SQL - Basics

Reading Material

PW SKILLS

Introduction to MySQL

• What is SQL and MySQL?

SQL (Structured Query Language): SQL is a standard programming language specifically designed for managing and manipulating relational databases. It is used to perform various operations such as querying, updating, and managing data. SQL is the backbone of any database-related operation and is integral to interacting with relational database management systems (RDBMS).

Example of SQL:

- **Querying Data:** SELECT * FROM employees WHERE department = 'Sales';
- **Updating Data:** UPDATE employees SET salary = 50000 WHERE id = 101;

MySQL: MySQL is an open-source relational database management system (RDBMS) that uses SQL as its standard language. It is one of the most popular RDBMS, especially for web applications, due to its reliability, speed, and ease of use. MySQL is used to store, retrieve, and manage data for various applications, from simple websites to complex data warehouses.

Example of MySQL in Use:

- A website that tracks user accounts, posts, comments, and likes might use MySQL to manage its database. All interactions with the data, like adding a new post or retrieving user comments, would involve SQL commands executed against a MySQL database.

• RDBMS concept

An RDBMS is a software system that manages databases based on the relational model introduced by E. F. Codd. In an RDBMS, data is stored in tables, which are structured into rows and columns. Each table in the database is related to other tables through keys, making it easy to retrieve and manipulate data.

Key Concepts of RDBMS:

1. **Tables:** Organized into rows and columns, where each row represents a record and each column represents a field.
2. **Primary Key:** A unique identifier for each record in a table.
3. **Foreign Key:** A field in one table that uniquely identifies a row of another table, creating a relationship between the two tables.
4. **Normalization:** The process of organizing data to reduce redundancy and improve data integrity.

Example:
Consider a students table with fields student_id (Primary Key), name, and age, and a courses table with course_id (Primary Key) and course_name. A third table, enrollments, might use student_id and course_id as foreign keys to track which students are enrolled in which courses.

• MySQL architecture

MySQL's architecture is designed to provide high performance, flexibility, and ease of use. The architecture includes several layers, each responsible for different aspects of database management.

- ## Components of MySQL Architecture:

1. **Client Layer:** The top layer where clients (applications or users) interact with the database using SQL queries.
2. **SQL Layer:** This layer processes SQL commands, handling tasks like parsing, analysis, and optimization of queries.
3. **Storage Engine Layer:** The layer responsible for storage and retrieval of data. MySQL supports various storage engines like InnoDB and MyISAM, each optimized for different types of workloads.
4. **File System Layer:** The lowest layer, which interacts with the file system to store data on disk.

Example of MySQL Architecture in Action:
When a user executes a query like
SELECT * FROM orders WHERE order_date > '2023-01-01';,
the request is processed by the client layer, passed through the SQL layer where it's optimized and executed, and finally, the storage engine retrieves the relevant data from the database files.

- ## Creating and using a database

1. **Creating a Database:** A database in MySQL is a collection of tables and other database objects. Creating a database is one of the first steps when setting up a new application or data storage system.

SQL Command to Create a Database:
CREATE DATABASE mydatabase;
This command creates a new database named mydatabase.

2. **Using a Database:** After creating a database, you can select it for use by running:
USE mydatabase;
This command sets the current database context to mydatabase, meaning any subsequent SQL commands will apply to this database.

Example:
If you're building a blog platform, you might create a database named blog to store tables like posts, comments, and users. After creating the database, you would use the USE blog; command to start working with it.

- ## Connecting to MySQL using different tools

Connecting to MySQL: There are various tools and methods to connect to a MySQL database, each suitable for different use cases, from command-line tools to graphical user interfaces (GUIs).

Common Tools for Connecting to MySQL:
1. **MySQL Command-Line Client:** A terminal-based interface to execute SQL commands directly.
   - Example Command: mysql -u root -p to log in as the root user.

2. **MySQL Workbench:** A GUI tool that allows for database design, SQL development, and server administration.
   - Example Use: Visualize and manage your database schema, run queries, and perform server maintenance tasks.

3. **PHPMyAdmin:** A web-based tool for managing MySQL databases, often used in web hosting environments.
   - Example Use: Manage databases, users, and permissions through a web interface.

Example:
A developer might use MySQL Workbench to design a database schema visually, create tables, and write SQL queries, while a server administrator might use the command-line client to perform backups and monitor database performance.

SQL Commands

- **Data Definition Language (DDL): CREATE, ALTER, DROP, TRUNCATE**

Data Definition Language (DDL) commands are used to define, modify, and manage the structure of database objects such as tables, indexes, and schemas. DDL commands are essential for setting up the initial database structure and for making changes to that structure over time.

1.1. CREATE

The CREATE command is used to create new database objects such as tables, indexes, views, and databases. This command is fundamental when setting up a new database, as it allows you to define the structure that will store your data.

Syntax:

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

Example:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

In this example, a table named employees is created with four columns: id, name, department, and salary. The id column is defined as an integer and is the primary key, meaning it uniquely identifies each record in the table. The name column is a variable-length string (VARCHAR) with a maximum of 100 characters and is required (NOT NULL). The department column is also a VARCHAR, while the salary column is a decimal number with up to 10 digits, including two decimal places.

1.2. ALTER

The ALTER command is used to modify the structure of an existing database object. This includes adding, deleting, or modifying columns in a table, or altering other object properties like indexes or constraints.

Syntax:

```sql
ALTER TABLE table_name
ADD column_name datatype;
```

or

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

Example:

```sql
ALTER TABLE employees ADD COLUMN hire_date DATE;
```

This command adds a new column hire_date of type DATE to the existing employees table. The ALTER command is versatile and allows you to make changes to a table's structure without having to recreate the table entirely.

1.3. DROP

The DROP command is used to delete existing database objects, such as tables, indexes, or databases. When an object is dropped, it is permanently removed from the database, and all the data within it is lost.

Syntax:

```sql
DROP TABLE table_name;
```

Example:

```sql
DROP TABLE employees;
```

This command deletes the employees table from the database. After executing this command, the table and all its data are permanently removed, making it an irreversible action.

1.4. TRUNCATE

The TRUNCATE command is used to remove all rows from a table, effectively emptying the table. Unlike the DELETE command, TRUNCATE does not log individual row deletions and is faster, but it does not allow for a WHERE clause.

Syntax:

```sql
TRUNCATE TABLE table_name;
```

Example:

```sql
TRUNCATE TABLE employees;
```

This command removes all records from the employees table, but the table structure remains intact. It's a quick way to clear all data from a table without dropping it.

- **Data Manipulation Language (DML): INSERT, UPDATE, DELETE**

Data Manipulation Language (DML) commands are used to manage data within database tables. DML commands allow you to insert, update, delete, and retrieve data. These commands are essential for interacting with the data stored in the database.

2.1. INSERT

The INSERT command is used to add new records (rows) to a table. This command allows you to populate tables with data.

Syntax:

```sql
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```sql
INSERT INTO employees (id, name, department, salary)
VALUES (101, 'John Doe', 'Sales', 50000);
```

This command inserts a new row into the employees table with the specified values for id, name, department, and salary. The values correspond to the columns listed in the INSERT statement.

2.2. UPDATE

The UPDATE command is used to modify existing records in a table. This command allows you to change the data stored in one or more rows based on specific conditions.

Syntax:

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

```sql
UPDATE employees
SET salary = 55000
WHERE id = 101;
```

This command updates the salary of the employee with id = 101 to 55,000. The WHERE clause ensures that only the specified record is updated.

2.3. DELETE

The DELETE command is used to remove one or more records from a table. This command allows you to delete specific rows based on a condition.

Syntax:

```sql
DELETE FROM table_name
WHERE condition;
```

Example:

```sql
DELETE FROM employees
WHERE id = 101;
```

This command deletes the row from the employees table where the id is 101. The WHERE clause ensures that only the specified row is deleted. If the WHERE clause is omitted, all rows in the table will be deleted.

• Data Query Language (DQL): SELECT

Data Query Language (DQL) is primarily concerned with retrieving data from the database. The primary command in DQL is SELECT, which is used to query data from one or more tables.

3.1. SELECT

The SELECT command is used to retrieve data from a database. It is one of the most commonly used SQL commands and allows you to specify the columns you want to view, apply filters, sort the results, and more.

Syntax:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```sql
SELECT name, department
FROM employees
WHERE salary > 50000;
```

This command retrieves the name and department columns from the employees table where the salary is greater than 50,000. The SELECT statement can be as simple or complex as needed, including joins, subqueries, and aggregations.

Advanced Example:

```sql
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;
```

This command counts the number of employees in each department and groups the results by department. The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows.

- **Transaction Control Language (TCL) :COMMIT, ROLLBACK**

Transaction Control Language (TCL) commands are used to manage transactions in a database. Transactions are sequences of one or more SQL operations treated as a single unit of work, which must be either fully completed or fully rolled back.

4.1. COMMIT

The COMMIT command is used to save all the changes made in the current transaction permanently to the database. Once a transaction is committed, the changes cannot be rolled back.

Syntax:

```sql
COMMIT;
```

Example:

```sql
BEGIN TRANSACTION;
UPDATE employees SET salary = 60000 WHERE id = 102;
COMMIT;
```

In this example, the COMMIT command is used after an UPDATE operation to save the changes made to the salary of the employee with id = 102. Without the COMMIT, the changes would not be permanently saved.

4.2. ROLLBACK

The ROLLBACK command is used to undo changes made in the current transaction. It reverts the database to the state it was in before the transaction began.

Syntax:

```sql
ROLLBACK;
```

Example:

```sql
BEGIN TRANSACTION;
UPDATE employees SET salary = 60000 WHERE id = 102;
ROLLBACK;
```

In this example, the ROLLBACK command is used to undo the changes made by the UPDATE operation. The database will revert to its previous state, and the salary of the employee with id = 102 will remain unchanged.

• Data Control Language (DCL): GRANT, REVOKE

Data Control Language (DCL) commands are used to control access to data in a database. DCL commands manage permissions and security by allowing or restricting user access to database objects.

5.1. GRANT

The GRANT command is used to give users specific privileges on database objects. These privileges can include the ability to SELECT, INSERT, UPDATE, DELETE, or perform other operations on database objects.

Syntax:

```sql
GRANT privilege ON object TO user;
```

Example:

```sql
GRANT SELECT, INSERT ON employees TO user1;
```

This command grants the user user1 the ability to SELECT (view) and INSERT (add) records in the employees table. The GRANT command is essential for managing who can perform specific actions within the database.

5.2. REVOKE

The REVOKE command is used to remove previously granted privileges from a user. It is the opposite of the GRANT command and is used to restrict access to database objects.

Syntax:

```sql
REVOKE privilege ON object FROM user;
```

Example:

```sql
REVOKE SELECT, INSERT ON employees FROM user1;
```

This command removes the SELECT and INSERT privileges from user1 on the employees table. After executing this command, user1 will no longer be able to view or add records to the employees table.

SQL Data Types & Operators

In SQL, data types and operators are foundational concepts that define the nature of the data you can store in a database and how you can manipulate it. Understanding data types is crucial for structuring tables, while operators allow you to perform calculations, comparisons, and logical operations on the data. Let's dive into each category of data types and operators with detailed explanations and examples.

Numeric data types (INT, FLOAT, DECIMAL)

Numeric data types are used to store numbers, which can be integers or real numbers with fractional parts. They are essential for storing any kind of numerical information, such as quantities, prices, or IDs.

a. INT (Integer):

The INT data type is used to store whole numbers (integers) without any decimal points. It is commonly used for counting items, such as the number of products in stock or the ID of a user.

Syntax:

```sql
column_name INT;
```

Example:

```sql
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    quantity_in_stock INT
);
```

In this example, the product_id and quantity_in_stock columns are defined as INT, meaning they will store integer values. INT can store both positive and negative whole numbers, with a typical range from -2,147,483,648 to 2,147,483,647.

b. FLOAT (Floating-Point Number):
The FLOAT data type is used to store real numbers with decimal points. It is useful for representing values that require precision, such as measurements or currency.

Syntax:

```sql
column_name FLOAT;
```

Example:

```sql
CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    amount FLOAT
);
```

In this example, the amount column is defined as FLOAT, which allows it to store numbers with decimal points. For instance, if a sale amount is $123.45, it can be accurately stored using the FLOAT data type.

c. DECIMAL (Fixed-Point Number):
The DECIMAL data type is similar to FLOAT but is used to store numbers with a fixed number of decimal places. It is often used for financial data where exact precision is crucial, such as prices or interest rates.

Syntax:

```sql
column_name DECIMAL(precision, scale);
```

Example:

```sql
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    balance DECIMAL(10, 2)
);
```

In this example, the balance column is defined as DECIMAL(10, 2), which means it can store numbers with up to 10 digits in total, including 2 digits after the decimal point (e.g., 12345.67). The DECIMAL data type ensures that operations involving money are handled accurately.

- **Character data types (CHAR, VARCHAR, TEXT)**

Character data types are used to store text or string data, such as names, addresses, and other alphanumeric information. These data types are essential for handling any form of textual data.

a. CHAR (Fixed-Length String):
The CHAR data type is used to store fixed-length strings. If the data entered is shorter than the specified length, it will be padded with spaces. This data type is useful when storing strings of a consistent length, such as country codes.

Syntax:

```sql
column_name CHAR(length);
```

Example:

```sql
CREATE TABLE countries (
    country_code CHAR(3) PRIMARY KEY,
    country_name VARCHAR(50)
);
```

In this example, the country_code column is defined as CHAR(3), meaning it will store strings that are exactly 3 characters long. If a value shorter than 3 characters is inserted, it will be padded with spaces.

b. VARCHAR (Variable-Length String):
The VARCHAR data type is used to store variable-length strings. Unlike CHAR, VARCHAR only uses as much space as needed for the actual data, making it more efficient for storing text that can vary in length.

Syntax:

```sql
column_name VARCHAR(length);
```

Example:

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    email VARCHAR(255)
);
```

In this example, the customer_name and email columns are defined as VARCHAR(100) and VARCHAR(255), respectively. These columns can store strings up to 100 and 255 characters long, but they will only use as much space as the length of the actual data.

c. TEXT (Large Text String):
The TEXT data type is used to store large amounts of text. It is ideal for storing lengthy textual data such as descriptions, comments, or notes.

Syntax:

```sql
column_name TEXT;
```

Example:

```sql
CREATE TABLE blog_posts (
    post_id INT PRIMARY KEY,
    title VARCHAR(255),
    content TEXT
);
```

In this example, the content column is defined as TEXT, allowing it to store a large block of text, such as a full blog post. The TEXT data type is ideal for columns where the amount of text can vary significantly and may be quite large.

- **Date and time data types (DATE, TIME, DATETIME)**

Date and time data types are used to store date, time, or both date and time values. These data types are crucial for managing information related to time, such as timestamps, birthdays, or deadlines.

a. DATE (Date Only):
Definition: The DATE data type is used to store calendar dates without any time component. The format is typically YYYY-MM-DD.

In this example, the date_of_birth column is defined as DATE, meaning it will store the birth date of employees in the YYYY-MM-DD format (e.g., 1990-05-20).

b. TIME (Time Only):
Definition: The TIME data type is used to store time values without any date component. The format is typically HH:MM:SS.

Syntax:

```sql
column_name TIME;
```

Example:

```sql
CREATE TABLE shifts (
    shift_id INT PRIMARY KEY,
    start_time TIME,
    end_time TIME
);
```

In this example, the start_time and end_time columns are defined as TIME, meaning they will store the start and end times of shifts in the HH:MM:SS format (e.g., 09:30:00).

c. DATETIME (Date and Time):
Definition: The DATETIME data type is used to store both date and time in a single field. The format is typically YYYY-MM-DD HH:MM:SS.

Syntax:

```sql
column_name DATETIME;
```

Example:

```sql
CREATE TABLE appointments (
    appointment_id INT PRIMARY KEY,
    appointment_time DATETIME,
    description TEXT
);
```

In this example, the appointment_time column is defined as DATETIME, meaning it will store both the date and time of an appointment (e.g., 2024-08-12 14:30:00). This data type is useful for recording events that include both date and time.

- **Arithmetic operators (+, -, *, /)**

Arithmetic operators are used to perform mathematical operations on numeric data. They are commonly used in calculations involving numeric columns.

a. Addition (+):
Definition: The + operator is used to add two numbers.

Example:

SELECT price, quantity, (price * quantity) AS total_cost

FROM orders;

In this example, the - operator is used to calculate the final_price by subtracting the discount from the original_price for each sale.

c. Multiplication (*):
Definition: The * operator is used to multiply two numbers.

Example:

SELECT product_id, price, quantity, (price * quantity) AS total_cost
FROM order_items;

In this example, the * operator is used to calculate the total_cost by multiplying the price and quantity columns for each item in an order.

d. Division (/):
Definition: The / operator is used to divide one number by another.

Example:
SELECT total_amount, number_of_items, (total_amount / number_of_items) AS average_price FROM invoices;
In this example, the / operator is used to calculate the average_price by dividing the total_amount by the number_of_items on each invoice.

- **Comparison operators (=, <>, <, >, <=, >=)**

Comparison operators are used to compare two values and return a result based on whether the comparison is true or false. These operators are essential for filtering data in queries.

a. Equal to (=):
Definition: The = operator is used to check if two values are equal.

Example:

```
SELECT * FROM employees
WHERE department_id = 3;
```

In this example, the = operator is used to filter the employees who belong to department 3.

b. Not equal to (<> or !=):
Definition: The <> or != operator is used to check if two values are not equal.

Example:

```
SELECT *
FROM employees
WHERE department_id <> 3;
```

In this example, the <> operator is used to filter the employees who do not belong to department 3.

c. Greater than (>):
Definition: The > operator is used to check if one value is greater than another.

Example:

```
SELECT *
FROM products
WHERE price > 100;
```

In this example, the > operator is used to filter the products that have a price greater than $100.

d. Less than (<):
Definition: The < operator is used to check if one value is less than another.

Example:

```
SELECT *
FROM products
WHERE price < 100;
```

In this example, the < operator is used to filter the products that have a price less than $100.

e. Greater than or equal to (>=):
Definition: The >= operator is used to check if one value is greater than or equal to another.

Example:

```
SELECT *
FROM orders
WHERE order_date >= '2024-01-01';
```

In this example, the >= operator is used to filter the orders that were placed on or after January 1, 2024.

f. Less than or equal to (<=):
Definition: The <= operator is used to check if one value is less than or equal to another.

Example:

```
SELECT *
FROM orders
WHERE order_date <= '2024-01-01';
```

In this example, the <= operator is used to filter the orders that were placed on or before January 1, 2024.

- **Logical operators (AND, OR, NOT)**

Logical operators are used to combine multiple conditions in a query, allowing you to filter data based on more complex criteria.

a. AND:
Definition: The AND operator is used to combine multiple conditions in a query, where all conditions must be true for a record to be included in the result set.

Example:

```
SELECT *
FROM employees
WHERE department_id = 3 AND salary > 50000;
```

In this example, the AND operator is used to filter employees who belong to department 3 and have a salary greater than $50,000. Both conditions must be true for an employee to be included in the result.

b. OR:
Definition: The OR operator is used to combine multiple conditions in a query, where at least one condition must be true for a record to be included in the result set.

Example:

```
SELECT *
FROM employees
WHERE department_id = 3 OR salary > 50000;
```

In this example, the OR operator is used to filter employees who either belong to department 3 or have a salary greater than $50,000. If either condition is true, the employee will be included in the result.

c. NOT:
Definition: The NOT operator is used to negate a condition, returning records where the condition is not true.

Example:

```
SELECT *
FROM employees
WHERE NOT department_id = 3;
```

In this example, the NOT operator is used to filter employees who do not belong to department 3. This effectively excludes all employees from department 3 from the result set.

Constraints

Constraints in SQL are rules applied to columns or tables to enforce data integrity and ensure the accuracy and reliability of the data within a database. They help maintain the correctness and consistency of data by restricting the types of data that can be inserted or actions that can be performed. Constraints are essential for designing robust databases, as they prevent invalid data from being entered and ensure that relationships between tables are maintained correctly. Let's explore each type of constraint in detail, with explanations and examples.

- **Primary Key and Unique Key constraints**

1.1. Primary Key Constraint
Definition: The PRIMARY KEY constraint is used to uniquely identify each record in a table. It ensures that no duplicate values exist in the column(s) defined as the primary key and that each record has a unique identifier. A primary key also implicitly applies a NOT NULL constraint, meaning that the column cannot have NULL values.

Characteristics:
1. Only one primary key can be defined per table.
2. A primary key can consist of a single column or a combination of multiple columns (composite key).
3. It uniquely identifies each row in the table.

Syntax:

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

Example:

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

In this example, the employee_id column is defined as the PRIMARY KEY, meaning it will uniquely identify each employee in the employees table. No two employees can have the same employee_id, and every employee_id must contain a value (i.e., it cannot be NULL).

1.2. Unique Key Constraint
Definition: The UNIQUE constraint ensures that all values in a column or a set of columns are unique across the entire table. Unlike the primary key, a table can have multiple unique keys, and unique columns can accept NULL values (except for SQL Server, where multiple NULLs are not allowed).

Characteristics:
1. Enforces uniqueness of data in the column(s).
2. Multiple unique constraints can be applied to a table.
3. Allows NULL values (except in some SQL implementations like SQL Server).

Syntax:

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

Example:

```sql
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(100) UNIQUE
);
```

In this example, both the username and email columns are defined with the UNIQUE constraint. This ensures that each username and email address is unique across all users in the users table. No two users can have the same username or email address, preventing duplicate records.

- **Foreign Key constraint**

The FOREIGN KEY constraint is used to establish and enforce a link between two tables. A foreign key in one table points to the primary key (or a unique key) in another table, creating a parent-child relationship between the two tables. This constraint ensures referential integrity, meaning that a record in the child table cannot exist without a corresponding record in the parent table.

Characteristics:

1. Establishes a relationship between two tables.
2. Ensures that the value in the foreign key column matches a value in the primary key or unique key column of the referenced table.
3. Prevents actions that would violate the referential integrity (e.g., deleting a parent record with existing child records).

Syntax:

```sql
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    FOREIGN KEY (column_name) REFERENCES parent_table(column_name)
);
```

Example:

```sql
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

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

In this example, the department_id column in the employees table is defined as a FOREIGN KEY that references the department_id column in the departments table. This ensures that every employee is assigned to a valid department, and prevents an employee from being assigned to a non-existent department.

- **Not Null constraint**

The NOT NULL constraint ensures that a column cannot have a NULL value. This constraint is crucial when a column must always have a valid value for every record. It is commonly used for columns that are essential to the business logic, such as names, dates, or identifiers.

Characteristics:

1. Ensures that a column must always contain a value.
2. Prevents the insertion of NULL values into the column.
3. Often used in combination with other constraints like PRIMARY KEY or UNIQUE.

Syntax:

```sql
CREATE TABLE table_name (
    column1 data_type NOT NULL,
    column2 data_type,
    ...
);
```

Example:

```sql
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE NOT NULL,
    customer_id INT NOT NULL
);
```

In this example, the order_date and customer_id columns are defined with the NOT NULL constraint. This ensures that every order must have a valid date and be associated with a customer. No order can be placed without this essential information.

- **Check constraint**

The CHECK constraint is used to limit the values that can be inserted into a column based on a specific condition. It allows you to define a rule that the data in the column must meet, such as a range of values or a specific pattern. This constraint is useful for enforcing business rules at the database level.

Characteristics:

1. Validates data against a condition before it is inserted or updated.
2. Ensures that the data meets specific criteria defined by the condition.
3. Can be applied to one or more columns.

Syntax:

```sql
CREATE TABLE table_name (
    column1 data_type CHECK (condition),
    column2 data_type,
    ...
);
```

Example:

```sql
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) CHECK (price > 0),
    stock_quantity INT CHECK (stock_quantity >= 0)
);
```

In this example, the price column has a CHECK constraint that ensures the price of a product must be greater than 0. Similarly, the stock_quantity column has a CHECK constraint that ensures the quantity in stock cannot be negative. These constraints help enforce business rules, ensuring that the data remains valid and logical.

- **Default constraint**

Definition: The DEFAULT constraint is used to assign a default value to a column when no value is provided during an insert operation. This ensures that a column always has a valid value, even if the user does not supply one. The default value can be a static value or an expression.

Characteristics:

1. Provides a default value for a column when no value is explicitly provided.
2. Ensures that a column is never left empty unless explicitly set to NULL (if allowed).
3. Useful for columns where a common value is frequently used, such as status flags or timestamps.

Syntax:

```sql
CREATE TABLE table_name (
    column1 data_type DEFAULT default_value,
    column2 data_type,
    ...
);
```

Example:

```sql
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Pending'
);
```

In this example, the order_date column is defined with a DEFAULT constraint that assigns the current date as the default value. The status column is given a default value of 'Pending'. If a new order is inserted without specifying these values, the database automatically fills them in with the default values, ensuring that the order has a date and a status.

Joins & Set Operators

Joins and set operators are fundamental SQL concepts used to combine and manipulate data from multiple tables or queries. They allow you to retrieve, compare, and integrate data across different sources within a relational database, facilitating complex data analysis and reporting. Understanding these concepts is essential for working effectively with relational databases. Below is a detailed explanation of each type of join and set operator, with examples and step-by-step explanations.

- **Types of joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN)**

Joins are used in SQL to combine rows from two or more tables based on a related column between them. By using joins, you can retrieve data from multiple tables as if the data existed in a single table.

1.1. INNER JOIN
Definition: An INNER JOIN returns only the rows where there is a match between the columns in both tables being joined. If no match is found, those rows are excluded from the result set.

Characteristics:
1. Only returns matching rows from both tables.
2. Non-matching rows are excluded from the result.

Syntax:

```sql
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example:

```sql
SELECT employees.employee_id, employees.first_name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

In this example, the INNER JOIN is used to combine the employees and departments tables. The join condition is based on the department_id column, which is common to both tables. The result includes only those employees who have a matching department in the departments table, displaying the employee's ID, first name, and department name.

1.2. LEFT JOIN (or LEFT OUTER JOIN)

Definition: A LEFT JOIN returns all the rows from the left table and the matching rows from the right table. If there is no match, the result will include NULL values for columns from the right table.

Characteristics:

1. Returns all rows from the left table.
2. Returns matching rows from the right table.
3. Non-matching rows from the right table will appear as NULL in the result.

Syntax:

```sql
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example:

```sql
SELECT employees.employee_id, employees.first_name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

In this example, the LEFT JOIN returns all employees, regardless of whether they have a matching department in the departments table. If an employee does not belong to any department, the department_name column will display NULL.

1.3. RIGHT JOIN (or RIGHT OUTER JOIN)

Definition: A RIGHT JOIN is the opposite of a LEFT JOIN. It returns all rows from the right table and the matching rows from the left table. If there is no match, the result will include NULL values for columns from the left table.

Characteristics:

1. Returns all rows from the right table.
2. Returns matching rows from the left table.
3. Non-matching rows from the left table will appear as NULL in the result.

Syntax:

```sql
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Example:

```sql
SELECT employees.employee_id, employees.first_name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

In this example, the RIGHT JOIN returns all departments, including those that do not have any employees assigned. If no employee belongs to a department, the employee_id and first_name columns will display NULL.

1.4. FULL OUTER JOIN

Definition: A FULL OUTER JOIN returns all rows when there is a match in either table. It combines the results of both LEFT JOIN and RIGHT JOIN. If there is no match, the result will include NULL values for columns from both tables.

Characteristics:

1. Returns all rows from both tables.
2. Non-matching rows from both tables will appear as NULL in the result.

Syntax:

```sql
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

Example:

```sql
SELECT employees.employee_id, employees.first_name, departments.department_name
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.department_id;
```

In this example, the FULL OUTER JOIN returns all employees and all departments, even if there are no matches between them. If an employee does not belong to any department, the department_name will display NULL, and if a department has no employees, the employee_id and first_name will display NULL.

- **Set operators (UNION, UNION ALL, INTERSECT, EXCEPT)**

Set operators are used to combine the results of two or more SELECT queries. The rows from each query are combined into a single result set based on the specified set operation. Set operators can help in comparing and combining results from different queries, making them powerful tools for data analysis.

2.1. UNION

Definition: The UNION operator is used to combine the result sets of two or more SELECT queries. It removes duplicate rows from the result set, ensuring that only unique rows are returned.

Characteristics:

1. Combines the results of multiple SELECT queries
2. Removes duplicate rows from the combined result.
3. The number and order of columns in each SELECT query must match.

Syntax:

```sql
SELECT column_names FROM table1
UNION
SELECT column_names FROM table2;
```

Example:

```sql
SELECT city FROM customers
UNION
SELECT city FROM suppliers;
```

In this example, the UNION operator combines the cities listed in both the customers and suppliers tables. The result will include only unique city names, removing any duplicates.

2.2. UNION ALL

Definition: The UNION ALL operator is similar to the UNION operator, but it does not remove duplicate rows. It returns all rows from the combined result set, including duplicates.

Characteristics:

1. Combines the results of multiple SELECT queries.
2. Does not remove duplicate rows.
3. The number and order of columns in each SELECT query must match.

Syntax:

```sql
SELECT column_names FROM table1
UNION ALL
SELECT column_names FROM table2;
```

Example:

```sql
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers;
```

In this example, the UNION ALL operator combines the cities from both the customers and suppliers tables without removing duplicates. If a city appears in both tables, it will appear twice in the result set.

2.3. INTERSECT

Definition: The INTERSECT operator returns only the rows that are common to the result sets of both SELECT queries. It effectively filters out rows that do not appear in both queries.

Characteristics:

1. Returns only the common rows from the result sets of two SELECT queries.
2. The number and order of columns in each SELECT query must match.

Syntax:

```sql
SELECT column_names FROM table1
INTERSECT
SELECT column_names FROM table2;
```

Example:

```sql
SELECT city FROM customers
INTERSECT
SELECT city FROM suppliers;
```

In this example, the INTERSECT operator returns only the cities that are present in both the customers and suppliers tables. The result includes only the cities that both customers and suppliers share.

2.4. EXCEPT (or MINUS in some SQL dialects)

Definition: The EXCEPT (or MINUS) operator returns the rows that are in the result set of the first SELECT query but not in the result set of the second SELECT query. It effectively subtracts the results of the second query from the first.

Characteristics:

1. Returns rows from the first query that are not present in the second query.
2. The number and order of columns in each SELECT query must match.

Syntax:

```sql
SELECT column_names FROM table1
EXCEPT
SELECT column_names FROM table2;
```

Example:

```sql
SELECT city FROM customers
EXCEPT
SELECT city FROM suppliers;
```

In this example, the EXCEPT operator returns the cities that are present in the customers table but not in the suppliers table. The result excludes any cities that appear in both tables.

Subqueries

Subqueries, also known as nested queries or inner queries, are queries within another SQL query. They are powerful tools used to perform complex operations by breaking them down into simpler, more manageable steps. Subqueries allow you to retrieve data that will be used in the main query as a condition to further refine the result set. They are essential for sophisticated data retrieval and manipulation in SQL.

- ## Introduction to subqueries

A subquery is a query embedded within the WHERE, FROM, SELECT, or HAVING clause of another SQL query. The subquery provides a set of results that the main (outer) query uses to execute its logic. Subqueries can be used in various ways, such as filtering data, calculating values, or even returning a complete dataset for further querying.

Characteristics:

1. **Nested Structure:** Subqueries are embedded within another query.
2. **Isolation:** Subqueries execute independently before their results are passed to the outer query.
3. **Flexibility:** Subqueries can return a single value, a list of values, or a complete table.

Example:

SELECT employee_id, first_name, last_name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE department_name = 'Sales');

In this example, the subquery (SELECT department_id FROM departments WHERE department_name = 'Sales') retrieves the department_id of the 'Sales' department. The main query then selects all employees who belong to that department.

- **Single-row subqueries**

A single-row subquery returns only one row and one column. These subqueries are typically used with comparison operators like =, >, <, >=, and <=.

Characteristics:

1. Returns a single value.
2. Used with comparison operators in the main query.
3. If the subquery returns more than one row, it will result in an error.

Syntax:

```sql
SELECT column1, column2
FROM table1
WHERE column1 = (SELECT single_column FROM table2 WHERE condition);
```

Example:

```sql
SELECT first_name, last_name, salary
FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees WHERE department_id = 10);
```

In this example, the subquery (SELECT MAX(salary) FROM employees WHERE department_id = 10) retrieves the highest salary in department 10. The main query then selects the first name, last name, and salary of the employee(s) earning that salary.

- **Multiple-row subqueries**

A multiple-row subquery returns more than one row of results. These subqueries are typically used with operators like IN, ANY, ALL, and EXISTS to compare the main query's results against the list of values returned by the subquery.

Characteristics:

1. Returns multiple rows.
2. Used with set operators like IN, ANY, ALL, or EXISTS.
3. Allows the main query to match against multiple values.

Syntax:

```sql
SELECT column1, column2
FROM table1
WHERE column1 IN (SELECT column1 FROM table2 WHERE condition);
```

Example:

```sql
SELECT employee_id, first_name, last_name
FROM employees
WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 1700);
```

In this example, the subquery (SELECT department_id FROM departments WHERE location_id = 1700) returns all department_ids located in the location with location_id = 1700. The main query then selects all employees who work in those departments.

- **Correlated subqueries**

A correlated subquery is a subquery that references columns from the outer query. Unlike other subqueries, a correlated subquery is executed repeatedly, once for each row processed by the outer query. This makes it more complex and computationally expensive but also very powerful for certain types of data retrieval.

Characteristics:

1. Depends on the outer query for its values.
2. Executed once for each row in the outer query.
3. Used for row-by-row comparisons.

Syntax:

```
SELECT column1, column2
FROM table1 alias1
WHERE column1 = (SELECT column1 FROM table2 alias2 WHERE alias1.column = alias2.column AND condition);
```

Example:

```
SELECT employee_id, first_name, salary
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.department_id = e2.department_id);
```

In this example, the subquery (SELECT AVG(salary) FROM employees e2 WHERE e1.department_id = e2.department_id) calculates the average salary for each department. The main query then selects employees who earn more than the average salary in their respective departments. Here, the subquery is correlated because it references e1.department_id from the outer query.