Introduction to SQL

Contents

Prepared by:
Mr. Rupesh Kumar

Assistant Professor

KRMU

Introduction to SQL

Structured Query Language (SQL) is a standardized programming language used for managing relational databases and performing various operations on the data they store. It is essential for accessing and manipulating the vast stores of data within a relational database management system (RDBMS), which organizes data into tables that can be linked—or related—based on common data attributes.

Databases

A database in SQL is a collection of tables that store a specific set of structured data. Each table in a database resembles a spreadsheet and consists of rows and columns, where rows represent individual records and columns represent the attributes of the data.

Column Types

Columns in SQL tables are defined to store a specific type of data. The primary data types include:

- **`INT`**: An integer number.
- **`VARCHAR`**: A variable-length string.
- **`TEXT`**: A long string of text.
- **`DATE/TIME`**: Date and time values.
- **`FLOAT`**: Floating-point numbers.
- **`BOOLEAN`**: True or false values.

Each column in a table is set up to store a specific type of data, which ensures data integrity and accuracy.

Tables

Tables are the fundamental structure in SQL databases, where data is stored in rows and columns. A table has a unique name and consists of columns and rows. Columns have a defined data type, and rows contain the individual records.

Assume database named **`school`**.

Creating the `students` Table

First, create the **`students`** table with columns for **`id`**, **`name`**, **`grade`**, and **`age`**:

```
CREATE TABLE students (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    grade VARCHAR(2),
    age INT
);
```

Note: This SQL statement creates a new table named **`students`** with four columns: **`id`**, **`name`**, **`grade`**, and **`age`**. The **`id`** column is designated as the primary key, which means it must contain a unique value for each row.

Inserting Data into the `students` Table

Next, we'll insert some data into the table:

```
INSERT INTO students (id, name, grade, age) VALUES (1, 'Alice', 'A', 14);
INSERT INTO students (id, name, grade, age) VALUES (2, 'Bob', 'B', 13);
INSERT INTO students (id, name, grade, age) VALUES (3, 'Charlie', 'A',
15);
```

These commands add three rows to the **`students`** table, each representing a student with their corresponding **`id`**, **`name`**, **`grade`**, and **`age`**.

Within this database, there's a table named **`students`** structured as follows:

- **`id`** (INT)
- **`name`** (VARCHAR)
- **`grade`** (VARCHAR)
- **`age`** (INT)

- The **`students`** table look like this:

Viewing the Table in Tabular Format

In most SQL database management systems (DBMS), when you execute a SELECT query, the output is automatically displayed in a tabular format. For instance, to view all entries in the **`students`** table, you would use:

```
SELECT * FROM students;
```

```
| id | name    | grade | age |
|----|---------|-------|-----|
| 1  | Alice   | A     | 14  |
| 2  | Bob     | B     | 13  |
| 3  | Charlie | A     | 15  |
```

SELECT: Working with SQL in the Terminal

The **`SELECT`** statement is used to query the database and retrieve selected data, allowing you to specify exactly which data you want to fetch from a table.

For example, **`SELECT * FROM students;`** retrieves all the records from the 'users' table.

This command fetches all rows from the **`students`** table.

```
| id | name    | grade | age |
|----|---------|-------|-----|
| 1  | Alice   | A     | 14  |
| 2  | Bob     | B     | 13  |
| 3  | Charlie | A     | 15  |
```

To retrieve only the names and grades of students:

```
SELECT name, grade FROM students;
| name    | grade |
|---------|-------|
| Alice   | A     |
| Bob     | B     |
| Charlie | A     |
```

Functions

SQL includes built-in functions to perform calculations on data, such as **`COUNT()`**, **`SUM()`**, **`AVG()`**, **`MIN()`**, and **`MAX()`**. For instance, **`SELECT AVG(price) FROM products;`** calculates the average price of all products.

COUNT: To count the total number of students:

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

AVG: To calculate the average age of the students:

```
SELECT AVG(age) FROM students;
```

MAX: To find the maximum age among the students:

```
SELECT MAX(age) FROM students;
```

UPDATE

The `UPDATE` statement modifies existing records in a table. For example, `UPDATE students SET grade = 'A' WHERE id = 1;` changes the grade of the student with id 1 to 'A'.

```
UPDATE students SET grade = 'A' WHERE name = 'Bob';
```

The table will now look like:

```
| id | name    | grade | age |

|----|---------|-------|-----|

| 1  | Alice   | A     | 14  |

| 2  | Bob     | A     | 13  |

| 3  | Charlie | A     | 15  |
```

DELETE

The `DELETE` statement removes existing records from a table. For example, `DELETE FROM customers WHERE age < 18;` removes all records of customers younger than 18 years.

To delete the record for any student who is 15 years old:

```
DELETE FROM students WHERE age = 15;

| id | name  | grade | age |

|----|-------|-------|-----|

| 1  | Alice | A     | 14  |

| 2  | Bob   | A     | 13  |
```

Charlie's record is removed from the `students` table because he is 15 years old.

Other Clauses

SQL includes various clauses like `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`, which help refine and sort the data retrieved or manipulated.

WHERE: To select data conditionally, use the `WHERE` clause. For example, to find students with grade 'A':

```
SELECT * FROM students WHERE grade = 'A';

| id | name  | grade | age |

|----|-------|-------|-----|

| 1  | Alice | A     | 14  |

| 2  | Bob   | A     | 13  |
```

ORDER BY: To sort the results, use the `ORDER BY` clause. For example, to order students by age:

```
SELECT * FROM students ORDER BY age;
```

```
| id | name  | grade | age |

|----|-------|-------|-----|

| 2  | Bob   | A     | 13  |

| 1  | Alice | A     | 14  |
```

Joining Tables: JOIN Query

The `JOIN` clause is used to combine rows from two or more tables based on a related column between them. For example, `SELECT * FROM orders JOIN customers ON orders.customer_id = customers.id;` retrieves all orders along with the customer information for each order.

Let's assume there's another table `enrollments` with the following columns:

- `student_id` (INT)

- `course_name` (VARCHAR)

student_id	course_name
1	Mathematics
2	Science

First, we need to insert data into `enrollments`:

```
INSERT INTO enrollments (student_id, course_name) VALUES (1, 'Mathematics');

INSERT INTO enrollments (student_id, course_name) VALUES (2, 'Science');
```

To display student names along with their enrolled courses, we use a JOIN query:

```
SELECT students.name, enrollments.course_name

FROM students

JOIN enrollments ON students.id = enrollments.student_id;
```

```
| name  | course_name |

|-------|-------------|

| Alice | Mathematics |

| Bob   | Science     |
```

This query retrieves a list of students along with the courses they are enrolled in.

Indexing

Indexing is used to speed up the retrieval of records. An index is created on a column in a table, providing a quick way to look up data based on the values of that column.

For example, to create an index on the `age` column:

```
CREATE INDEX idx_age ON students(age);
```

This index helps the database quickly locate data in the `students` table based on the `age` column.

SQL Vulnerabilities

SQL is prone to various vulnerabilities, the most notable being SQL injection, where an attacker can execute malicious SQL statements that control a web application's database server, given that the application is not properly sanitizing the input to SQL statements.

For instance, consider a scenario where user input is directly included in an SQL query:

```
SELECT * FROM students WHERE name = '[user_input]';
```

If the user inputs something malicious like `' OR '1'='1`, it could lead to an SQL injection, where unauthorized data can be accessed or manipulated.

To prevent this, always validate and sanitize inputs, use prepared statements, and employ parameterized queries to ensure that input is treated in a safe manner, not as part of the SQL command.