Structured Query Language:

- ## 1. What is SQL?
  - SQL is a language used to communicate with relational databases. It allows you to create, read, update, and delete data — commonly known as CRUD operations**.**
  - SQL or Relational databases are used to store and manage the data objects that are related to one another.
  - A system used to manage these relational databases is known as Relational Database Management System (RDBMS).
- ## What Is a Database?
  - A database is a structured collection of data that is stored in a computer system.
  - A Database may contain many tables.
- ## What is RDBMS?
  - RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- ## What is a Table?
  - The data in an RDBMS is stored in database objects known as tables. This table is basically a collection of related data entries and it consists of numerous columns and rows.
  - a table is the most common and simplest form of data storage in a relational database

ID	Name	Age	Salary	City	Country
1	Ramesh	32	2000.00	Hyderabad	India
2	Mukesh	40	5000.00	New York	USA
3	Sumit	45	4500.00	Muscat	Oman
4	Kaushik	25	2500.00	Kolkata	India
5	Hardik	29	3500.00	Bhopal	India
6	Komal	38	3500.00	Saharanpur	India
7	Ayush	25	3500.00	Delhi	India
8	Javed	29	3700.00	Delhi	India

  - What is a Field → like ID, Name, Age, Salary, City and Country.
  - What is a Record or a Row?
    - A record is also called as a row of data is each individual entry that exists in a table

ID	Name	Age	Salary	City	Country
1	Ramesh	32	2000.00	Hyderabad	India

- ## Keys?

- **UNIQUE Key:** Ensures that all the values in a column are different.here null can also present.

```
CREATE TABLE table_name(
    column1 datatype UNIQUE KEY,
    column2 datatype UNIQUE KEY,
    .....
    .....
    columnN datatype
);
```

- **Primary Key:** A Primary Key is a column that uniquely identifies each row in a table.
    - **Characteristics:**
        - Must be **unique** for each row.
        - Cannot be **NULL**.
        - Each table can have **only one** primary key.

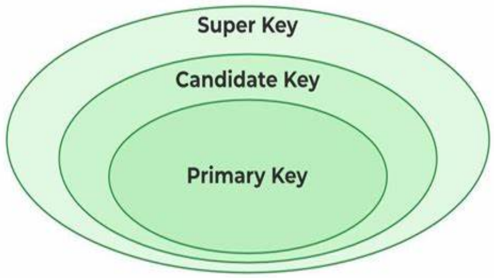
    - CREATE TABLE Customers (
      CustomerID INT PRIMARY KEY,
      Name VARCHAR(100),
      Email VARCHAR(100)
      );
        - `CustomerID` is the primary key — no two customers can have the same `CustomerID`.

- **Foreign Key:** A **Foreign Key** is a column that **references the primary key** of another table. It's used to create a **relationship between two tables**.
    - **Characteristics:**
        - Ensures **referential integrity** between related tables.
        - Can have **duplicate** values.
        - Can contain **NULLs**, unless specified otherwise.

    - ```
      CREATE TABLE Orders (
          OrderID INT PRIMARY KEY,
          OrderDate DATE,
          CustomerID INT,
          FOREIGN KEY (CustomerID) REFERENCES
      Customers(CustomerID)
      );
      ```

      In this example:

        - `CustomerID` in `Orders` is a foreign key.
        - It references `CustomerID` in the `Customers` table.
        - This means each order must be linked to a valid customer.
      ```

- **Candidate Key** — any column or set of columns that can uniquely identify each row.
    - There can be **one or more candidate keys** in a table.
      But **only one** of them is chosen as the **Primary Key**.
    - So Candidate Key Satisfy all conditions of primary key.
    - 

- **Composite Primary Key**
    - In many-to-many relationships (like students enrolled in courses), **no single column uniquely identifies a row**.
    - So we **combine two columns** to form a **composite primary key**.

Enrollment Table (Before Normalization)

StudentID	CourseID	Grade
1	CS101	A
2	CS101	B
1	MATH201	A-

Here:

- No single column (`StudentID` or `CourseID`) can uniquely identify a row.
- But the **pair** `(StudentID, CourseID)` **is unique** for each row.

- ✅ So `(StudentID, CourseID)` is a **composite primary key**.

- ## Database Normalization:
    - Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process –
        1. Eliminating redundant data, for example, storing the same data in more than one table.
        2. Ensuring data dependencies make sense.
    - Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure.
    - Normalization guidelines are divided into normal forms; think of a form as the format or the way a database structure is laid out.
    - **Normal Forms (NF):**
        - **First Normal Form (1NF)**
            - **Rule: No repeating groups or arrays. Each field should hold atomic (indivisible) values.**

❌ Not in 1NF:

OrderID	CustomerName	Products
1	Alice	Pen, Notebook

✅ In 1NF:

OrderID	CustomerName	Product
1	Alice	Pen
1	Alice	Notebook

- **Second Normal Form (2NF)**
  - A table is in 2NF if:
    1. It is already in 1NF.
    2. Every non-key column is fully dependent on the whole primary key, not just part of it.
    3. A **partial dependency** happens when a column depends **only on part** of a **composite primary key** , not the whole key.
       🎓 Original Table (1NF but **Not in 2NF**)

StudentID	CourseID	StudentName	CourseName	Instructor
1	CS101	Alice	SQL Basics	Mr. Smith
2	CS101	Bob	SQL Basics	Mr. Smith

    4.
    5. **Composite Primary Key**: `(StudentID, CourseID)`

       Problem:

       a. **StudentName depends only on StudentID**
       b. **CourseName and Instructor depend only on CourseID.**
       c. ➡️ So we have **partial dependencies** – columns depending on **part** of the key, not the **whole key**.

✅ Converting to 2NF (Remove Partial Dependencies)

🔹 **Step 1: Split the data into separate tables**

🧍 **Students Table**

StudentID	StudentName
1	Alice
2	Bob

📚 **Courses Table**

CourseID	CourseName	Instructor
CS101	SQL Basics	Mr. Smith

🟦 **Enrollment Table (links students to courses)**

StudentID	CourseID
1	CS101
2	CS101

6.
    a. Now each non-key column depends fully on its primary key. → StudentName depends on StudentID , Instructor and CourseName depends on CourseID , CourseID depends on StudentID.
    b. No partial dependencies remain.

- If the **primary key is just one column**, you're **already in 2NF** (because partial dependency is not possible).
    - But if the **key is composite**, like `(StudentID, CourseID)`, then:
    - Every non-key column must depend on **both** parts.
    - If a column depends only on `StudentID`, it **violates 2NF**.

- **What is 3NF?**
    - **A table is in Third Normal Form (3NF) if:**
        1. It is in Second Normal Form (2NF) ✅

        2. There is no transitive dependency — i.e., non-key columns do not depend on other non-key columns (or) non-key columns should depend on key columns only.

    - **What is a transitive dependency?**
        - A transitive dependency means a non-key column depends on another non-key column, not directly on the primary key.

- ## 🧱 Example of a Table **NOT in 3NF**

EmpID	EmpName	DeptID	DeptName
101	Alice	D01	HR
102	Bob	D02	IT

  - **Primary Key**: EmpID
  - EmpName and DeptID depend on EmpID ✅
  - But DeptName depends on DeptID and DeptID depends on EmpID ❌
  - EmpID → DeptID → DeptName (**transitive dependency**)
  - **So**, this is Not in 3NF

  ## ✅ How to Fix It (Convert to 3NF)

  Split the table into two:

  ### 1. Employees table

EmpID	EmpName	DeptID
101	Alice	D01
102	Bob	D02

  ### 2. Departments table

DeptID	DeptName
D01	HR
D02	IT

  Now:

  - `DeptName` depends **only on DeptID**, not on `EmpID`
  - No transitive dependencies ✅
- - Table is now in **3NF**

2 NF VS 3 NF: Let Consider a question Which

  - ✅ *Satisfy 2NF: No partial dependencies*
  - ❌ *Fail 3NF: Has a transitive dependency*

  ### 📘 Example Table: `Employees`

EmpID	EmpName	DeptID	DeptName
101	Alice	D01	HR
102	Bob	D02	IT
103	Carol	D01	HR

- 🔑 **Primary Key**: `EmpID` (a single-column key)

- **Why This is in 2NF:**
  - ***All non-key attributes (`EmpName`, `DeptID`, `DeptName`) depend on the whole primary key (`EmpID`).***
  - ***No partial dependency exists (since key is not composite).***
  - ***DeptName depends on DeptID and DeptID depends on EmpID ,However it indirectly depends on EmpID.***
- **Why This is NOT in 3NF:**
  - `DeptName` depends on `DeptID`, not directly on `EmpID`.
  - But `DeptID` depends on `EmpID` → So, there's a transitive dependency:

▪ How to Fix It (Convert to 3NF):

🔹 Split into two tables:

1. `Employees`:

EmpID	EmpName	DeptID
101	Alice	D01
102	Bob	D02
103	Carol	D01

2. `Departments`:

DeptID	DeptName
D01	HR
D02	IT

- Now:
  - All non-key columns depend **only on Primary keys**
  - No transitive dependencies ✅ → **3NF achieved**

- **What is Denormalization in DBMS:**Denormalization is the process of intentionally introducing redundancy into a database by combining tables that were separated during normalization. This is done mainly to improve performance, especially read/query speed.

⟶ **Definition:Denormalization is the process of joining normalized tables into fewer tables by adding redundant data, in order to reduce JOINs and improve query performance.**

⟶ Why Denormalize?

While normalization helps reduce redundancy and improve data integrity, it can lead to:

- Many small tables
- Complex JOIN operations
- Slower read performance

→ **Denormalization improves:**

- Query speed (fewer JOINs)
- Read-heavy performance

→ **But it sacrifices:**

- Storage efficiency
- Data consistency
- Update performance (more risk of anomalies)

Example:

Normalized Tables:

🎯 Normalized Tables:

1. Customers

CustomerID	Name
1	Alice
2	Bob

2. Orders

OrderID	CustomerID	Product
101	1	Laptop
102	2	Phone

To get the customer's name for an order, you'd need a JOIN:

```
SELECT Orders.OrderID, Customers.Name, Orders.Product

FROM Orders

JOIN Customers ON Orders.CustomerID =
Customers.CustomerID;
```

Denormalized Table:

OrderID	CustomerID	Name	Product
101	1	Alice	Laptop
102	2	Bob	Phone

- **No JOIN needed**
- **But now Customer Name is duplicated**
- **If Alice changes her name, it must be updated in every order row**

Key Trade-offs:

Benefit	Cost
Faster SELECT queries	Higher storage (redundancy)
Fewer JOINs	More chance of **inconsistency**
Simpler queries	Harder updates (data is repeated)

When to Use Denormalization:

- In read-heavy systems (data warehouses, analytics)
- When JOINs slow down performance
- When data changes rarely
- For reporting and summary tables

- ## What is a Transaction:
  - A **transaction** is a sequence of one or more database operations (like insert, update, delete) executed as a **single logical unit of work**.
  - **Example of a Transaction:**

    Imagine transferring money from Account A to Account B:

    - Step 1: Deduct $100 from Account A
    - Step 2: Add $100 to Account B

    Both steps must succeed together. If step 2 fails, step 1 should be undone to keep data consistent.

Why Use Transactions?

- To **maintain data integrity** during complex operations
- To **handle failures gracefully** (rollback on errors)
- To **enable concurrent access safely**

- **Transaction States**
  - A transaction goes through several states during its lifecycle:

State	Description
Active	Transaction is executing its operations.
Partially Committed	All operations are done, waiting to commit.
Committed	Changes are permanently saved in the database.
Failed	Transaction encountered an error and must be rolled back.
Aborted	Transaction rolled back, database restored to previous state.

  - **Concurrency Control**

When multiple transactions run **simultaneously**, concurrency control ensures **correctness and consistency** by handling:

- **Isolation:** Transactions should not interfere with each other.
- **Serializability:** The effect of concurrent transactions is the same as if they ran one after the other.

(Or)Handling **concurrency** in a DBMS is crucial to ensure **data consistency**, **correctness**, and **isolation** when **multiple transactions access the database at the same time**.

- **What Is Concurrency Control?**

Concurrency control ensures that when multiple transactions run concurrently, the result is **correct** and **the same as if they were run one after the other (serially)**.

⚠️ Problems Without Concurrency Control:

Problem	Description
Dirty Read	A transaction reads data modified by another transaction that hasn't committed.
Lost Update	Two transactions read the same data and both update it, but one update is lost.
Non-repeatable Read	A transaction reads the same row twice and gets different data because another transaction modified it in between.
Phantom Read	A transaction reads a set of rows matching a condition, but another transaction inserts new rows that now also match that condition.

Common techniques:

1. Lock-Based Protocols

- Locks prevent other transactions from accessing data being used.

Lock Type	Description
Shared Lock (S)	Allows reading; multiple can coexist.
Exclusive Lock (X)	Allows read and write; only one allowed.

-

Two-Phase Locking (2PL)

- Phase 1: Growing – acquire all needed locks.
- Phase 2: Shrinking – release locks; no new locks can be acquired.
  This ensures serializability.

Transaction T1:

```sql
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
-- Lock is acquired here!
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
COMMIT;
```

Transaction T2 (starts before T1 commits):

```sql
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
-- This will wait (or be blocked) until T1 releases the lock
UPDATE accounts SET balance = balance + 50 WHERE id = 1;
COMMIT;
```

-
- What happens:
  - T2 waits for T1 to release the exclusive lock.
  - No dirty read or lost update occurs.
- **In SQL, locks are implicitly acquired when using certain clauses like:**

  🔹 `SELECT ... FOR UPDATE`

  This tells the DBMS:

  "I want to read this row and I plan to update it, so lock it exclusively."

  🔹 `UPDATE`, `INSERT`, or `DELETE`

These automatically acquire exclusive locks on affected rows or tables

- **Timestamp-Based Protocol?**

  Each transaction is assigned a unique timestamp when it starts. This timestamp determines the order in which transactions should logically execute — older transactions go first.It ensures serializability without using locks.

Key Rules:

Each data item (e.g., variable or row) tracks:

- **`read_TS(X)`: Latest timestamp of any transaction that read `X`.**
- **`write_TS(X)`: Latest timestamp of any transaction that wrote to `X`.**

  Then, for any transaction T:

Operation	Rule to check
`read(X)`	Allowed if T's timestamp ≥ `write_TS(X)`
`write(X)`	Allowed if T's timestamp ≥ both `read_TS(X)` and `write_TS(X)`

- If a rule is violated, the transaction is **rolled back (aborted)** and restarted with a **new timestamp**.

Example:

We have a data item X (e.g., a bank balance) starting at ₹1000.

Initial timestamps:

- `read_TS(X) = 0`
- `write_TS(X) = 0`

⏱️ Transaction Start Times

Transaction	Timestamp
T1	5
T2	10
T3	15

Transaction Actions

Step	Transaction	Operation	Timestamp	read_TS(X)	write_TS(X)	Action	Result
1	T1	`read(X)`	5	0	0	5 ≥ 0 ✅	Allowed
				→ 5		Update read_TS	
2	T2	`write(X)`	10	5	0	10 ≥ 5 and 10 ≥ 0 ✅	Allowed
				5	→ 10	Update write_TS	
3	T3	`read(X)`	15	5	10	15 ≥ 10 ✅	Allowed
				→ 15	10	Update read_TS	
4	T1	`write(X)`	5	15	10	5 < 15 ❌ (read_TS) and 5 < 10 ❌ (write_TS)	**Rejected**
						Rollback T1	❌
5	T3	`write(X)`	15	15	10	15 ≥ 15 and 15 ≥ 10 ✅	Allowed
				15	→ 15	Update write_TS	✅

Final State of X:

- `read_TS(X) = 15`
- `write_TS(X) = 15`
- **Last successful write by T3**

Why Use a Timestamp-Based Protocol?

- **No need for locks**
- **No waiting or deadlocks**
- **Ensures serializability**
- **Works well when conflicts are rare and reads dominate**

What Is Optimistic Concurrency Control (OCC)?

Optimistic Concurrency Control assumes:Conflicts between transactions are rare, so don't lock anything upfront.

Instead of using locks, OCC allows transactions to execute freely and only check for conflicts at the end, during a validation phase.

OCC Phases:

Each transaction goes through three phases:

1. Read Phase

- The transaction reads data and performs computations.

- All updates are stored locally (not written to the database yet).

2. Validation Phase

- Before committing, the system checks:
  "Did any other transaction modify the data I used?"

3. Write Phase

- If validation succeeds, the updates are written to the database.
- If validation fails, the transaction is rolled back and restarted.

Example:

- **Two Transactions: T1, T2,**
- Let's say we have a table with one row:

ID	Balance
1	₹1000

Scenario

🟢 T1 starts:

- **Read Phase: Reads `balance = 1000` and stores it locally.**
- **Do some calculations (e.g., withdraw ₹100).**

🟡 T2 starts during T1:

- **Read Phase: Also reads `balance = 1000`.**
- **Deposits ₹200.**

🟡 T2 ends and enters Validation Phase:

- **No one has written anything yet, so validation passes ✅**
- **Write Phase: Balance becomes ₹1200.**

🔴 T1 now tries to commit:

- **Validation Phase: Compares `balance = 1000` (what T1 read) with current DB value (`₹1200`)**
- **Mismatch detected ❌ — someone else (T2) modified the data after T1 read it.**
- **→ T1 is aborted and restarted**

Why Use OCC?

- Best for systems with many reads and few writes (e.g., analytics dashboards).
- Avoids locking overhead.
- Prevents deadlocks.

🔄 OCC vs. 2PL vs. Timestamp

Feature	OCC	2PL (Locks)	Timestamp
Uses Locks	❌ No	✅ Yes	❌ No
Deadlock	❌ Never	✅ Possible	❌ Never
Best Use Case	Read-heavy systems	Mixed workloads	Ordered access needed
Conflict Check	At the end (validation)	During execution	At each read/write

4. Multiversion Concurrency Control (MVCC) [Same Like ReadWriteArrayList in Java concurrency]

MVCC allows multiple transactions to access the **same data at the same time** by storing **multiple versions** of a data item.

It solves the problem of **read-write conflicts** by giving:

- **Readers a consistent snapshot** of the data.
- **Writers a new version** of the data, without blocking readers.

You can **read old versions** of data without waiting for writes to finish. **(Or)**

Idea:

- **Each write creates a new version.**
- **Readers see the data as it was when their transaction started.**

🧾 Example:

Initial Table:

id	balance	version
1	1000	1

T1 starts: Reads version 1

T2 starts after T1:

- `UPDATE accounts SET balance = 1100 WHERE id = 1;`

DBMS creates a new version (version 2) with balance = 1100
T1 still sees version 1 (balance = 1000)

- **Even though T2 updated it**

This prevents non-repeatable reads and allows true read and write isolation

- **Key Properties of Transactions (ACID):**

1. **Atomicity**
   - All steps in a transaction are completed or none are.
   - If any step fails, the entire transaction is rolled back.

   Example:
   Transferring $100 from Account A to Account B.

   - Step 1: Deduct $100 from Account A
   - Step 2: Add $100 to Account B

   If Step 2 fails after Step 1 succeeded, the database must undo Step 1 (rollback) so the money isn't lost.

2. **Consistency**

   - A transaction takes the database from one **valid state** to another, maintaining all rules and constraints.

     Example:
     Suppose there's a rule: **Account balance cannot be negative**.

     Before transaction:
     Account A balance = $200

     Transaction:
     Withdraw $250

     - This violates the rule → transaction is aborted or rolled back.(here the valid state is greater than or equal to 0).
     - The database remains consistent (no negative balance).

3. **Isolation**

   - Transactions run independently without interference from others.
   - Intermediate states are not visible to other transactions.

- **Example:**
  Two transactions run simultaneously:
  i. T1 transfers $100 from Account A to B.
  ii. T2 reads the balance of Account A.

  Isolation ensures T2 sees either the balance **before** or **after** T1, but never an inconsistent intermediate state (like after deduction but before addition).

4. **Durability**

   - Once a transaction is committed, its changes are permanent, even if there is a system failure.

   Example:
   Once a transaction commits, like a money transfer, the changes persist even if:

   - The system crashes immediately after commit.
   - Power failure happens.

   The DBMS ensures the changes are saved on disk and restored when back online.

- **What is SQL Syntax?**
  - **Case Sensitivity**
    - The most important point to be noted here is that SQL is case insensitive, which means **SELECT** and **Select** have same meaning in SQL statements
    - But column names and values are case insensitive. Ex: EmpID(column Name) should be EmpID not empID or Empid or anyother.
  - All the SQL statements require a **semicolon (;)** at the end of each statement.
  - **SQL CREATE DATABASE Statement:**
    - `CREATE DATABASE databasename;`
  - **SQL USE Statement:**
    - Once the database is created, it needs to be used in order to start storing the data accordingly.
    - `Database is collection tables (keep in mind).`
    - `you can work with a database without explicitly using USE, but:`

- In SQL tools like MySQL, SQL Server, or MariaDB:
    - USE database_name; tells the system **which database to run queries on.**
    - If you **don't use USE**, then:
        - You must **specify the database name** each time like this:
            - SELECT * FROM my_database.customers;
            - My_database → It is Database.
            - Customers → It is table.

- **SQL Backup Database Statement**
    - ```
      BACKUP DATABASE database_name
      TO DISK = 'filepath'
      GO
      ```

- **Restore Database From Backup**
    - ```
      RESTORE DATABASE database_name
      FROM DISK = 'filepath';
      GO
      ```

- **SQL CREATE TABLE Statement:**
    - ```
      CREATE TABLE table_name(
         column1 datatype,
         column2 datatype,
         column3 datatype,
         .....
         columnN datatype,
         PRIMARY KEY( one or more columns )
      );
      ```

    - SQL CREATE TABLE IF NOT EXISTS:
        - ```
          CREATE TABLE IF NOT EXISTS table_name(
             column1 datatype,
             column2 datatype,
             column3 datatype,
             .....
             columnN datatype,
             PRIMARY KEY( one or more columns )
          );
          ```

- Creating a Table from an Existing Table:
  - ```
    CREATE TABLE NEW_TABLE_NAME AS
    SELECT [column1, column2...columnN]
    FROM EXISTING_TABLE_NAME
    WHERE Condition;
    ```

- Example: Here NOT NULL ensures we do not insert null.
  ```
  CREATE TABLE CUSTOMERS(
     ID   INT              NOT NULL,
     NAME VARCHAR (20)     NOT NULL,
     AGE  INT              NOT NULL,
     ADDRESS  CHAR (25) ,
     SALARY   DECIMAL (18, 2),
     PRIMARY KEY (ID)
  );
  ```

SQL INSERT INTO Statement:

- ```
  INSERT INTO table_name(
  column1,column2....columnN)
  VALUES ( value1, value2....valueN);
  ```
- Example: We can insert more rows at a time.
  ```
  INSERT INTO CUSTOMERS VALUES
  (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 ),
  (2, 'Khilan', 25, 'Delhi', 1500),
  (3, 'kaushik', 23, 'Kota', 2000),
  (4, 'Chaitali', 25, 'Mumbai', 6500),
  (5, 'Hardik', 27, 'Bhopal', 8500),
  (6, 'Komal', 22, 'Hyderabad', 4500),
  (7, 'Muffy', 24, 'Indore', 10000);
  ```

- Inserting Data into a Table Using Another
  - ```
    INSERT INTO first_table_name
    (column_name(s))
    SELECT column1, column2, ...columnN
    FROM second_table_name
    [WHERE condition];
    ```

  - ```
    CREATE TABLE BUYERS(
       ID   INT              NOT NULL,
    ```

```
    NAME VARCHAR (20)       NOT NULL,
    AGE  INT                NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

- ```
  INSERT INTO BUYERS (ID, NAME, AGE, ADDRESS, SALARY)
  SELECT * FROM CUSTOMERS;
  ```

- Insert whole Table:
- ```
  INSERT INTO first_table_name TABLE second_table_name;
  ```

- **Insert Into... Select Statement:**
  - ```
    INSERT INTO table_new
    SELECT (column1, column2, ...columnN)
    FROM table_old;
    ```

- **SQL SELECT Statement:**
  - ```
    SELECT column1, column2....columnN FROM table_name;
    ```
  - ```
    SELECT * FROM CUSTOMERS; --> For Selecting All Rows.
    ```
- **SQL UPDATE Statement:**
  - ```
    UPDATE table_name SET column1 = value1, column2 = value2....columnN=valueN WHERE  CONDITION;
    ```
  - **Example:**
    - ```
      UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;
      ```
  - SQL RENAME TABLE Statement:
    - ```
      RENAME TABLE table_name TO new_table_name;
      ```
    - Or `ALTER TABLE table_name RENAME [TO|AS] new_table_name`
    - Example:
    - ```
      ALTER TABLE BUYERS RENAME TO CUSTOMERS;
      ```
- **SQL DROP TABLE Statement:**
  - SQL **DROP TABLE Is Used To Drop Tables.**
  - ```
    DROP TABLE table_name;
    ```

- SQL **DROP DATABASE** statement is used to delete an existing database along with all the data such as tables, views, indexes, stored procedures, and constraints.
- `DROP` is used to **completely delete a table**, including all of its data and structure.
- This will **completely delete** the `CUSTOMERS` table.
- You cannot access the table or any of its data after this.
  - `DROP DATABASE IF EXISTS DatabaseName;`
  - Deleting Multiple Databases
  - `DROP DATABASE testDB3, testDB4;`

- **SQL TRUNCATE TABLE Statement:**
  - The TRUNCATE TABLE statement is implemented in SQL to delete the data of the table but not the table itself.
  - When this SQL statement is used, the table stays in the database like an empty table.
  - `TRUNCATE TABLE table_name;`

- **DELETE – Remove Rows (Data)**
  - **DELETE is used to remove specific rows (records) from a table, but keeps the table and its structure.**
  - `DELETE FROM table_name WHERE condition;`
  - **Example:**
    - `DELETE FROM CUSTOMERS WHERE CustomerID = 1;`
    - **This will remove only the row where CustomerID = 1 from the CUSTOMERS table.**
    - **The table and other rows remain.**

- **SQL ALTER TABLE Statement:**
  - `ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_type};`
  - Example:

```
ALTER TABLE CUSTOMERS ADD SEX1 char(1);

ALTER TABLE table_name DROP COLUMN column_name;
```

  - ALTER TABLE – RENAME COLUMN:
    - `ALTER TABLE table_name RENAME COLUMN old_column_name to new_column_name;`
  - `ALTER TABLE table_name RENAME TO new_table_name;`

- ## SQL DISTINCT Clause:
  - The DISTINCT clause in a database is used to identify the non-duplicate data from a column.
  - ```
    SELECT DISTINCT column1, column2....columnN FROM
    table_name;
    ```
  - DISTINCT Keyword with COUNT() Function
    - ```
      SELECT COUNT(DISTINCT column_name)

      FROM table_name WHERE condition;
      ```

- ## SQL WHERE Clause:

  🔍 Key Difference:

Clause	Filters...	Used With	Happens...
`WHERE`	Individual rows	Before grouping	First
`HAVING`	Groups (aggregated data)	After `GROUP BY`	After aggregation

  - Note:We should use where before the group by.
    ```
    SELECT Customer, SUM(Quantity) AS TotalQuantity
    FROM Orders
    WHERE Product = 'Banana'
    GROUP BY Customer
    HAVING SUM(Quantity) >= 5;
    ```
  - ```
    SELECT column1, column2....columnN
    FROM   table_name
    WHERE  CONDITION;
    ```

- ## SQL AND/OR Operators:
  - ```
    SELECT column1, column2....columnN
    FROM   table_name
    WHERE  CONDITION-1 {AND|OR} CONDITION-2;
    ```
  - Example:
    - ```
      SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE
      SALARY > 2000 AND age < 25;
      ```

- ## SQL IN/NOT IN Clause:
  - ```
    SELECT column1, column2....columnN
    FROM   table_name
    WHERE  column_name IN (val-1, val-2,...val-N);
    ```

- ```
  SELECT column1, column2....columnN
  FROM   table_name
  WHERE  column_name NOT IN (val-1,
  val-2,...val-N);
  ```

- Example:
  - ```
    SELECT * FROM CUSTOMERS
    WHERE NAME IN ('Khilan', 'Hardik', 'Muffy');
    ```

- **SQL ANY/ALL Clause:**

  ✅ Example:

  Assume this table: `products`

price
200
300
500

  ```sql
  SELECT *
  FROM products
  WHERE price > ANY (SELECT price FROM products WHERE price < 300);
  ```

  ➡️ This checks:

  - `price > 200` → returns rows with price 300 and 500.

  - ```sql
    SELECT *
    FROM products
    WHERE price > ALL (SELECT price FROM products WHERE price < 300);
    ```
    - **returns rows with price 300 and 500.**
  - **ANY Clause Check if atleast Any one true – consider if we get 100,200 in above table price >any (100,200) means our price is 110 then 110 > any(100,200) → true as 110 >100.**
  - **ALL Clause Check that all should pass. 110 > ALL (100,200) → false.**

- **SQL BETWEEN Clause:**
  - ```
    SELECT column1, column2....columnN
    FROM   table_name
    ```

```
WHERE  column_name BETWEEN val-1 AND val-2;
```

- Example:
- `SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 20 AND 25;`

- **SQL LIKE Clause :**
  - ```
        SELECT column1, column2....columnN
        FROM   table_name
        WHERE  column_name LIKE PATTERN;
    ```
  - **We can also use NOT before LIKE.**
  - **PATTERN - In this we use wildcards**
    - **% — The percent sign represents zero, one or multiple characters. Ex:- 2% May be 20 or 200 or 220 or any other.**
    - **_ — The underscore represents a single number or character. Ex:- 2_ May be 22 or 23 or 2a But not 222.**
  - **Example:**
    - `SELECT * FROM CUSTOMERS WHERE SALARY LIKE '200%';`
- **SQL ORDER BY Clause:**
  - ```
    SELECT column1, column2....columnN
    FROM   table_name
    WHERE  CONDITION
    ORDER BY column_name {ASC|DESC};
    ```
  - Example:
    - `SELECT * FROM CUSTOMERS ORDER BY NAME ASC;`
  - ORDER BY Clause on Multiple Columns:
    - `SELECT * FROM CUSTOMERS ORDER BY AGE ASC, SALARY DESC;`
  - ORDER BY with LIMIT Clause
    - ```
      SELECT column1, column2, ...
      FROM table_name
      ORDER BY column_name1 [ASC | DESC], column_name2 [ASC | DESC], ...
      LIMIT N;
      ```
      ```

- ORDER BY with WHERE Clause
    - ```
      SELECT * FROM CUSTOMERS
      WHERE AGE = 25 ORDER BY NAME DESC;
      ```
- We **cannot use ASC or DESC without ORDER BY** in SQL.

- **SQL COUNT Function:**
    - The COUNT Function gives the number of non-null values present in the specified column.
    - ```
      SELECT COUNT(column_name)
      FROM   table_name;
      ```

- **SQL GROUP BY Clause:**
    - **GROUP BY is used in SQL to group rows that have the same values in one or more columns, so that we can apply aggregate functions (like COUNT, SUM, AVG, etc.) to each group.**
    - ```
      SELECT column1, AGGREGATE_FUNCTION(column2)
      FROM table_name
      GROUP BY column1;
      ```
      Example:

      📘 Example Table: `Sales`

ID	Product	Quantity
1	Apple	10
2	Banana	5
3	Apple	7
4	Banana	3
5	Orange	8
    - ```
      SELECT Product, SUM(Quantity)
      FROM Sales
      GROUP BY Product;
      ```

🔍 Output:

Product	SUM(Quantity)
Apple	17
Banana	8
Orange	8

- **Example: For Grouping Two columns.**

ID	Product	Region	Quantity
1	Apple	East	10
2	Banana	West	5
3	Apple	East	7
4	Banana	East	3
5	Apple	West	8
6	Banana	West	6

```
SELECT Product, Region, SUM(Quantity) AS TotalQty
FROM Sales
GROUP BY Product, Region;
```

🔍 Output:

Product	Region	TotalQty
Apple	East	17
Apple	West	8
Banana	East	3
Banana	West	11

- **GROUP BY with ORDER BY Clause**

```
SELECT column1, column2, ...,
aggregate_function(columnX) AS alias
FROM table
GROUP BY column1, column2, ...
```

```
ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;
```

- **Example:**
  - ```
    SELECT AGE, MIN(SALARY) AS MIN_SALARY
    FROM CUSTOMERS
    GROUP BY AGE ORDER BY MIN_SALARY DESC;
    ```

- SQL HAVING Clause
  - What is HAVING?
  - The HAVING clause is used to filter groups created by the GROUP BY clause.
  - It allows you to apply conditions on aggregate functions like SUM(), COUNT(), AVG(),MIN(),MAX() etc.
  - Think of HAVING as a WHERE clause for groups.

```
SELECT column1, column2, ..
aggregate_function(column_Name)
FROM table_name
GROUP BY column1, column2 ..
HAVING condition;
```

Important Notes:

- HAVING is applied after aggregation.
- If you want to filter rows before aggregation, use WHERE.
- You can use HAVING without GROUP BY — it will treat the entire result as a single group
- Note:We should use having after the group by.
  SELECT Customer, SUM(Quantity) AS TotalQuantity
  FROM Orders
  WHERE Product = 'Banana'
  GROUP BY Customer
  HAVING SUM(Quantity) >= 5;
- **The following code block shows the position of the HAVING Clause - when we write a query first select should come after that from and where and so on.**
  1. `SELECT`
  2. `FROM`
  3. `Joins`
```

4. `WHERE`
5. `GROUP BY`
6. `HAVING`
7. `ORDER BY`

- **SQL Constraints:**
    - SQL Constraints are the rules applied to a data columns or the complete table to limit the type of data that can go into a table.
    - **SQL Create Constraints**
        - ```
          CREATE TABLE table_name (

              column1 datatype constraint,

              column2 datatype constraint,

              ....

              columnN datatype constraint

          );
          ```
        - **Example**
            1. ```
               CREATE TABLE CUSTOMERS (

                   ID INT NOT NULL UNIQUE,

                   NAME VARCHAR (20) NOT NULL DEFAULT
               'Not Available',

                   AGE INT NOT NULL CHECK(AGE>=18),

                   ADDRESS CHAR (25),

                   SALARY DECIMAL (18, 2),

               PRIMARY KEY (ID),

               CUSTOMER INT FOREIGN KEY REFERENCES
               CUSTOMER_ID(ID)

               );
               ```
            2. **Here In** Above NOT NULL,UNIQUE,DEFAULT,CHECK,PRIMARY KEY,FOREIGN KEY ARE **Constraints.**

- **What is an SQL View?**

- An SQL View is a virtual table based on the result of a `SELECT` query. It does not store data physically; instead, it displays data dynamically from one or more real tables.
  - ```
    CREATE VIEW view_name AS

    SELECT column1, column2....

    FROM table_name

    WHERE [condition];
    ```
  - **Update View Table:**
    - ```
      UPDATE view_name

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

      WHERE [condition];
      ```
  - **Drop view table:[] → optional**
  - `DROP VIEW [IF EXISTS] view_name;`
  - **DELETE,RENAME → Same like a normal table.**
- **SQL TOP Clause:**
  - ```
    SELECT TOP value column_name(s)

    FROM table_name

    WHERE [condition];
    ```
  - **Example:Return top 4 and 40 percent rows.**
    - `SELECT TOP 4 * FROM CUSTOMERS;`
    - `SELECT TOP 40 PERCENT * FROM CUSTOMERS ORDER BY SALARY;`
    - `DELETE TOP(2) FROM CUSTOMERS WHERE NAME LIKE 'K%';`
- **`SQL EXISTS` Clause:**
    ```

`Customers`:

customer_id	name
1	Alice
2	Bob
3	Charlie

`Orders`:

order_id	customer_id	product
101	1	Phone
102	2	Laptop

```
SELECT name
FROM Customers c
WHERE EXISTS (
    SELECT 1
    FROM Orders o
    WHERE o.customer_id = c.customer_id
);--here we can select any thing (we choice 1)
```

- Here we select a row if the inner query returns true.we move from first to last row for every row we check the inner query.

✅ Final Output:

name
Alice
Bob

- **SQL CASE Clause:**

```
CASE
   WHEN condition1 THEN statement1,
   WHEN condition2 THEN statement2,
   WHEN condition THEN statementN
   ELSE result -- optional by default id return null
```

```
END;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

- ```
  SELECT NAME, AGE,
  CASE
  WHEN AGE > 30 THEN 'Gen X'
  WHEN AGE > 25 THEN 'Gen Y'
  WHEN AGE > 22 THEN 'Gen Z'
  ELSE 'Gen Alpha'
  END AS Generation
  FROM CUSTOMERS;
  Output:-
  ```

NAME	AGE	Generation
Ramesh	32	Gen X
Khilan	25	Gen Z
Kaushik	23	Gen Z
Chaitali	25	Gen Z
Hardik	27	Gen Y
Komal	22	Gen Alpha
Muffy	24	Gen Z

- **SQL UNION/UNION ALL Operator**
  - To use the UNION operator on multiple tables, all these tables must be union compatible.
  - Both used to retrieve the rows from multiple tables and return them as one single table.
  ```

- The difference between these two operators is that UNION only returns distinct rows while UNION ALL returns all the rows present in the tables.
  - The same number of columns selected with the same datatype.
  - These columns must also be in the same order.
  - They need not have the same number of rows.
- *The column names in the final result set will be based on the column names selected in the first SELECT statement. If you want to use a different name for a column in the final result set, you can use an alias in the SELECT statement.*
- ```
  SELECT column1 [, column2 ]

  FROM table1 [, table2 ]

  [WHERE condition]

  UNION

  SELECT column1 [, column2 ]

  FROM table1 [, table2 ]

  [WHERE condition];
  ```

- Union All:

  ```
  SELECT * FROM table1

  UNION ALL

  SELECT * FROM table2;
  ```

- Union with order by:

  ```
  SELECT ID, SALARY FROM CUSTOMERS WHERE ID > 5

  UNION

  SELECT CUSTOMER_ID, AMOUNT FROM ORDERS WHERE
  CUSTOMER_ID > 2

  ORDER BY SALARY;
  ```

- Example:
  - `SELECT SALARY FROM CUSTOMERS UNION SELECT AMOUNT FROM ORDERS;`

- ***SQL INTERSECT Operator:***
  ```

- The **INTERSECT** operator in SQL is used to retrieve the records that are identical/common between the result sets of two or more tables.

Name	Age
Sara	26
Dev	22
Jay	29
Aarohi	30

INTERSECT

Name	Salary
Dev	3000
Rahul	2000
Aarohi	5000
Rohan	4000

=

Name
Dev
Aarohi

```
SELECT column1, column2,..., columnN
FROM table1
INTERSECT
SELECT column1, column2,..., columnN
FROM table2
```

- *Example:*

```
SELECT NAME, AGE, HOBBY FROM STUDENTS_HOBBY

INTERSECT

SELECT NAME, AGE, HOBBY FROM STUDENTS;
```

- **The SQL EXCEPT Operator**
  - *The **EXCEPT** operator in SQL is used to retrieve all the unique records from the left operand (query), except the records that are present in the result set of the right operand (query).*

NAME	AGE
Sara	26
Dev	22
Jay	29
Sara	26

EXCEPT

NAME	SALARY
DEV	3000
Kalyan	2000
Aarohi	5000

=

NAME
Sara
Jay

```
SELECT column1, column2,..., columnN
```

```
FROM table1, table2,..., tableN
[Conditions] //optional
EXCEPT
SELECT column1, column2,..., columnN
FROM table1, table2,..., tableN
[Conditions] //optional
```

Example:

```
SELECT NAME, HOBBY, AGE FROM STUDENTS
EXCEPT
SELECT NAME, HOBBY, AGE FROM STUDENTS_HOBBY;
```

- **What are SQL Data types?**
  - An SQL data type refers to the type of data which can be stored in a column of a database table.
  - For example, if you want to store student name in a column then you should give column name something like student_name and it's data type will be char(50) which means it can store a string of characters up to 50 characters.
  - The data type provide guidelines for SQL to understand what type of data is expected inside each column, and hence, prevents the user from entering any unexpected or invalid data in a column.
  - Types of SQL Data Types:
    - Character/String Data Types:
      1. CHAR(n) → If We store a data whose size is less then n.then remove is padded. 0<=n<=255
         - Example:
         - CREATE TABLE ExampleChar (Code CHAR(5));
         - INSERT INTO ExampleChar VALUES ('USA');
         - Stored as `'USA  '` (padded with 2 spaces to make length 5).
         - Length will always be 5, even if data is shorter.
      2. VARCHAR(n) → Here no padding is there.0<=n<=65535
         - CREATE TABLE ExampleVarchar (Name VARCHAR(5));
         - INSERT INTO ExampleVarchar VALUES ('USA');
         - Stored as `'USA'` (only 3 bytes, no padding).

- Length depends on actual string length.

- Numeric Data Types:
  1. INT → A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295.
     - CREATE TABLE Students (ID INT,Age INT);
     - → Example: ID = 1, Age = 20
  2. DECIMAL(p, s) → Fixed-precision numbers.
     - CREATE TABLE Products (Price DECIMAL(6,2));
     - →Example: Price = 1234.56 (6 digits total, 2 after decimal)
  3. FLOAT → Approximate decimal values.
     - CREATE TABLE Physics (Gravity FLOAT);
     - → Example: Gravity = 9.81
  4. TEXT → Used for long text.
     - CREATE TABLE Articles (Content TEXT);
     - →Example: 'This is a full article body with many characters...'
- Date and Time Types:
  1. **DATE** → Stores calendar date (YYYY-MM-DD)
     - `CREATE TABLE Employees (JoinDate DATE);`
     - → Example: `'2023-01-15'`
  2. **TIME** → Stores time of day (HH:MM:SS)
     - `CREATE TABLE Schedules (StartTime TIME);`
     - → Example: `'08:30:00'`
  3. **DATETIME or TIMESTAMP** → Stores both date and time.
  4. `CREATE TABLE Events (EventTime DATETIME);`
  5. → Example: `'2025-05-23 14:00:00'`
- **Boolean Type:**
  1. **BOOLEAN or BOOL**
     - Stores TRUE or FALSE values.
     - `CREATE TABLE Members (IsActive BOOLEAN);`
     - `Example: IsActive = TRUE`
- **What is ENUM:**ENUM is a **string data type** that allows you to **define a list of allowed values** for a column.
  1. It ensures that the column can **only store one of those values** — like a built-in dropdown list!
  2. `CREATE TABLE Shirts (Size ENUM('S', 'M', 'L', 'XL'));`

        - The `Size` column can only contain one of these: `'S'`, `'M'`, `'L'`, or `'XL'`.

- **What is SQL Operator?**
    - Types of Operator in SQL:
        - SQL Arithmetic Operators:

1.

+	Addition	10 + 20 = 30
-	Subtraction	20 - 30 = -10
*	Multiplication	10 * 20 = 200
/	Division	20 / 10 = 2
%	Modulus	5 % 2 = 1

        - SQL Comparison Operators:

1.

=	Equal to	5 = 5 returns TRUE
!=	Not equal	5 != 6 returns TRUE
<>	Not equal	5 <> 4 returns TRUE
>	Greater than	4 > 5 returns FALSE
<	Less than	4 < 5 returns TRUE
>=	Greater than or equal to	4 >= 5 returns FALSE
<=	Less than or equal to	4 <= 5 returns TRUE
!<	Not less than	4 !< 5 returns FALSE
!>	Not greater than	4 !> 5 returns TRUE

        - SQL Logical Operators:

1.

ALL	TRUE if all of a set of comparisons are TRUE.
AND	TRUE if all the conditions separated by AND are TRUE.
ANY	TRUE if any one of a set of comparisons are TRUE.
BETWEEN	TRUE if the operand lies within the range of comparisons.
EXISTS	TRUE if the subquery returns one or more records
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern specially with wildcard.
NOT	Reverses the value of any other Boolean operator.
OR	TRUE if any of the conditions separated by OR is TRUE
IS NULL	TRUE if the expression value is NULL.
SOME	TRUE if some of a set of comparisons are TRUE.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

- **SQL - Comments**
    - There are two types of comments used.
        - Single-line comments
        -
```
-- Will fetch all the table records

SELECT * from table;
```

- Multi-line comments
  1. ```
     /* This is a
        multi-line
        comment */
     ```

- DataBase Vs Tables

```
📁 Database
├── 📄 Table1
|     ├── Row1
|     ├── Row2
|     └── ...
├── 📄 Table2
|     ├── Row1
|     └── ...
└── ...
```

- ### What is an Index?
  - An **index** is a **database structure** that improves the **speed of data retrieval** on a table.
    - Without an index: Full table scan
    - With an index: Jump directly to needed rows
  - **How Does an Index Work (Internally)?**
    - Most SQL databases use B-Trees (or B+ Trees) (B → Balanced) for indexes.
    - **Benefits:**
      - **Logarithmic time** complexity: O(log n)
      - **Efficient for range queries**
      - Great for large datasets (even millions of rows)
    - CREATE INDEX idx_name ON Employees(Name);
    - Now this query is very fast: (Here Employees → Table_Name and Name Column_Name)
    - SELECT * FROM Employees WHERE Name = 'Alice';
    - `DROP INDEX index_name ON table_name;`
    - We can declare Columns as indexes → Composite Index.
    - **Types:**
```

🧱 4. Types of Indexes

Type	Description
Primary Index	Created automatically on primary key
Unique Index	Ensures all values are unique
Composite Index	Index on multiple columns
Clustered Index	Sorts the actual table rows (1 per table in SQL Server)
Non-clustered Index	A separate structure with pointers to rows
Full-text Index	For searching text (like documents)
Bitmap Index	Efficient for low-cardinality columns (used in analytics)
Spatial Index	For geographic data (used in GIS systems)

- **When Should You Use Indexes?**
  1. ✅ **Use Indexes:**
     - **On columns used in `WHERE`, `JOIN`, `ORDER BY`, `GROUP BY`**
     - **On foreign key columns**
     - **On frequently searched or sorted fields**
  - ❌ **Avoid Indexes:**
     - **On columns with many duplicate values (e.g. gender)**
     - **On columns that are rarely queried**
     - **On very small tables — scanning is fast anyway**
     - **Too many indexes can slow INSERT/UPDATE/DELETE because they must also update the index**

- **SQL Join Clause:**
  - A **JOIN** in SQL is used to **combine rows** from two or more tables, based on a **related column** between them.
  - (OR) SQL **Join** clause is used to combine data from two or more tables in a database. When the related data is stored across multiple tables, joins help you to retrieve records combining the fields from these tables using their foreign keys.
  - *When you retrieve a table using joins, the resultant table displayed is not stored anywhere in the database.*
  - JOINS —-- (I) INNER JOIN (II)OUTER JOIN –(a)LEFT JOIN (b) RIGHT JOIN (c) FULL JOIN (III) OTHER – (a) SELF JOIN (b) CROSS JOIN.
  - Let Consider An Example For Better Understand:
    - Table 1:

- 🔵 Table: `Customers`

customer_id	name
1	Alice
2	Bob
3	Carol
4	David

- Table 2:

  🟢 Table: `Orders`

order_id	customer_id	product
101	1	Phone
102	2	Laptop
103	2	Keyboard
104	5	Tablet

- **1️⃣ INNER JOIN**
  - **Code** :
    - SELECT *
      FROM Customers
      INNER JOIN Orders
      ON Customers.customer_id = Orders.customer_id;

    🟢 Result:

customer_id	name	order_id	customer_id (Orders)	product
1	Alice	101	1	Phone
2	Bob	102	2	Laptop
2	Bob	103	2	Keyboard

- **2️⃣ LEFT JOIN**
  - **Code:**

```
SELECT *
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id =
Orders.customer_id;
```

🟢 Result:

customer_id	name	order_id	customer_id (Orders)	product
1	Alice	101	1	Phone
2	Bob	102	2	Laptop
2	Bob	103	2	Keyboard
3	Carol	NULL	NULL	NULL
4	David	NULL	NULL	NULL

- ## 3️⃣ RIGHT JOIN
  - **Code**

```
SELECT *
FROM Customers
RIGHT JOIN Orders
ON Customers.customer_id =
Orders.customer_id;
```

🟢 Result:

customer_id (Customers)	name	order_id	customer_id	product
1	Alice	101	1	Phone
2	Bob	102	2	Laptop
2	Bob	103	2	Keyboard
NULL	NULL	104	5	Tablet

- ## 4️⃣ FULL OUTER JOIN (Simulated with `UNION`)
  - **Code**

```
SELECT *
FROM Customers
LEFT JOIN Orders ON
Customers.customer_id =
Orders.customer_id
```

```
UNION
SELECT *
FROM Customers
RIGHT JOIN Orders ON
Customers.customer_id =
Orders.customer_id;
```

🟢 Result:

customer_id	name	order_id	customer_id (Orders)	product
1	Alice	101	1	Phone
2	Bob	102	2	Laptop
2	Bob	103	2	Keyboard
3	Carol	NULL	NULL	NULL
4	David	NULL	NULL	NULL
NULL	NULL	104	5	Tablet

- ## 5️⃣ CROSS JOIN
  - **Code**
    - ```
      SELECT *
      FROM Customers
      CROSS JOIN Orders;
      ```

🟢 **Result: (only first 5 rows shown here)**

customer_id	name	order_id	customer_id (Orders)	product
1	Alice	101	1	Phone
1	Alice	102	2	Laptop
1	Alice	103	2	Keyboard
1	Alice	104	5	Tablet
2	Bob	101	1	Phone
...	...	...	...	...

> Total rows = 4 customers × 4 orders = **16 rows**

- **SQL DELETE… JOIN Clause:**
  - The purpose of Joins in SQL is to combine records of two or more tables based on common columns/fields. Once the tables are joined,
      ```

performing the deletion operation on the obtained result-set will delete records from all the original tables at a time.

- For example, consider a database of an educational institution. It consists of various tables: Departments, StudentDetails, LibraryPasses, LaboratoryPasses etc. When a set of students are graduated, all their details from the organizational tables need to be removed, as they are unwanted. However, removing the details separately from multiple tables can be cumbersome.
- Syntax:
  - ```
    DELETE table(s)
    FROM table1 JOIN table2
    ON table1.common_field =
    table2.common_field;
    ```
- Example:
  - ```
    DELETE a
    FROM CUSTOMERS AS a INNER JOIN ORDERS AS b
    ON a.ID = b.CUSTOMER_ID;
    ```