MySQL Basics & Commands

MySQL is a **relational database management system (RDBMS)** used to store, manage, and retrieve structured data efficiently. You interact with MySQL using SQL (Structured Query Language).

1️⃣ Logging into MySQL from Command Prompt (cmd)

Step 1: Open Command Prompt

- **Windows:** Press `Win + R`, type `cmd`, and hit `Enter`.
- **Linux/macOS:** Open the **Terminal**.

Step 2: Connect to MySQL

```
mysql -u root -p
```

- `-u root` → Specifies the **username** (`root` is the default admin user).
- `-p` → Prompts for a **password** (Enter the MySQL password when asked).

✅ **Successful Login Output:**

```
Welcome to the MySQL monitor. Commands end with ; or \g.
mysql>
```

Now, you can execute MySQL commands.

2️⃣ MySQL Commands for Databases

1. Show all existing databases

```
SHOW DATABASES;
```

2. Create a new database

```
CREATE DATABASE mydatabase;
```

3. Use a specific database

Before working with tables, you need to **select** the database:

```
USE mydatabase;
```

✅ **Output:**

```
Database changed
```

Now, all commands will apply to `mydatabase`.

4. Drop (Delete) a Database

```
DROP DATABASE mydatabase;
```

- This **permanently deletes** the database and all its tables.
- **Be careful!** This action **cannot be undone**.

✅ **Output:**

```
Query OK, 1 row affected (0.01 sec)
```

4️⃣ Exiting MySQL

To exit MySQL and return to the command prompt, type:

```
EXIT;

QUIT;
```

Summary of Commands

Command	Description

`mysql -u root -p`	Log in to MySQL
`SHOW DATABASES;`	List all databases
`CREATE DATABASE dbname;`	Create a new database
`USE dbname;`	Switch to a database
`DROP DATABASE dbname;`	Delete a database
`EXIT;`	Exit MySQL

Let me know if you need more details! 🚀

List of All Data Types in MySQL

MySQL has three main categories of data types:

1. **Numeric Data Types**
2. **Date and Time Data Types**
3. **String (Character) Data Types**

1. Numeric Data Types

Data Type	Description
`TINYINT`	1-byte integer (-128 to 127)
`SMALLINT`	2-byte integer (-32,768 to 32,767)
`MEDIUMINT`	3-byte integer (-8,388,608 to 8,388,607)
`INT` / `INTEGER`	4-byte integer (-2,147,483,648 to 2,147,483,647)
`BIGINT`	8-byte integer (-9 quintillion to 9 quintillion)
`DECIMAL(m, d)`	Exact fixed-point number (e.g., 10.2 means 10 digits, 2 decimal places)
`FLOAT`	Approximate floating-point number (4 bytes)

`DOUBLE` / `REAL`	Approximate floating-point number (8 bytes)
`BIT`(n)	Stores binary values (up to 64-bit)

2. Date and Time Data Types

Data Type	Description
`DATE`	Stores date (`YYYY-MM-DD`)
`DATETIME`	Stores date and time (`YYYY-MM-DD HH:MM:SS`)
`TIMESTAMP`	Stores timestamp (UTC-based, auto-updates)
`TIME`	Stores time (`HH:MM:SS`)
`YEAR`	Stores year (4 digits, e.g., `2025`)

3. String (Character) Data Types

Data Type	Description
`CHAR(n)`	Fixed-length string (0-255 characters)
`VARCHAR(n)`	Variable-length string (0-65,535 characters)
`TEXT`	Large text field (64 KB)
`TINYTEXT`	Very small text (255 bytes)
`MEDIUMTEXT`	Medium text (16 MB)
`LONGTEXT`	Large text (4 GB)
`BLOB`	Binary Large Object (stores binary data)
`TINYBLOB`	Small binary object (255 bytes)
`MEDIUMBLOB`	Medium binary object (16 MB)

`LONGBLOB`	Large binary object (4 GB)
`ENUM`	String with predefined values (`ENUM('small', 'medium', 'large')`)
`SET`	Multiple predefined values (`SET('a', 'b', 'c')`)

1. Numeric Data Types Table

```
CREATE TABLE NumericData (
    id INT PRIMARY KEY AUTO_INCREMENT,
    tiny_int_col TINYINT,
    small_int_col SMALLINT,
    medium_int_col MEDIUMINT,
    int_col INT,
    big_int_col BIGINT,
    decimal_col DECIMAL(10,2),
    float_col FLOAT,
    double_col DOUBLE
);
```

Insert Data

```
INSERT INTO NumericData (
    tiny_int_col, small_int_col, medium_int_col, int_col, big_int_col,
    decimal_col, float_col, double_col
) VALUES
(127, 32000, 8000000, 2147483647, 9223372036854775807,
 12345.67, 123.45, 123456.789);
```

Check Table Structure

```
DESC NumericData;
```

---

2. Date and Time Data Types Table

```
CREATE TABLE DateTimeData (
    id INT PRIMARY KEY AUTO_INCREMENT,
    date_col DATE,
    datetime_col DATETIME,
    timestamp_col TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    time_col TIME,
    year_col YEAR
);


INSERT INTO DateTimeData (
    date_col, datetime_col, time_col, year_col
) VALUES
('2025-03-20', '2025-03-20 14:30:00', '14:30:00', 2025);
```

Check Table Structure

```
DESC DateTimeData;
```

---

3. String (Character) Data Types Table

```
CREATE TABLE StringData (
    id INT PRIMARY KEY AUTO_INCREMENT,
    char_col CHAR(10),
    varchar_col VARCHAR(50),
    text_col TEXT,
    tinytext_col TINYTEXT,
    mediumtext_col MEDIUMTEXT,
    longtext_col LONGTEXT,
    enum_col ENUM('low', 'medium', 'high'),
    set_col SET('A', 'B', 'C', 'D')
);


INSERT INTO StringData (
    char_col, varchar_col, text_col, tinytext_col, mediumtext_col,
longtext_col,
    enum_col, set_col
) VALUES
('Hello', 'Hello World', 'This is a TEXT column', 'Tiny text data',
 'Medium text data', 'Long text data', 'high', 'A,B');
```

Check Table Structure

```
DESC StringData;
```

Summary

✔ We created **three separate tables** for different data types.
✔ We inserted **sample data** into each table.
✔ The `DESC` command helps **verify the structure** of each table.

This approach keeps data well-organized and easy to manage. Let me know if you need any modifications! 🚀

Using the `DESC` Command

The `DESC` command is used to describe the structure of a table.

sql
Copy code

```
DESC DataTypesDemo;
```

Importance of `DESC` Command

- Shows **column names**, **data types**, and **NULL constraints**.
- Displays **default values** (e.g., `DEFAULT CURRENT_TIMESTAMP`).
- Indicates **primary keys, indexes, and auto-increment properties**.
- Helps in debugging table structures before performing operations.

SQL Constraints: Definition & Usage

SQL **constraints** are rules applied to table columns to ensure **data integrity, accuracy, and reliability**. Constraints **restrict** the type of data that can be inserted into a table.

Types of SQL Constraints

SQL provides the following constraints:

1. **NOT NULL**
2. **UNIQUE**
3. **PRIMARY KEY**
4. **FOREIGN KEY**
5. **CHECK**
6. **DEFAULT**
7. **AUTO_INCREMENT** (MySQL-specific)
8. **INDEX** (Not exactly a constraint but helps optimize searches)

---

1. NOT NULL Constraint

- **Ensures that a column cannot store NULL values.**
- **Use case:** When a column must always have a value (e.g., `email`, `username`).

```
CREATE TABLE Users (

    id INT PRIMARY KEY AUTO_INCREMENT,

    name VARCHAR(50) NOT NULL,

    email VARCHAR(100) NOT NULL

);
```

- Here, `name` and `email` **must always have values** (i.e., they cannot be NULL).

---

2. UNIQUE Constraint

- **Ensures that all values in a column are different.**
- **Use case:** To prevent duplicate entries (e.g., unique email, unique username).

Example

sql

Copy code

```
CREATE TABLE Employees (

    emp_id INT PRIMARY KEY AUTO_INCREMENT,

    email VARCHAR(100) UNIQUE

);
```

- Here, `email` **must be unique**, so **no two employees can have the same email**.

---

3. PRIMARY KEY Constraint

- **Uniquely identifies each record in a table.**
- A **PRIMARY KEY** is a combination of `NOT NULL + UNIQUE`.
- **Use case:** To uniquely identify records in a table.

Example

sql

Copy code

```
CREATE TABLE Products (

    product_id INT PRIMARY KEY,

    name VARCHAR(100) NOT NULL,

    price DECIMAL(10,2)
);
```

🔹 Here, `product_id` is the **PRIMARY KEY**, meaning:

- Each `product_id` must be **unique**.
- Each `product_id` **cannot be NULL**.

🔹 **Composite Primary Key** (multiple columns can be a primary key):

sql

Copy code

```
CREATE TABLE Orders (

    order_id INT,

    product_id INT,

    quantity INT,

    PRIMARY KEY (order_id, product_id)

);
```

🔹 Here, **both** `order_id` and `product_id` together form a **PRIMARY KEY**.

---

4. FOREIGN KEY Constraint

🔹 **Links two tables together (parent-child relationship).**
🔹 The foreign key column **must refer to a PRIMARY KEY in another table**.
🔹 **Use case:** Ensuring referential integrity (e.g., orders must be linked to valid customers).

Example

sql

Copy code

```
CREATE TABLE Customers (

    customer_id INT PRIMARY KEY,

    name VARCHAR(100) NOT NULL

);


CREATE TABLE Orders (

    order_id INT PRIMARY KEY,

    customer_id INT,

    order_date DATE,

    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)

);
```

🔹 Here, `Orders.customer_id` is a **FOREIGN KEY**, which means:

- `customer_id` **must exist in the `Customers` table**.
- If a customer is deleted, we need to **handle dependencies** (e.g., `CASCADE`, `SET NULL`).

🔹 **Handling deletions:**

sql

Copy code

```
FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE
CASCADE;
```

- If a customer is deleted, all their orders are deleted too.

sql

Copy code

```
FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE
SET NULL;
```

- If a customer is deleted, `customer_id` in `Orders` is set to `NULL`.

---

5. CHECK Constraint

🔹 **Ensures that column values meet a specific condition.**
🔹 **Use case:** To enforce business rules (e.g., salary should be > 0, age should be > 18).

Example

sql

Copy code

```
CREATE TABLE Employees (

    emp_id INT PRIMARY KEY,

    name VARCHAR(50) NOT NULL,

    age INT CHECK (age >= 18),

    salary DECIMAL(10,2) CHECK (salary > 0)

);
```

🔹 Here:

- `age` **must be 18 or more**.
- `salary` **must be greater than 0**.

---

6. DEFAULT Constraint

- **Provides a default value if no value is given during insertion.**
- **Use case:** To set default values (e.g., default status as "pending" for orders).

Example

sql

Copy code

```
CREATE TABLE Orders (

    order_id INT PRIMARY KEY,

    status VARCHAR(20) DEFAULT 'Pending'

);
```

- If we insert a new order **without specifying `status`**, it will **automatically** get the value `'Pending'`.

sql

Copy code

```
INSERT INTO Orders (order_id) VALUES (1);

-- The status will be 'Pending'
```

---

7. AUTO_INCREMENT Constraint (MySQL only)

- **Automatically generates unique numbers for each row.**
- **Use case:** Used for primary keys to create unique IDs.

```
CREATE TABLE Employees (

    emp_id INT PRIMARY KEY AUTO_INCREMENT,
 name VARCHAR(100) NOT NULL
);
```

- Here, `emp_id` **automatically increments** for each new row.

sql

Copy code

```
INSERT INTO Employees (name) VALUES ('Alice');
INSERT INTO Employees (name) VALUES ('Bob');
```

- The `emp_id` values will be **1 and 2** respectively.

---

8. INDEX (Not a constraint, but important)

- **Speeds up searches and queries.**
- **Use case:** When querying large tables frequently based on a specific column.

```
CREATE INDEX idx_name ON Employees(name);
```

- Now, searching for employees **by name** will be **faster**.

Where and When to Use Constraints?

Constraint	Usage Scenario
NOT NULL	When a column **must always have a value** (e.g., `username`, `email`)

UNIQUE	To ensure **no duplicate values** (e.g., `email`, `phone number`)
PRIMARY KEY	To **uniquely identify** each row (e.g., `user_id`, `order_id`)
FOREIGN KEY	To **link tables** (e.g., `customer_id` in `Orders` refers to `Customers`)
CHECK	To **validate values** (e.g., `age >= 18`, `salary > 0`)
DEFAULT	To **set default values** (e.g., `status` as `'Pending'`)
AUTO_INCREMENT	To **generate unique IDs automatically** (e.g., `order_id`, `emp_id`)
INDEX	To **speed up queries** (e.g., searching by `name` in `Employees`)

Foreign Key and Reference Key – Explanation & Usage

What is a Foreign Key?

A foreign key is a column (or set of columns) in one table that establishes a relationship with the primary key in another table. It ensures that the values in the foreign key column must exist in the referenced table.

What is a Reference Key?

A reference key is simply the primary key of the referenced table. The foreign key in the child table refers to this primary key in the parent table.

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,

    dept_name VARCHAR(50)

);


CREATE TABLE Employees (

    emp_id INT PRIMARY KEY,

    emp_name VARCHAR(50),

    dept_id INT,

    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)

);
```

Here,

- **`dept_id` in Departments is the reference key (primary key).**
- **`dept_id` in Employees is the foreign key that refers to the reference key.**

Why is a Foreign Key Used?

1. **Maintains Data Integrity**
    - **Ensures only valid data is inserted into the foreign key column.**
    - **Example: An employee must belong to an existing department.**
2. **Establishes Relationships Between Tables**
    - **Links tables logically, avoiding duplicate data.**
    - **Example: Employees belong to departments instead of repeating department names in every employee record.**
3. **Ensures Referential Integrity**
    - **Prevents orphan records (child records without a valid parent).**
    - **Example: You can't assign an employee to a non-existent department.**

4. **Helps in Cascading Operations**
    - **Allows automatic updates or deletions when the parent record changes.**
    - **Example: If a department is deleted, all employees in that department can also be deleted (`ON DELETE CASCADE`).**

Summary:

Term	Definition
Foreign Key	**Column in a child table that references a primary key in another table.**
Reference Key	**The primary key in the parent table being referenced.**
Purpose	**Enforces data integrity, maintains relationships, and ensures referential integrity.**

Would you like more examples or a deeper explanation? 😊

DDL (Data Definition Language) Commands in SQL

DDL (Data Definition Language) is used to define and manage the **structure** of database objects like tables, indexes, and constraints.

📌 **List of All DDL Commands in SQL**

Command	Description

`CREATE`	Creates **new database objects** (tables, views, indexes, etc.).
`ALTER`	Modifies **existing database objects** (add, delete, modify columns).
`DROP`	Deletes an **entire database object** (table, view, index).
`TRUNCATE`	Deletes **all records** from a table but keeps the table structure.
`RENAME`	Renames an existing database object.
`COMMENT`	Adds a **comment** to database objects (table, column, etc.).

📖 Explanation with Examples

1️⃣ `CREATE` – Create a Table

sql

Copy code

```sql
CREATE TABLE Employees (

    emp_id INT PRIMARY KEY,

    name VARCHAR(50),

    age INT,

    department VARCHAR(50)

);
```

- **Creates a table `Employees`** with `emp_id`, `name`, `age`, and `department`.

2️⃣ALTER – Modify Table Structure

(a) Add a Column

sql

Copy code

```
ALTER TABLE Employees ADD salary DECIMAL(10,2);
```

- Adds a new column `salary` to the `Employees` table.

(b) Modify Column Data Type

sql

Copy code

```
ALTER TABLE Employees MODIFY name VARCHAR(100);
```

- Changes the data type of `name` to `VARCHAR(100)`.

(c) Rename a Column

```
ALTER TABLE Employees CHANGE salary monthly_salary DECIMAL(10,2);
```

- Renames column `salary` to `monthly_salary`.

(d) Delete a Column

```
ALTER TABLE Employees DROP COLUMN age;
```

- Removes the `age` column from the `Employees` table.

3️⃣ `DROP` – Delete an Object

```
DROP TABLE Employees;
```

🔹 **Deletes the entire table `Employees` permanently** (all data and structure).

```
DROP DATABASE CompanyDB;
```

🔹 **Deletes the entire database `CompanyDB`**.

4️⃣ `TRUNCATE` – Delete All Records

```
TRUNCATE TABLE Employees;
```

🔹 **Deletes all records** from the `Employees` table but **keeps the table structure**.

📌 **Difference between `DELETE` and `TRUNCATE`:**

Feature	DELETE	TRUNCATE
Removes Specific Rows?	✅ Yes (With `WHERE`)	❌ No (Deletes all rows)
Keeps Table Structure?	✅ Yes	✅ Yes
Can Be Rolled Back?	✅ Yes (If within a transaction)	❌ No
Performance	🚀 Slower (logs each row deletion)	⚡ Faster (resets table directly)

5️⃣ `RENAME` – Rename Table or Column

```
RENAME TABLE Employees TO Staff;
```

🔹 Renames table `Employees` to `Staff`.

6️⃣ `COMMENT` – Add Comments to Tables & Columns

```
COMMENT ON COLUMN Employees.name IS 'Employee Full Name';
```

🔹 Adds a **comment** on the `name` column of `Employees` table.

🎯 Quick Recap

DDL Command	Purpose
`CREATE`	Creates a new **database object**.
`ALTER`	Modifies an **existing object** (add, modify, delete column).
`DROP`	Deletes an entire **object permanently**.
`TRUNCATE`	Deletes **all records** but **keeps the structure**.
`RENAME`	Renames a **table or column**.

`COMMENT`	Adds a **description** to an object.

Would you like me to cover **DML (Data Manipulation Language)** next? 😊🚀

UPDATE and DELETE Constraints on FOREIGN KEY in SQL

1. ON DELETE CASCADE

Effect:

Automatically deletes child records when the parent record is deleted.

Tables & Sample Data:

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id) ON DELETE CASCADE
);

INSERT INTO Departments VALUES (1, 'HR'), (2, 'IT');
INSERT INTO Employees VALUES (101, 'Alice', 1), (102, 'Bob', 2), (103, 'Charlie', 1);
```

Operation:

```
DELETE FROM Departments WHERE dept_id = 1;
```

Output:

```
SELECT * FROM Employees;
-- | emp_id | emp_name | dept_id |
-- |--------|---------|---------|
-- | 102    | Bob     | 2       |
```

✅ **Explanation**: Employees in dept_id = 1 (Alice and Charlie) are automatically removed.

---

2. ON DELETE SET NULL

Effect:

Sets the child foreign key to NULL when the parent record is deleted.

Tables & Sample Data:

```
CREATE TABLE Projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(50)
);

CREATE TABLE Tasks (
    task_id INT PRIMARY KEY,
    task_name VARCHAR(50),
    project_id INT,
    FOREIGN KEY (project_id) REFERENCES Projects(project_id) ON DELETE SET NULL
);

INSERT INTO Projects VALUES (100, 'Website Redesign'), (200, 'App Launch');
INSERT INTO Tasks VALUES (1, 'Design Homepage', 100), (2, 'Test API', 200);
```

Operation:

```
DELETE FROM Projects WHERE project_id = 100;
```

Output:

```
SELECT * FROM Tasks;
-- | task_id | task_name       | project_id |
-- |---------|-----------------|------------|
-- | 1       | Design Homepage | NULL       |
-- | 2       | Test API        | 200        |
```

✅ **Explanation**: The task linked to project_id = 100 keeps its data, but its project_id is set to NULL.

---

3. ON DELETE SET DEFAULT

Effect:

Resets the child foreign key to a default value when the parent record is deleted.

Tables & Sample Data:

```
CREATE TABLE Categories (
    category_id INT PRIMARY KEY DEFAULT 0,
    category_name VARCHAR(50)
);

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    category_id INT DEFAULT 0,
    FOREIGN KEY (category_id) REFERENCES Categories(category_id) ON DELETE SET DEFAULT
);

INSERT INTO Categories VALUES (0, 'Uncategorized'), (1, 'Electronics');
INSERT INTO Products VALUES (500, 'Laptop', 1), (501, 'Desk', 0);
```

Operation:

```
DELETE FROM Categories WHERE category_id = 1;
```

Output:

```
SELECT * FROM Products;
-- | product_id | product_name | category_id |
-- |------------|-------------|-------------|
-- | 500       | Laptop     | 0          |

-- | 501       | Desk       | 0          |
```

✅ **Explanation**: Products in the deleted category are reassigned to the default category_id = 0.

---

4. ON DELETE RESTRICT

Effect:

Prevents deletion if child records exist.

Tables & Sample Data:

```
CREATE TABLE Authors (
    author_id INT PRIMARY KEY,
    author_name VARCHAR(50)
);

CREATE TABLE Books (
    book_id INT PRIMARY KEY,
    book_title VARCHAR(50),
    author_id INT,
    FOREIGN KEY (author_id) REFERENCES Authors(author_id) ON DELETE RESTRICT
);

INSERT INTO Authors VALUES (1, 'J.K. Rowling');
INSERT INTO Books VALUES (100, 'Harry Potter', 1);
```

Operation:

```
DELETE FROM Authors WHERE author_id = 1;
```

Output:

```
-- ❌ Error: "Cannot delete or update a parent row: a foreign key constraint fails"
```

✅ **Explanation**: The deletion is blocked because the author has linked books.

Summary of FOREIGN KEY Constraints:

Constraint	Parent Operation	Child Effect
ON DELETE CASCADE	Delete	Child records are deleted
ON DELETE SET NULL	Delete	Child foreign key set to NULL
ON DELETE SET DEFAULT	Delete	Child foreign key set to default
ON DELETE RESTRICT	Delete	Parent deletion is blocked
ON DELETE NO ACTION	Delete	Same as RESTRICT (deletion blocked)
ON UPDATE CASCADE	Update	Child foreign key is updated

ON UPDATE SET NULL	Update	Child foreign key set to NULL
ON UPDATE SET DEFAULT	Update	Child foreign key set to default
ON UPDATE RESTRICT	Update	Parent update is blocked
ON UPDATE NO ACTION	Update	Same as RESTRICT (update blocked)

These examples illustrate how **foreign key constraints** help maintain data integrity in relational databases.

Here's an optimized version for **ON UPDATE** foreign key constraints, formatted for Google Docs:

Practical Examples of FOREIGN KEY Constraints – ON UPDATE

Foreign key constraints maintain data integrity when updating primary keys in relational databases. Below are examples with sample data and outputs for different **ON UPDATE** actions.

1. ON UPDATE CASCADE

Effect: Updates child foreign keys when the parent's primary key changes.

Tables & Data:

sql

Copy code

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50)
);


CREATE TABLE Enrollments (

    enrollment_id INT PRIMARY KEY,

    student_id INT,

    course VARCHAR(50),

    FOREIGN KEY (student_id) REFERENCES Students(student_id) ON UPDATE
CASCADE

);


INSERT INTO Students VALUES (1, 'Alice');

INSERT INTO Enrollments VALUES (100, 1, 'Math');
```

Operation:

sql

Copy code

```
UPDATE Students SET student_id = 10 WHERE student_id = 1;
```

Output:

sql

Copy code

```
SELECT * FROM Enrollments;

-- | enrollment_id | student_id | course |

-- |---------------|------------|--------|
-- | 100          | 10         | Math    | -- Updated automatically
```

✅ **Explanation:** The student_id in the child table updates to match the new value in the parent table.

---

2. ON UPDATE SET NULL

Effect: Sets child foreign keys to NULL when the parent's primary key changes.

Tables & Data:

sql

Copy code

```
CREATE TABLE Teams (

    team_id INT PRIMARY KEY,

    team_name VARCHAR(50)

);


CREATE TABLE Members (

    member_id INT PRIMARY KEY,

    team_id INT,

    name VARCHAR(50),

    FOREIGN KEY (team_id) REFERENCES Teams(team_id) ON UPDATE SET NULL

);


INSERT INTO Teams VALUES (1, 'Developers');

INSERT INTO Members VALUES (101, 1, 'Bob');
```

Operation:

sql

Copy code

```
UPDATE Teams SET team_id = 100 WHERE team_id = 1;
```

Output:

sql

Copy code

```
SELECT * FROM Members;
-- | member_id | team_id | name |
-- |-----------|---------|------|
-- | 101       | NULL    | Bob  | -- team_id set to NULL
```

✅ **Explanation:** The team_id is set to NULL because the referenced value changed.

---

3. ON UPDATE SET DEFAULT

Effect: Sets child foreign keys to a default value when the parent's primary key changes.

Tables & Data:

sql

Copy code

```
CREATE TABLE Suppliers (
    supplier_id INT PRIMARY KEY DEFAULT 0,
    supplier_name VARCHAR(50)
);


CREATE TABLE Products (

    product_id INT PRIMARY KEY,

    supplier_id INT DEFAULT 0,

    FOREIGN KEY (supplier_id) REFERENCES Suppliers(supplier_id) ON
UPDATE SET DEFAULT

);


-- Insert default supplier

INSERT INTO Suppliers VALUES (0, 'No Supplier'), (1, 'Tech Corp');

INSERT INTO Products VALUES (500, 1), (501, 0);
```

Operation:

sql

Copy code

```
UPDATE Suppliers SET supplier_id = 100 WHERE supplier_id = 1;
```

Output:

sql

Copy code

```
SELECT * FROM Products;

-- | product_id | supplier_id |

-- |------------|-------------|
-- | 500        | 0           | -- Reset to default (No Supplier)

-- | 501        | 0           |
```

✅ **Explanation:** The child records linked to the changed parent key are reset to the default value (0).

---

4. ON UPDATE RESTRICT

Effect: Prevents parent key updates if child records exist.

Tables & Data:

sql

Copy code

```
CREATE TABLE Users (

    user_id INT PRIMARY KEY,

    name VARCHAR(50)

);


CREATE TABLE Posts (

    post_id INT PRIMARY KEY,

    user_id INT,

    content TEXT,

    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON UPDATE RESTRICT

);


INSERT INTO Users VALUES (1, 'Alice');
INSERT INTO Posts VALUES (100, 1, 'Hello World');
```

Operation:

sql

Copy code

```
UPDATE Users SET user_id = 10 WHERE user_id = 1;
```

Output:

sql

Copy code

```
-- ❌ Error: "Cannot delete or update a parent row: a foreign key
constraint fails"
```

✅ **Explanation:** The update is blocked because the user has associated posts.

---

5. ON UPDATE NO ACTION

Effect: Same as RESTRICT – prevents updates if child records exist.

Tables & Data:

sql

Copy code

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

```
CREATE TABLE Salaries (

    salary_id INT PRIMARY KEY,

    emp_id INT,

    amount DECIMAL(10,2),

    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id) ON UPDATE NO
ACTION

);


INSERT INTO Employees VALUES (1, 'Alice');

INSERT INTO Salaries VALUES (100, 1, 5000);
```

Operation:

sql

Copy code

```
UPDATE Employees SET emp_id = 10 WHERE emp_id = 1;
```

Output:

sql

Copy code

```
-- ❌ Error: "Cannot delete or update a parent row: a foreign key
constraint fails"
```

✅ **Explanation:** The update is blocked because ON UPDATE NO ACTION prevents modification.

Summary of ON UPDATE Effects

Constraint	Parent Update Effect	Child Effect
ON UPDATE CASCADE	Parent key updated	Child records updated
ON UPDATE SET NULL	Parent key updated	Child foreign key set to NULL
ON UPDATE SET DEFAULT	Parent key updated	Child foreign key set to default
ON UPDATE RESTRICT	Parent key update blocked	No changes allowed
ON UPDATE NO ACTION	Parent key update blocked	No changes allowed

---

This structured breakdown makes it easy to understand how **ON UPDATE** constraints enforce referential integrity. Let me know if you need modifications! 🚀

Final Thoughts

✅ **Understanding `ON DELETE` and `ON UPDATE` helps prevent orphan records and maintain consistency.**
✅ **Choosing the right constraint depends on business rules and data relationships.**
✅ **Use `CASCADE` carefully to avoid accidental mass deletions.**

Let me know if you need further clarifications or code examples! 🚀

SQL `INSERT`, `DELETE`, and `UPDATE` with Multiple Examples

SQL provides three essential commands to manipulate data in tables:

1. `INSERT` → **Adds new records.**
2. `UPDATE` → **Modifies existing records.**
3. `DELETE` → **Removes records.**

We will go through detailed examples with real-world scenarios.

---

1. `INSERT` Statement

Used to add new records into a table.

Syntax

sql

Copy code

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

Example 1: Inserting a Single Record

sql

Copy code

```
CREATE TABLE Employees (

    emp_id INT PRIMARY KEY AUTO_INCREMENT,

    name VARCHAR(50) NOT NULL,
    department VARCHAR(50),

    salary DECIMAL(10,2)

);


INSERT INTO Employees (name, department, salary)

VALUES ('Alice', 'HR', 50000.00);
```

✅ `emp_id` **is auto-generated, so we don't need to provide it.**

Example 2: Inserting Multiple Records

```
INSERT INTO Employees (name, department, salary) VALUES

('John Doe', 'IT', 70000.00),

('Emma Watson', 'Finance', 60000.00),

('Mike Brown', 'IT', 75000.00);
```

✅ **Inserting multiple rows in a single query improves performance.**

Example 3: `INSERT` from Another Table

sql

Copy code

```
CREATE TABLE Employee_Backup AS

SELECT * FROM Employees WHERE department = 'IT';
```

✅ This creates a backup table with only IT employees.

---

2. UPDATE Statement

Used to modify existing records.

Syntax

sql

Copy code

```sql
UPDATE table_name

SET column1 = value1, column2 = value2, ...

WHERE condition;
```

⚠️ Without `WHERE`, all rows will be updated!

Example 1: Updating a Single Record

sql

Copy code

```sql
UPDATE Employees

SET salary = 80000

WHERE name = 'John Doe';
```

✅ John Doe’s salary is now updated to 80,000.

Example 2: Updating Multiple Rows

sql

Copy code

```
UPDATE Employees

SET salary = salary * 1.10

WHERE department = 'IT';
```

✅ **Increases the salary by 10% for all IT employees.**

Example 3: Updating Based on Another Table

sql

Copy code

```
UPDATE Employees e

JOIN Employee_Backup b ON e.emp_id = b.emp_id

SET e.salary = b.salary * 1.05;
```

✅ **Increases the salary of employees in the backup table by 5%.**

---

3. `DELETE` Statement

Used to remove records.

Syntax

sql

Copy code

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

⚠️ **Without `WHERE`, all rows will be deleted!**

Example 1: Deleting a Single Record

sql

Copy code

```
DELETE FROM Employees

WHERE name = 'Mike Brown';
```

✅ **Mike Brown is removed from the table.**

Example 2: Deleting Multiple Rows

sql

Copy code

```
DELETE FROM Employees

WHERE department = 'HR';
```

✅ **All employees in HR are deleted.**

Example 3: Deleting All Records (`TRUNCATE`)

sql

Copy code

```
TRUNCATE TABLE Employees;
```

✅ **Removes all records but keeps the table structure.**

Example 4: `DELETE` with `JOIN`

sql

Copy code

```
DELETE e FROM Employees e

JOIN Employee_Backup b ON e.emp_id = b.emp_id;
```

✅ **Deletes employees that exist in both `Employees` and `Employee_Backup`.**

Final Thoughts

Command	**Use Case**
`INSERT`	**Add new records**
`UPDATE`	**Modify existing records**

`DELETE`	**Remove specific records**
`TRUNCATE`	**Remove all records (faster than `DELETE`)**

🚀 Let me know if you need more practice queries or explanations! 😊

EXAMPLE :

```
use onetomany;
create table ss(
id int,
name varchar(10)
);
create table mm(
id int,
name varchar(10)
);

INSERT INTO ss (id, name) VALUES
(1, 'it'), (2, 'it'), (3, 'it'), (4, 'it'), (5, 'it'), (6, 'hr'), (7, 'sales'), (8, 'admin'), (9, 'finance'), (10, 'marketing');
insert into mm select *from ss where name='it';
select *from mm;
```

DELETE OPERATION DELETE ENTIRE ROW NOT SINGLE COLUMN USE UPDATE TO CHANGE OR DELETE ONE COLUMN

Here’s the exact SQL script to create tables, insert data, and demonstrate the `DELETE` and `UPDATE` operations using three-table joins.

1️⃣Create Tables and Insert Sample Data

sql

Copy code

```
-- Creating Customers Table
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(50),
    status VARCHAR(20) -- 'Active' or 'Inactive'
);

-- Creating Products Table
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    is_discontinued BOOLEAN -- 1 = discontinued, 0 = available
);

-- Creating Orders Table
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    product_id INT,
    order_date DATE
);

-- Insert Data into Customers
INSERT INTO Customers (customer_id, name, status) VALUES
(1, 'Alice', 'Active'),
(2, 'Bob', 'Inactive'),
(3, 'Charlie', 'Inactive');

-- Insert Data into Products
INSERT INTO Products (product_id, product_name, is_discontinued)
VALUES
```

```
(101, 'Laptop', 0),
(102, 'Keyboard', 1), -- Discontinued
(103, 'Mouse', 1);    -- Discontinued

-- Insert Data into Orders
INSERT INTO Orders (order_id, customer_id, product_id, order_date)
VALUES
(1001, 1, 101, '2023-01-01'), -- Active customer, non-discontinued
product (retained)
(1002, 2, 102, '2023-02-01'), -- Inactive customer, discontinued
product (deleted)
(1003, 3, 103, '2023-03-01'); -- Inactive customer, discontinued
product (deleted)
```

2️⃣ DELETE Query with Three-Table JOIN

sql

Copy code

```
DELETE o FROM Orders o
JOIN Customers c ON o.customer_id = c.customer_id
JOIN Products p ON o.product_id = p.product_id
WHERE c.status = 'Inactive' -- Delete orders from inactive customers
  AND p.is_discontinued = 1; -- Delete orders for discontinued
products
```

✅ Expected Result After Deletion:

sql

Copy code

```
SELECT * FROM Orders;
```

order_id	customer_id	product_id	order_date
1001	1	101	2023-01-01

💡 Explanation:

- Orders placed by inactive customers (`Bob` and `Charlie`) for discontinued products (`Keyboard` and `Mouse`) are **deleted**.
- Only the order for **Alice (Active customer) with Laptop (not discontinued)** remains.

3️⃣ Create More Tables for UPDATE Example

sql

Copy code

```
-- Creating Suppliers Table
CREATE TABLE Suppliers (
    supplier_id INT PRIMARY KEY,
    supplier_name VARCHAR(50),
    cost_rating VARCHAR(10) -- 'Low', 'Medium', 'High'
);

-- Creating Categories Table
CREATE TABLE Categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50)
);

-- Creating Products Table (Extended with supplier and category)
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    price DECIMAL(10,2),
    supplier_id INT,
    category_id INT
);

-- Insert Data into Suppliers
INSERT INTO Suppliers (supplier_id, supplier_name, cost_rating) VALUES
(1, 'TechGlobal', 'High'),
(2, 'EcoSupplies', 'Low');

-- Insert Data into Categories
INSERT INTO Categories (category_id, category_name) VALUES
(10, 'Electronics'),
(20, 'Accessories');

-- Insert Data into Products
INSERT INTO Products (product_id, product_name, price, supplier_id,
category_id) VALUES
(101, '4K Monitor', 299.99, 1, 10), -- High-cost supplier, Electronics
(102, 'Wireless Mouse', 19.99, 2, 20), -- Low-cost supplier,
Accessories
(103, 'Gaming Chair', 199.99, 1, 10); -- High-cost supplier,
Electronics
```

4️⃣UPDATE Query with Three-Table JOIN

sql
Copy code

```
UPDATE Products p
JOIN Suppliers s ON p.supplier_id = s.supplier_id
JOIN Categories c ON p.category_id = c.category_id
SET p.price = p.price * 0.9 -- Apply 10% discount
WHERE s.cost_rating = 'High' -- Target expensive suppliers
  AND c.category_name = 'Electronics'; -- Target Electronics category
```

✅ Expected Result After Update:

sql
Copy code

```
SELECT * FROM Products;
```

product_id	product_name	price	supplier_id	category_id
101	4K Monitor	269.99	1	10
102	Wireless Mouse	19.99	2	20
103	Gaming Chair	179.99	1	10

💡 Explanation:

- **Only products in the "Electronics" category and supplied by "High" cost suppliers (TechGlobal) are updated** with a **10% price discount**.
- The price of **"4K Monitor" and "Gaming Chair"** is reduced.
- The **"Wireless Mouse" remains unchanged** because its supplier has a **low cost rating**.

🚀 Summary of Key Concepts

Operation	Query	Purpose
DELETE	`DELETE o FROM Orders o JOIN Customers c ...`	Delete orders from inactive customers for discontinued products.
UPDATE	`UPDATE Products p JOIN Suppliers s ...`	Apply a **10% discount** to **expensive electronics** from high-cost suppliers.

Now you can **run these queries in MySQL** and test them with the given sample data. Let me know if you need modifications or additional explanations! 🚀🔥

The **SELECT** keyword in SQL is used to retrieve data from a database. Below is a comprehensive breakdown of its syntax, including clauses and optional components. The order of clauses **must follow the sequence outlined here**.

Basic Syntax

```
SELECT [DISTINCT | ALL] column1, column2, ...
FROM table_name
[WHERE condition]
[GROUP BY column(s)]
[HAVING condition]
[ORDER BY column(s) [ASC | DESC]]
[LIMIT n]  -- MySQL/PostgreSQL
[OFFSET m] -- MySQL/PostgreSQL
```

SQL Operators Table

Category	Operator	Description	Example

Comparison Operators	`=`	Equal to	`WHERE salary = 50000`
	`!=` or `<>`	Not equal to	`WHERE department != 'IT'`
	`>`	Greater than	`WHERE salary > 60000`
	`<`	Less than	`WHERE hire_date < '2022-01-01'`
	`>=`	Greater than or equal to	`WHERE age >= 30`
	`<=`	Less than or equal to	`WHERE salary <= 45000`
Logical Operators	`AND`	All conditions must be true	`WHERE department = 'IT' AND salary > 60000`
	`OR`	At least one condition is true	`WHERE department = 'HR' OR department = 'Sales'`
	`NOT`	Negates a condition	`WHERE NOT department = 'IT'`
Arithmetic Operators	`+`	Addition	`SELECT salary + 5000 AS new_salary`
	`-`	Subtraction	`SELECT salary - tax AS net_salary`
	`*`	Multiplication	`SELECT quantity * price AS total`
	`/`	Division	`SELECT salary / 12 AS monthly_pay`
Pattern Matching	`%`	Matches any sequence of characters	`WHERE name LIKE 'A%'` (Starts with "A")
	`_`	Matches a single character	`WHERE name LIKE '_ob'` (e.g., "Bob")
Membership Operators	`IN`	Value exists in a list	`WHERE department IN ('IT', 'HR')`
	`NOT IN`	Value does not exist in list	`WHERE department NOT IN ('Sales')`
Range Operators	`BETWEEN`	Between two values (inclusive)	`WHERE salary BETWEEN 45000 AND 60000`

	`NOT BETWEEN`	Outside a range	`WHERE hire_date NOT BETWEEN '2020' AND '2022'`
NULL Check Operators	`IS NULL`	Checks for NULL	`WHERE hire_date IS NULL`
	`IS NOT NULL`	Checks for non-NULL	`WHERE department IS NOT NULL`
String Operators	`		or+`
Set Operators	`UNION`	Combines results (removes duplicates)	`SELECT name FROM IT UNION SELECT name FROM HR`
	`UNION ALL`	Combines results (allows duplicates)	`SELECT name FROM IT UNION ALL SELECT name FROM HR`
	`INTERSECT`	Returns common rows	`SELECT name FROM IT INTERSECT SELECT name FROM HR`
	`EXCEPT`	Returns rows from the first query not in the second	`SELECT name FROM IT EXCEPT SELECT name FROM HR`
Bitwise Operators	`&`	Bitwise AND	`SELECT 5 & 3` (Result: 1)
	`	`	Bitwise OR
	`^`	Bitwise XOR	`SELECT 5 ^ 3` (Result: 6)
Special Operators	`EXISTS`	Checks if a subquery returns rows	`WHERE EXISTS (SELECT 1 FROM employees)`
	`DISTINCT`	Returns unique values	`SELECT DISTINCT department FROM employees`
	`CASE`	Conditional logic	`SELECT CASE WHEN salary > 50000 THEN 'High' ELSE 'Low' END`

Pattern Matching in SQL with `SELECT`

Pattern matching in SQL is primarily done using the `LIKE` operator, which is used to filter results based on a specific pattern. The two main wildcard characters used in `LIKE` are:

- `%` (percent sign) – Matches **zero, one, or multiple** characters.
- `_` (underscore) – Matches **a single** character.

Sample Table: `employees`

We will use this table for examples:

id	name	department	email
1	Alice	IT	alice@company.com
2	Bob	HR	bob.hr@company.com
3	Charlie	IT	charlie@company.com
4	Diana	Finance	diana@finance.com
5	Eve	HR	eve123@company.com

---

1. Using `%` for Pattern Matching

Example 1: Names that start with 'A'

```
SELECT * FROM employees WHERE name LIKE 'A%';
```

🔹 **Explanation:**

- The `%` after 'A' means "any number of characters after A".
- This will return names like **Alice**, **Alex**, and **Andrew**.

Example 2: Emails that contain 'company'

```
SELECT * FROM employees WHERE email LIKE '%company%';
```

🔹 **Explanation:**

- `%company%` means "contains the word 'company' anywhere in the string".
- This will return emails like **alice@company.com**, **bob.hr@company.com**, and **eve123@company.com**.

---

2. Using `_` for Single Character Matching

Example 3: Names with 'o' as the second letter

```
SELECT * FROM employees WHERE name LIKE '_o%';
```

- `_o%` means "any name where the second letter is 'o'".
- This will return names like **Bob** but not **Alice** or **Charlie**.

Example 4: Emails that start with exactly four characters before '@'

```
SELECT * FROM employees WHERE email LIKE '____@%';
```

- `____@%` means "exactly four characters before the '@' symbol".
- This will return emails like **"diana@finance.com"**, but not **"alice@company.com"** because "alice" has 5 characters.

3. Combining `%` and `_` for Complex Patterns

Example 5: Emails ending with '.com' but having a digit before '@'

```
SELECT * FROM employees WHERE email LIKE '%[0-9]@%.com';
```

🔹 **Explanation:**

- `%[0-9]@%.com` means:
  - `%` → Any number of characters before the digit.
  - `[0-9]` → A digit must appear before `@`.
  - `%` → Any characters after `@` (domain name).

    - `.com` → Ends with ".com".
- This will return **eve123@company.com**, but **not** "alice@company.com" (since Alice's email has no digit).

✅ **Output:**

id	name	department	email
5	Eve	HR	eve123@company.com

---

4. Using `NOT LIKE` for Exclusion

Example 6: Employees whose names do NOT start with 'A'

sql

CopyEdit

```
SELECT * FROM employees WHERE name NOT LIKE 'A%';
```

🔹 **Explanation:**

- `NOT LIKE 'A%'` excludes names that start with 'A'.

Conclusion

- `%` → Matches **any number of characters** (including zero).
- `_` → Matches **a single character**.
- `LIKE` → Finds matches based on a pattern.
- `NOT LIKE` → Excludes matches based on a pattern.

Let me know if you need more examples! 🚀

Membership Operators in SQL (`IN` and `NOT IN`) with `SELECT`

Membership operators are used to check if a value exists within a specified list of values. These operators include:

- `IN` → Checks if a value exists in a given list.
- `NOT IN` → Checks if a value does **not** exist in a given list.

Sample Table: `employees`

We will use this table for examples:

id	name	department	salary	hire_date	age
1	Alice	IT	50000	2021-03-15	28
2	Bob	HR	60000	2020-07-22	35
3	Charlie	IT	45000	2023-01-10	30
4	Diana	Finance	75000	2019-11-05	42
5	Eve	HR	48000	2022-09-30	29

---

1. Using `IN` Operator

Example 1: Selecting employees from IT and HR departments

```
SELECT * FROM employees WHERE department IN ('IT', 'HR');
```

🔹 **Explanation:**

This query returns only those employees whose `department` is either **IT** or **HR**.

```
SELECT * FROM employees WHERE department = 'IT' OR
department = 'HR';
```

✅ **Output:**

id	name	department	salary	hire_date	age
1	Alice	IT	50000	2021-03-15	28
2	Bob	HR	60000	2020-07-22	35
3	Charlie	IT	45000	2023-01-10	30
5	Eve	HR	48000	2022-09-30	29

Example 2: Selecting employees hired in specific years

```
SELECT * FROM employees WHERE YEAR(hire_date) IN (2020, 2021);
```

🔹 **Explanation:**

- Filters employees hired in **2020 or 2021**.

✅ **Output:**

id	name	department	salary	hire_date	age
1	Alice	IT	50000	2021-03-15	28
2	Bob	HR	60000	2020-07-22	35

---

2. Using `NOT IN` Operator

Used to filter rows where the column value is **not** present in the given list.

Example 3: Selecting employees NOT from IT and HR departments

```
SELECT * FROM employees WHERE department NOT IN ('IT', 'HR');
```

Excludes employees from **IT** and **HR** departments.

Example 4: Selecting employees whose salary is NOT in a specific set

```
SELECT * FROM employees WHERE salary NOT IN (45000, 50000);
```

- Filters out employees who have a salary of **45000** or **50000**.

---

3. Using `IN` with Subqueries

We can also use `IN` with subqueries to match values dynamically.

Example 5: Select employees whose department has employees earning more than 60000

```
SELECT * FROM employees WHERE department IN
    (SELECT department FROM employees WHERE salary > 60000);
```

🔹 **Explanation:**

- Finds departments where at least one employee earns more than **60000**.
- Returns all employees from those departments.

✅ **Output:**

id	name	department	salary	hire_date	Age
2	Bob	HR	60000	2020-07-22	35
4	Diana	Finance	75000	2019-11-05	42
5	Eve	HR	48000	2022-09-30	29

Conclusion

- `IN` checks if a value **exists** in a given list.
- `NOT IN` checks if a value **does not exist** in a given list.
- `IN` can be used with **subqueries** to filter dynamically.

Would you like more complex queries? 🚀

Here are different **Range Operators (`BETWEEN` and `NOT BETWEEN`)** with explanations and examples using the `SELECT` statement.

---

1. Using `BETWEEN` with Numeric Values

📌 **Find employees with salaries between 45,000 and 60,000.**

```
SELECT * FROM employees WHERE salary BETWEEN 45000 AND 60000;
```

- This query selects employees whose **salary is between 45,000 and 60,000**, inclusive of both values.
- Equivalent to:

```
SELECT * FROM employees WHERE salary >= 45000 AND salary
<=60000;
```

---

2. Using `NOT BETWEEN` with Numeric Values

📌 **Find employees whose salaries are NOT between 45,000 and 60,000.**

```
SELECT * FROM employees

WHERE salary NOT BETWEEN 45000 AND 60000;
```

3. Using `BETWEEN` with Dates

📌 **Find employees hired between 2020 and 2022.**

```
SELECT * FROM employees

WHERE hire_date BETWEEN '2020-01-01' AND '2022-12-31';
```

4. Using NOT BETWEEN with Dates

📌 **Find employees hired BEFORE 2020 or AFTER 2022.**

```
SELECT * FROM employees

WHERE hire_date NOT BETWEEN '2020-01-01' AND '2022-12-31';
```

---

5. Using BETWEEN with Age

📌 **Find employees aged between 28 and 35.**

```
SELECT * FROM employees  WHERE age BETWEEN 28 AND 35;
```

6. Using BETWEEN with String Values (Alphabetic Range)

📌 **Find employees whose names fall alphabetically between 'B' and 'D'.**

```
SELECT * FROM employees

WHERE name BETWEEN 'B' AND 'D';
```

---

7. Using NOT BETWEEN with String Values

📌 **Find employees whose names DO NOT fall alphabetically between 'B' and 'D'.**

```
SELECT * FROM employees WHERE name NOT BETWEEN 'B' AND 'D';
```

Key Takeaways

✅ `BETWEEN` includes both **start** and **end** values in the range.
✅ `NOT BETWEEN` excludes the given range.
✅ Works with **numbers, dates, and strings**.
✅ **Alphabetic comparison** works in SQL but depends on collation settings.
✅ **Use quotes (`'  '`) for dates and string comparisons.**

Would you like more advanced examples? 🚀

Here are different **NULL Check Operators (`IS NULL` and `IS NOT NULL`)** with explanations and examples using the `SELECT` statement.

---

1. Using `IS NULL`

📌 **Find employees whose hire date is not available (NULL values).**

```
SELECT * FROM employees  WHERE hire_date IS NULL;
```

🔹 **Explanation:**

- This query selects employees **where the `hire_date` column has NULL values** (i.e., no data is recorded).

✅ **Output:**

id	name	salary	department	hire_date	age
6	Frank	55000	IT	NULL	32

---

2. Using `IS NOT NULL`

📌 **Find employees who have a hire date (not NULL).**

```
SELECT * FROM employees  WHERE hire_date IS NOT NULL;
```

```
OUTPUT:
```

Id	hire date
1	2-3-25

3. Using `IS NULL` with `UPDATE`

📌 **Assign a default hire date (`'2024-01-01'`) to employees where hire_date is NULL.**

```
UPDATE employees

SET hire_date = '2024-01-01'

WHERE hire_date IS NULL;
```

🔹 **Explanation:**

- Updates all rows where `hire_date` is **NULL** and replaces it with `'2024-01-01'`.

---

4. Using `IS NULL` with `DELETE`

📌 **Remove all employees who don't have a hire date.**

```
DELETE FROM employees

WHERE hire_date IS NULL;
```

🔹 **Explanation:**

- Deletes records **where `hire_date` is NULL** (i.e., employees without recorded hiring information).

---

5. Using `IS NULL` in an `IF` Condition

📌 **Select employees and replace NULL hire dates with ‘Not Available’.**

```
SELECT name,

       COALESCE(hire_date, 'Not Available') AS hire_date

FROM employees;
```

🔹 **Explanation:**

- The `COALESCE()` function **replaces NULL values** with `'Not Available'`.

✅ **Output:**

name	hire_date
Alice	2021-03-15
Bob	2020-07-22
Charlie	2023-01-10
Diana	2019-11-05
Eve	2022-09-30

Frank	Not Available

Key Takeaways

✅ `IS NULL` checks for missing (NULL) values.
✅ `IS NOT NULL` filters out NULL values.
✅ **Use** `COALESCE()` to replace NULL with a default value.
✅ **NULL is not the same as an empty string (`''`) or zero (`0`).**
✅ `NULL` **values don't work with = or** `!=` (e.g., `WHERE hire_date = NULL` **won't work!**).

Set Operators in SQL with Examples (`UNION`, `UNION ALL`, `INTERSECT`, `EXCEPT`)

Set operators are used to **combine results** from two or more SELECT queries.

---

1. `UNION` (Removes Duplicates)

📌 **Combine IT and HR employees into one list (without duplicates).**

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION

SELECT name, department FROM employees WHERE department = 'HR';
```

🔹 **Explanation:**

- Combines IT and HR department employees.
- **Removes duplicates** automatically.

✅ **Output:**

name	department
Alice	IT
Charlie	IT
Bob	HR
Eve	HR

---

2. `UNION ALL` (Keeps Duplicates)

📌 **Same as `UNION`, but keeps duplicates.**

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION ALL

SELECT name, department FROM employees WHERE department = 'HR';
```

✅ **Output (with duplicates if any exist):**

name	department

Alice	IT
Charlie	IT
Bob	HR
Eve	HR

---

3. `INTERSECT` (Common Records)

📌 **Find employees working in both IT and HR departments.**

```
SELECT name FROM employees WHERE department = 'IT'

INTERSECT

SELECT name FROM employees WHERE department = 'HR';
```

🔹 **Explanation:**

- Returns **only employees who exist in both queries**.
- Some databases (like MySQL) don't support `INTERSECT` directly. You can simulate it with `INNER JOIN`.

✅ **Output:**

name

(No common employees in this case)

---

4. EXCEPT (Records in First Query, But Not in Second)

📌 **Find employees who are in IT but NOT in HR.**

sql

CopyEdit

```sql
SELECT name FROM employees WHERE department = 'IT'

EXCEPT

SELECT name FROM employees WHERE department = 'HR';
```

🔹 **Explanation:**

- Returns employees in IT **but NOT in HR**.

✅ **Output:**

name
Alice
Charlie

---

5. `EXCEPT` with `UNION` Example

📌 **Get employees from IT or HR, but exclude those from Finance.**

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION

SELECT name, department FROM employees WHERE department = 'HR'

EXCEPT

SELECT name, department FROM employees WHERE department =
'Finance';
```

✅ **Output:**

name	department
Alice	IT
Charlie	IT
Bob	HR
Eve	HR

Key Takeaways

✅ **`UNION`** – Combines results **without duplicates**.
✅ **`UNION ALL`** – Combines results **with duplicates**.
✅ **`INTERSECT`** – Returns only **common records**.
✅ **`EXCEPT`** – Returns records from the **first query that are NOT in the second**.

Would you like advanced queries using **set operators with filtering or sorting?** 🚀

Step-by-Step Explanation of Set Operators in SQL

Set operators **combine results** from two or more `SELECT` statements. Here's how each command works internally:

---

1. `UNION` (Combines Results and Removes Duplicates

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION

SELECT name, department FROM employees WHERE department = 'HR';
```

Step-by-Step Execution:

1. **First Query Execution** (**`SELECT name, department FROM employees WHERE department = 'IT'`**)
    - The database fetches all employees from the IT department.

```
Alice    IT

Charlie  IT
```

    -

2. **Second Query Execution** (`SELECT name, department FROM employees WHERE department = 'HR'`)
    - The database fetches all employees from the HR department.

```
Bob     HR

Eve     HR
```

-

3. **Combining Results (Before Removing Duplicates)**

The database **merges** both result sets:
nginx
CopyEdit

```
Alice    IT

Charlie  IT

Bob      HR

Eve      HR
```

-

4. **Removing Duplicates**
    - If there were duplicate values in both queries, `UNION` **removes them automatically**.

Final Output (After Deduplication)

nginx
CopyEdit

```
Alice    IT

Charlie  IT

Bob      HR
```

```
Eve      HR
```

5.

---

2. `UNION ALL` (Combines Results and Keeps Duplicates)

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION ALL

SELECT name, department FROM employees WHERE department = 'HR';
```

Step-by-Step Execution:

1. **Executes the first `SELECT` query** to fetch IT employees.
2. **Executes the second `SELECT` query** to fetch HR employees.
3. **Merges the results WITHOUT removing duplicates**.

Final Output (Duplicates Remain):

```
Alice    IT

Charlie  IT

Bob      HR

Eve      HR
```

4.

🔹 **Difference from `UNION`?**

- **`UNION ALL` is faster** because it **skips duplicate removal**.
- **Duplicates appear if there are any**.

---

3. `INTERSECT` (Finds Common Records in Both Queries)

🔹 **Some databases (like MySQL) do not support `INTERSECT` directly**, but we can simulate it using `INNER JOIN`.

```
SELECT name FROM employees WHERE department = 'IT'

INTERSECT

SELECT name FROM employees WHERE department = 'HR';
```

Step-by-Step Execution:

1. **First Query Execution (`SELECT name FROM employees WHERE department = 'IT'`)**

```
Alice

Charlie
```

    ○

2. **Second Query Execution (`SELECT name FROM employees WHERE department = 'HR'`)**

Fetches HR employees:

```
Bob

Eve
```

    ○

3. **Finding Common Records**

  - The database **compares both result sets** and finds names that exist in both.
  - **In this case, no common names exist**.

Final Output:

```
(No results)
```

4.

📌 **If "Alice" worked in both IT and HR, the output would be:**

nginx

CopyEdit

```
Alice
```

---

4. EXCEPT (Finds Records in First Query but NOT in Second Query)

Query:

sql

CopyEdit

```
SELECT name FROM employees WHERE department = 'IT'

EXCEPT

SELECT name FROM employees WHERE department = 'HR';
```

Step-by-Step Execution:

Executes the first query (`SELECT name FROM employees WHERE department = 'IT'`)

```
Alice

Charlie
```

1.

Executes the second query (`SELECT name FROM employees WHERE department = 'HR'`)

```
Bob

Eve
```

2.
3. **Removing Matching Records**
    - The database **removes any records from the first result set that also appear in the second result set**.
    - Since "Alice" and "Charlie" do NOT appear in the second list, they remain.

Final Output:

```
Alice

Charlie
```

4.

📌 **If Alice were in HR too, she would be removed from the output.**

---

5. `EXCEPT` with `UNION` Example

📌 **Get employees from IT or HR, but exclude Finance employees.**

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION

SELECT name, department FROM employees WHERE department = 'HR'

EXCEPT

SELECT name, department FROM employees WHERE department =
'Finance';
```

Step-by-Step Execution:

`UNION` **(Combines IT and HR employees without duplicates)**

```
Alice    IT

Charlie  IT

Bob      HR

Eve      HR

Diana    Finance
```

1.

`EXCEPT` **(Removes Finance employees from the above list)**

```
Alice    IT

Charlie  IT

Bob      HR

Eve      HR
```

2.

Key Takeaways

✅ **`UNION`** – Combines results **without duplicates**.
✅ **`UNION ALL`** – Combines results **with duplicates**.
✅ **`INTERSECT`** – Returns only **common records**.
✅ **`EXCEPT`** – Returns records from the **first query that are NOT in the second**.

🔹 `UNION ALL` is **faster than `UNION`** because it **skips duplicate checking**.
🔹 `INTERSECT` and `EXCEPT` **work only in databases that support them** (not MySQL).

Would you like examples using real-world scenarios like sales, customers, or orders? 🚀

How `EXCEPT` Works with `UNION` in SQL?

Let's analyze how `EXCEPT` behaves when used with `UNION`. Specifically, does `EXCEPT` check both **name and department** or just one of them?

SQL Query:

sql

CopyEdit

```
SELECT name, department FROM employees WHERE department = 'IT'

UNION

SELECT name, department FROM employees WHERE department = 'HR'

EXCEPT
```

```
SELECT name, department FROM employees WHERE department =
'Finance';
```

Step-by-Step Execution:

Step 1: Execute First Query (IT Department)

```
SELECT name, department FROM employees WHERE department = 'IT';
```

Result Set 1 (IT employees):

```
Alice    IT

Charlie  IT
```

1.

Step 2: Execute Second Query (HR Department)

sql
CopyEdit
```
SELECT name, department FROM employees WHERE department = 'HR';
```

Result Set 2 (HR employees):

nginx
CopyEdit
```
Bob      HR

Eve      HR
```

2.

Step 3: Apply `UNION` (Combine IT and HR without Duplicates)

nginx

CopyEdit

```
Alice    IT

Charlie  IT

Bob      HR

Eve      HR
```

3.

Step 4: Execute `EXCEPT` Query (Finance Employees)

```
SELECT name, department FROM employees WHERE department =
'Finance';
```

Finance Employees (To Be Excluded):

nginx
CopyEdit

```
Diana    Finance
```

4.
5. **Step 5: Apply `EXCEPT`**
    - `EXCEPT` **removes rows that are an exact match** in both **columns (`name` and `department`)**.
    - Since `"Diana Finance"` is **not** in the previous result set, **no records are removed**.

Final Output

nginx

CopyEdit

```
Alice    IT
```

```
Charlie   IT

Bob       HR

Eve       HR
```

Does `EXCEPT` Check Both Name and Department?

Yes! `EXCEPT` compares **both columns** (i.e., `name` and `department`) **together**.

- **If only the `name` matches but the `department` does not**, the record is **not removed**.
- **If both `name` and `department` match**, the record is removed.

Example to Prove This:

If our `Finance` department had an employee also named **Charlie**, like this:

```
Charlie   Finance
```

And our original dataset had:

```
Charlie   IT
```

Then `"Charlie Finance"` **would not** remove `"Charlie IT"`, because `EXCEPT` **checks both columns together**.

Key Takeaways

✅ **EXCEPT checks both columns** (not just one).
✅ If both `name` and `department` match, the record is removed.
✅ If only `name` matches but the `department` is different, the record remains.

Would you like me to demonstrate this with a modified dataset where `EXCEPT` actually removes some records? 🚀

Bitwise Operators in SQL

Bitwise operators perform operations at the **binary level** on integers. These are useful for tasks like permissions, flags, or low-level data manipulation.

---

1. List of Bitwise Operators in SQL

Operator	Description	Example	Result (Binary Calculation)
`&` (Bitwise AND)	Compares bits and returns `1` if both bits are `1`	`SELECT 5 & 3;`	`1` (`0101 & 0011 = 0001`)
`	` (Bitwise OR)	Compares bits and returns `1` if at least one bit is `1`	`SELECT 5
`^` (Bitwise XOR)	Returns `1` if bits are different (`0` if same)	`SELECT 5 ^ 3;`	`6` (`0101 ^ 0011 = 0110`)

`~` (Bitwise NOT)	Flips all bits (1 becomes 0, 0 becomes 1)	`SELECT ~5;`	`-6` (depends on system representation)
`<<` (Left Shift)	Shifts bits left by `n` places (multiplies by `2^n`)	`SELECT 5 << 1;`	`10` (`0101 << 1 = 1010`)
`>>` (Right Shift)	Shifts bits right by `n` places (divides by `2^n`)	`SELECT 5 >> 1;`	`2` (`0010 >> 1 = 0010`)

Operator	**Description**	**Example**	**Output (5, 3)**
`&`	Bitwise AND	`SELECT 5 & 3;`	1
`	`	Bitwise OR	`SELECT 5
`^`	Bitwise XOR	`SELECT 5 ^ 3;`	6
`~`	Bitwise NOT	`SELECT ~5;`	-6
`<<`	Left Shift	`SELECT 5 << 1;`	10

`>>`	Right Shift	`SELECT 5 >> 1;`	2

Would you like more real-world use cases? 🚀

You said:

now do with special opertopr

ChatGPT said:

Special Operators in SQL

Special operators in SQL perform advanced filtering, existence checks, and conditional logic.

---

1. List of Special Operators

Operator	Description	Example
`EXISTS`	Checks if a subquery returns any rows	`WHERE EXISTS (SELECT 1 FROM employees WHERE salary > 50000);`
`DISTINCT`	Returns unique values in a column	`SELECT DISTINCT department FROM employees;`
`CASE`	Implements conditional logic	`SELECT name, salary, CASE WHEN salary > 50000 THEN 'High' ELSE 'Low' END AS salary_level FROM employees;`

---

2. Examples with Explanation

Example 1: `EXISTS` (Checking for Existence in Another Table)

📌 **Find employees who have made at least one sale.**

sql

CopyEdit

```
SELECT name

FROM employees e

WHERE EXISTS (SELECT 1 FROM sales s WHERE s.employee_id = e.id);
```

How it works:

- The **subquery** (`SELECT 1 FROM sales WHERE s.employee_id = e.id`) checks if an employee has a sale.
- `EXISTS` returns **TRUE** if at least one row exists in `sales`.
- If `EXISTS` is **TRUE**, the employee is included in the result.

✅ **Output:**

nginx

CopyEdit

```
name

Alice

Bob

Eve
```

(Only employees with sales are listed.)

Example 2: `DISTINCT` (Getting Unique Values)

📌 **Get a list of unique departments from employees.**

```
SELECT DISTINCT department FROM employees;
```

How it works:

- Removes duplicate department values.

Example 3: `CASE` (Conditional Logic)

📌 **Classify employees based on salary.**

```
SELECT name, salary,

  CASE

    WHEN salary > 60000 THEN 'High Salary'

    WHEN salary BETWEEN 40000 AND 60000 THEN 'Medium Salary'

    ELSE 'Low Salary'

  END AS salary_category

FROM employees;
```

🔍 What is NOT EXISTS?

`NOT EXISTS` returns `TRUE` if the **subquery returns no rows**.
It's typically used to **find records in one table that don't have matching entries in another**.

📘 Syntax:

sql

CopyEdit

```
SELECT * FROM table1

WHERE NOT EXISTS (

  SELECT 1 FROM table2 WHERE table1.id = table2.id

);
```

🎯 Example Scenario:

Let’s say we have two tables:

employees

emp_id	name
1	Alice
2	Bob
3	Charlie

attendances

emp_id	date

1	2025-04-20
3	2025-04-20

❓ Goal:

> Find employees who **did NOT mark attendance** on 2025-04-20

✅ Query Using NOT EXISTS:

sql

CopyEdit

```
SELECT * FROM employees e
WHERE NOT EXISTS (
  SELECT 1 FROM attendances a
  WHERE a.emp_id = e.emp_id
  AND a.date = '2025-04-20'
);
```

✅ Output:

emp_id	**name**

```
2       Bob
```

🧠 **Explanation:**

- It checks for each employee whether there's a matching entry in the attendances table for that date.
- If not found → the employee is returned.

How it works:

- **CASE** checks each salary and assigns a category.
- If `salary > 60000`, output = "**High Salary**".
- If `salary BETWEEN 40000 AND 60000`, output = "**Medium Salary**".
- Otherwise, output = "**Low Salary**".

✅ **Output:**

```
name      salary     salary_category

Alice     50000      Medium Salary

Bob       70000      High Salary

Charlie   40000      Medium Salary

Diana     80000      High Salary

Eve       35000      Low Salary
```

(Each employee is categorized.)

---

3. Summary of Special Operators

Operator	Purpose	Example Usage
`EXISTS`	Checks for existence of related data	`WHERE EXISTS (SELECT 1 FROM sales WHERE employee_id = e.id);`
`DISTINCT`	Removes duplicate values	`SELECT DISTINCT department FROM employees;`
`CASE`	Implements conditional logic	`SELECT name, CASE WHEN salary > 50000 THEN 'High' ELSE 'Low' END FROM employees;`

📘 Use Case With a Table:

Let’s say you have a `students` table:

```
SELECT name, marks, age FROM students WHERE marks > 60 AND age <
20;
```

This will fetch students **with marks over 60 AND under 20 years of age**.

JOINS

Table: Employees

EmpID	EmpName	DeptID
101	Alice	1
102	Bob	2
103	Charlie	3
104	David	NULL

Table: Departments

DeptID	DeptName
1	HR
2	IT
4	Marketing

Absolutely! Let's break down what's *actually happening under the hood* for each join type using the `Employees` and `Departments` tables.

🧩 1. CROSS JOIN

🔧 Process:

- This is like saying:
  *"Combine every row from `Employees` with **every single** row from `Departments`, no conditions at all."*

- It's like a **cartesian product** in math.

🔄 **How it works:**

- 4 `Employees` × 3 `Departments` = **12 combinations**

- Doesn't care about `DeptID`, just blindly pairs each row.

🧮 2. THETA JOIN (using ON condition)

🔧 **Process:**

- This is a **conditional join** where you specify the logic for matching rows.

- In our case:
  "Only join rows where `Employees.DeptID = Departments.DeptID`."

🔄 **How it works:**

- It goes row-by-row and checks:
  "Hey, does this employee's `DeptID` match any department's `DeptID`?"

- If yes 👉 it joins them into a result row.

- If not 👉 it skips that employee.

✅ **Result:**

- Only Alice (DeptID = 1) and Bob (DeptID = 2) have matching departments.

- Charlie (DeptID = 3) and David (NULL) are left out because there's no match.

🧬 3. NATURAL JOIN

🔧 Process:

- It's like a smart theta join. It automatically finds columns with the **same name** in both tables.
- Then it **implicitly** joins using those columns.

Under the hood:

SQL sees that both tables have a `DeptID`, so it assumes you want:

sql
CopyEdit

```
ON Employees.DeptID = Departments.DeptID
```

-

⚠️ Caution:

- If tables share multiple columns with the same name (e.g. both had `Location`), it will join on **all** of them.
- So while natural join is convenient, it can be **risky** if you're not careful.

👀 Summary Table

Join Type	Matches?	Condition Required?	Rows Included

Cross Join	No matching needed	❌ No	All combinations (cartesian)
Theta Join	Custom condition (`=`)	✅ Yes	Only matched rows based on logic
Natural Join	Auto match by column name	❌ No (implicit)	Only matched rows

---

Want to go next into **LEFT / RIGHT / FULL OUTER JOINs**? Those show what happens to unmatched data.

🧩 1. CROSS JOIN

```
SELECT * FROM Employees CROSS JOIN Departments;
```

📋 **Output:**

EmpID	EmpName	DeptID (Emp)	DeptID (Dept)	DeptName
101	Alice	1	1	HR
101	Alice	1	2	IT

101	Alice	1	4	Marketing
102	Bob	2	1	HR
102	Bob	2	2	IT
102	Bob	2	4	Marketing
103	Charlie	3	1	HR
103	Charlie	3	2	IT
103	Charlie	3	4	Marketing
104	David	NULL	1	HR
104	David	NULL	2	IT
104	David	NULL	4	Marketing

🧮 2. THETA JOIN (Using ON condition)

This is usually a condition-based join. We'll use `Employees.DeptID = Departments.DeptID`.

```
SELECT * FROM Employees E JOIN Departments D ON E.DeptID =
D.DeptID;
```

📋 **Output:**

EmpID	EmpName	DeptID	DeptID	DeptName
101	Alice	1	1	HR
102	Bob	2	2	IT

Note: Only rows with matching `DeptID` are returned.

🧬 3. NATURAL JOIN

A **natural join** automatically joins using columns with the same name (`DeptID` in this case).

```
SELECT * FROM Employees NATURAL JOIN Departments;
```

📋 **Output:**

EmpID	EmpName	DeptID	DeptName

101	Alice	1	HR
102	Bob	2	IT

Note: Same result as theta join here because both used `DeptID` as the common column.

1. **LEFT OUTER JOIN**
2. **RIGHT OUTER JOIN**
3. **FULL OUTER JOIN**

We’ll use the same `Employees` and `Departments` tables.

🌐 1. LEFT OUTER JOIN

```
SELECT * FROM Employees E LEFT JOIN Departments D ON E.DeptID =
D.DeptID;
```

🔧 What it does:

- Returns **all rows** from `Employees`
- Adds matching `Departments` info when available
- If no match, fills with **NULLs**

📋 Output:

EmpID	EmpName	DeptID	DeptID	DeptName
101	Alice	1	1	HR
102	Bob	2	2	IT
103	Charlie	3	NULL	NULL
104	David	NULL	NULL	NULL

✅ **Explanation**:

- Charlie's `DeptID = 3` doesn't exist in Departments → NULLs
- David has `NULL` DeptID → can't match anything → NULLs

🔁 2. RIGHT OUTER JOIN

📄 **Query:**

sql

CopyEdit

```
SELECT *

FROM Employees E

RIGHT JOIN Departments D
```

```
ON E.DeptID = D.DeptID;
```

🔧 What it does:

- Returns **all rows** from `Departments`
- Adds matching `Employees` info when available
- If no match, fills with **NULLs**

📋 Output:

EmpID	EmpName	DeptID	DeptID	DeptName
101	Alice	1	1	HR
102	Bob	2	2	IT
NULL	NULL	NULL	4	Marketing

✅ **Explanation**:

- DeptID 4 (Marketing) has no employee → shows up with NULL employee data.

🌍 3. FULL OUTER JOIN

```
SELECT * FROM Employees E FULL OUTER JOIN Departments D

ON E.DeptID = D.DeptID;
```

🔧 What it does:

- Returns **all rows** from both tables
- Matches where it can
- Fills in NULLs where there's no match

📋 Output:

EmpID	EmpName	DeptID	DeptID	DeptName
101	Alice	1	1	HR
102	Bob	2	2	IT
103	Charlie	3	NULL	NULL
104	David	NULL	NULL	NULL
NULL	NULL	NULL	4	Marketing

✅ Explanation:

- Combines left + right outer join
- Everyone is included from both sides

UP TO HERE ALL JOINS USE ONLY TWO TABLES

A subquery (or nested query) is a query inside another query.

It usually helps answer questions like:

- **“Get all employees who earn more than the average salary”**
- **“Get departments that don’t have any employees”**
- **“Find the employee with the highest salary”**

Subqueries can be used in:

- `SELECT`
- `FROM`
- `WHERE`
- `HAVING`

🔄 Types of Subqueries

Type	Description
Scalar Subquery	Returns a single value (1 row, 1 column)

Row Subquery	**Returns one row of multiple columns**
Table Subquery	**Returns a full table (can be used like a normal table)**
Correlated Subquery	**Depends on outer query for each row**

📌 **Examples with Sample Table:** `Employees`

EmpID	EmpName	DeptID	Salary
101	Alice	1	50000
102	Bob	2	60000
103	Charlie	1	70000
104	David	3	45000

🔍 Question: Get employees who earn more than the average salary.

```
SELECT * FROM Employees WHERE Salary > (

    SELECT AVG(Salary) FROM Employees );
```

💡 **Explanation:**

- The inner query gets average salary: `56250`
- Outer query returns Alice, Charlie, and Bob

🔍 **Question: Show each employee with the overall average salary.**

```
SELECT EmpName, Salary, (SELECT AVG(Salary) FROM Employees) AS AvgSalary FROM Employees;
```

🔍 **Question: Show employees from a temporary "high earners" table.**

```
SELECT * FROM ( SELECT * FROM Employee WHERE Salary > 55000) AS HighEarners;
```

💡 **Explanation:**

- Inner query creates a temporary table of high earners
- Outer query just selects from it

✅ 4. Correlated Subquery

🔍 **Question: Get employees who earn more than anyone else in their department.**

```
SELECT * FROM Employees E1 WHERE Salary > (SELECT MAX(Salary)

   FROM Employees E2 WHERE E1.DeptID = E2.DeptID AND E1.EmpID <> E2.EmpID );
```

💡 **Explanation:**

- Inner query changes for each row
- Filters employees who have the highest salary in their own department

---

📝 **Practice Questions:**

1. **Get employees who earn more than the average salary.**
2. **List departments that have no employees (use subquery with `NOT IN`).**
3. **Show names of employees who earn less than Bob.**
4. **Find employee(s) with the highest salary.**
5. **Get employees whose salary is equal to the maximum in their department.**

Sample Data

1. Employees Table

employee_id	employee_name	department_id
1	Alice	101
2	Bob	102

2. Departments Table

department_id	department_name

101	Engineering
102	Marketing

3. ProjectAssignments Table

assignment_id	employee_id	project_id	assignment_start_date
1	1	201	2023-01-15
2	2	202	2023-02-01

4. Projects Table

project_id	project_name
201	Project X
202	Project Y

5. Salaries Table

salary_id	employee_id	salary_amount
1	1	75000
2	2	65000

SQL Query

sql

```
SELECT
  e.employee_name,
  d.department_name,
  p.project_name,
  pa.assignment_start_date,
  s.salary_amount
FROM Employees e
INNER JOIN Departments d ON e.department_id = d.department_id
INNER JOIN ProjectAssignments pa ON e.employee_id = pa.employee_id
INNER JOIN Projects p ON pa.project_id = p.project_id
INNER JOIN Salaries s ON e.employee_id = s.employee_id;
```

Output

employee_name	department_name	project_name	assignment_start_date	salary_amount
Alice	Engineering	Project X	2023-01-15	75000

Bob	Marketing	Project Y	2023-02-01	65000

You can copy and paste the above content directly into Google Docs, and it should maintain its formatting well!

1. Goal of the Query

This query retrieves a list of employees, their department, the projects they're assigned to, and their salaries. It combines data from five tables to create a comprehensive report.

---

2. Breakdown of Each Command

a. `FROM Employees e`

- **Purpose: Start with the `Employees` table (aliased as `e`) as the primary source of data.**
- **Why: The query focuses on employees, so this table acts as the "anchor" for joining other related tables.**

---

b. `INNER JOIN Departments d ON e.department_id = d.department_id`

- **Purpose: Link employees to their departments.**
- **Why:**
    - **Every employee belongs to a department (assumed by the `INNER JOIN`).**
    - **If an employee has no department (`department_id` is `NULL`), they are excluded from results.**
    - **Columns like `department_name` come from this join.**

---

c. `INNER JOIN ProjectAssignments pa ON e.employee_id = pa.employee_id`

- **Purpose: Connect employees to their project assignments.**
- **Why:**
    - **Each employee may be assigned to one or more projects (via the `ProjectAssignments` table).**
    - **`INNER JOIN` ensures only employees with project assignments are included.**

    - If an employee has no projects, they are excluded from results.

---

d. `INNER JOIN Projects p ON pa.project_id = p.project_id`

- Purpose: Fetch details about the projects employees are assigned to.
- Why:
    - The `ProjectAssignments` table only contains `project_id`, so we need the `Projects` table to get meaningful names (e.g., `project_name`).
    - `INNER JOIN` ensures only valid projects (with matching `project_id`) are included.

---

e. `INNER JOIN Salaries s ON e.employee_id = s.employee_id`

- Purpose: Retrieve salary information for each employee.
- Why:
    - Assumes every employee has a salary record.
    - If an employee has no salary entry (e.g., `Salaries` table lacks their `employee_id`), they are excluded from results.

---

3. Why `INNER JOIN` Is Used Everywhere

- Mandatory Relationships:
  This query assumes strict business rules:
    - All employees must belong to a department.
    - All employees must have at least one project assignment.
    - All employees must have a salary record.
    - `INNER JOIN` enforces these rules by excluding rows with missing relationships.

SQL GROUP BY and HAVING Guide

1. GROUP BY Clause

Purpose

Groups rows with the same values in specified columns into summary rows.

Syntax

```
sql
SELECT column1, aggregate_function(column2)
FROM table
GROUP BY column1;
```

2. HAVING Clause

Purpose

Filters groups created by `GROUP BY` based on a condition (like `WHERE` but for groups).

Key Difference from WHERE

- **`WHERE` filters rows before grouping.**
- **`HAVING` filters groups after aggregation.**

Syntax

```
sql
SELECT column1, aggregate_function(column2)
FROM table
GROUP BY column1
HAVING condition;
```

3. Aggregate Functions

Function	Description	Example

`COUNT()`	Counts the number of rows.	`COUNT(order_id)`
`SUM()`	Sums values in a numeric column.	`SUM(amount)`
`AVG()`	Calculates the average.	`AVG(salary)`
`MIN()`	Returns the smallest value.	`MIN(price)`
`MAX()`	Returns the largest value.	`MAX(quantity)`
`STDDEV()`	Standard deviation of values.	`STDDEV(sales)`
`VARIANCE()`	Variance of values.	`VARIANCE(revenue)`

4. Examples with Data, Queries, and Outputs

Sample Data: `sales` Table

sale_id	product	amount	region
1	Laptop	1000	East

2	Phone	500	East
3	Laptop	1200	West
4	Monitor	300	West
5	Phone	450	East

Example 1: `COUNT` with `GROUP BY` and `HAVING`

Goal: Find products sold more than once.

Query:

```
sql
SELECT product, COUNT(*) AS total_sales
FROM sales
GROUP BY product
HAVING COUNT(*) > 1;
```

Output:

product	total_sales
Laptop	2
Phone	2

Explanation:

- Groups data by `product`.
- `COUNT(*)` calculates total sales per product.
- `HAVING` filters out products with ≤ 1 sale.

Example 2: `SUM` with `GROUP BY` and `HAVING`

Goal: Find regions with total sales over $1000.
Query:
sql

```
SELECT region, SUM(amount) AS total_revenue
FROM sales
GROUP BY region
HAVING SUM(amount) > 1000;
```

Output:

region	total_revenue
East	1950
West	1500

Explanation:

- Groups data by `region`.
- `SUM(amount)` calculates total revenue per region.
- `HAVING` filters regions with total revenue ≤ $1000.

Example 3: `AVG` with `GROUP BY` and `HAVING`

Goal: Find products with an average sale price ≥ $700.
Query:
sql

```
SELECT product, AVG(amount) AS avg_price
FROM sales
GROUP BY product
```

```
HAVING AVG(amount) >= 700;
```

Output:

product	avg_price
Laptop	1100

Explanation:

- Groups data by `product`.
- `AVG(amount)` calculates average price per product.
- `HAVING` filters products with average price < $700.

Example 4: `MIN/MAX` with `GROUP BY` and `HAVING`

Goal: Find regions where the smallest sale is < $400.

Query:

```
sql
SELECT region, MIN(amount) AS min_sale
FROM sales
GROUP BY region
HAVING MIN(amount) < 400;
```

Output:

region	min_sale
West	300

Explanation:

- Groups data by `region`.
- `MIN(amount)` finds the smallest sale in each region.
- `HAVING` filters regions where the smallest sale ≥ $400.

Example 5: Combining Multiple Aggregates

Goal: List products with total sales > $1000 and max sale > $1000.
Query:
sql

```
SELECT product,
       SUM(amount) AS total_sales,
       MAX(amount) AS max_sale
FROM sales
GROUP BY product
HAVING SUM(amount) > 1000 AND MAX(amount) > 1000;
```

Output:

product	total_sales	max_sale
Laptop	2200	1200

Explanation:

- **Groups data by `product`.**
- `SUM(amount)` **and** `MAX(amount)` **calculate total and max sales.**
- `HAVING` **filters products that meet both conditions.**

5. Key Rules

1. **Columns in `SELECT`:**
    - **Must be in the `GROUP BY` clause or used in an aggregate function.**
    - **Example:** `SELECT region, SUM(amount)` **is valid if** `region` **is in** `GROUP BY`**.**
2. **Order of Execution:**
    - `WHERE` → `GROUP BY` → `HAVING` → `ORDER BY`.
3. **Use Cases:**
    - `GROUP BY`**: Summarize data (e.g., sales by region).**
    - `HAVING`**: Filter aggregated results (e.g., total sales > $1000).**

6. HAVING vs WHERE

HAVING Clause	WHERE Clause
Filters groups after aggregation.	Filters rows before grouping.
Used with aggregate functions.	Cannot use aggregate functions.
Executed after `GROUP BY`.	Executed before `GROUP BY`.

Copy and paste this entire structure into Google Docs. The markdown tables and code blocks will retain their formatting. Let me know if you need further adjustments!

SQL Guide: GROUP BY, Aggregates, and ORDER BY

Dataset: `sales` table

sale_id	product	amount	region	sale_date
1	Laptop	1000	East	2023-01-01
2	Phone	500	East	2023-01-02
3	Laptop	1200	West	2023-01-03

4	Monitor	300	West	2023-01-04
5	Phone	450	East	2023-01-05
6	Laptop	900	East	2023-01-06
7	Monitor	350	East	2023-01-07

Example 1: GROUP BY + SUM + ORDER BY

Goal

Total sales per product, ordered by highest sales first.

SQL Query

```sql
SELECT
  product,
  SUM(amount) AS total_sales
FROM sales
GROUP BY product
ORDER BY total_sales DESC;
```

Output

product	total_sales

Laptop	3100
Phone	950
Monitor	650

Example 2: GROUP BY + AVG + HAVING + ORDER BY

Regions with average sale amount > $400, ordered by region.

SQL Query

sql

```sql
SELECT
  region,
  AVG(amount) AS avg_sale
FROM sales
GROUP BY region
HAVING AVG(amount) > 400
ORDER BY region;
```

Output

region	avg_sale
East	575.00

West	750.00

1. Tables: Add gridlines for clarity.

Key Concepts

Concept	Description
GROUP BY	Groups rows with identical values in specified columns.
Aggregates	Functions like `SUM`, `AVG`, `COUNT` to compute summary statistics.
HAVING	Filters groups after aggregation (used with `GROUP BY`).
ORDER BY	Sorts results in ascending (`ASC`) or descending (`DESC`) order.

LIMIT and OFFSET in MySQL

In MySQL, `LIMIT` and `OFFSET` are used to control the number of rows returned by a query. They are commonly used in pagination, where you fetch a subset of results instead of retrieving all data at once.

1. LIMIT

The `LIMIT` clause specifies the maximum number of rows to return.

Syntax:

sql
CopyEdit
```
SELECT column_names FROM table_name LIMIT number_of_rows;
```

Example:

sql
CopyEdit
```
SELECT * FROM customers LIMIT 5;
```

✅ **Explanation:**

- This query returns only the first 5 rows from the `customers` table.

---

2. LIMIT with OFFSET

- The `OFFSET` clause skips a specified number of rows before starting to return rows.
- It is useful for pagination.

Syntax:

sql
CopyEdit
```
SELECT column_names FROM table_name LIMIT number_of_rows OFFSET
start_row;


SELECT column_names FROM table_name LIMIT start_row,
number_of_rows;
```

> Note: The second syntax (LIMIT start, count) is a MySQL-specific way of using `LIMIT` and `OFFSET`.

Example:

sql
CopyEdit
```
SELECT * FROM customers LIMIT 5 OFFSET 10;
```

✅ **Explanation:**

- This query skips the first 10 rows and then fetches the next 5 rows.

---

3. Pagination Example

Scenario: Fetch customer records in pages of 5 rows each

Page	SQL Query
Page 1	`SELECT * FROM customers LIMIT 5 OFFSET 0;`
Page 2	`SELECT * FROM customers LIMIT 5 OFFSET 5;`
Page 3	`SELECT * FROM customers LIMIT 5 OFFSET 10;`

✅ **Explanation:**

- Page 1 fetches rows 1-5 (OFFSET 0, LIMIT 5)
- Page 2 fetches rows 6-10 (OFFSET 5, LIMIT 5)
- Page 3 fetches rows 11-15 (OFFSET 10, LIMIT 5)

---

4. Alternative: Using LIMIT with Start and Count

sql
CopyEdit
```
SELECT * FROM customers LIMIT 10, 5;
```

✅ **Explanation:**

- This is the same as `LIMIT 5 OFFSET 10`, meaning it skips 10 rows and fetches 5 rows.

---

5. Sorting with LIMIT & OFFSET

Before paginating, always use `ORDER BY` to ensure a consistent order of results.

sql
CopyEdit

```
SELECT * FROM customers ORDER BY name ASC LIMIT 5 OFFSET 10;
```

✅ **Best Practice:** Sorting avoids unexpected changes in data order when paginating.

---

6. Real-World Use Case: Fetching Products for an E-Commerce Website

Suppose you have a products table and want to display 10 products per page.

sql
CopyEdit

```
SELECT * FROM products ORDER BY product_name ASC LIMIT 10 OFFSET
20;
```

✅ **Explanation:**

- This fetches Page 3 of results (assuming 10 results per page).
- OFFSET 20 means skipping first 20 products and fetching next 10.

Summary

Clause	Purpose
LIMIT X	**Retrieves only X rows**
LIMIT X OFFSET Y	**Skips Y rows and fetches X rows**
LIMIT Y, X	**Another way to write `LIMIT X OFFSET Y` (MySQL-specific)**

Would you like an example with real sample data? 😊

Here's a structured exploration of LIMIT and OFFSET in combination with aggregate functions, GROUP BY, and ORDER BY using practical examples:

Sample Data: sales Table

sale_id	product	amount	region	customer_id
1	Laptop	1000	East	101
2	Phone	500	East	102
3	Laptop	1200	West	103
4	Monitor	300	West	104
5	Phone	450	East	101

6	Laptop	900	East	102
7	Monitor	350	East	103

Example 1: Top 3 Products by Total Sales

Goal: Show the top 3 products by total sales.

```
sql
SELECT product, SUM(amount) AS total_sales
FROM sales
GROUP BY product
ORDER BY total_sales DESC
LIMIT 3;
```

Output:

product	total_sales
Laptop	3100
Phone	950
Monitor	650

Explanation:

- GROUP BY product groups sales by product.
- SUM(amount) calculates total sales per product.
- ORDER BY total_sales DESC sorts products from highest to lowest.
- LIMIT 3 returns only the top 3 rows.

Example 2: Second-Highest Average Sale by Region

Goal: Find the region with the second-highest average sale.

```
sql
SELECT region, AVG(amount) AS avg_sale
FROM sales
GROUP BY region
ORDER BY avg_sale DESC
LIMIT 1 OFFSET 1;
```

Output:

region	avg_sale
East	575.00

Explanation:

- GROUP BY region groups sales by region.
- AVG(amount) calculates average sales per region.
- ORDER BY avg_sale DESC sorts regions by highest average first.
- LIMIT 1 OFFSET 1 skips the first row (West: 750) and returns the second row (East: 575).

Example 3: Paginate Customers by Number of Orders

Goal: Show the 3rd and 4th customers by total orders (pagination).

sql

```
SELECT customer_id, COUNT(*) AS total_orders
FROM sales
GROUP BY customer_id
ORDER BY total_orders DESC
LIMIT 2 OFFSET 2;
```

Output:

customer_id	total_orders
103	2
104	1

Explanation:

- GROUP BY customer_id groups sales by customer.
- COUNT(*) counts orders per customer.
- ORDER BY total_orders DESC sorts customers by most orders first.
- LIMIT 2 OFFSET 2 skips the first 2 rows and returns the next 2.

Example 4: Top 2 Regions by Total Sales (Excluding First)

Goal: Skip the top region and show the next 2 (if they exist).

sql

```
SELECT region, SUM(amount) AS total_sales
FROM sales
```

```
GROUP BY region
ORDER BY total_sales DESC
LIMIT 2 OFFSET 1;
```

Output:

region	total_sales
East	2700

Explanation:

- GROUP BY region groups sales by region.
- SUM(amount) calculates total sales per region.
- ORDER BY total_sales DESC sorts regions by highest sales first.
- LIMIT 2 OFFSET 1 skips the first region (West: 1500) and returns the next region (East: 2700).
  (Only one region remains after OFFSET 1.)

Example 5: Products with the Lowest Single Sale

Goal: Find products with the smallest single sale, returning only 1 result.

```
sql
SELECT product, MIN(amount) AS min_sale
FROM sales
GROUP BY product
ORDER BY min_sale ASC
LIMIT 1;
```

Output:

product	min_sale
Monitor	300

Explanation:

- GROUP BY product groups sales by product.
- MIN(amount) finds the smallest sale per product.
- ORDER BY min_sale ASC sorts products from smallest to largest minimum sale.
- LIMIT 1 returns the product with the smallest sale.

Key Takeaways

- LIMIT: Restricts the number of rows returned. Often used with ORDER BY to get "Top N" or "Bottom N" results.
- OFFSET: Skips a specified number of rows. Useful for pagination (e.g., page 2 of results).
- Order of Execution: GROUP BY → Aggregations → ORDER BY → LIMIT/OFFSET.
- Common Use Cases: Paginating aggregated results, finding ranked results.

Pitfalls to Avoid

- Missing ORDER BY: Without sorting, LIMIT/OFFSET may return random rows.
- Large OFFSET: Skipping many rows can be slow; use keyset pagination for large datasets.
- Ambiguous Groups: Ensure GROUP BY includes all non-aggregated columns in SELECT.

Here’s a structured guide on using the CASE statement in MySQL, complete with sample data, SQL queries, outputs, and explanations:

Practical Examples of Using the CASE Statement in MySQL

Sample Data: `students` Table

student_id	name	marks	gender
1	Alice	85	F
2	Bob	45	M
3	Charlie	92	M
4	Diana	72	F
5	Eva	60	F

Example 1: Categorize Grades

Goal: Assign grades based on marks.

SQL Query

```
sql
SELECT
  name,
  marks,
  CASE
    WHEN marks >= 90 THEN 'A'
    WHEN marks >= 80 THEN 'B'
    WHEN marks >= 60 THEN 'C'
    ELSE 'Fail'
  END AS grade
FROM students;
```

Output:

name	marks	grade
Alice	85	B
Bob	45	Fail
Charlie	92	A
Diana	72	C
Eva	60	C

Explanation:

The CASE statement evaluates marks and assigns grades conditionally.

Example 2: Conditional Aggregation

Goal: Count students by gender and pass/fail status.

SQL Query

`sql`

```
SELECT
  gender,
  COUNT(CASE WHEN marks >= 60 THEN 1 ELSE NULL END) AS passed,
  COUNT(CASE WHEN marks < 60 THEN 1 ELSE NULL END) AS failed
FROM students
GROUP BY gender;
```

Output:

gender	passed	failed
F	3	0
M	1	1

Explanation:

COUNT ignores NULL, so only valid conditions are counted.

- **Female students (F): 3 passed, 0 failed.**
- **Male students (M): 1 passed (Charlie), 1 failed (Bob).**

Example 3: Update Records Conditionally

Goal: Increase marks by 5 for students scoring below 50.

SQL Query

`sql`

```
UPDATE students
SET marks = CASE
  WHEN marks < 50 THEN marks + 5
  ELSE marks
END;
```

Result:

- Bob’s marks become 50 (from 45).
- Other students’ marks remain unchanged.

Example 4: Order by Custom Priority

Goal: Sort students with marks > 80 first, then others.

SQL Query

```
sql
SELECT name, marks
FROM students
ORDER BY
  CASE
    WHEN marks > 80 THEN 1
    ELSE 2
  END;
```

Output:

name	marks
Alice	85

Charlie	92
Bob	45
Diana	72
Eva	60

Explanation:

Students with marks > 80 (Alice, Charlie) appear first.

Example 5: Pivot Data with CASE

Goal: Show total marks by gender and category (High/Low).

SQL Query

```
sql
SELECT
  gender,
  SUM(CASE WHEN marks >= 70 THEN marks ELSE 0 END) AS high_marks,
  SUM(CASE WHEN marks < 70 THEN marks ELSE 0 END) AS low_marks
FROM students
GROUP BY gender;
```

Output:

gender	high_marks	low_marks

F	157	60
M	92	45

Explanation:

- **Female students: Alice (85) + Diana (72) = 157; Eva (60) = 60.**
- **Male students: Charlie (92); Bob (45) = 45.**

Key Notes

Syntax:

```
sql
CASE
   WHEN condition1 THEN result1
   WHEN condition2 THEN result2
   ...
   ELSE default_result
END
```

Use Cases:

- Categorizing data.
- Conditional updates.
- Custom sorting.
- Pivoting rows to columns.

ELSE Clause:

Optional but recommended to handle unmatched cases.
Feel free to copy and paste this content into Google Docs! Let me know if you'd like more advanced examples or any other modifications! 😊

SQL Command to Determine Triangle Type

Sample Data: `triangles` Table

a	b	c
3	4	5
5	5	5
5	5	7
2	3	4
1	2	3

SQL Command

sql

```
SELECT
  a, b, c,
  -- Outer CASE: Check if it's a valid triangle
  CASE
    WHEN (a + b > c) AND (a + c > b) AND (b + c > a) THEN
      -- Inner CASE: Determine the type of triangle
      CASE
        WHEN a = b AND b = c THEN 'Equilateral'
        WHEN a = b OR b = c OR a = c THEN 'Isosceles'
        WHEN POWER(a, 2) + POWER(b, 2) = POWER(c, 2) OR
```

```
             POWER(b, 2) + POWER(c, 2) = POWER(a, 2) OR
             POWER(a, 2) + POWER(c, 2) = POWER(b, 2) THEN
'Right-Angled'
        ELSE 'Scalene'
      END
    ELSE 'Not a Triangle'
  END AS triangle_type
FROM triangles;
```

Output

a	b	c	triangle_type
3	4	5	Right-Angled
5	5	5	Equilateral
5	5	7	Isosceles
2	3	4	Scalene
1	2	3	Not a Triangle

Explanation

Outer CASE:

- **Checks if the sides satisfy the triangle inequality theorem:**

  - $a+b>c$
  - $a+b>c$
  - $a+c>b$
  - $a+c>b$
  - $b+c>a$
  - $b+c>a$
- If not satisfied, returns Not a Triangle.

Inner CASE:

- Equilateral: All sides are equal
- $(a=b=c)$
- $(a=b=c)$.
- Isosceles: Exactly two sides are equal
- $(a=b$
- $(a=b$,
- $b=c$
- $b=c$, or
- $a=c)$
- $a=c)$.
- Right-Angled: Satisfies the Pythagorean theorem (e.g.,
- $a2+b2=c2$
- Scalene: No sides are equal and not right-angled.

Key Notes

1. Order of Conditions: The CASE statement evaluates conditions sequentially. For example, Equilateral is checked before Isosceles because all equilateral triangles are technically isosceles, but we want the more specific classification first.
2. Right-Angled Check: The Pythagorean theorem is checked for all permutations of sides (since any side could be the hypotenuse).
3. Scalene: Falls under "none of the above" after checking other types.

Feel free to copy and paste this content into Google Docs! Let me know if you’d like to test with more examples or need any modifications! 😊