Prathamesh Arvind Jadhav

SQL for Placement

LEC-1: SQL Basics

SQL Basics:

1. What is SQL?

- **SQL** stands for **Structured Query Language**.
- It is a **domain-specific language** used to **access, manipulate, and manage** data stored in **relational databases**.
- SQL is **not a database**, but a **language used to interact** with relational databases.

---

2. CRUD Operations in SQL

CRUD refers to the **four basic operations** used to interact with database records:

Operation	SQL Command	Description
C - Create	INSERT	Adds new records (tuples) to a table (relation).
R - Read	SELECT	Retrieves existing data from the table.
U - Update	UPDATE	Modifies existing data in the table.
D - Delete	DELETE	Removes data from the table (specific row or rows).

---

3. What is RDBMS (Relational Database Management System)?

- RDBMS is a **software system** that helps to **store, manage, and retrieve** data using the **relational model**.
- Data is stored in the form of **tables** (also called **relations**).
- RDBMSs implement SQL to interact with the stored data.

Examples of RDBMS:

- **MySQL** (open-source)
- **MS SQL Server**
- **Oracle DB**
- **IBM DB2**

Key Concepts:

- A **table** or **relation** is the **basic data storage unit** in an RDBMS.
- **MySQL** is an RDBMS that uses **SQL** to perform all CRUD operations.
- It follows the **client-server model**:

- **Client**: CLI tool, GUI application, or frontend interface.
- **Server**: MySQL engine that processes queries and manages the database.

---

4. Difference Between SQL and MySQL

SQL	MySQL
Language used to interact with relational databases.	A relational database management system.
Used for querying and managing data (INSERT, SELECT, etc.).	Uses SQL internally to store and manipulate data.
Not a software or tool by itself.	A complete software system to manage databases.

---

5. SQL Data Types

Data types define **what kind of data** can be stored in a column.

Character/String Types:

Data Type	Description	Size
CHAR(n)	Fixed-length string	0–255 characters
VARCHAR(n)	Variable-length string	0–255 characters
TINYTEXT	Small text string	Up to 255 characters
TEXT	Large text	Up to 65,535 characters
BLOB	Binary Large Object (binary data)	Up to 65,535 bytes
MEDIUMTEXT	Medium-length text	Up to 16,777,215 characters
MEDIUMBLOB	Medium binary	Up to 16,777,215 bytes
LONGTEXT	Very large text	Up to 4,294,967,295 characters
LONGBLOB	Very large binary	Up to 4,294,967,295 bytes

Integer Types:

Data Type	Range
TINYINT	-128 to 127
SMALLINT	-32,768 to 32,767
MEDIUMINT	-8,388,608 to 8,388,607
INT	-2,147,483,648 to 2,147,483,647
BIGINT	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Note: Use UNSIGNED to store only positive values and increase the upper limit.

Floating-Point Types:

Data Type	Precision
FLOAT	Approx. 7 digits
DOUBLE	Approx. 15 digits
DECIMAL(p, s)	Fixed precision (stored as string)

Date and Time Types:

Data Type	Format
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	Similar to DATETIME, auto-updated
TIME	HH:MM:SS
YEAR	YYYY

Other Data Types:

Data Type	Description
ENUM('x','y')	One value from a predefined list
SET('x','y','z')	One or more values from a list
BOOLEAN	True (1) or False (0)
BIT(n)	Stores n bits ($n \leq 64$)

---

6. General Notes:

- **Size Hierarchy:** TINY < SMALL < MEDIUM < INT < BIGINT
- **Variable-length types** (like VARCHAR) **optimize space** by only using the space needed.
- Always choose the **appropriate data type** to improve performance and save storage.

LEC-2: Types of SQL Commands

Types of SQL Commands

SQL commands are categorized based on their purpose in interacting with the database. There are **five main types**:

---

1. DDL – Data Definition Language

Purpose: Used to **define and modify** the structure of database objects like tables, schemas, or views.

Command	Description
CREATE	Creates a new **table**, **database**, **index**, or **view**. *Example*: CREATE TABLE Students (ID INT, Name VARCHAR(50));
ALTER	Modifies an existing table structure like **adding**, **removing**, or **modifying** columns. *Example*: ALTER TABLE Students ADD Email VARCHAR(100);
DROP	Deletes a **table**, **database**, or **view** completely. *Example*: DROP TABLE Students;
TRUNCATE	Deletes **all rows** from a table but keeps the structure. *Example*: TRUNCATE TABLE Students;
RENAME	Renames a **table**, **column**, or **database**. *Example*: RENAME TABLE Students TO Alumni;

---

2. DQL / DRL – Data Query Language / Data Retrieval Language

Purpose: Used to **query** or **retrieve data** from a database.

Command	Description
SELECT	Used to **fetch data** from one or more tables. *Example*: SELECT Name FROM Students WHERE ID = 101;

Note: While commonly referred to as DQL or DRL, this category contains only the SELECT command but is critical in SQL operations.

---

3. DML – Data Manipulation Language

Purpose: Used to **insert**, **update**, and **delete** data in database tables.

Command	Description
INSERT	Adds new rows into a table. *Example*: INSERT INTO Students (ID, Name) VALUES (101, 'John');
UPDATE	Modifies existing records. *Example*: UPDATE Students SET Name = 'Alice' WHERE ID = 101;
DELETE	Removes one or more records from a table. *Example*: DELETE FROM Students WHERE ID = 101;

---

4. DCL – Data Control Language

Purpose: Used to **control access** to data and permissions in the database.

Command	Description
GRANT	Gives user-specific privileges to access or manipulate the database. *Example*: GRANT SELECT ON Students TO user1;
REVOKE	Removes previously granted privileges. *Example*: REVOKE SELECT ON Students FROM user1;

---

5. TCL – Transaction Control Language

Purpose: Used to **manage transactions** to ensure data integrity and consistency.

Command	Description

Command	Description
START TRANSACTION	Begins a database transaction.
COMMIT	Saves all changes made during the transaction permanently. *Example*: COMMIT;
ROLLBACK	Cancels all changes made in the current transaction. *Example*: ROLLBACK;
SAVEPOINT	Sets a checkpoint in a transaction to **rollback** to that point later. *Example*: SAVEPOINT sp1; → ROLLBACK TO sp1;

 Summary Table

Type	Full Form	Purpose	Key Commands
DDL	Data Definition Language	Define/alter structure	CREATE, ALTER, DROP, TRUNCATE, RENAME
DQL/DRL	Data (Query/Retrieval) Language	Retrieve data	SELECT
DML	Data Manipulation Language	Modify data	INSERT, UPDATE, DELETE
DCL	Data Control Language	Access control	GRANT, REVOKE
TCL	Transaction Control Language	Transaction management	START TRANSACTION, COMMIT, ROLLBACK, SAVEPOINT

LEC-3: DDL (Data Definition Language)

MANAGING DATABASES USING DDL (Data Definition Language)

What is DDL?

DDL (Data Definition Language) is a subset of SQL used to define and manage the **structure of databases and database objects** such as schemas, tables, indexes, and views.

---

1. Creation of Database

To store and manage data, we first need to **create a database**. Here's how you do that:

CREATE DATABASE

CREATE DATABASE IF NOT EXISTS mydb;

- **Purpose:** Creates a new database.
- **IF NOT EXISTS:** Prevents errors if the database already exists.
- **mydb** is the name of the database you want to create.

Example:

CREATE DATABASE IF NOT EXISTS school_db;

✔ This command creates a database named school_db only if it doesn't already exist.

---

2. Selecting the Active Database

USE

USE mydb;

- **Purpose:** Tells the SQL engine which database to use for the upcoming operations like CREATE TABLE, INSERT, etc.
- This is essential when multiple databases exist on the server.

Example:

USE school_db;

✔ After this, any table creation or manipulation will happen within school_db.

□ 3. Dropping a Database

□ DROP DATABASE

DROP DATABASE IF EXISTS mydb;

- **Purpose:** Permanently deletes a database and all its contents.
- **IF EXISTS:** Prevents an error if the database does not exist.

□ **Caution:** This will **delete all tables and data** inside the database.

□ **Example:**

DROP DATABASE IF EXISTS school_db;

✔ Deletes school_db if it exists.

□ 4. Listing All Databases

□ SHOW DATABASES

SHOW DATABASES;

- **Purpose:** Lists all the databases present on the server.
- Useful to verify if a database exists before using or dropping it.

□ 5. Listing Tables in the Selected Database

□ SHOW TABLES

SHOW TABLES;

- **Purpose:** Displays all the tables in the **currently selected** (using USE) database.

□ **Example:**

USE school_db;

SHOW TABLES;

 Summary of DDL Commands for Managing DB:

Command	Description
CREATE DATABASE IF NOT EXISTS db_name;	Creates a new database if it doesn't already exist.
USE db_name;	Selects the database to run further SQL commands on.
DROP DATABASE IF EXISTS db_name;	Deletes the database if it exists.
SHOW DATABASES;	Lists all databases on the server.
SHOW TABLES;	Lists all tables within the selected database.

LEC-4: DATA RETRIEVAL LANGUAGE (DRL / DQL)

DATA RETRIEVAL LANGUAGE (DRL / DQL)

□ What is DRL?

DRL (Data Retrieval Language), also known as **DQL (Data Query Language)**, is used to **fetch/query data** from a relational database.

---

1. Basic Syntax

SELECT <column_names> FROM <table_name>;

- SELECT → Tells the database what columns you want.
- FROM → Specifies the table to fetch the data from.

□ Example:

SELECT name, age FROM students;

---

2. Order of Execution (RIGHT to LEFT)

Internally, SQL executes statements in a specific logical order (not necessarily top-down):

Execution Order:

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT

This is why you must define the table in FROM before applying WHERE or GROUP BY.

---

3. SELECT Without FROM (DUAL Table)

Yes, we can use SELECT without FROM using **DUAL**, a dummy table in MySQL.

Purpose of DUAL Table

- Used to **evaluate expressions**, **functions**, or **constants** without needing actual tables.

□ Examples:

```
SELECT 55 + 11;
SELECT NOW();          -- current timestamp
SELECT UCASE('hello');   -- HELLO
```

These use the internal DUAL table implicitly.

---

4. WHERE Clause

Used to filter rows based on a condition.

 Example:

```
SELECT * FROM customers WHERE age > 18;
```

✔ Shows only customers whose age is greater than 18.

---

5. BETWEEN Clause

Filters records within a range (inclusive of boundaries).

 Example:

```
SELECT * FROM customers WHERE age BETWEEN 0 AND 100;
```

✔ Includes age = 0 and age = 100.

---

6. IN Clause

Reduces the need for multiple OR conditions.

 Example:

```
SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');
```

✔ Instead of writing WHERE officer_name = 'A' OR officer_name = 'B'...

---

7. Logical Operators: AND / OR / NOT

Operator	Description
AND	Both conditions must be true
OR	At least one condition must be true
NOT	Negates the condition

 Examples:

```
SELECT * FROM customer WHERE age > 18 AND city = 'Delhi';
SELECT * FROM customer WHERE age < 18 OR city = 'Mumbai';
SELECT * FROM customer WHERE id NOT IN (1, 2, 3);
```

---

8. IS NULL Clause

To find rows with missing/NULL values.

 Example:

```
SELECT * FROM customer WHERE prime_status IS NULL;
```

---

9. Pattern Searching (Wildcards using LIKE)

Wildcard	Meaning
%	Any number of characters (0 or more)
_	Exactly one character

 Example:

```
SELECT * FROM customer WHERE name LIKE '%p_';
```

✔ Matches names ending in **‘p’ + any single character** (e.g., "deep", "keep").

---

10. ORDER BY Clause

Sorts the output by one or more columns.

SELECT * FROM customer ORDER BY name DESC;

Keyword	Meaning
ASC	Ascending (default)
DESC	Descending

---

11. GROUP BY Clause

Used to group rows with the same value into summary rows.

SELECT country, COUNT(*) FROM customer GROUP BY country;

- You **must include** in GROUP BY all non-aggregated columns from SELECT.
- Often used with aggregate functions:

Function	Use
COUNT()	Number of rows
SUM()	Total value
AVG()	Average value
MIN()	Smallest value
MAX()	Largest value

 Example:

SELECT city, COUNT(*) FROM customers GROUP BY city;

---

12. DISTINCT Keyword

Returns only unique/distinct values in a column.

SELECT DISTINCT city FROM customers;

Same as:

SELECT city FROM customers GROUP BY city;

- SQL is smart enough to treat GROUP BY without aggregation as DISTINCT.

---

13. GROUP BY with HAVING

Filters groups after aggregation. Used after GROUP BY, unlike WHERE.

 Example:

```
SELECT COUNT(customer_id), country
FROM customer
GROUP BY country
HAVING COUNT(customer_id) > 50;
```

Clause	Filters On
WHERE	Individual Rows
HAVING	Aggregated Groups

LEC-5: SQL Constraints

□ SQL Constraints (DDL)

Constraints are **rules enforced on data in tables** to ensure **accuracy and integrity** of the database.

They are usually defined at the time of table creation using **CREATE TABLE**, or can be added later using **ALTER TABLE**.

---

1. Primary Key (PK)

➢ Definition:

- A **Primary Key** uniquely identifies each record in a table.
- It ensures **two things**:
  - **NOT NULL**: Cannot be empty.
  - **UNIQUE**: No duplicates allowed.
- A table can have **only one Primary Key** (which can consist of single or multiple columns - **composite key**).

➢ Example:

```
CREATE TABLE customer (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   age INT
);
```

- Here, id is the primary key, so each customer must have a unique id and it **cannot be NULL**.

---

2. Foreign Key (FK)

➢ Definition:

- A **Foreign Key** in one table points to a **Primary Key** in another table.
- Used to maintain **referential integrity** between two related tables.
- A table can have **multiple foreign keys**.

➢ **Example:**

```
CREATE TABLE orders (
   id INT PRIMARY KEY,
   delivery_date DATE,
   order_placed_date DATE,
   cust_id INT,
   FOREIGN KEY (cust_id) REFERENCES customer(id)
);
```

- Here, cust_id in the orders table refers to the id column in the customer table.
- This ensures that an order can only be placed for an existing customer.

➢ **Note:**

- A column can be **both a Primary Key and a Foreign Key**, e.g., in weak entities or junction tables (many-to-many relationships).

---

3. UNIQUE

➢ **Definition:**

- Ensures **all values in a column are different**.
- Unlike Primary Key, the **UNIQUE constraint can accept NULLs**, and a table can have **multiple UNIQUE constraints**.

➢ **Example:**

```
CREATE TABLE customer (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   email VARCHAR(1024) UNIQUE
);
```

- email must be unique, but it **can be NULL** if not provided.

---

4. CHECK

➢ **Definition:**

- Used to **limit the range of values** that can be placed in a column.

- Validates the condition before inserting or updating data.

➢ **Example:**

```
CREATE TABLE customer (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   age INT,
   CONSTRAINT age_check CHECK (age > 12)
);
```

- This ensures that the age of any customer must be **greater than 12**.

➢ **Note:**

- age_check is a custom name for the constraint.
- If you don't name it, MySQL or the DBMS will generate a default name.

---

5. DEFAULT

➢ **Definition:**

- Assigns a **default value** to a column **if no value is specified** during insertion.

➢ **Example:**

```
CREATE TABLE account (
   id INT PRIMARY KEY,
   account_holder VARCHAR(100),
   saving_rate DOUBLE NOT NULL DEFAULT 4.25
);
```

- If the saving_rate is not specified in the INSERT command, it will default to 4.25.

---

6. Multiple Constraints Together

- You can combine constraints in one column:

```
CREATE TABLE employee (
   emp_id INT PRIMARY KEY,
   email VARCHAR(255) UNIQUE,
```

```
   salary DECIMAL(10,2) CHECK (salary >= 30000),
   department_id INT,
   FOREIGN KEY (department_id) REFERENCES department(dept_id)
);
```

---

7. Can a Column Be Both PK and FK?

Yes!

- In certain designs like **junction tables** or **weak entities**, a column may be:
  - **Primary Key**: Uniquely identifies the record.
  - **Foreign Key**: Connects to another table's primary key.

➤ **Example:**

```
CREATE TABLE student_login (
   student_id INT PRIMARY KEY,
   password VARCHAR(255),
   FOREIGN KEY (student_id) REFERENCES student(id)
);
```

- Here, student_id is **both a PK and FK**, ensuring only valid student accounts are created.

LEC-6: ALTER OPERATIONS (DDL)

ALTER OPERATIONS (DDL)

The ALTER command allows you to **change the structure of an existing table** without deleting the table or losing its data.

---

1. Purpose of ALTER

- To **change the schema** (structure) of a table.
- Modify columns by adding, dropping, changing datatype, renaming columns or tables.

---

2. ADD

➢ **What it does:**

- Adds a **new column(s)** to an existing table.

➢ **Syntax:**

```
ALTER TABLE table_name
ADD new_col_name datatype;
```

- You can add multiple columns in one statement:

```
ALTER TABLE table_name
ADD new_col_name1 datatype, ADD new_col_name2 datatype;
```

➢ **Example:**

```
ALTER TABLE customer
ADD age INT NOT NULL;
```

- Adds a new column age of type INT to the customer table, and it cannot be NULL.

---

3. MODIFY

➢ **What it does:**

- Changes the **datatype** or definition of an existing column.
- Useful when you want to change size or data type (e.g., VARCHAR to CHAR).

➢ **Syntax:**

```
ALTER TABLE table_name
MODIFY column_name new_datatype;
```

➢ **Example:**

```
ALTER TABLE customer
MODIFY name CHAR(1024);
```

- Changes the data type of the name column to CHAR(1024) from whatever it was before.

---

4. CHANGE COLUMN

➢ **What it does:**

- Renames a column and/or changes its datatype in one go.

➢ **Syntax:**

```
ALTER TABLE table_name
CHANGE COLUMN old_column_name new_column_name new_datatype;
```

➢ **Example:**

```
ALTER TABLE customer
CHANGE COLUMN name customer_name VARCHAR(1024);
```

- Renames the column name to customer_name and changes its datatype to VARCHAR(1024).

---

5. DROP COLUMN

➢ **What it does:**

- Removes a column completely from the table.

➢ **Syntax:**

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

➢ **Example:**

```
ALTER TABLE customer
DROP COLUMN middle_name;
```

- Deletes the column middle_name from the customer table.

---

6. RENAME TABLE

➢ **What it does:**

- Changes the **name of the entire table**.

➢ **Syntax:**

```
ALTER TABLE old_table_name
RENAME TO new_table_name;
```

➢ **Example:**

```
ALTER TABLE customer
RENAME TO customer_details;
```

- Renames the table customer to customer_details.

Notes:

- The exact syntax might vary slightly depending on the SQL database (MySQL, PostgreSQL, Oracle, etc.).
- MODIFY is often used in MySQL, while some databases prefer ALTER COLUMN for modifying.
- When renaming a column, **both new column name and datatype must be specified** in MySQL with CHANGE COLUMN.

LEC-7: DATA MANIPULATION LANGUAGE (DML)

DATA MANIPULATION LANGUAGE (DML)

DML commands help you **work with the data inside database tables**. Unlike DDL, which changes table structure, DML changes the data stored.

---

1. INSERT

- Used to **add new rows (records)** into a table.
- You specify columns and their corresponding values.

Syntax:

```
INSERT INTO table_name (col1, col2, col3)
VALUES (val1, val2, val3), (val4, val5, val6);
```

- You can insert **multiple rows** in one statement by separating values with commas.

Example:

```
INSERT INTO student (id, name, age)
VALUES (1, 'Alice', 20), (2, 'Bob', 22);
```

- Adds two new rows into the student table.

---

2. UPDATE

- Used to **modify existing records** in a table.
- Specify columns to update and conditions to limit which rows are changed.

Syntax:

```
UPDATE table_name
SET col1 = value1, col2 = value2
WHERE condition;
```

Example (update a specific row):

```
UPDATE student
SET age = 21
```

```
WHERE id = 1;
```

- Changes age of the student with id=1 to 21.

Update multiple rows example:

```
UPDATE student
SET standard = standard + 1;
```

- Increments the standard column for **all rows** by 1.

---

3. ON UPDATE CASCADE

- Related to **foreign key constraints**.
- When the **primary key (PK)** of a parent table is updated, this option **automatically updates corresponding foreign keys (FK)** in the child table.
- Keeps data consistent without manual updates.

Example:

If you have tables:

- customer(id, name)
- order(order_id, cust_id) where cust_id is FK referencing customer(id)

```
CREATE TABLE order (
  order_id INT PRIMARY KEY,
  delivery_date DATE,
  cust_id INT,
  FOREIGN KEY (cust_id) REFERENCES customer(id) ON UPDATE CASCADE
);
```

- If a customer.id is updated, corresponding order.cust_id is automatically updated.

---

4. DELETE

- Used to **remove rows** from a table.
- Use a condition to delete specific rows or no condition to delete all rows.

Syntax:

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

Example (delete specific row):

DELETE FROM student WHERE id = 1;

- Deletes student with id=1.

Delete all rows:

DELETE FROM table_name;

- Removes **all rows** from the table but keeps the table structure intact.

---

5. DELETE CASCADE

- Another **foreign key constraint option**.
- When a **parent row is deleted**, all child rows referencing it in other tables are **automatically deleted**.
- Helps avoid orphan records in child tables.

Example:

```
CREATE TABLE order (
  order_id INT PRIMARY KEY,
  delivery_date DATE,
  cust_id INT,
  FOREIGN KEY (cust_id) REFERENCES customer(id) ON DELETE CASCADE
);
```

- If a customer row is deleted, all order rows with that cust_id are deleted automatically.

---

6. ON DELETE SET NULL

- Another FK constraint option.
- When the parent row is deleted, the FK column in the child row is set to **NULL**.
- This requires that FK columns allow NULL values.

Example:

```
CREATE TABLE order (
  order_id INT PRIMARY KEY,
  delivery_date DATE,
  cust_id INT,
```

```
  FOREIGN KEY (cust_id) REFERENCES customer(id) ON DELETE SET NULL
);
```

- Deleting a customer sets cust_id to NULL in corresponding orders.

---

7. REPLACE

- A MySQL-specific DML command combining **INSERT and UPDATE**.
- If a row with the **same primary key or unique key exists**, it **deletes** the old row and **inserts** the new row.
- If no such row exists, it simply inserts the new row.
- Acts like an **upsert** (update or insert).

Syntax:

```
REPLACE INTO table_name (col1, col2)
VALUES (val1, val2);
```

or

```
REPLACE INTO table_name SET col1 = val1, col2 = val2;
```

Example:

```
REPLACE INTO student (id, class) VALUES (4, 3);
```

- If student with id=4 exists, it's replaced with new values.
- Otherwise, a new row is inserted.

LEC-8: JOINING TABLES

JOINING TABLES

Relational databases (RDBMS) are designed to store data in tables, which are related to each other through **keys**. To get meaningful insights or combine data from multiple tables, we use **joins**.

---

1. Why Join Tables?

- RDBMS stores data across multiple tables to reduce redundancy.

- You use **foreign keys (FK)** to reference related data in other tables.
- Joining tables helps you **combine related rows from two or more tables** into a single result set.

---

2. Foreign Key (FK) Reference

- FK in one table refers to the **primary key (PK)** of another table.
- FK enforces referential integrity and allows joining tables meaningfully.

---

3. INNER JOIN

- Returns rows where **matching values exist in both tables**.
- Only rows with matching keys appear in the result.
- Most common join type.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.key = table2.key
INNER JOIN table3 ON table2.key2 = table3.key2
...;
```

Example:

```
SELECT customer.name, order.order_id
FROM customer
INNER JOIN order ON customer.id = order.cust_id;
```

- Returns only customers who have placed orders.

---

4. Alias in SQL

- Temporary names to make queries easier and cleaner.
- Can rename columns or tables within a query for convenience.

Examples:

```
SELECT col_name AS alias_name FROM table_name;
SELECT col1, col2 FROM table_name AS t;
```

E.g.,

```
SELECT c.name AS customer_name, o.order_id
FROM customer AS c
INNER JOIN order AS o ON c.id = o.cust_id;
```

---

5. OUTER JOIN

Returns matching rows **plus** unmatched rows from one or both tables.

---

a) LEFT JOIN (or LEFT OUTER JOIN)

- Returns **all rows from the left table** plus matching rows from the right table.
- If no match, right table columns show NULL.

Syntax:

```
SELECT columns
FROM left_table
LEFT JOIN right_table ON condition;
```

Example:

```
SELECT customer.name, order.order_id
FROM customer
LEFT JOIN order ON customer.id = order.cust_id;
```

- Shows all customers even if they haven't placed any orders.

---

b) RIGHT JOIN (or RIGHT OUTER JOIN)

- Returns **all rows from the right table** plus matching rows from the left table.
- If no match, left table columns show NULL.

Syntax:

```
SELECT columns
FROM left_table
RIGHT JOIN right_table ON condition;
```

---

c) FULL JOIN (or FULL OUTER JOIN)

- Returns **all rows** when there is a match in **either** left or right table.
- Shows NULLs for missing matches on either side.
- Not directly supported in MySQL, but can be emulated.

MySQL Emulation:

```
SELECT columns FROM table1 LEFT JOIN table2 ON condition
UNION
SELECT columns FROM table1 RIGHT JOIN table2 ON condition;
```

- UNION removes duplicates; UNION ALL includes duplicates.

---

6. CROSS JOIN

- Returns **Cartesian product** — every row from first table combined with every row from second table.
- Number of rows in result = rows in table1 × rows in table2.
- Rarely used in practice unless explicitly required.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

---

7. SELF JOIN

- Join a table **to itself**.
- Useful for comparing rows within the same table or hierarchical data.

Syntax:

```
SELECT a.columns, b.columns
FROM table AS a
INNER JOIN table AS b ON a.some_col = b.some_col;
```

Example:

```
SELECT e1.name AS Employee, e2.name AS Manager
FROM employees AS e1
INNER JOIN employees AS e2 ON e1.manager_id = e2.id;
```

---

8. Join Without Explicit JOIN Keyword

- Old-style join syntax uses **comma** separated tables with join condition in WHERE clause.

Syntax:

```
SELECT columns
FROM table1, table2
WHERE table1.key = table2.key;
```

Example:

```
SELECT artist_name, album_name, year_recorded
FROM artist, album
WHERE artist.id = album.artist_id;
```

- This is equivalent to an INNER JOIN.

LEC-9: SET OPERATIONS

SET OPERATIONS

Set operations allow you to combine results from two or more SELECT statements into a single result. These operations work **row-wise** and treat the result sets like mathematical sets, often returning **distinct rows** by default.

Key Differences: JOIN vs SET OPERATIONS

Aspect	JOIN	SET OPERATIONS
Purpose	Combines tables **based on matching column(s)**	Combines results of multiple **SELECT queries**
Result Combination	**Column-wise** (adds columns)	**Row-wise** (adds rows)
Data Type Requirements	Columns used in join must have compatible data	Columns in each SELECT must have compatible data

Aspect	JOIN	SET OPERATIONS
Number of Columns	Can differ	Must be the **same** in each SELECT
Output Rows	Based on join condition, may include duplicates	Usually returns **distinct rows** by default

---

1. UNION

- Combines the result sets of two or more SELECT statements **vertically**.
- Removes duplicate rows (returns **distinct** rows).
- The number of columns and their order must be the **same** in all SELECT statements.
- Data types in corresponding columns should be compatible.

Syntax:

```
SELECT col1, col2, ... FROM table1
UNION
SELECT col1, col2, ... FROM table2;
```

Example:

```
SELECT id, name FROM employees
UNION
SELECT id, name FROM managers;
```

- This returns all distinct employees and managers combined.

---

2. INTERSECT

- Returns only the rows that **appear in both** SELECT result sets.
- Gives the **common rows** between two tables.
- Not always directly supported (e.g., MySQL), but can be emulated.

Emulation using INNER JOIN:

```
SELECT DISTINCT t1.column_list
FROM table1 t1
INNER JOIN table2 t2 ON t1.common_column = t2.common_column;
```

Example:

```
SELECT DISTINCT id FROM employees
INNER JOIN managers USING(id);
```

- Returns IDs common to both employees and managers.

---

3. MINUS (or EXCEPT)

- Returns rows from the **first SELECT** that are **not present** in the second SELECT.
- Shows what is in table1 but **not** in table2.
- Not always directly supported, but can be emulated.

Emulation using LEFT JOIN and NULL check:

```
SELECT t1.column_list
FROM table1 t1
LEFT JOIN table2 t2 ON t1.common_column = t2.common_column
WHERE t2.common_column IS NULL;
```

Example:

```
SELECT id FROM employees
LEFT JOIN managers USING(id)
WHERE managers.id IS NULL;
```

- Returns employees who are **not** managers.

Important Notes:

- **Column count and data types:** For UNION, INTERSECT, and MINUS, each SELECT must return the **same number of columns** with compatible data types.
- **Duplicates:** UNION removes duplicates; UNION ALL includes duplicates.
- **Column order:** Must be consistent across SELECTs.
- **Use cases:** Set operations are useful for merging, finding common, or excluding data across tables.

LEC-10: SUB QUERIES

SUB QUERIES

A **subquery** (or nested query) is a query nested inside another query — the **outer query** depends on the result of this **inner query**. Subqueries are an alternative to JOINs and help retrieve data based on conditions derived from other tables or the same table.

Key Points:

- A subquery is written inside parentheses ().
- It can return a single value, a list of values, or a whole table.
- Subqueries are commonly used inside WHERE, FROM, and SELECT clauses.
- Can be **correlated** (dependent on outer query) or **non-correlated** (independent).

Basic Syntax:

```
SELECT column_list
FROM table_name
WHERE column_name OPERATOR
(SELECT column_list FROM another_table WHERE condition);
```

Example:

```
SELECT * FROM employees WHERE department_id IN
(SELECT department_id FROM departments WHERE location = 'New York');
```

- Here, the inner query finds all departments in New York.
- The outer query finds all employees who work in those departments.

Where Subqueries Appear:

1. Inside WHERE Clause

- Most common place.
- Filters outer query results based on inner query.

```
SELECT * FROM students WHERE student_id IN (SELECT student_id FROM courses
WHERE course_name = 'Math');
```

2. Inside FROM Clause

- Acts like a temporary table (also called a **derived table**).
- You can run aggregate functions or further filtering on this temp result.

```
SELECT MAX(rating) FROM (SELECT * FROM movies WHERE country = 'India') AS temp;
```

- The inner subquery gets all Indian movies.
- The outer query finds the max rating from these movies.

3. Inside SELECT Clause

- Used to get values from another table for each row of the outer query.

```
SELECT
  employee_name,
  (SELECT department_name FROM departments WHERE departments.department_id =
employees.department_id) AS dept_name
FROM employees;
```

Derived Subquery

- Subquery inside FROM clause acts like a virtual table.

```
SELECT new_table.col1, new_table.col2
FROM (SELECT col1, col2 FROM table_name WHERE condition) AS new_table;
```

Correlated Subqueries

- The inner query depends on each row of the outer query.
- Executes once for every row processed by the outer query.

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

- For each employee, it compares salary to the average salary in their department.

JOINs vs SUBQUERIES

Aspect	JOIN	SUBQUERY
Speed	Faster	Generally slower
Complexity	More complex, harder to understand	Simpler and easier to write/read
Execution	DBMS handles the calculations	User controls calculation responsibility
Use cases	Good for combining tables horizontally	Good for filtering and nested queries
Optimization	Requires choosing best join type	Easier to implement, but less optimized

LEC-11: MySQL VIEWS

MySQL VIEWS

1. What is a View?

- A **View** is a **virtual table** in the database.
- It **does not store data itself** but shows data dynamically from one or more base tables.
- It has **rows and columns like a real table**, but its content is defined by a query on the underlying tables.

---

2. How Does a View Work?

- Views are created using **SELECT queries** that can join multiple tables.
- When you query a view, it runs the underlying query and shows you the current data.
- Any **changes in the base tables** (inserts, updates, deletes) are **automatically reflected in the view** because the view always fetches fresh data from those tables.

---

3. Why Use Views?

- To **simplify complex queries** by saving them as a reusable virtual table.
- To **restrict access** to sensitive data by showing only specific columns or rows.
- To **present data in a particular format** without changing the actual tables.

- Views can be used in queries just like normal tables.

---

4. Creating, Altering, and Dropping Views

- **Create a view:**

  CREATE VIEW view_name AS
  SELECT columns FROM tables [WHERE conditions];

- **Alter a view:** (modify the query behind the view)

  ALTER VIEW view_name AS
  SELECT columns FROM tables WHERE conditions;

- **Drop a view:**

  DROP VIEW IF EXISTS view_name;

---

5. Example of a View Using Join

Suppose you have two tables courses and contact and want a view that shows course name, trainer, and trainer's email.

CREATE VIEW Trainer AS
SELECT c.course_name, c.trainer, t.email
FROM courses c
JOIN contact t ON c.id = t.id;

- Here, Trainer is a view that combines data from courses and contact.
- Whenever you query Trainer, it shows joined data from those tables.

---

6. Additional Notes

- Views can be **exported or imported** similarly to tables (schemas can be saved as .csv or .json files).
- Views **do not store data**, so they do not consume much disk space.
- Since views are query results, they may sometimes be slower than querying base tables directly, especially for complex views.

LEC-12: SQL Functions

SQL Functions

---

1. Aggregate Functions

Aggregate functions perform calculations on multiple rows and return a single result. They are commonly used with GROUP BY but can also be used alone.

- **COUNT()**: Returns the number of rows or non-null values in a column.

  SELECT COUNT(*) FROM employees;

- **SUM()**: Adds up all the values in a numeric column.

  SELECT SUM(salary) FROM employees;

- **AVG()**: Calculates the average (mean) of a numeric column.

  SELECT AVG(age) FROM employees;

- **MAX()**: Returns the maximum value in a column.

  SELECT MAX(salary) FROM employees;

- **MIN()**: Returns the minimum value in a column.

  SELECT MIN(age) FROM employees;

---

2. String Functions

String functions are used to manipulate text data.

- **CONCAT()**: Concatenates (joins) two or more strings into one.

  SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;

- **LENGTH()**: Returns the length (number of characters) of a string.

  SELECT LENGTH('Hello World'); -- Returns 11

- **UPPER()**: Converts all characters in a string to uppercase.

```
SELECT UPPER('hello');  -- Returns 'HELLO'
```

- **LOWER()**: Converts all characters in a string to lowercase.

  ```
  SELECT LOWER('HELLO');  -- Returns 'hello'
  ```

- **SUBSTRING()**: Extracts a substring from a string starting at a specific position and optionally for a specific length.

  ```
  SELECT SUBSTRING('Hello World', 7, 5);  -- Returns 'World'
  ```

---

3. Date Functions

Date functions allow you to work with date and time data.

- **NOW()**: Returns the current date and time.

  ```
  SELECT NOW();
  ```

- **CURDATE()**: Returns the current date (without time).

  ```
  SELECT CURDATE();
  ```

- **YEAR()**: Extracts the year part from a date.

  ```
  SELECT YEAR('2025-05-20');  -- Returns 2025
  ```

- **MONTH()**: Extracts the month part from a date.

  ```
  SELECT MONTH('2025-05-20');  -- Returns 5
  ```

---

4. Math Functions

Math functions perform mathematical operations.

- **ROUND()**: Rounds a number to the nearest integer or to a specified number of decimal places.

  ```
  SELECT ROUND(3.14159, 2);  -- Returns 3.14
  ```

- **FLOOR()**: Returns the largest integer less than or equal to a number (rounds down).

  ```
  SELECT FLOOR(3.9);  -- Returns 3
  ```

- **CEIL()** or **CEILING()**: Returns the smallest integer greater than or equal to a number (rounds up).

  SELECT CEIL(3.1); -- Returns 4

- **ABS()**: Returns the absolute (non-negative) value of a number.

  SELECT ABS(-10); -- Returns 10

LEC-12: Advanced SQL Concepts

Advanced SQL Concepts

---

1. Indexes

- **What is an Index?**
  An index is a database object that improves the speed of data retrieval operations on a table at the cost of additional writes and storage. Think of it like an index in a book — it helps you find information quickly without scanning every page.
- **How it works:**
  When a query searches for data using columns with an index, the database uses the index to quickly locate the rows instead of scanning the whole table (a full table scan).
- **Types of Indexes:**
  - **Primary Index:** Automatically created on the primary key.
  - **Unique Index:** Ensures all values in the indexed column(s) are unique.
  - **Composite Index:** An index on multiple columns.
  - **Full-text Index:** Used for text searching.
- **Example:**

  CREATE INDEX idx_employee_name ON employees(last_name);

- **Benefits:**
  Faster SELECT queries, especially with large tables.
- **Trade-off:**
  Slower INSERT, UPDATE, and DELETE because indexes must be updated.

---

2. Stored Procedures

- **What is a Stored Procedure?**
  A stored procedure is a precompiled SQL program stored in the database. It can contain SQL statements and procedural logic, and can accept parameters.
- **Why use Stored Procedures?**
  - Encapsulate complex operations.
  - Promote code reuse and maintainability.
  - Reduce network traffic (multiple SQL statements can be executed in one call).
  - Provide security by controlling access to data.
- **Basic Structure:**

```
DELIMITER //
CREATE PROCEDURE GetEmployeeByID(IN emp_id INT)
BEGIN
  SELECT * FROM employees WHERE employee_id = emp_id;
END //
DELIMITER ;
```

- **Calling a Procedure:**

```
CALL GetEmployeeByID(101);
```

---

3. Triggers

- **What is a Trigger?**
  A trigger is a special kind of stored procedure that automatically executes in response to certain events on a table, such as INSERT, UPDATE, or DELETE.
- **When are Triggers used?**
  - Enforce complex business rules.
  - Maintain audit trails (track changes).
  - Synchronize tables.
  - Validate data automatically.
- **Types of Triggers:**
  - **BEFORE Trigger:** Executes before the triggering event.
  - **AFTER Trigger:** Executes after the triggering event.
- **Example:**

```
CREATE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  SET NEW.created_at = NOW();
END;
```

This example automatically sets a created_at timestamp whenever a new employee record is inserted.

---

4. Transactions

- **What is a Transaction?**
  A transaction is a sequence of one or more SQL operations treated as a single logical unit of work. Transactions ensure **data integrity** even if there are failures or concurrent access.
- **ACID Properties of Transactions:**
  - **Atomicity:** All operations succeed or none do.
  - **Consistency:** Database moves from one valid state to another.
  - **Isolation:** Transactions run independently without interference.
  - **Durability:** Once committed, changes are permanent.
- **Common Transaction Commands:**
  - BEGIN or START TRANSACTION: Begins a new transaction.
  - COMMIT: Saves all changes made during the transaction.
  - ROLLBACK: Undoes all changes if something goes wrong.
- **Example:**

  START TRANSACTION;

  UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

  COMMIT;

  This ensures that money is deducted from one account and added to another as a single atomic operation — if either update fails, none of the changes are applied.