SQL Notes

LinkedIn : Shambhuling Konapure

Data, Database, and DBMS

1. Data

- **Definition:** Any information that can be stored, processed, or analyzed. It is systematic record of a specific quantity.
- **Types:**
    - Structured (e.g., tables, spreadsheets)
    - Unstructured (e.g., images, videos, text)
    - Semi-structured (e.g., JSON, XML)
- **Example:**
    - A customer's name, age, and purchase history in an e-commerce store.

2. Database

- **Definition:** A structured collection of data stored electronically.
- **Purpose:** Efficient storage, retrieval, and management of data.
- **Types:**
    - **Relational Databases:** MySQL, PostgreSQL
    - **NoSQL Databases:** MongoDB, Firebase
- **Example:**
    - A bank’s database storing customer details, account balances, and transaction history.

3. DBMS (Database Management System)

- **Definition:** Software that interacts with the database to manage data efficiently. It is middle
- **Functions:**
  - Data storage, retrieval, and updating
  - Security and access control
  - Backup and recovery
- **Types:**
  - **Relational DBMS (RDBMS):** MySQL, Oracle, SQL Server
  - **Non-Relational DBMS:** MongoDB, Cassandra
- **Example:**
  - A **MySQL** server managing an e-commerce website's product catalog and orders.

Types of DBMS

1. Relational Database Management System (RDBMS)

- Data is stored in structured tables with rows and columns.
- Uses SQL for querying and managing data.
- Follows ACID (Atomicity, Consistency, Isolation, Durability) properties.
- Supports **relationships** between tables using primary and foreign keys.
- Suitable for structured and complex transactional data.
- Examples: **MySQL, PostgreSQL, Oracle, SQL Server.**
- Example Use Cases **:** Banking systems, ERP software, e-commerce platforms.

2. Non-Relational Database Management System (Non-RDBMS / NoSQL)

- Data is stored in flexible formats like JSON, key-value pairs, graphs, or documents.
- Does **not** follow a strict schema like RDBMS.
- Scalable for large datasets and real-time applications.
- Uses BASE (Basically Available, Soft-state, Eventually consistent) properties.
- Faster read/write operations compared to RDBMS for certain use cases.
- Examples: **MongoDB (Document-based), Redis (Key-Value), Neo4j (Graph-based).**
- Example Use Cases **:** Social media analytics, real-time recommendations, big data processing.

SQL (Structured Query Language)

What is SQL?

- SQL stands for **Structured Query Language**, used for managing and querying relational databases.
- It is a **declarative language**, meaning you specify **what** you want to do rather than **how** to do it.
- SQL allows users to **insert, update, delete, and retrieve** data from a database.
- SQL follows **ACID properties** (Atomicity, Consistency, Isolation, Durability) for transaction management.
- It is a **standardized language**, supported by databases like MySQL, PostgreSQL, SQL Server, and Oracle.
- SQL is divided into multiple categories: **DDL (Data Definition Language), DML (Data Manipulation Language), DCL (Data Control Language), TCL (Transaction Control Language).**
- SQL is **case-insensitive** for keywords (e.g., `SELECT` is the same as `select`), but table/column names may be case-sensitive depending on the database.
- It is widely used in **web applications, data analytics, and enterprise systems** for structured data management.

History of SQL

- **1970s:** Developed by **IBM** researchers Donald D. Chamberlin and Raymond F. Boyce as SEQUEL (Structured English Query Language).
- **1979: Oracle** became the first company to commercialize SQL-based relational database systems.
- **1986:** SQL was standardized by **ANSI (American National Standards Institute)** as SQL-86.
- **Present:** SQL has evolved with various enhancements and is used globally in almost all relational database systems.

MySQL

What is MySQL?

- MySQL is an **open-source relational database management system (RDBMS)** developed by **Oracle Corporation**.
- It is based on **SQL (Structured Query Language)** and is used for managing structured data.
- MySQL is widely used in **web applications, data warehousing, e-commerce, and enterprise solutions**.
- It supports **cross-platform compatibility** and integrates with various programming languages like Python, Java, and PHP.

Features of MySQL (In Short)

- **Open Source:** Free to use and customizable under the GNU General Public License (GPL).
- **High Performance:** Optimized for speed and reliability, even with large datasets.
- **Scalability:** Can handle **small to large-scale** applications efficiently.
- **Security:** Provides strong data protection with **user authentication and encryption**.
- **Multi-User Access:** Supports concurrent connections and role-based access.

- **Replication & Clustering:** Enables data redundancy and high availability.
- **Transaction Support:** Supports **ACID transactions** for reliable data management.
- **Cross-Platform Compatibility:** Works on **Windows, Linux, and macOS**.

MySQL Data Types

MySQL provides various **data types** to store different kinds of values. They are categorized into:

1. Numeric Data Types

- **TINYINT (1 byte):** Stores small integers (-128 to 127 or 0 to 255 unsigned).
- **SMALLINT (2 bytes):** Stores values from -32,768 to 32,767.
- **MEDIUMINT (3 bytes):** Stores values from -8,388,608 to 8,388,607.
- **INT (INTEGER) (4 bytes):** Stores values from -2,147,483,648 to 2,147,483,647.
- **BIGINT (8 bytes):** Stores large values up to 9 quintillion.
- **DECIMAL(M, D) / NUMERIC(M, D) (Varies):** Stores exact numeric values (M = total digits, D = decimals).
- **FLOAT (4 bytes):** Stores floating-point numbers with single precision.
- **DOUBLE (8 bytes):** Stores floating-point numbers with double precision.
- **BIT (1+ bytes):** Stores binary values (e.g., `BIT(8)` stores 8-bit binary).

Signed & Unsigned Data Types

- **Signed Data Type:** Supports both negative and positive values (e.g., `INT` ranges from -2,147,483,648 to 2,147,483,647).
- **Unsigned Data Type:** Only supports positive values, effectively doubling the maximum positive range (e.g., `UNSIGNED INT` ranges from 0 to 4,294,967,295).

2. String (Character) Data Types

- **CHAR(n) (1 to 255 bytes):** Fixed-length string (e.g., `CHAR(10)` always stores 10 characters).
- **VARCHAR(n) (1 to 65,535 bytes):** Variable-length string (e.g., `VARCHAR(50)`).
- **TEXT (0 to 65,535 bytes):** Large text data.
  - **TINYTEXT (255 bytes)**
  - **TEXT (64 KB)**
  - **MEDIUMTEXT (16 MB)**
  - **LONGTEXT (4 GB)**
- **BLOB (0 to 65,535 bytes):** Stores binary data (images, files).

    - **TINYBLOB (255 bytes)**
    - **BLOB (64 KB)**
    - **MEDIUMBLOB (16 MB)**
    - **LONGBLOB (4 GB)**
- **ENUM (1 or 2 bytes):** Stores one value from a predefined list (e.g., `'Male'`, `'Female'`).
- **SET (1 to 8 bytes):** Stores multiple values from a predefined list.

3. Date & Time Data Types

- **DATE (3 bytes):** Stores date (`YYYY-MM-DD`, e.g., `2024-03-20`).
- **DATETIME (8 bytes):** Stores date & time (`YYYY-MM-DD HH:MM:SS`).
- **TIMESTAMP (4 bytes):** Stores UTC timestamp (`YYYY-MM-DD HH:MM:SS`).
- **TIME (3 bytes):** Stores time (`HH:MM:SS`).
- **YEAR (1 byte):** Stores only the year (`YYYY`, e.g., `2025`).

Constraints in MySQL

Constraints are rules applied to table columns to **enforce data integrity** and **maintain accuracy** in the database. They help prevent invalid data from being entered.

Types of Constraints

1. NOT NULL

- Ensures a column cannot store **NULL** values.
- It can be applied on Multiple columns.
- Used when a field must always have data, such as `name` or `email`.

2. UNIQUE

- Ensures all values in a column are **distinct**, preventing duplicates.
- Accept only one null Value.
- Not allows duplicates.
- Allows multiple `NULL` values unless combined with `NOT NULL`.

3. PRIMARY KEY

- A combination of **NOT NULL** and **UNIQUE**, uniquely identifying each record.
- A table can have **only one** primary key, but it can consist of multiple columns (**composite key**).

4. FOREIGN KEY

- Establishes a relationship between tables by linking a column to the **PRIMARY KEY** of another table.
- Ensures that values in the child table must exist in the parent table.

5. CHECK

- Ensures column values meet a specific condition.
- Used for enforcing constraints like age restrictions or value ranges.

6. DEFAULT

- Assigns a **default value** if no value is provided during insertion.
- Useful for fields like timestamps, status columns, or boolean flags.

7. AUTO_INCREMENT

- Automatically generates a **unique number** for a column, commonly used for primary keys.
- Starts from 1 and increments automatically.

Creating Database

- A database is a structured collection of data that allows efficient storage, retrieval, and management.
- SQL provides the `CREATE DATABASE` statement to create a new database.
- Each database must have a unique name within the database server.
- By default, databases are created with default settings, but you can specify additional configurations.
- It is necessary to have the required privileges to create a database.
- The database name should follow the naming conventions and avoid reserved keywords.
- Once created, the database can store tables, views, procedures, and other objects.
- It is recommended to check if the database exists before creating a new one.

Syntax:

```
CREATE DATABASE database_name;

CREATE DATABASE IF NOT EXIST database_name;
```

Example:

```
CREATE DATABASE Batch76;
```

Use Database

- The `USE` statement is used to select a database for performing queries.
- After selecting a database, all subsequent SQL operations will be executed within that database.
- It eliminates the need to specify the database name in every query.
- The `USE` statement does not create a new database; it only selects an existing one.
- If the specified database does not exist, an error will be thrown.

- A database must be selected before creating tables, inserting data, or running queries.

Syntax:

```
USE database_name;
```

Example:

```
USE Batch76;
```

Creating Table

- A table is a structured format in a database that stores data in rows and columns.
- The `CREATE TABLE` statement is used to define a new table in SQL.
- Each column in a table must have a specified data type (e.g., `INT`, `VARCHAR`, `DATE`).
- Constraints such as `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK`, and `DEFAULT` can be applied to enforce data integrity.
- The `AUTO_INCREMENT` attribute can be used for automatically generating unique values in a column.
- A table must be created inside an existing database; ensure the database is selected using `USE database_name`.
- The column names should be meaningful and follow proper naming conventions.
- The structure of a table can be modified later using the `ALTER TABLE` command.

Syntax:

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

Example (Referencing the Students Table):

```
CREATE TABLE Students(
    RollNo INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(30) NOT NULL,
    Address VARCHAR(100),
    MobileNumber CHAR(10) NOT NULL,
    Email VARCHAR(30) UNIQUE,
    Fees INT CHECK(fees >= 20000),
    College VARCHAR(30),
    Department VARCHAR(30),
    AddmissionDate DATE NOT NULL,
    Batch VARCHAR(10),
    DOB DATE,
    PassingYear YEAR,
    JavaMarks INT,
    AptiMarks INT,
    SQLMarks INT,
    PlacementStatus VARCHAR(10) DEFAULT "NOT-PLACED"
);
```

Show Existing Tables

- The `SHOW TABLES` command is used to list all tables present in the currently selected database.
- It helps in verifying whether a table exists before performing operations on it.

Syntax:

```
SHOW TABLES;
```

Example:

```
USE school;
SHOW TABLES;
```

Describe Table Structure

- The `DESCRIBE` or `DESC` command is used to display the structure of a table.
- It provides information about column names, data types, constraints, and whether a column allows `NULL` values.

Syntax:

```
DESCRIBE table_name;
```

or

```
DESC table_name;
```

Example:

```
DESC Students;
```

Output:

Field	Type	Null	Key	Default	Extra
RollNo	INT	NO	PRI	NULL	AUTO_INCREMENT
Name	VARCHAR(30)	NO		NULL	
Address	VARCHAR(100)	YES		NULL	
MobileNumber	CHAR(10)	NO		NULL	

Email	VARCHAR(30)	YES	UNI	NULL	
Fees	INT	YES		NULL	CHECK (Fees>=20000)
College	VARCHAR(30)	YES		NULL	
Department	VARCHAR(30)	YES		NULL	
AddmissionDate	DATE	NO		NULL	
Batch	VARCHAR(10)	YES		NULL	
DOB	DATE	YES		NULL	
PassingYear	YEAR	YES		NULL	
JavaMarks	INT	YES		NULL	
AptiMarks	INT	YES		NULL	
SQLMarks	INT	YES		NULL	
PlacementStatus	VARCHAR(10)	YES		NOT-PLACED	

Inserting Values into a Table

- The `INSERT INTO` statement is used to add new records to a table.
- It can be done in different ways based on the requirement.
- There are four types of insertion methods:

Syntax and Examples

1. Inserting Values by Specifying Column Names (Type 1)

- This method allows inserting data by specifying column names explicitly.
- Recommended when inserting values for all or specific columns.

Syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Example (from `Students` table):

```
INSERT INTO Students (RollNo, Name, Address, MobileNumber, Email, Fees, College, Department, AddmissionDate, Batch, DOB, PassingYear, JavaMarks, AptiMarks, SQLMarks, PlacementStatus)
VALUES (1, "Ajay", "Pune", "9989899977", "ajay@gmail.com", 44000, "COEP", "IT", "2020-07-23", "Batch76", "2003-10-10", 2025, 66, 78, 45, "NOT-PLACED");
```

2. Inserting Values Without Column Names (Type 2)

- All values must be provided in the same order as table columns.
- Not recommended if table structure changes frequently.

Syntax:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Example:

```
INSERT INTO Students
VALUES (2, "Vijay", "Kolhapur", "9076675544", "vijay@gmail.com", 30000, "Ch. Shivaji Univer
sity", "ETC", "2021-01-27", "Batch77", "2002-05-10", 2024, 68, 77, 89, "PLACED");
```

---

3. Inserting Partial Data with Default or NULL Values (Type 3)

- Only selected column values are inserted; other columns get default or `NULL` values.

Syntax:

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

Example:

```
INSERT INTO Students (Name, MobileNumber, Email, Fees, AddmissionDate)
VALUES ("Sujay", "9876455533", "sujay@gmail.com", 60000, "2019-07-23");
```

- Here, unspecified columns will have `NULL` or default values.

---

4. Inserting Multiple Records in a Single Query (Type 4)

- Useful for bulk insertion, reduces query execution time.

Syntax:

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

Example:

```
INSERT INTO Students
VALUES
(4, "Vinay", "Satara", "9546378899", "Vinay@gmail.com", 47000, "WCOES", "CIVIL", "2020-0
7-23", "Batch75", "2000-03-04", 2024, 55, 78, 45, "NOT-PLACED"),
(5, "Akaay", "Sangli", "9546334499", "Akaay@gmail.com", 57000, "ADCOE", "MECH", "2017-03-15", "Batch73", "1998-04-13", 2021, 75, 88, 95, "PLACED");
```

- Multiple records are inserted at once, reducing execution overhead.

DQL - Data Query Language

- DQL (Data Query Language) is used to retrieve data from a database.
- The `SELECT` statement is the primary command used in DQL.
- It allows fetching all or specific columns from a table.
- Filtering, sorting, and aggregation can be applied using different clauses.
- Queries can include conditions to fetch relevant data.
- The retrieved data does not modify the table; it only displays records.

Syntax of SELECT Statement

```
SELECT column1, column2, ... FROM table_name;
```

- Used to fetch specific columns.

```
SELECT * FROM table_name;
```

- Used to fetch all columns (represents all).

Examples (Using `Students` Table)

1. Display All Records

```
SELECT * FROM Students;
```

- Retrieves all columns and rows from the `Students` table.

2. Display Specific Columns

```
SELECT Name, Batch, MobileNumber, Fees FROM Students;
```

- Fetches only `Name` , `Batch` , `MobileNumber` , and `Fees` columns.

```
SELECT RollNo, Name, Batch FROM Students;
```

- Retrieves `RollNo` , `Name` , and `Batch` .

```
SELECT Name, Email, MobileNumber FROM Students;
```

- Displays student names along with their email and mobile number.

WHERE Clause in SQL

- The `WHERE` clause is used to filter records based on specified conditions.
- It works on a **Boolean logic** (TRUE or FALSE) to determine which rows to display.
- It is commonly used with `SELECT`, `UPDATE`, and `DELETE` statements.
- Multiple conditions can be applied using `AND`, `OR`, and `NOT` operators.
- The `WHERE` clause supports comparison (`=`, `>`, `<`, `>=`, `<=`, `!=`) and pattern matching (`LIKE`).
- Filtering data using `WHERE` improves query performance by reducing the number of processed rows.

Syntax

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

- Retrieves records where the specified condition is met.

Examples (Using `Students` Table)

1. Retrieve Student with a Specific Roll Number

```
SELECT * FROM Students WHERE RollNo = 1;
```

- Displays details of the student whose `RollNo` is `1`.

2. Retrieve Students from a Specific College

```
SELECT * FROM Students WHERE College = "COEP";
```

- Fetches all students who belong to **COEP** college.

3. Retrieve Students from a Specific Address

```
SELECT * FROM Students WHERE Address = "Pune";
```

- Displays all students whose address is **Pune**.

Operators in SQL

SQL operators are used to perform operations on values and filter records based on conditions.

1. Relational (Comparison) Operators

- Used to compare two values in a condition.
- Returns `TRUE` if the condition is satisfied; otherwise, returns `FALSE`.
- Common relational operators: `<`, `>`, `<=`, `>=`, `=`, `!=`, `<>`.

Relational Operators Explanation & Use Cases

- `=` **(Equal to)** → Checks if two values are the same.
- `!=` **or** `<>` **(Not Equal to)** → Checks if two values are different.
- `>` **(Greater than)** → Checks if a value is greater than another.
- `<` **(Less than)** → Checks if a value is smaller than another.
- `>=` **(Greater than or Equal to)** → Checks if a value is greater than or equal to another.
- `<=` **(Less than or Equal to)** → Checks if a value is smaller than or equal to another.

Examples (Using `Students` Table)

```
SELECT * FROM Students WHERE PassingYear >= 2023;   -- Fetches students passing in 2023 or later.
SELECT * FROM Students WHERE JavaMarks >= 50;       -- Fetches students who scored at least 50 in Java.
SELECT * FROM Students WHERE RollNo != 1;           -- Fetches all students except RollNo 1.
SELECT * FROM Students WHERE Fees <= 50000;         -- Fetches students who paid fees of 50,000 or less.
SELECT * FROM Students WHERE College <> "COEP";     -- Fetches students not from "COEP" (<> is same as !=).
```

Difference Between `=` and `==` in SQL

Operator	Meaning	Usage	Example
`=`	Assignment or Comparison	Used to compare values in SQL queries	`SELECT * FROM Students WHERE College = 'COEP';`
`==`	Not used in SQL	SQL does not recognize `==` as a valid comparison operator	❌ Not valid in SQL

Explanation:

1. `=` **(Equal To)**
   - In SQL, `=` is the **only** valid operator for comparison.
   - Used in `WHERE`, `SELECT`, `UPDATE`, `DELETE`, etc.
2. `==` **(Double Equal Sign)**

- SQL **does not support** `==`.
- It is used in some programming languages like Python, Java, and C for equality checks.

Correct Example in SQL:

```
SELECT * FROM Students WHERE College = 'COEP';
```

Incorrect Usage in SQL:

```
SELECT * FROM Students WHERE College == 'COEP'; -- ❌ Error
```

---

2. Logical Operators

- Used to combine multiple conditions in a SQL query.
- Returns `TRUE` or `FALSE` based on combined conditions.
- Common logical operators: `AND`, `OR`, `NOT`.

Logical Operators Explanation & Use Cases

- `AND` / `&&` **(Logical AND)** → Returns `TRUE` if both conditions are met.
- `OR` / `||` **(Logical OR)** → Returns `TRUE` if at least one condition is met.
- `NOT` **(Logical NOT)** → Reverses the Boolean result (TRUE → FALSE, FALSE → TRUE).

Examples (Using `Students` Table)

1. `AND` Operator (Both Conditions Must be True)

```
SELECT * FROM Students WHERE College="ADCOE" AND Batch="Batch73";
-- Fetches students who are from "ADCOE" and belong to "Batch73".
```

```
SELECT * FROM Students WHERE Batch="Batch76" && JavaMarks >= 50;
-- Fetches students from "Batch76" who scored at least 50 in Java.
```

```
SELECT * FROM Students WHERE College="ADCOE" AND PlacementStatus="PLACED";
-- Fetches placed students from "ADCOE".
```

2. `OR` Operator (At Least One Condition Must be True)

```
SELECT * FROM Students WHERE JavaMarks > 70 OR Fees > 40000;
-- Fetches students who either scored above 70 in Java or paid more than 40,000 in fees.
```

3. `NOT` Operator (Negates the Condition)

```
SELECT * FROM Students WHERE NOT (JavaMarks > 70);
-- Fetches students who scored 70 or below in Java (opposite of `JavaMarks > 70`).
```

How OR and AND Work Internally in SQL?

SQL processes logical operators using a concept called "short-circuit evaluation." Here's how each operator works:

AND Operator Evaluation

- Evaluates conditions from left to right
- If the first condition is FALSE, stops immediately (short-circuits) and returns FALSE
- Only evaluates the second condition if the first condition is TRUE
- Returns TRUE only if all conditions are TRUE

```
SELECT * FROM Students WHERE JavaMarks > 50 AND PythonMarks > 60;
-- If JavaMarks check fails, PythonMarks won't be evaluated
```

OR Operator Evaluation

- Evaluates conditions from left to right
- If the first condition is TRUE, stops immediately (short-circuits) and returns TRUE
- Only evaluates the second condition if the first condition is FALSE
- Returns TRUE if any condition is TRUE

```
SELECT * FROM Students WHERE JavaMarks > 90 OR PythonMarks > 85;
-- If JavaMarks check passes, PythonMarks won't be evaluated
```

Performance Implications

- **Best Practice:** Put the most selective conditions first in AND operations
- **Best Practice:** Put the most likely-to-be-true conditions first in OR operations
- This optimization helps SQL engine to short-circuit earlier and reduce processing time

---

3. Arithmetic Operators

- Used for mathematical calculations in SQL queries.
- Common arithmetic operators:
  - `+` (Addition)
  - (Subtraction)
  - (Multiplication)
  - `/` (Division)

    - `%` (Modulus - returns remainder)
- Can be used directly in `SELECT` queries for calculations.
- Can be combined with `AS` to give an alias (renaming the result column).
- Used with `WHERE` to filter records based on calculations.

Syntax:

```
SELECT column_name, column_name OPERATOR value AS alias_name FROM table_name;
```

Queries (Using `Students` Table):

Addition (`+`) - Increase marks by 10

```
SELECT Name, Batch, JavaMarks, JavaMarks + 10 FROM Students;
```

Addition with Alias (`+` with `AS`)

```
SELECT Name, Batch, JavaMarks, JavaMarks + 10 AS GraceMarks FROM Students;
```

Subtraction (`-`) - Deduct reward from fees

```
SELECT Name, Batch, JavaMarks, Fees, Fees - 2000 AS Reward FROM Students WHERE Jav
aMarks >= 60;
```

Percentage Calculation (`*` , `/`) - Calculate 15% discount on fees

```
SELECT RollNo, Name, Batch, JavaMarks, Fees,
    (15 / 100) * Fees AS Discount,
    (Fees - ((15 / 100) * Fees)) AS FinalFees
FROM Students;
```

DISTINCT in SQL

- Used to **eliminate duplicate** records in a result set.
- Returns only **unique values** from the specified column(s).
- Can be applied to **single or multiple columns**.
- When applied to **multiple columns**, it considers **unique combinations** of values from those columns.
- Helps in **data analysis** by filtering out repeated values.

---

2. Syntax:

```
SELECT DISTINCT column_name FROM table_name;
SELECT DISTINCT column1, column2 FROM table_name;
```

3. Queries (Using `Students` Table):

Get Unique Batches:

```
SELECT DISTINCT Batch FROM Students;
```

Get Unique Colleges:

```
SELECT DISTINCT College FROM Students;
```

Get Unique Combinations of Department and PassingYear:

```
SELECT DISTINCT Department, PassingYear FROM Students;
```

- This will return unique department and passing year pairs.
- NOTE : we can mention "Distinct" keyword once in the query.

IN and BETWEEN Operators

1. IN Operator:

- Used to **filter data** by matching values in a specified list.
- **Simplifies multiple OR conditions** into a single statement.
- Can be used with **numeric, string, and date values**.

Syntax:

```
SELECT * FROM table_name WHERE column_name IN (value1, value2, ...);
```

Queries Using `Students` Table:

Get students from Pune and Kolhapur:

```
SELECT * FROM Students WHERE Address IN ("Pune", "Kolhapur");
```

Get students whose Fees match specific values:

```
SELECT * FROM Students WHERE Fees IN (34000, 30000, 45000);
```

2. BETWEEN Operator:

- Used to **filter data within a specific range**.
- Requires specifying both **starting and ending values**.
- Can be used with **numbers, dates, and text-based data**.

- **Inclusive** for numbers and dates, **exclusive** for text.

Syntax:

```
SELECT * FROM table_name WHERE column_name BETWEEN value1 AND value2;
```

Queries Using `Students` Table:

Get students with RollNo between 5 and 10:

```
SELECT * FROM Students WHERE RollNo BETWEEN 5 AND 10;
```

Get students whose Fees range between 40,000 and 50,000:

```
SELECT * FROM Students WHERE Fees BETWEEN 40000 AND 50000;
```

Get students admitted between 2020 and 2024:

```
SELECT * FROM Students WHERE AddmissionDate BETWEEN "2020-01-01" AND "2024-12-31";
```

Get students born between 1996 and 1999:

```
SELECT * FROM Students WHERE DOB BETWEEN "1996-01-01" AND "1999-12-12";
```

Get students whose names start between 'A' and 'N' (Excludes 'N'):

```
SELECT * FROM Students WHERE Name BETWEEN "A" AND "N";
```

Get students whose email starts between 'H' and 'T' (Excludes 'T'):

```
SELECT * FROM Students WHERE Email BETWEEN "H" AND "T";
```

LIKE Operator in SQL

1. Definition:

- The `LIKE` operator is used to **filter data** based on a **specific pattern**.
- Useful for **partial and exact** searches in **text-based columns**.

2. Wildcards Used in LIKE Operator:

Wildcard	Description	Example
`%`	Represents **0, 1, or multiple** characters	`"A%"` → Starts with "A"
`_`	Represents **exactly 1 character**	`"_e%"` → Second character is "e"

`BINARY`	Makes search **case-sensitive**	`"BINARY A%"` → Starts with uppercase "A"

3. Syntax:

```
SELECT * FROM table_name WHERE column_name LIKE 'pattern';
```

4. Examples (Using `Students` Table):

Find students whose names start with 'S':

```
SELECT * FROM Students WHERE Name LIKE "S%";
```

Find students whose address ends with 'ne':

```
SELECT * FROM Students WHERE Address LIKE "%ne";
```

Find students whose email contains 'gmail.com':

```
SELECT * FROM Students WHERE Email LIKE "%gmail.com";
```

Find students whose mobile number contains '99' anywhere:

```
SELECT * FROM Students WHERE MobileNumber LIKE "%99%";
```

Find students whose email contains 'sh':

```
SELECT * FROM Students WHERE Email LIKE "%sh%";
```

5. Using `_` for Single Character Matching:

Find colleges where the second letter is 'e':

```
SELECT * FROM Students WHERE College LIKE "_e%";
```

Find addresses where the second letter is 'a':

```
SELECT * FROM Students WHERE Address LIKE "_a%";
```

Find names where the second letter is 'a' and ends with 'sh':

```
SELECT * FROM Students WHERE Name LIKE "_a%sh";
```

Find colleges with exactly four characters:

```
SELECT * FROM Students WHERE College LIKE "____";
```

6. Case-Sensitive Search using `BINARY`:

Find students whose name starts with uppercase 'A' (case-sensitive):

```
SELECT * FROM Students WHERE BINARY Name LIKE "A%";
```

REGEXP (Regular Expression) in SQL

1. Definition:

- The `REGEXP` operator is used to **filter data** by **matching text patterns** using **regular expressions**.
- More **powerful than** `LIKE`, but may **impact performance** when used on large datasets.

---

2. Regular Expression Symbols & Meaning:

Symbol	Meaning	Example
`^`	Matches **start** of a string	`"^a"` → Starts with "a"
`$`	Matches **end** of a string	`"a$"` → Ends with "a"
`[]`	Matches **a range of characters**	`"[6-9]"` → Any digit between 6-9
`{}`	Specifies **size/length** of match	`"{2}"` → Exactly 2 characters
`\`	Escapes special characters	`"\\.com$"` → Ends with ".com"

---

3. Syntax:

```
SELECT * FROM table_name WHERE column_name REGEXP 'pattern';
```

---

4. Examples (Using `Students` Table):

Find students whose name starts with 'A':

```
SELECT * FROM Students WHERE Name REGEXP "^a";
```

Find students whose name ends with 'a':

```
SELECT * FROM Students WHERE Name REGEXP "a$";
```

Find students whose email ends with '@gmail.com':

```
SELECT * FROM Students WHERE Email REGEXP "@gmail.com$";
```

Find students whose name contains 'jay':

```
SELECT * FROM Students WHERE Name REGEXP "jay";
```

---

5. Using REGEXP for Mobile Number Validation:

Find invalid mobile numbers (starting with 0-5):

```
SELECT * FROM Students WHERE MobileNumber REGEXP "^[0-5]{1}";
```

Find mobile numbers that contain '99' anywhere:

```
SELECT * FROM Students WHERE MobileNumber REGEXP "[9]{2}";
```

Find valid Indian mobile numbers (start with 6-9 and have 10 digits):

```
SELECT * FROM Students WHERE MobileNumber REGEXP "^[6-9]{1}[0-9]{9}";
```

Alternative way to find valid mobile numbers (checks anywhere in the string):

```
SELECT * FROM Students WHERE MobileNumber REGEXP "[6-9]{1}[0-9]{9}";
```

---

6. Using REGEXP for Email Validation:

Find emails ending with '.com':

```
SELECT * FROM Students WHERE Email REGEXP "\\.com$";
```

Alternative way to find emails ending with 'com':

```
SELECT * FROM Students WHERE Email REGEXP "com$";
```

Incorrect pattern (won't return results):

```
SELECT * FROM Students WHERE Email REGEXP "\\com$";
```

ORDER BY Clause in SQL

---

1. Definition:

- The `ORDER BY` clause is used to **sort data** in **ascending (`ASC`) or descending (`DESC`) order**.
- Can be applied to **one or multiple columns**.
- **Default sorting order is** `ASC` if not specified.

---

2. Syntax:

```
SELECT * FROM table_name ORDER BY column_name [ASC | DESC];
```

- `ASC` → **Ascending order (default)**
- `DESC` → **Descending order**

---

3. Examples (Using `Students` Table):

Sort students by `Fees` in descending order:

```
SELECT * FROM Students ORDER BY Fees DESC;
```

Sort students by `DOB` in ascending order:

```
SELECT * FROM Students ORDER BY DOB ASC;
```

Sort students by `Name` (default ascending order):

```
SELECT * FROM Students ORDER BY Name;
```

Sort students by `Name`, then `Fees` (when names are the same, sort by Fees):

```
SELECT * FROM Students ORDER BY Name, Fees;
```

LIMIT Clause in SQL

1. **Definition**
   - The LIMIT clause is used to **restrict the number of records** in the output.
   - Useful for **pagination** and displaying a specific number of rows.
2. **Syntax**

   ```
   SELECT * FROM table_name LIMIT offset, rowcount;
   ```

   - **offset** → Number of records to skip (optional).
   - **rowcount** → Number of records to display.
3. **Use Cases & Examples (Using Students Table)**
   - **Retrieve the first 3 records**

     ```
     SELECT * FROM Students LIMIT 3;
     ```

   - **Pagination: Skip first 3 records, show next 4**

     ```
     SELECT * FROM Students LIMIT 3, 4;
     ```

   - **Show 3 records per page**

```
SELECT * FROM Students LIMIT 3;      -- Page 1 (First 3 records)
SELECT * FROM Students LIMIT 3, 3;   -- Page 2 (Next 3 records)
SELECT * FROM Students LIMIT 6, 3;   -- Page 3
SELECT * FROM Students LIMIT 9, 3;   -- Page 4
SELECT * FROM Students LIMIT 12, 3;  -- Page 5
```

- **Find the student with the highest JavaMarks**

```
SELECT * FROM Students ORDER BY JavaMarks DESC LIMIT 1;
```

4. **Key Points**
   - If only one value is provided (LIMIT n), it returns the first 'n' rows.
   - Used for **pagination** by skipping records using an offset.
   - Often combined with **ORDER BY** to fetch top or bottom records.
   - Helps in **optimizing queries** by reducing unnecessary data retrieval.

DML - Data Manipulation Language

What is DML?

DML (**Data Manipulation Language**) is a category of SQL commands used to **modify and manage data** within a database. It is responsible for handling data inside tables, including inserting, updating, and deleting records.

DML **does not define database structures** (unlike DDL - Data Definition Language). Instead, it focuses on modifying existing data without affecting the table's schema.

Key Features of DML

- **Allows users to insert, update, delete, and retrieve data.**
- **Affects only data, not database structure.**
- **Can be rolled back if used inside a transaction (`COMMIT` and `ROLLBACK`).**
- **Helps in managing real-time data changes efficiently.**

DML Commands

1. INSERT - Adding Data

The `INSERT` statement is used to add new records into a table.

Syntax:

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

Example:

```
INSERT INTO Students (RollNo, Name, Batch, Fees) VALUES (10, 'Rahul', 'Batch76', 40000);
```

🔹 **Note:** If values for all columns are provided, column names can be skipped:

```
INSERT INTO Students VALUES (11, 'Amit', 'Batch75', 45000);
```

---

2. UPDATE - Modifying Existing Data

The `UPDATE` statement is used to modify existing records in a table.

Syntax:

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

Examples:

```
UPDATE Students SET Batch = "Batch76" WHERE Batch = "Batch71";
UPDATE Students SET Fees = Fees - 2000 WHERE JavaMarks <= 60;
UPDATE Students SET Email = "ajay@gmail.com" WHERE RollNo = 1;
```

🔹 **Important Notes:**

- Always use a `WHERE` clause to avoid updating all records.
- **MySQL Safe Updates:** If MySQL prevents updates without `WHERE`, disable safe updates:

  ```
  SET SQL_SAFE_UPDATES = 0;
  ```

---

3. DELETE - Removing Data

The `DELETE` statement removes specific records from a table.

Syntax:

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

Example:

```
DELETE FROM Students WHERE RollNo = 15;
```

Important Notes:

- **Without `WHERE`, all records in the table will be deleted!**
- If you want to remove **all records but keep the table structure**, use:

  ```
  DELETE FROM Students;
  ```

- To **reset auto-increment values**, use:

  ```
  TRUNCATE TABLE Students;
  ```

DDL - Data Definition Language

What is DDL?

DDL (**Data Definition Language**) is a category of SQL commands used to **define, modify, or manage the structure (schema) of a database**. It affects the structure of tables but does **not** manipulate actual data within them.

Features of DDL

- **Defines and modifies database schema.**
- **Affects table structures but not the actual data.**
- **Includes commands like CREATE, ALTER, DROP, TRUNCATE, and RENAME.**
- **Changes made using DDL commands are auto-committed (cannot be rolled back).**

DDL Commands

1. CREATE - Creating a Table

The `CREATE` statement is used to create new tables.

Syntax:

```
CREATE TABLE tableName (
    column1 datatype constraint,
    column2 datatype constraint
);
```

Example:

```
CREATE TABLE Students (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50),
    Email VARCHAR(100),
    Fees INT
);
```

---

2. ALTER - Modifying an Existing Table

The `ALTER` statement allows modifying a table’s structure.

a) ADD COLUMN - Adding a New Column

```
ALTER TABLE Students ADD TotalFeesPaid INT;  -- Adds at the end
ALTER TABLE Students ADD TotalFeesPaid INT AFTER Fees;  -- Adds after Fees
ALTER TABLE Students ADD TotalFeesPaid INT FIRST;  -- Adds at the beginning
```

b) DROP COLUMN - Removing a Column

```
ALTER TABLE Students DROP COLUMN TotalFeesPaid;
```

c) MODIFY COLUMN - Changing Data Type, Size, or Constraints

```
ALTER TABLE Students MODIFY MobileNumber VARCHAR(10);  -- Change data type
ALTER TABLE Students MODIFY Address VARCHAR(90);  -- Change column size
ALTER TABLE Students MODIFY TotalFeesPaid INT UNIQUE;  -- Add unique constraint
```

d) RENAME COLUMN - Changing a Column Name

```
ALTER TABLE Students CHANGE COLUMN TotalFeesPaid TotalFees INT;
```

e) ADD PRIMARY KEY - Assigning a Primary Key to an Existing Table

```
ALTER TABLE StudentInfo ADD PRIMARY KEY (RollNo);
```

f) DROP PRIMARY KEY - Removing the Primary Key

```
ALTER TABLE StudentInfo DROP PRIMARY KEY;
```

---

3. DROP - Deleting a Table

The `DROP` statement **completely removes** a table, including its structure and all records.

Syntax:

```
DROP TABLE tableName;
```

Examples:

```
DROP TABLE Course;
DROP TABLE IF EXISTS Course;  -- Prevents error if the table does not exist
```

Important Notes:

- The table is permanently deleted and cannot be recovered.

- Foreign key constraints must be dropped before dropping a referenced table.

---

4. TRUNCATE - Removing All Records but Keeping the Structure

The `TRUNCATE` statement **deletes all records from a table but retains its structure**.

Syntax:

```
TRUNCATE TABLE tableName;
```

Example:

```
TRUNCATE TABLE Student;
```

🔹 **Key Differences Between `DROP` and `TRUNCATE`**

Feature	DROP	TRUNCATE
Deletes records	✅	✅
Deletes table structure	✅	❌
Resets AUTO_INCREMENT	❌	✅
Can be rolled back	❌	❌
Works with foreign key constraints	❌	❌ (needs disabling first)

---

5. RENAME - Changing Table Name

The `RENAME TABLE` statement renames an existing table.

Syntax:

```
RENAME TABLE oldTableName TO newTableName;
```

Example:

```
RENAME TABLE Student TO StudentInfo;
```

In short DDL

Command	Purpose	Example
`CREATE`	Creates a new table	`CREATE TABLE Students (RollNo INT, Name VARCHAR(50));`
`ALTER`	Modifies an existing table	`ALTER TABLE Students ADD Email VARCHAR(100);`
`DROP`	Deletes a table permanently	`DROP TABLE Students;`
`TRUNCATE`	Deletes all records but keeps structure	`TRUNCATE TABLE Students;`
`RENAME`	Renames a table	`RENAME TABLE Student TO StudentInfo;`

Foreign Key in SQL

What is a Foreign Key?

A **foreign key** is a **constraint** that creates a relationship between two tables by linking a column in one table to the **primary key** of another table. It helps maintain **referential integrity**, ensuring that relationships between tables remain valid.

Key Features of Foreign Key

- Establishes a **link** between two tables.
- The **primary key** of one table becomes the **foreign key** in another.
- Prevents **orphan records** (i.e., ensures that a referenced value exists in the parent table).
- Supports `ON DELETE CASCADE` and `ON DELETE SET NULL` for automatic handling of deletions in the parent table.

Creating Tables with a Foreign Key

1. Creating the Parent Table (`StudentInfo`)

```
CREATE TABLE StudentInfo (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(30),
    Address VARCHAR(100),
    MobileNumber CHAR(10),
    Email VARCHAR(50),
    Fees INT,
    Batch VARCHAR(20)
);
```

- Creates a table `StudentInfo` with `RollNo` as the **Primary Key**.

2. Creating the Child Table (`StudentMarks`)

```
CREATE TABLE StudentMarks (
    RollNo INT,
    JavaMarks INT,
    AptiMarks INT,
    FOREIGN KEY (RollNo) REFERENCES StudentInfo(RollNo)
);
```

- Creates a table `StudentMarks` .
- `RollNo` in `StudentMarks` acts as a **Foreign Key** referencing `RollNo` in `StudentInfo` .

Retrieving Data from Both Tables

1. Displaying Student Details

```
SELECT * FROM StudentInfo;
```

- Displays all student details from `StudentInfo`.

2. Displaying Student Marks

```
SELECT * FROM StudentMarks;
```

- Displays all student marks from `StudentMarks`.

3. Retrieving Student Details with Marks

```
SELECT * FROM StudentInfo AS si
JOIN StudentMarks AS sm
ON si.RollNo = sm.RollNo;
```

- Retrieves student details along with their marks by joining both tables.

Deleting Records with Foreign Key Constraints

1. Allowed: Deleting from Child Table (`StudentMarks`)

```
DELETE FROM StudentMarks WHERE RollNo = 210;
```

- Deletion is **allowed** because `StudentMarks` is a child table.

2. Allowed: Deleting from Parent Table (`StudentInfo`)

```
DELETE FROM StudentInfo WHERE RollNo = 210;
```

- Deletion is **allowed** if there is **no reference** to `RollNo = 210` in `StudentMarks`.

3. Not Allowed: Deleting from Parent Table (`StudentInfo`)

```
DELETE FROM StudentInfo WHERE RollNo = 207;
```

- **Not allowed** if `RollNo = 207` exists in `StudentMarks`.
- To allow deletion, use **`ON DELETE CASCADE`**.

Using `ON DELETE CASCADE` to Auto-Delete Dependent Records

```
CREATE TABLE StudentMarks (
    RollNo INT,
    JavaMarks INT,
    AptiMarks INT,
    FOREIGN KEY (RollNo) REFERENCES StudentInfo(RollNo) ON DELETE CASCADE
);
```

- When a student is deleted from `StudentInfo`, their marks in `StudentMarks` are **automatically deleted**.

Deleting Multiple Records in `StudentMarks`

```
DELETE FROM StudentMarks WHERE RollNo IN (203, 206, 208);
```

- Deletes records for students with **RollNo 203, 206, and 208** from `StudentMarks`.

Summary of Foreign Key Constraints

Constraint	Description
`FOREIGN KEY`	Establishes a relationship between two tables.
`REFERENCES`	Specifies the parent table and column to reference.
`ON DELETE CASCADE`	Automatically deletes child records when the parent record is deleted.
`ON DELETE SET NULL`	Sets the foreign key to `NULL` when the parent record is deleted.
`ON UPDATE CASCADE`	Updates child table records when the parent table key changes.

Joins in SQL

What is a Join?

A **JOIN** is used to combine rows from two or more tables based on a **common column**. It helps retrieve meaningful relationships between datasets stored in different tables.

Types of Joins in SQL

1. Inner Join

- Retrieves **only matching** data from both tables.
- Rows that do not have a match in both tables are **excluded**.

Example: Inner Join on `StudentInfo` and `StudentMarks`

```
SELECT * FROM StudentInfo
INNER JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo;
```

- Retrieves students **only if they have marks recorded** in `StudentMarks`.

Example: Inner Join with Selected Columns

```
SELECT StudentInfo.RollNo, StudentInfo.Name, StudentInfo.MobileNumber,
    StudentMarks.JavaMarks + AptiMarks AS TotalMarks
FROM StudentInfo
JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo;
```

- Retrieves student `RollNo`, `Name`, `MobileNumber`, and **total marks**.

Example: Using Table Aliases

```
SELECT si.RollNo, si.Name, sm.JavaMarks + AptiMarks AS TotalMarks
FROM StudentInfo AS si
JOIN StudentMarks AS sm
ON si.RollNo = sm.RollNo;
```

- Uses **table aliases** (`si`, `sm`) for readability.

Example: Inner Join on `Employees` and `Department`

```
SELECT * FROM Employees e
INNER JOIN Department d
ON e.DeptId = d.DeptId;
```

- Retrieves employee details **only if they belong to a department**.

---

2. Outer Join

A. Left Join (Left Outer Join)

- Retrieves **all records** from the **left table** and **matching records** from the right table.
- If no match is found, it **returns NULL** for right table columns.

Example: Left Join on `StudentInfo` and `StudentMarks`

```
SELECT * FROM StudentInfo
LEFT JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo;
```

- Retrieves **all students** and their marks (if available).

Example: Left Join with Selected Columns

```
SELECT si.RollNo, si.Name, sm.JavaMarks, sm.AptiMarks
FROM StudentInfo si
LEFT JOIN StudentMarks sm
ON si.RollNo = sm.RollNo;
```

- If a student **does not have marks**, `JavaMarks` and `AptiMarks` **will be NULL**.

Example: Left Join on `Employees` and `Department`

```
SELECT * FROM Employees e
LEFT JOIN Department d
ON e.DeptId = d.DeptId;
```

- Retrieves **all employees** and their department details (if available).

B. Right Join (Right Outer Join)

- Retrieves **all records** from the **right table** and **matching records** from the left table.
- If no match is found, it **returns NULL** for left table columns.

Example: Right Join on `StudentInfo` and `StudentMarks`

```
SELECT * FROM StudentInfo
RIGHT JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo;
```

- Retrieves **all records from `StudentMarks`** and the **matching records from `StudentInfo`**.

Example: Right Join with Selected Columns

```
SELECT si.RollNo, si.Name, sm.JavaMarks, sm.AptiMarks
FROM StudentInfo si
RIGHT JOIN StudentMarks sm
ON si.RollNo = sm.RollNo;
```

- If a student exists in `StudentMarks` but **not in `StudentInfo`**, `Name` will be `NULL`.

Example: Right Join on `Employees` and `Department`

```
SELECT * FROM Employees e
RIGHT JOIN Department d
ON e.DeptId = d.DeptId;
```

- Retrieves **all departments** and their **employees (if available)**.

C. Full Join (Full Outer Join)

- Retrieves **all records from both tables**.
- If no match is found, it **returns NULL** in the columns from the missing table.
- **MySQL does not support FULL JOIN directly**, but we can achieve it using `UNION`.

Example: Full Join using `UNION`

```
SELECT * FROM StudentInfo
LEFT JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo
UNION
SELECT * FROM StudentInfo
RIGHT JOIN StudentMarks
ON StudentInfo.RollNo = StudentMarks.RollNo;
```

- Combines **Left Join** and **Right Join** results to achieve a **Full Join**.

Example: Full Join on `Employees` and `Department`

```
SELECT * FROM Employees e
LEFT JOIN Department d
ON e.DeptId = d.DeptId
UNION
SELECT * FROM Employees e
RIGHT JOIN Department d
ON e.DeptId = d.DeptId;
```

- Ensures that **all employees and all departments** are retrieved.

---

3. Cross Join

- Returns the **Cartesian product** of both tables.
- Each row from the first table **combines with every row** from the second table.
- **No condition is required** for the join.

Example: Cross Join on `StudentInfo` and `StudentMarks`

```
SELECT * FROM StudentInfo
CROSS JOIN StudentMarks;
```

- If `StudentInfo` has **5 rows** and `StudentMarks` has **3 rows**, the result will have **5 × 3 = 15 rows**.

Example: Cross Join on `Employees` and `Department`

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

- Each employee will be **paired with every department**.

---

4. Self Join

- Joins a **table with itself**.
- Used to find **hierarchical relationships** like **employee-manager** or **category-subcategory**.

- Requires **table aliases** to differentiate instances.

Example: Creating `Emp` Table

```
CREATE TABLE Emp (
    EmpId INT PRIMARY KEY,
    Name VARCHAR(50),
    ManagerId INT
);
```

Example: Inserting Data

```
INSERT INTO Emp (EmpId, Name) VALUES (1, "Gitesh");
INSERT INTO Emp (EmpId, Name, ManagerId) VALUES
(2, "Aditya", 1),
(3, "Alice", 2),
(4, "Bob", 3),
(5, "Mahesh", 1),
(6, "Peter", 2);
```

Example: Self Join to Find Employees and Their Managers

```
SELECT e1.Name AS Employee, e2.Name AS Manager
FROM Emp e1
LEFT JOIN Emp e2
ON e1.ManagerId = e2.EmpId;
```

- `e1` represents the employee, and `e2` represents their manager.
- If an employee **has no manager**, `Manager` will be `NULL`.

Example: Displaying All Employees

```
SELECT * FROM Emp;
```

- Retrieves all employee details, including `EmpId`, `Name`, and `ManagerId`.

Summary of Joins

Join Type	Description
Inner Join	Retrieves **only matching records** from both tables.
Left Join	Retrieves **all records from the left table** and matching records from the right. Missing values are `NULL`.
Right Join	Retrieves **all records from the right table** and matching records from the left. Missing values are `NULL`.
Full Join	Retrieves **all records from both tables**. Missing values are `NULL`. (Achieved using `UNION` in MySQL).

Cross Join	Returns the **Cartesian product** of two tables.
Self Join	Joins a **table with itself**, used for hierarchical data.

Aggregate Functions in SQL

Definition: Aggregate functions perform calculations on multiple rows of a column and return a single value.

They are often used with the `GROUP BY` clause.

1. COUNT()

- Returns the number of rows that match a specified condition.

```
SELECT COUNT(*) FROM Students;  -- Counts all rows in the table
SELECT COUNT(JavaMarks) FROM Students;  -- Counts non-null JavaMarks
SELECT COUNT(Department) FROM Students;  -- Counts non-null Department values
SELECT COUNT(College) FROM Students;  -- Counts non-null College values
SELECT COUNT(*) FROM Students WHERE Batch = "Batch76";  -- Counts students in Batch76
SELECT COUNT(*) - COUNT(Department) AS NullCount FROM Students;  -- Counts null values in Department
```

2. MAX()

- Returns the highest value in a column.

```
SELECT MAX(JavaMarks) FROM Students;  -- Highest JavaMarks
SELECT MAX(JavaMarks) FROM Students WHERE Batch = "Batch76";  -- Highest JavaMarks in Batch76
SELECT MAX(DOB) FROM Students;  -- Latest Date of Birth
SELECT MAX(PassingYear) FROM Students;  -- Latest Passing Year
SELECT MAX(Address) FROM Students;  -- Alphabetically last Address
SELECT MAX(Name) FROM Students;  -- Alphabetically last Name
```

3. MIN()

- Returns the lowest value in a column.

```
SELECT MIN(JavaMarks) FROM Students;  -- Lowest JavaMarks
SELECT MIN(Fees) FROM Students;  -- Minimum Fees
SELECT MIN(Fees) FROM Students WHERE Batch = "Batch76";  -- Minimum Fees in Batch76
SELECT MIN(JavaMarks) FROM Students WHERE Batch = "Batch76";  -- Minimum JavaMarks in Batch76
SELECT MIN(DOB) FROM Students;  -- Oldest Date of Birth
SELECT MIN(PassingYear) FROM Students;  -- Earliest Passing Year
SELECT MIN(Address) FROM Students;  -- Alphabetically first Address
SELECT MIN(Name) FROM Students;  -- Alphabetically first Name
```

4. SUM() - Returns the total sum of numeric column values.

```
SELECT SUM(Fees) FROM Students;  -- Total Fees
SELECT SUM(JavaMarks) FROM Students;  -- Total JavaMarks
SELECT SUM(Fees) FROM Students WHERE Batch = "Batch76";  -- Total Fees in Batch76

-- The following will work but might not make sense:
SELECT SUM(DOB) FROM Students;  -- Dates are treated as numbers
```

5. AVG() - Returns the average value of a numeric column.

```
SELECT AVG(AptiMarks) FROM Students;  -- Average Aptitude Marks
SELECT AVG(Fees) FROM Students;  -- Average Fees
SELECT AVG(JavaMarks) FROM Students WHERE Batch = "Batch76";  -- Average JavaMarks i
n Batch76

-- The following will work but might not make sense:
SELECT AVG(DOB) FROM Students;  -- Dates are treated as numbers
SELECT AVG(Name) FROM Students;  -- Will return 0 as Name is not numeric
```

ROUND() Function in SQL

Definition

The `ROUND()` function in SQL is used to round a numeric value to a specified number of decimal places.

Syntax

```
ROUND(expression, decimal_places)
```

- `expression` : The numeric value to be rounded.
- `decimal_places` : The number of decimal places to round the value to. If omitted, it defaults to 0 (rounds to the nearest integer).

Examples

1. Rounding AVG() Values

```
SELECT ROUND(AVG(JavaMarks), 2) FROM Students;  -- Rounds average JavaMarks to 2 de
cimal places
```

```
SELECT ROUND(AVG(Fees), 1) FROM Students;  -- Rounds average Fees to 1 decimal place
```

2. Rounding a Specific Value

```
SELECT ROUND(123.567, 2);  -- Returns 123.57
SELECT ROUND(123.567, 0);  -- Returns 124
SELECT ROUND(123.567, -1);  -- Returns 120 (rounds to nearest 10)
```

3. Rounding Negative Values

```
SELECT ROUND(-45.789, 1);  -- Returns -45.8
SELECT ROUND(-45.789, 0);  -- Returns -46
```

4. Using ROUND() with Columns

```
SELECT Name, ROUND(Fees, 2) FROM Students;  -- Rounds Fees to 2 decimal places
SELECT Name, ROUND(JavaMarks / 2, 1) FROM Students;  -- Rounds JavaMarks divided by 2 to 1 decimal place
```

GROUP BY and HAVING Clause in SQL

1. GROUP BY Clause

- The `GROUP BY` clause is used to group rows that have the same values in specified columns.
- It is often used with aggregate functions (`SUM()` , `AVG()` , `COUNT()` , `MAX()` , `MIN()`).

Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name;
```

Examples

```
-- Group students by Batch and sum their Fees
SELECT Batch, SUM(Fees) FROM Students GROUP BY Batch;

-- Count students in each Batch
SELECT Batch, COUNT(*) FROM Students GROUP BY Batch;

-- Count students in each Department
SELECT Department, COUNT(*) FROM Students GROUP BY Department;
```

```
-- Get the average JavaMarks for each College
SELECT College, AVG(JavaMarks) FROM Students GROUP BY College;

-- Find the minimum JavaMarks in each Batch
SELECT Batch, MIN(JavaMarks) FROM Students GROUP BY Batch;
```

---

2. HAVING Clause

- The `HAVING` clause is used to filter grouped data.
- It is similar to `WHERE`, but `HAVING` works on aggregated results.

Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name
HAVING condition;
```

Examples

```
-- Show only departments with more than 1 student
SELECT Department, COUNT(*) FROM Students
GROUP BY Department
HAVING COUNT(*) > 1;

-- Show Batches where the average Fees is 50,000 or more
SELECT Batch, AVG(Fees) AS avgFees
FROM Students
GROUP BY Batch
HAVING avgFees >= 50000;
```

Key Differences: GROUP BY vs HAVING

Feature	GROUP BY	HAVING
Purpose	Groups rows	Filters grouped results
Works With	Columns and aggregate functions	Aggregate functions only
Used With	`SELECT`	`GROUP BY`

Subqueries in SQL

A **subquery** is a query within another query. It is enclosed in parentheses and executed first before the main query.

Types of Subqueries

1. **Scalar Subquery** (returns a single value)

2. **Multi-row Subquery** (returns multiple values)
3. **Nested Subquery** (a subquery inside another subquery)
4. **Correlated Subquery** (depends on the outer query)

---

1. Basic Subqueries

Find students who scored equal to or more than Vijay in Java

```
SELECT * FROM Students
WHERE JavaMarks >= (SELECT JavaMarks FROM Students WHERE Name = 'Vijay');
```

Find students who pay more fees than the average fees

```
SELECT * FROM Students
WHERE Fees > (SELECT AVG(Fees) FROM Students);
```

Find students who joined after Vinay's admission date

```
SELECT * FROM Students
WHERE AdmissionDate > (SELECT AdmissionDate FROM Students WHERE Name = 'Vinay');
```

Find names of students who live in the same address as Ajay

```
SELECT Name FROM Students
WHERE Address IN (SELECT Address FROM Students WHERE Name = 'Ajay');
```

---

2. Multi-row Subqueries

Find students who belong to a department where at least one student scored above 80 in Java

```
SELECT * FROM Students
WHERE Department IN (SELECT Department FROM Students WHERE JavaMarks > 80);
```

Find students who scored the same as any student in Batch 'Batch76'

```
SELECT * FROM Students
WHERE JavaMarks IN (SELECT JavaMarks FROM Students WHERE Batch = 'Batch76');
```

Find students whose fees match any student's fees from the Computer Science department

```
SELECT * FROM Students
WHERE Fees IN (SELECT Fees FROM Students WHERE Department = 'Computer Science');
```

---

3. Correlated Subqueries

A **correlated subquery** is executed once for each row in the outer query.

Find students who have the highest JavaMarks in their batch

```
SELECT * FROM Students S1
WHERE JavaMarks = (SELECT MAX(JavaMarks) FROM Students S2 WHERE S1.Batch = S2.Ba
tch);
```

Find students whose fees are greater than the average fees of their department

```
SELECT * FROM Students S1
WHERE Fees > (SELECT AVG(Fees) FROM Students S2 WHERE S1.Department = S2.Departme
nt);
```

Find students who have the lowest Aptitude marks in their college

```
SELECT * FROM Students S1
WHERE AptiMarks = (SELECT MIN(AptiMarks) FROM Students S2 WHERE S1.College = S2.Col
lege);
```

---

4. Nested Subqueries

A **nested subquery** contains another subquery inside it.

Find students who belong to a department where the average JavaMarks is higher than the total average JavaMarks of all students

```
SELECT * FROM Students
WHERE Department IN (
   SELECT Department FROM Students
   GROUP BY Department
   HAVING AVG(JavaMarks) > (SELECT AVG(JavaMarks) FROM Students)
);
```

Find students who belong to a college where the highest fee is more than 1,00,000

```
SELECT * FROM Students
WHERE College IN (
   SELECT College FROM Students
   GROUP BY College
```

```
    HAVING MAX(Fees) > 100000
);
```

Indexing in SQL

An **index** is a database structure that enhances the speed of **data retrieval operations** by creating a lookup table. However, it comes with **trade-offs**, such as **increased storage requirements** and **slower write operations** (`INSERT` , `UPDATE` , `DELETE`).

Indexes are used to speed up **data retrieval operations** in a database by maintaining a data structure (like **B-Trees or Hash indexes**) for efficient lookup.

Advantages:

Faster search queries (retrieval operations)

Reduces the number of rows scanned in a table

Disadvantages:

Extra storage space required

Slower `INSERT` , `UPDATE` , and `DELETE` operations (as indexes need to be updated)

How Queries Perform Before and After Using Indexing

Before Indexing (Without Index)

- The database performs a **full table scan** for every query.
- As the dataset grows, the query execution **becomes slower**.
- Filtering and searching operations **take more time** since every row is checked individually.

After Indexing (With Index)

- The database uses an **optimized search mechanism** like **B-Trees** or **Hash indexing**.
- Queries execute **much faster** since they use the index instead of scanning the entire table.
- Indexes improve the performance of **SELECT** queries but slow down **INSERT, UPDATE, DELETE** operations due to the overhead of maintaining the index.

Types of Indexes in SQL

1. Primary Index

- **Before Indexing:** Searching for a record using a **PRIMARY KEY** requires scanning the entire table.
- **After Indexing:** The database **automatically creates an index** on the primary key column, making lookups much faster.

```
SHOW INDEXES IN Students;
```

2. Unique Index

- **Before Indexing:** Checking for duplicate values requires scanning all existing records.
- **After Indexing:** The database can **quickly verify uniqueness**, reducing the time taken for insert operations.

```
CREATE UNIQUE INDEX Index_Email ON Students(Email);
DROP INDEX Index_Email ON Students;
EXPLAIN SELECT * FROM Students WHERE Email = "vijay@gmail.com";
```

---

3. Simple Index

- **Before Indexing:** Searching for records in columns with duplicate values requires a **full table scan**.
- **After Indexing:** The index helps the database **locate records faster**, improving `SELECT` query performance.

```
CREATE INDEX Index_College ON Students(College);
SELECT * FROM Students WHERE College = "COEP";
EXPLAIN SELECT * FROM Students WHERE College = "COEP";
```

---

4. Full-Text Index

- **Before Indexing:** Searching for keywords in long text fields like **remarks or articles** is slow as it requires scanning every row.
- **After Indexing:** The **MATCH() AGAINST()** function speeds up searches, allowing **efficient text-based lookups**.

```
CREATE FULLTEXT INDEX index_remark ON Students(Remark);
```

Example Queries

Natural Mode:

```
SELECT * FROM Students WHERE MATCH(Remark) AGAINST("Java");
SELECT * FROM Students WHERE MATCH(Remark) AGAINST("Java HTML");
```

Boolean Mode:

```
SELECT * FROM Students WHERE MATCH(Remark) AGAINST("+Java +HTML" IN BOOLEAN MODE);
SELECT * FROM Students WHERE MATCH(Remark) AGAINST("-Java +HTML" IN BOOLEAN MODE);
```

---

5. Composite Index (Multi-Column Index)

- **Before Indexing:** If a query filters by multiple columns, the database scans each column separately, making the query slower.

- **After Indexing:** The composite index improves performance when searching with **multiple conditions**, reducing execution time.

```
CREATE INDEX idx_students_name ON Students (Name, College);
```

Example Queries

Efficient (Index is used):

```
SELECT * FROM Students WHERE Name = 'Ajay' AND College = 'COEP';
SELECT * FROM Students WHERE Name = 'Ajay';
```

Inefficient (Index is NOT used):

```
SELECT * FROM Students WHERE College = 'COEP'; -- Does not follow the leftmost prefix rul
e
```

Performance Comparison (With and Without Indexing)

Query	Without Index (Full Table Scan)	With Index (Optimized Search)
`SELECT * FROM Students WHERE Name = "Ajay";`	Slow (scans all rows)	Fast (uses index for direct lookup)
`SELECT * FROM Students WHERE Email = "vijay@gmail.com";`	Slow (checks all rows)	Fast (unique index allows instant lookup)
`SELECT * FROM Students WHERE MATCH(Remark) AGAINST("Java");`	Slow (full text search)	Fast (optimized full-text search)
`SELECT * FROM Students WHERE Name = "Ajay" AND College = "COEP";`	Slow (checks both columns separately)	Fast (uses composite index for efficient filtering)

Best Practices for Using Indexes

1. **Index frequently searched columns** that appear in `WHERE`, `JOIN`, `ORDER BY`, and `GROUP BY` clauses.
2. **Use composite indexes** when multiple columns are often queried together.
3. **Avoid excessive indexing**, as it can **slow down insert and update operations**.
4. **Regularly monitor and optimize indexes** to ensure they improve performance effectively.

SQL Triggers

A **trigger** is a block of SQL code that automatically executes in response to specific **events** (INSERT, UPDATE, DELETE) on a table. It helps in **enforcing business rules, auditing changes, and automating actions**.

Characteristics of Triggers

- **Automatically executed** on predefined events.
- Can be triggered **before or after** an operation.

- Works on **INSERT, UPDATE, DELETE** actions.
- Can enforce **constraints** and maintain **data integrity**.

Types of Triggers

Trigger Type	Description
BEFORE INSERT	Executes **before** inserting a new record into the table.
AFTER INSERT	Executes **after** a new record has been inserted.
BEFORE UPDATE	Executes **before** an existing record is updated.
AFTER UPDATE	Executes **after** an existing record is updated.
BEFORE DELETE	Executes **before** deleting a record from the table.
AFTER DELETE	Executes **after** a record has been deleted from the table.

1. BEFORE INSERT Trigger

Executes before inserting data into the `Students` table. Used to **validate or modify** data before it is added.

```
DELIMITER &&
CREATE TRIGGER before_insert
BEFORE INSERT ON Students
FOR EACH ROW
BEGIN
   IF NEW.Fees < 30000 THEN
      SIGNAL SQLSTATE "43000"
      SET MESSAGE_TEXT = 'Fees cannot be less than 30000';
   END IF;
END &&
DELIMITER ;
```

Test Query:

```
INSERT INTO Students(Name, Fees, MobileNumber, Email)
VALUES ("Anit", 20000, 8978675645, "anit@gmail.com");  -- This will trigger an error
```

Error Output:

`Fees cannot be less than 30000`

2. AFTER INSERT Trigger

Executes after inserting a new record. Used to **log changes or maintain a duplicate record**.

```
CREATE TRIGGER after_insert
AFTER INSERT ON Students
FOR EACH ROW
BEGIN
   INSERT INTO StudentLogs(RollNo, Name, Address, MobileNumber)
```

```
    VALUES (NEW.RollNo, NEW.Name, NEW.Address, NEW.MobileNumber);
END;
```

Effect:

- Every new student inserted into `Students` will also be recorded in `StudentLogs`.

---

3. BEFORE UPDATE Trigger

Executes before updating an existing record. Used to **validate** data before the update.

```
DELIMITER &&
CREATE TRIGGER before_update
BEFORE UPDATE ON Students
FOR EACH ROW
BEGIN
    IF NEW.Fees < 30000 THEN
        SIGNAL SQLSTATE "43000"
        SET MESSAGE_TEXT = "Fees cannot be less than 30000";
    END IF;
END &&
DELIMITER ;
```

Test Query:

```
UPDATE Students SET Fees = 25000 WHERE RollNo = 12;  -- This will trigger an error
```

Error Output:

`Fees cannot be less than 30000`

---

4. AFTER UPDATE Trigger

Executes after an update occurs. Used to **log changes or update related records**.

```
CREATE TRIGGER after_update
AFTER UPDATE ON Students
FOR EACH ROW
BEGIN
    UPDATE StudentLogs SET Name = NEW.Name WHERE RollNo = NEW.RollNo;
END;
```

Effect:

- Whenever a student's name is updated in `Students`, it is also updated in `StudentLogs`.

---

5. BEFORE DELETE Trigger

Executes before deleting a record. Used to **prevent accidental deletions**.

```
DELIMITER //
CREATE TRIGGER before_delete
BEFORE DELETE ON Students
FOR EACH ROW
BEGIN
  SIGNAL SQLSTATE "43000"
  SET MESSAGE_TEXT = "Cannot delete any data";
END //
DELIMITER ;
```

Test Query:

```
DELETE FROM Students WHERE RollNo = 13;  -- This will trigger an error
```

Error Output:

`Cannot delete any data`

---

6. AFTER DELETE Trigger

Executes after deleting a record. Used to **log deletions** or **archive data**.

```
DELIMITER &&
CREATE TRIGGER after_delete
AFTER DELETE ON Students
FOR EACH ROW
BEGIN
  INSERT INTO DeletedStudents(RollNo, Name, Address, MobileNumber)
  VALUES (OLD.RollNo, OLD.Name, OLD.Address, OLD.MobileNumber);
END &&
DELIMITER ;
```

Effect:

- When a record is deleted from `Students`, it is stored in `DeletedStudents` for reference.

Managing Triggers

Show Triggers in the Database

```
SHOW TRIGGERS;
```

Drop a Trigger

```
DROP TRIGGER IF EXISTS before_insert;
DROP TRIGGER IF EXISTS before_update;
DROP TRIGGER IF EXISTS before_delete;
```

Performance Considerations

- **Advantages**
    - Ensures **data integrity** by enforcing rules.
    - Automates **logging and auditing**.
    - Reduces **manual intervention** for validations.
- **Disadvantages**
    - Can **slow down performance** due to additional processing.
    - Debugging can be **complex**, as triggers execute **implicitly**.
    - Can **increase system complexity**, making maintenance harder.

SQL Procedures

1. What is a Procedure?

- A **procedure** is an SQL query stored in a database.
- It is used to **execute multiple SQL statements** as a single unit.
- Similar to **methods or functions** in programming.
- Helps in **reducing network traffic** by executing on the database server.
- Increases **security** as users can execute procedures without accessing table details.
- **Reusability** – A stored procedure can be used multiple times.

---

2. Syntax of Creating a Procedure

```
CREATE PROCEDURE procedure_name()
BEGIN
    -- SQL statements
END;
```

Calling a Procedure:

```
CALL procedure_name();
```

---

3. Example: Get All Students Data

```
DELIMITER //
CREATE PROCEDURE getAllStudents()
BEGIN
    SELECT * FROM Students;
END //
```

```
DELIMITER ;

CALL getAllStudents();
```

Explanation:

- `DELIMITER //` changes the default delimiter to `//` to allow multiple statements inside `BEGIN...END`.
- `CREATE PROCEDURE getAllStudents()` defines a stored procedure.
- `SELECT * FROM Students;` retrieves all student records.
- `CALL getAllStudents();` executes the procedure.

---

4. Types of Parameters in Procedures

Procedures can accept parameters for dynamic queries. There are **three types**:

4.1. IN Parameter (Input Parameter)

- Used to pass values **into** a procedure.
- The value of `IN` parameters **cannot be changed** inside the procedure.

Example: Fetch Student Details by Roll Number

```
DELIMITER &&
CREATE PROCEDURE getStudentByRollNo(IN num INT)
BEGIN
   SELECT * FROM Students WHERE RollNo = num;
END &&
DELIMITER ;

CALL getStudentByRollNo(7);
```

Explanation:

- `IN num INT` accepts an integer input (Roll Number).
- The procedure retrieves details of the student with `RollNo = num`.
- `CALL getStudentByRollNo(7);` fetches data for Roll No **7**.

4.2. OUT Parameter (Output Parameter)

- Used to **return** values **from** a procedure.
- It **does not accept input**, only stores the result.

Example: Get Total Marks of All Students

```
DELIMITER &&
CREATE PROCEDURE getAllMarks(OUT addition INT)
BEGIN
```

```
    SELECT SUM(JavaMarks) INTO addition FROM Students;
END &&
DELIMITER ;

CALL getAllMarks(@num);
SELECT @num;
```

Explanation:

- `OUT addition INT` is an output parameter to store the result.
- `SELECT SUM(JavaMarks) INTO addition;` calculates and stores total Java marks in `addition`.
- `CALL getAllMarks(@num);` executes the procedure and stores the result in `@num`.
- `SELECT @num;` displays the total Java marks.

4.3. INOUT Parameter (Input & Output)

- Accepts an input value and **modifies it inside** the procedure.
- The modified value is returned as output.

Example: Get Marks by Roll Number

```
DELIMITER &&
CREATE PROCEDURE getMarksByRoll(INOUT num INT)
BEGIN
    SELECT JavaMarks INTO num FROM Students WHERE RollNo = num;
END &&
DELIMITER ;

SET @num = 1;
CALL getMarksByRoll(@num);
SELECT @num;
```

Explanation:

- `INOUT num INT` is both an **input** and **output** parameter.
- `SELECT JavaMarks INTO num;` updates the input `num` with JavaMarks from the `Students` table.
- `SET @num = 1;` initializes the variable with **RollNo 1**.
- `CALL getMarksByRoll(@num);` modifies `@num` with **JavaMarks of Roll No 1**.
- `SELECT @num;` retrieves the modified value.

SQL Views

1. What is a View?

- A **view** is a **virtual table** that represents the **result of a stored SQL query**.
- It **does not store data physically** but displays data dynamically at runtime.
- Provides **access control** over data by limiting what users can see.
- Simplifies complex queries by **encapsulating them into a single view**.
- Helps in **data abstraction** by **hiding unnecessary details**.

---

2. Syntax of Creating a View

```
CREATE VIEW ViewName AS
SELECT ColumnNames FROM TableName;
```

- `ViewName` – Name of the virtual table.
- `SELECT ColumnNames FROM TableName;` – Defines what data the view will display.

---

3. Example: Creating a View for Vibrant Students

```
CREATE VIEW Vibrant_Students AS
SELECT Name, Email, College FROM Students;
```

Explanation:

- The **Vibrant_Students** view contains only **Name, Email, and College** from the `Students` table.
- This hides other sensitive information (e.g., mobile number, fees).
- The view can be accessed like a regular table:

```
SELECT Name, Email, College FROM Vibrant_Students;
```

---

4. Benefits of Using Views

- **Security & Access Control** – Restricts access to specific columns or rows.
- **Data Abstraction** – Hides unnecessary details and simplifies complex queries.
- **Reusability** – Instead of writing the same query multiple times, use a view.
- **Logical Independence** – If the base table changes, the view still works without modification.
- **Simplifies Joins & Aggregations** – Preprocesses complex queries.

---

5. Checking Views in the Database

```
SHOW FULL TABLES;
SHOW FULL TABLES WHERE Table_type = "VIEW";
```

- `SHOW FULL TABLES;` lists all tables and views in the database.
- `SHOW FULL TABLES WHERE Table_type = "VIEW";` filters only views.

6. Updating Data in a View (Occurs error)

```
UPDATE Vibrant_Students
SET Fees = "26000"
WHERE Name = "Sujay";
```

Error:

- This query **will not work** because the `Vibrant_Students` view **does not include the Fees column**.
- Views **must contain all columns involved in an UPDATE operation**.

7. Creating a View for Placed Students

```
CREATE VIEW PlacedStudents AS
SELECT * FROM Students WHERE PlacementStatus = "PLACED";
```

Explanation:

- The `PlacedStudents` view contains **only those students** whose `PlacementStatus` is `"PLACED"`.
- This view can be queried to get details of all placed students:

```
SELECT * FROM PlacedStudents;
```

8. Read-Only View vs Updatable View (Homework)

Read-Only View

- Some views **cannot be updated** if they involve:
  - Aggregations (`SUM()`, `AVG()`, `COUNT()`)
  - Joins between multiple tables
  - Use of `DISTINCT`, `GROUP BY`, or `HAVING`
- Example:

```
CREATE VIEW StudentSummary AS
SELECT College, COUNT(*) AS TotalStudents
FROM Students
GROUP BY College;
```

- This **cannot be updated** because of `COUNT(*)` and `GROUP BY`.

Updatable View

- A view **can be updated** if:
  - It is based on a **single table**.
  - It does **not use aggregations, GROUP BY, or DISTINCT**.
- Example:

```
CREATE VIEW UpdateableStudent AS
SELECT RollNo, Name, Fees FROM Students;
```

- You can **update the Fees** for a student:

```
UPDATE UpdateableStudent
SET Fees = 30000
WHERE RollNo = 5;
```

Transaction Control Language (TCL) :

---

1. What is TCL?

- **TCL (Transaction Control Language)** is used to **control transactions** in a database.
- A **transaction** is a **single unit of work** that consists of one or more SQL statements.
- Ensures **data consistency** and maintains the **integrity of the database**.
- Transactions follow the **ACID properties** (Atomicity, Consistency, Isolation, Durability).
- Used in scenarios like **banking (ATM transactions), online payments, and inventory management**.

---

2. Key TCL Commands

a) START TRANSACTION

- Marks the beginning of a transaction.
- Changes made after this command are **temporary** until a `COMMIT` or `ROLLBACK` is executed.

```
START TRANSACTION;
```

b) COMMIT

- **Saves all changes permanently** in the database.
- Once committed, changes **cannot be rolled back**.

```
COMMIT;
```

c) ROLLBACK

- **Undo all changes** made since the last `START TRANSACTION` or `SAVEPOINT`.
- Used in case of errors or failures.

```
ROLLBACK;
```

d) SAVEPOINT

- **Creates a checkpoint** within a transaction.
- Allows partial rollback to a specific point instead of rolling back the entire transaction.

```
SAVEPOINT save1;
```

e) RELEASE SAVEPOINT

- Deletes a specific savepoint, making it unavailable for rollback.

```
RELEASE SAVEPOINT save1;
```

---

3. Example: Bank Transaction

Scenario: Transferring money from one account to another

```
START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 101;
UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 102;

COMMIT;
```

- If both `UPDATE` statements execute successfully, the changes are **saved permanently** using `COMMIT`.
- If any error occurs, changes **must be undone** using `ROLLBACK`.

---

4. Example: Using ROLLBACK

Scenario: Error in Transaction

```
START TRANSACTION;

UPDATE Students SET Fees = Fees - 1000 WHERE RollNo = 5;
UPDATE Students SET Fees = Fees + 1000 WHERE RollNo = 6;

ROLLBACK;
```

- If an error occurs, `ROLLBACK` **undoes all changes** and restores the original state.

---

5. Example: Using SAVEPOINT

Scenario: Partial Rollback

```
START TRANSACTION;

UPDATE Employees SET Salary = Salary + 5000 WHERE EmpID = 1;
SAVEPOINT sp1;

UPDATE Employees SET Salary = Salary + 7000 WHERE EmpID = 2;
SAVEPOINT sp2;

ROLLBACK TO sp1;  -- Undo changes after sp1 but keep the first update

COMMIT;
```

- Changes before `SAVEPOINT sp1` are **saved**.
- Changes after `sp1` are **rolled back**.

ACID Properties in MySQL

ACID stands for **Atomicity, Consistency, Isolation, and Durability**. These properties ensure that database transactions are **reliable, accurate, and fault-tolerant**.

---

1. Atomicity

Definition:

- A transaction is treated as **a single unit**, meaning either **all operations complete successfully** or **none at all**.
- If one step fails, the entire transaction is **rolled back** to maintain consistency.

Example:

Imagine **transferring ₹500** from **Account A** to **Account B** using online banking.

1. **Step 1:** The bank **deducts ₹500** from **Account A**.
2. **Step 2:** The bank **adds ₹500** to **Account B**.
3. If **Step 2 fails** (e.g., server crash), the **₹500 should not be deducted** from Account A.

SQL Example:

```
START TRANSACTION;

-- Deduct ₹500 from Account A
UPDATE accounts SET balance = balance - 500 WHERE account_id = 101;
```

```
-- Add ₹500 to Account B
UPDATE accounts SET balance = balance + 500 WHERE account_id = 102;

COMMIT;
```

If **Step 2 fails**, MySQL **automatically rolls back** the transaction, ensuring that no money is lost.

---

2. Consistency

Definition:

- Ensures that **only valid data** is inserted into the database.
- Enforces **constraints** like **Primary Key, Foreign Key, and Data Types**.
- If a transaction violates a **constraint**, it is rolled back.

Example:

A college management system ensures that **a student must be assigned to an existing department**.

1. If a student tries to enroll in **a non-existent department**, the database **rejects** the transaction.
2. This prevents **inconsistent data** from being stored.

SQL Example:

```
INSERT INTO students (student_id, name, department_id)
VALUES (201, 'Amit Sharma', 5);
```

If **department_id = 5 does not exist**, MySQL **prevents the insertion**, maintaining consistency.

---

3. Isolation

Definition:

- Ensures that **concurrent transactions do not interfere** with each other.
- Prevents **dirty reads, lost updates, and incorrect balances**.

Example:

Imagine two people trying to **book the last movie ticket** at the same time.

1. **Person A** checks availability (1 ticket left).
2. **Person B** also checks availability (1 ticket left).
3. If both book at the same time, **only one should succeed**.

SQL Example:

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
```

```
-- Check available tickets
SELECT seats_available FROM movies WHERE movie_id = 101;

-- Reduce available tickets by 1
UPDATE movies SET seats_available = seats_available - 1 WHERE movie_id = 101;

COMMIT;
```

With **SERIALIZABLE isolation**, if **Person A's transaction is in progress**, Person B **must wait** until it's completed.

This prevents **overselling** of tickets.

---

4. Durability

Definition:

- Ensures that once a transaction is **committed**, its changes are **permanently saved**, even if the system crashes.

Example:

Imagine placing an order on **Amazon**:

1. You select a **mobile phone** and click **"Place Order"**.
2. The system **records the order details** and **updates the stock**.
3. Even if the server crashes **right after you place the order**, your order **must still exist** when you log in again.

SQL Example:

```
START TRANSACTION;

-- Insert order details
INSERT INTO orders (user_id, product_id, status)
VALUES (501, 301, 'Confirmed');

-- Update stock
UPDATE products SET stock = stock - 1 WHERE product_id = 301;

COMMIT;
```

Once the **COMMIT** is executed, the order is **permanently stored**, even if the system crashes.

Normalization in MySQL

What is Normalization?

Normalization is a process of organizing data in a database to:

- Reduce redundancy (duplicate data)

- Improve data consistency
- Ensure efficient data storage

It involves breaking a large table into smaller tables and establishing relationships using Primary Keys and Foreign Keys.

Why is Normalization Important?

Imagine a student database storing student and course details in a single table:

Student_ID	Name	Course	Instructor	Department
1	John	Math	Mr. A	Science
2	Alice	Science	Mr. B	Science
3	John	Science	Mr. B	Science
4	Alice	Math	Mr. A	Science

Problems in this Table:

1. **Data Duplication:**
    - "John" and "Alice" appear multiple times with different courses.
2. **Update Issues:**
    - If Instructor "Mr. A" changes, all rows where he appears must be updated.
3. **Insertion Issues:**
    - If a new course starts but has no students yet, it cannot be inserted.
4. **Deletion Issues:**
    - If the last student enrolled in "Math" is deleted, course details may also be lost.

Solution: Apply Normalization

Splitting data into multiple smaller tables can resolve these issues.

Normalization Forms (NF) – Step by Step

1st Normal Form (1NF) – Remove Repeating Data

- Each column should contain atomic (indivisible) values.
- No duplicate rows or multiple values in a single column.

We split the courses into separate rows instead of storing multiple values in a single column.

Student_ID	Name	Course_ID	Course
1	John	C101	Math
2	Alice	C102	Science
3	John	C102	Science
4	Alice	C101	Math

This ensures that each field contains only a single piece of data.

2nd Normal Form (2NF) – Remove Partial Dependency

- Every non-key column must depend on the whole primary key, not just part of it.
- Course details are separated into a new table since they depend only on Course_ID, not Student_ID.

Students Table:

Student_ID	Name
1	John
2	Alice

Courses Table:

Course_ID	Course	Instructor	Department
C101	Math	Mr. A	Science
C102	Science	Mr. B	Science

Enrollment Table (Mapping Students to Courses):

Student_ID	Course_ID
1	C101
2	C102
3	C102
4	C101

Now, course information is stored separately, reducing redundancy.

3rd Normal Form (3NF) – Remove Transitive Dependency

- Non-key columns should depend only on the Primary Key.
- "Department" depends on "Instructor," so it is moved to a separate table.

Instructor Table:

Instructor_ID	Name	Department
I01	Mr. A	Science
I02	Mr. B	Science

Now, instructor details are stored separately, making updates easier.

Final Structure (Fully Normalized Database)

1. Students Table

Student_ID	Name
1	John
2	Alice

2. Courses Table

Course_ID	Course	Instructor_ID
C101	Math	I01
C102	Science	I02

3. Enrollment Table (Mapping Students to Courses)

Student_ID	Course_ID
1	C101
2	C102
3	C102
4	C101

4. Instructors Table

Instructor_ID	Name	Department
I01	Mr. A	Science
I02	Mr. B	Science

Benefits of Normalization

- No data duplication, which saves storage space.
- Easy updates, as changing instructor details affects only one table.
- Efficient queries, since searching is faster with smaller, structured tables.
- No anomalies in insert, update, and delete operations.