen ciertos criterios antes de almacenarlos en la base de datos.

Ejemplo de uso de CHECK:

```
CREATE TABLE products (
product_id INT PRIMARY KEY,
name VARCHAR(100) NOT NULL,
price DECIMAL(10, 2) CHECK (price >= 0) -- Aplicar restricción CHECK a la columna
'price'
);
```

3.5. Restricción NOT NULL

`NOT NULL`: La restricción NOT NULL garantiza que una columna no pueda tener valores NULL. Esto asegura que la columna siempre tenga un valor, lo cual es útil cuando se requiere que una columna tenga datos en todo momento.

Ejemplo de uso de NOT NULL:

```
CREATE TABLE employees (
```

```
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL, -- Aplicar restricción NOT NULL a la columna
'first_name'
    last_name VARCHAR(50) NOT NULL -- Aplicar restricción NOT NULL a la columna
'last_name'
);
```

3.6. Restricción DEFAULT

`DEFAULT`: La restricción DEFAULT establece un valor predeterminado para una columna cuando no se especifica un valor durante la inserción de un registro.

Esto es útil cuando se desea proporcionar un **valor por defecto** para una columna en caso de que no se proporcione un valor al insertar un registro.

Ejemplo de uso de DEFAULT:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE DEFAULT CURRENT_DATE, -- Aplicar restricción DEFAULT a la columna
'order_date'
    status VARCHAR(20) DEFAULT 'Pending' -- Aplicar restricción DEFAULT a la columna
'status'
);
```