SQL

What is SQL?

- **SQL (Structured Query Language)** is a programming language used to **store, manage, and retrieve data** from *relational databases*.
- Databases using SQL include: **MySQL, MS Access, SQL Server, Oracle, Postgres, Sybase, Informix**, etc.
- SQL was developed by IBM in the **1970s**.
- SQL is **not** a database; it is only a **language used to communicate with databases**.

Why SQL?

- Used in almost every software domain: **banking, finance, education, security** etc.
- Easy to learn and essential for software developers.
- All relational databases follow SQL or a slight variation:
    - **MS SQL Server → T-SQL**
    - **Oracle → PL/SQL**
    - **MS Access → JET SQL**

SQL Commands

SQL commands are divided into these categories:

---

1. DDL (Data Definition Language)

Used to **define or modify database structure**.

Command	Purpose
CREATE	Creates tables, views, indexes, etc.
ALTER	Modifies an existing database object
DROP	Deletes a table or object entirely
TRUNCATE	Removes all rows from a table at once

---

2. DML (Data Manipulation Language)

Used to **manage the data** inside tables.

Command	Purpose
SELECT	Retrieves data
INSERT	Adds new records
UPDATE	Modifies existing records
DELETE	Removes records

---

3. DCL (Data Control Language)

Used to **control access** to the database.

Command	Purpose
GRANT	Gives permissions to a user
REVOKE	Removes permissions from a user

SQL Applications

SQL helps you to:

- Run queries on a database
- Define and manipulate stored data
- Create or delete databases and tables
- Add/manage database users
- Create views, procedures, and functions
- Set permissions on database objects

What is RDBMS?

- **RDBMS = Relational Database Management System**
- It is the base for SQL and all modern databases like **MySQL, Oracle, SQL Server, DB2, MS Access**, etc.
- RDBMS uses the **relational model** created by **E.F. Codd (1970)**.

Stores data in **tables** made of **rows and columns**.

Database Normalization (Organizing data properly)

Why normalize?

- Removes **duplicate data**
- Ensures **logical storage**
- Reduces **storage space** and improves **data consistency**

Normal Forms

1. **1NF (First Normal Form)**
   - No repeating groups
   - Each field contains **atomic (indivisible)** values
2. **2NF (Second Normal Form)**
   - Must be in **1NF**
   - No partial dependency (applies to composite keys)
3. **3NF (Third Normal Form)**
   - Must be in **2NF**
   - No transitive dependency (non-key fields should not depend on other non-key fields)

Usually databases up to 3NF are sufficient for most applications.

Common SQL Statements

SQL CREATE DATABASE

Creates a new database.

```
CREATE DATABASE sampleDB;
```

SQL USE (Select Database)

Selects a database to work with.

```
USE sampleDB;
```

SQL DROP DATABASE

Deletes the entire database.

```
DROP DATABASE sampleDB;
```

SQL CREATE TABLE

Creates a new table.

```
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

Inserts new records into the table.

```
INSERT INTO CUSTOMERS VALUES
(1, 'Ramesh', 32, 'Ahmedabad', 2000);
```

SQL SELECT

Retrieves data from a table.

```
SELECT * FROM CUSTOMERS;
```

SQL UPDATE

Updates existing records.

```
 UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;
```

SQL DELETE

Deletes a specific record.

```
DELETE FROM CUSTOMERS WHERE ID = 6;
```

SQL DROP TABLE

Deletes the table completely.

```
DROP TABLE CUSTOMERS;
```

TRUNCATE TABLE

Deletes all rows but keeps the table structure.

```
TRUNCATE TABLE CUSTOMERS;
```

SQL ALTER TABLE

Used to add, delete, or modify columns.

➤ Add Column

```
ALTER TABLE CUSTOMERS ADD SEX CHAR(1);
```

➤ Rename Table

```
ALTER TABLE CUSTOMERS RENAME TO NEW_CUSTOMERS;
```

SQL DISTINCT

Removes duplicate values.

```
SELECT DISTINCT SALARY FROM CUSTOMERS;
```

SQL WHERE Clause

Filters rows using conditions.

```
SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000;
```

SQL AND / OR

Apply multiple conditions.

```
SELECT * FROM CUSTOMERS
WHERE SALARY > 2000 AND AGE < 25;
```

SQL IN

Checks if a value matches any value in a list.

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

SQL BETWEEN

Filters values in a range.

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

SQL LIKE

Pattern matching using wildcards.

```
SELECT * FROM CUSTOMERS WHERE SALARY LIKE '200%';
```

SQL ORDER BY

Sorts result set (ASC or DESC).

```
SELECT * FROM CUSTOMERS ORDER BY NAME ASC;
```

SQL GROUP BY

Groups rows and performs calculations.

```
SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS, AGE;
```

SQL COUNT

Counts number of non-NULL values.

```
SELECT AGE, COUNT(Name)
FROM CUSTOMERS
GROUP BY AGE;
```

SQL HAVING

Filters groups created by GROUP BY.

```
SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS, AGE
HAVING TOTAL_SALARY >= 5000
ORDER BY TOTAL_SALARY DESC;
```

SQL - Data Types

What are SQL Data Types?

- SQL data types define **what kind of data** can be stored in a table column.
  Example: numbers, text, dates, binary data, etc.
- While creating a table, each column needs:
  1. **Column Name**
  2. **Data Type**

Example:

```
CREATE TABLE Customers (
  Name VARCHAR(25),
  Age INT
);
```

Here:

- `Name` can store up to 25 characters
- `Age` stores integer values

Data types help:

- Store correct data
- Prevent invalid entries
- Use memory efficiently

Main Categories of SQL Data Types

1. **String (text)**
2. **Numeric (numbers)**
3. **Date & Time**

Different databases (MySQL, Oracle, SQL Server, MS Access) support slightly different data types.

MS SQL Server Data Types

String Data Types

Type	Description
char(n)	Fixed length (max 8000)
varchar(n)	Variable length (max 8000)
varchar(max)	Very long text (up to 1GB)
Text	Up to 2GB text
Nchar	Fixed-length Unicode
Nvarchar	Variable-length Unicode
Ntext	Unicode large text
Binary	Fixed binary
Varbinary	Variable binary
varbinary(max)	Large binary
Image	Up to 2GB binary

Numeric Data Types

Type	Description
bit	0, 1, or NULL
tinyint	0–255
smallint	approx −32,000 to +32,000
int	large integers
bigint	very large integers
decimal(p,s)	Exact numeric
numeric(p,s)	Same as decimal
smallmoney	Small money values
money	Large money values

Type	Description
float(n)	Floating-point
real	Approx floating-point

Date & Time

Type	Description
Datetime	Date + time (3.33 ms accuracy)
datetime2	Higher precision date/time
Smalldatetime	1 minute accuracy
Date	Only date
Time	Only time
datetimeoffset	datettime2 + timezone
Timestamp	Auto-generated unique number (not actual time)

SQL - Operators

What is an SQL Operator?

An **SQL operator** is a symbol or keyword used in SQL statements—mainly in the **WHERE** clause—to perform operations such as:

- Mathematical calculations
- Comparing values
- Joining multiple conditions
- Checking patterns
- Checking NULL values

Operators help SQL decide **which rows to select**, **update**, or **delete**.

Types:

- **Unary operator** → works on **one** operand (example: unary +, unary -)
- **Binary operator** → works on **two** operands (example: `5 + 3`, `age > 20`)

1. SQL Arithmetic Operators

Used to do mathematical operations on numerical data.

Operator	Operation	Example	Result
+	Addition	10 + 20	30
-	Subtraction	20 - 30	-10
*	Multiplication	10 * 20	200
/	Division	20 / 10	2
%	Modulus (remainder)	5 % 2	1

SELECT Name,Salary + 5000 AS IncreasedSalary FROM Employees;

SELECT Name,Salary - 2000 AS ReducedSalary FROM Employees;

SELECT Name,Salary * 2 AS DoubleSalary FROM Employees;

SELECT Name,Salary / 1000 AS SalaryInThousands FROM Employees;

SELECT Name,Salary % 10000 AS SalaryRemainder FROM Employees

2. SQL Comparison Operators

Used to compare two values.
They return **TRUE**, **FALSE**, or **UNKNOWN** (if NULL is involved).

Operator	Meaning	Example	Result
=	Equal	5 = 5	TRUE
!=	Not equal	5 != 6	TRUE
<>	Not equal	5 <> 4	TRUE
>	Greater than	4 > 5	FALSE
<	Less than	4 < 5	TRUE

Operator	Meaning	Example	Result
>=	Greater or equal	4 >= 5	FALSE
<=	Less or equal	4 <= 5	TRUE
!<	Not less than	4 !< 5	FALSE
!>	Not greater than	4 !> 5	TRUE

SELECT Name, CASE WHEN Age >= 30 THEN 'Senior' ELSE 'Junior' END AS AgeGroup FROM Employees;

3. SQL Logical Operators

Used to combine conditions or check advanced logical conditions.
Return TRUE, FALSE, or UNKNOWN.

Operator	Description
ALL	TRUE if all comparisons in a set are TRUE
AND	TRUE only if **all** conditions are TRUE
ANY	TRUE if **any one** of the comparisons is TRUE
BETWEEN	TRUE if value is within a range
EXISTS	TRUE if a subquery returns any rows
IN	TRUE if value matches any value in a list
LIKE	TRUE if pattern matches (supports wildcards % and _)
NOT	Reverses (negates) a condition
OR	TRUE if **any** condition is TRUE
IS NULL	TRUE if value is NULL
SOME	Same as ANY
UNIQUE	TRUE if rows returned by subquery contain no duplicates

SELECT Name,Age,DepartmentID,CASE WHEN Age BETWEEN 30 AND 40 AND DepartmentID IN (1, 2) THEN 'Experienced in Core Depts'ELSE 'Other'END AS DepartmentStatus FROM Employees;

4. SQL Operator Precedence

Operator precedence means:
Which operator is executed first when multiple are used in a single expression.

▲ Highest → Lowest ▼

Precedence Order	Operators
1	+, – (unary)
2	*, /
3	+, –
4	Comparison operators (=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN)
5	NOT
6	AND
7	OR

AND vs OR (Most common mistake)

SELECT *FROM StudentsWHERE Grade = 10OR Grade = 11 AND City = 'London';

AND has higher precedence **than OR, so this is treated as:**

- WHERE Grade = 10 OR (Grade = 11 AND City = 'London');
- WHERE (Grade = 10 OR Grade = 11) AND City = 'London';

NOT vs AND

- SELECT *FROM UsersWHERE NOT IsActive = 1 AND IsAdmin = 1;
- WHERE (NOT IsActive = 1) AND IsAdmin = 1;
- NOT (IsActive = 1 AND IsAdmin = 1)
- WHERE NOT (IsActive = 1 AND IsAdmin = 1);

Arithmetic Precedence

- SELECT 10 + 5 * 2 AS Result;
- SELECT (10 + 5) * 2 AS Result;

Comparison + AND + OR Example

- SELECT *FROM Orders WHERE TotalAmount > 100 AND Status = 'Shipped' OR Priority 'High';
- WHERE (TotalAmount > 100 AND Status = 'Shipped') OR Priority = 'High';
- WHERE TotalAmount > 100 AND (Status = 'Shipped' OR Priority = 'High');

SQL - Expressions

What is an SQL Expression?

An **SQL Expression** is a combination of:

- **Values**
- **Operators**
- **SQL functions**

that **evaluates to a single value**.

You can think of expressions as **formulas** used inside SQL queries.

Expressions are mostly used in the **WHERE** clause to apply conditions and filter rows.

Syntax

```
SELECT column1, column2
FROM table_name
WHERE expression;
```

Types of SQL Expressions

SQL expressions are mainly divided into:

1. **Boolean Expressions**
2. **Numeric Expressions**
3. **Date & Time Expressions**

Let’s look at each.

---

1. SQL Boolean Expressions

What they do:

They compare values and return:

- **TRUE**
- **FALSE**
- **UNKNOWN** (if NULL is involved)

Boolean expressions use:

- Comparison operators (=, >, <, etc.)
- Logical operators (AND, OR, NOT)

Syntax:

```
SELECT columns
FROM table
WHERE boolean_expression;
```

Example Table: CUSTOMERS

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
7	Muffy	24	Indore	10000

Example Query:

```
SELECT * FROM CUSTOMERS WHERE SALARY = 10000;
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
7	Muffy	24	Indore	10000

---

2. SQL Numeric Expressions

What they do:

Numeric expressions use:

- Two numeric values
- Arithmetic operators (+, -, *, /, %)

They are used to calculate values.

Syntax:

```
SELECT numeric_expression AS alias
FROM table_name
```

```
WHERE numeric_expression;
```

Simple Example:

```
SELECT 15 + 6;
```

Output:

```
21
```

Using numeric functions:

```
SELECT COUNT(*) FROM CUSTOMERS;
```

Output:

```
7
```

---

3. SQL Date Expressions

What they do:

Used to compare dates or retrieve system date/time.

Useful for filtering data based on year, month, day, etc.

Syntax:

```
SELECT columns
FROM table
WHERE date_expression;
```

Example 1: Get current timestamp

```
SELECT CURRENT_TIMESTAMP;
```

Output Example:

```
2009-11-12 06:40:23
```

-- Current date and time

- SELECT GETDATE() AS CurrentDateTime;

-- Add days to a date

- SELECT DATEADD(DAY, 7, GETDATE()) AS NextWeek;

-- Difference between dates

- SELECT DATEDIFF(YEAR, '2000-01-01', GETDATE()) AS AgeInYears;

-- Filtering using date expression

- SELECT *FROM Orders WHERE OrderDate >= DATEADD(MONTH, -1, GETDATE());

SQL - Comments

What is a Comment?

A **comment** is a piece of text written inside code to:

- Explain the code
- Add notes for humans
- Temporarily disable execution of a statement

Comments are **ignored by the compiler/interpreter**, so:

- They do **not affect** the output
- They do **not execute** as code

SQL Comments

In SQL, comments are used to:

- Explain a part of the query
- Make SQL code easier to understand
- Skip (disable) the execution of a statement

MySQL supports **two types of comments**:

1. **Single-line comments**
2. **Multi-line comments**

---

1. Single-Line Comments

- Begin with `--` (two hyphens)
- Everything after `--` on the same line is ignored

Syntax

```
-- This is a single-line comment
```

Example 1: Comment describing the query

```
-- Will fetch all the table records
SELECT * FROM CUSTOMERS;
```

Example 2: Commenting part of a query

```
SELECT * FROM CUSTOMERS -- ORDER BY NAME ASC;
```

Here, **ORDER BY** is ignored.

Example 3: Ignoring a full statement

```
SELECT * FROM CUSTOMERS;
```

```
SELECT * FROM EMPLOYEES;
-- SELECT * FROM ORDERS WHERE ID = 6;
```

---

2. Multi-Line Comments

- Used for **multiple lines** or a **block of code**
- Starts with **`/*`** and ends with **`*/`**
- Everything between these symbols is ignored

Syntax

```
/* This is a
   multi-line
   comment */
```

Example 1: Query explanation using block comment

```
/* following query
   will fetch all
   table records */
SELECT * FROM CUSTOMERS;
```

Example 2: Commenting part of a query

```
SELECT ID, /* AGE, SALARY */
FROM CUSTOMERS
WHERE SALARY = 1500.00;
```

SQL Database

SQL Server – Database Commands

What is a Database?

A **database** is an organized collection of data stored in a computer system.
SQL Server uses **SQL (Structured Query Language)** to create, manage, and maintain databases.

CREATE DATABASE (DDL)

Used to create a new database.

Syntax

```
CREATE DATABASE DatabaseName;
```

Example

```
CREATE DATABASE testDB;
```

View All Databases (SQL Server)

SQL Server does **not** support `SHOW DATABASES`.

Method 1 – System View (Recommended)

```
SELECT name FROM sys.databases;
```

Method 2 – Stored Procedure

```
EXEC sp_databases;
```

USE Database

Before creating tables or inserting data, you must select the database.

Syntax

```
USE DatabaseName;
```

Example

```
USE testDB;
```

DROP DATABASE

Deletes the database **permanently** along with:

- Tables
- Views
- Stored procedures
- Data

⚠ This action cannot be undone

Syntax

```
DROP DATABASE DatabaseName;
```

Example

```
DROP DATABASE testDB;
```

Safe Drop (Recommended)

```
DROP DATABASE IF EXISTS testDB;
```

Rename Database (SQL Server Only)

SQL Server supports renaming databases.

Syntax

```
ALTER DATABASE OldName MODIFY NAME = NewName;
```

Example

```
ALTER DATABASE testDB MODIFY NAME = tutorialsDB;
```

Backup Database

Full Backup

```
BACKUP DATABASE testDB
TO DISK = 'D:\testDB.bak';
GO
```

Restore Database

Restore Backup

```
RESTORE DATABASE testDB
FROM DISK = 'D:\testDB.bak'
WITH REPLACE;
GO
```

SQL Server – CREATE TABLE & TABLE OPERATIONS

What is a Table?

- A **table** stores data in **rows (records)** and **columns (fields)**.
- Each column has a **name** and a **datatype**.
- Each table must have a **unique name** within the database.

CREATE TABLE Statement (DDL)

Used to create a new table in the selected database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype [constraint],
    column2 datatype [constraint],
    ...
);
```

Example – Creating a Table

```
CREATE TABLE Customers (
    ID INT NOT NULL,
    Name VARCHAR(20) NOT NULL,
```

```
    Age INT NOT NULL,
    Address VARCHAR(25),
    Salary DECIMAL(18,2),
    CONSTRAINT PK_Customers PRIMARY KEY (ID)
);
```

Key Points

- `NOT NULL` → column cannot have empty values
- `PRIMARY KEY`
  - Uniquely identifies each row
  - Cannot be NULL
  - No duplicate values allowed

Check Table Structure (SQL Server)

SQL Server does **not** support `DESC`.

Use this instead:

```
EXEC sp_help Customers;
```

or

```
SELECT column_name, data_type
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'Customers';
```

CREATE TABLE IF NOT EXISTS (SQL Server Way)

SQL Server does **not support** `CREATE TABLE IF NOT EXISTS` directly.

Correct SQL Server Method

```
IF NOT EXISTS (
    SELECT * FROM sys.tables WHERE name = 'Customers'
)
BEGIN
    CREATE TABLE Customers (
        ID INT PRIMARY KEY,
        Name VARCHAR(20) NOT NULL,
        Age INT NOT NULL
    );
END;
```

Create Table from Existing Table

Syntax

```
SELECT columns
```

```
INTO new_table
FROM existing_table
WHERE condition;
```

Example

```
SELECT ID, Salary
INTO SalaryDetails
FROM Customers;
```

✓ Creates a new table
✓ Copies **structure + data**

SQL Server – Show / List Tables

SQL Server does **not** support `SHOW TABLES`.

Method 1 – Using sys.tables (Recommended)

```
SELECT name FROM sys.tables;
```

Method 2 – INFORMATION_SCHEMA

```
SELECT table_name
FROM INFORMATION_SCHEMA.TABLES
WHERE table_type = 'BASE TABLE';
```

SQL Server – Rename Table

SQL Server uses **sp_rename**.

Syntax

```
EXEC sp_rename 'old_table_name', 'new_table_name';
```

Example

```
EXEC sp_rename 'Customers', 'Workers';
```

⚠ Update all:

- Views
- Stored procedures
- Triggers
  that reference the old table name.

SQL Server – TRUNCATE TABLE

What is TRUNCATE?

- Removes **all rows** from a table
- **Very fast**
- Keeps table structure
- Cannot be rolled back
- Requires higher privileges

Syntax

```
TRUNCATE TABLE Customers;
```

After Truncate

```
SELECT * FROM Customers;
```

Result → **No rows**

SQL - Clone Tables

Table Cloning in SQL Server

SQL Server has **no direct cloning command**, but we can use:

SELECT … INTO (simple clone + data)

- Creates a **new table and copies all data**.
- Does **NOT** copy:
  - Primary keys
  - Indexes
  - Constraints
  - AUTO_INCREMENT (IDENTITY)
- These must be added manually.

Syntax

```
SELECT * INTO new_table FROM original_table;
```

Example

```
SELECT * INTO NEW_CUSTOMERS FROM CUSTOMERS;
```

Verification

```
SELECT * FROM NEW_CUSTOMERS;
```

SQL - Temporary Tables

What are Temporary Tables?

- Temporary tables store **temporary data** during a session.

- You can perform all normal SQL operations (INSERT, UPDATE, SELECT, JOIN, etc.).
- They are **automatically deleted** when:
  - The client session ends, or
  - The script/program ends.
- They can also be deleted **manually** using `DROP TEMPORARY TABLE`.

Temporary Tables in SQL Server

SQL Server provides **two types**:

A. Local Temporary Tables

- Name starts with **#**
- Available **only to the session** that created it.
- Dropped automatically when:
  - Session ends, or
  - Stored procedure ends (if created inside).

Syntax

```
CREATE TABLE #table_name (...);
```

Example

```
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)
);
```

B. Global Temporary Tables

- Name starts with **##**
- Visible to **all sessions**.
- Dropped automatically when the **last session using it is closed**.

Syntax

```
CREATE TABLE ##table_name (...);
```

Example

```
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)
);
```

Dropping Temporary Tables in SQL Server

Local Temporary Table

```
DROP TABLE #Customers;
```

Global Temporary Table

```
DROP TABLE ##Buyers;
```

SQL - ALTER TABLE

`ALTER TABLE` is a **DDL (Data Definition Language)** command used to **modify the structure** of an existing database table.

It can be used to:

- Add/Delete columns
- Add/Delete indexes
- Change column datatype
- Rename columns or table
- Add/Delete constraints (Primary Key, Unique, etc.)
- Modify storage properties (engine, etc.)

Basic Syntax

```
ALTER TABLE table_name [alter_option ...];
```

---

1. ADD COLUMN

Syntax

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

Example

```
ALTER TABLE CUSTOMERS ADD SEX CHAR(1);
```

After adding, the new column appears with `NULL` values.

---

2. DROP COLUMN

Syntax

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

Example

```
ALTER TABLE CUSTOMERS DROP COLUMN SEX;
```

Column is permanently removed.

---

3. ADD INDEX

Syntax

```
ALTER TABLE table_name
ADD INDEX index_name (column_name);
```

Example

```
ALTER TABLE CUSTOMERS ADD INDEX name_index (NAME);
```

Indexes help speed up queries.

---

4. DROP INDEX

Syntax

```
ALTER TABLE table_name DROP INDEX index_name;
```

Example

```
ALTER TABLE CUSTOMERS DROP INDEX name_index;
```

---

5. ADD PRIMARY KEY

Syntax

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
PRIMARY KEY (column1, column2...);
```

Example

```
ALTER TABLE EMPLOYEES
ADD CONSTRAINT MyPrimaryKey
PRIMARY KEY(ID);
```

---

6. DROP PRIMARY KEY

Syntax

```
ALTER TABLE table_name DROP PRIMARY KEY;
```

Example

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

---

7. ADD UNIQUE CONSTRAINT

Syntax

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
UNIQUE(column1, column2...);
```

Example

```
ALTER TABLE EMPLOYEES ADD CONSTRAINT CONST UNIQUE(NAME);
```

---

8. DROP CONSTRAINT

(Used to remove UNIQUE or other named constraints.)

Syntax

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

Example

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT CONST;
```

---

9. RENAME COLUMN

Syntax

```
ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;
```

Example

```
ALTER TABLE CUSTOMERS RENAME COLUMN name TO full_name;
```

---

10. MODIFY COLUMN DATATYPE

SQL Server / MS Access

```
ALTER TABLE table_name ALTER COLUMN column_name datatype;
```

MySQL

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

Oracle

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

Example (MySQL)

```
ALTER TABLE CUSTOMERS MODIFY COLUMN ID DECIMAL(18,4);
```

Changes data type successfully

SQL - DROP Table

SQL Server – DROP TABLE

What is DROP TABLE?

`DROP TABLE` is a **DDL command** used to **permanently remove a table** from the database.

After DROP TABLE:

- Table structure is removed
- All data is deleted
- Indexes, constraints, triggers are removed
- Permissions on the table are removed
- **Cannot be rolled back**
- Causes an **implicit commit**

⚠ Use carefully in production systems

What DROP TABLE Does (SQL Server)

- Removes table definition
- Deletes all rows
- Drops:
  - Primary keys
  - Foreign keys
  - Indexes
  - Triggers
- Drops all partitions (if partitioned)
- Requires:
  - `ALTER` permission on the table
  - `CONTROL` permission on the schema

Syntax

```
DROP TABLE table_name;
```

Example

```
DROP TABLE Customers;
```

Verify Table Deletion (SQL Server)

SQL Server does **not support `DESC`**.

```
SELECT * FROM Customers;
```

Output:

```
Invalid object name 'Customers'.
```

Safer Way – DROP TABLE IF EXISTS

Prevents error if the table does not exist.

```
DROP TABLE IF EXISTS Customers;
```

SQL - Delete Table

What is DELETE?

`DELETE` is a **DML command** used to remove **rows only** from a table.

After DELETE:

- Table structure remains
- Indexes, constraints, triggers remain
- Can be rolled back (inside transaction)

DELETE Syntax

```
DELETE FROM table_name;
```

⚠ Without `WHERE`, **all rows are deleted**

DELETE With WHERE (Single Condition)

```
DELETE FROM Customers
WHERE Name = 'Hardik';
```

DELETE With Multiple Conditions

```
DELETE FROM Customers
WHERE Name = 'Komal' OR Address = 'Mumbai';
```

DELETE All Rows (Table Remains)

```
DELETE FROM Customers;
```

SQL Constraints

What Are SQL Constraints?

SQL constraints are **rules** applied to table columns to ensure that the data stored is **correct, reliable, and meaningful**.

They:

- Restrict what values can be inserted
- Prevent invalid or duplicate data
- Maintain relationships between tables
- Improve data quality and consistency

Constraints can be added:

- When creating a table (`CREATE TABLE`)
- Later using `ALTER TABLE`

Why Use Constraints?

Constraints help:

- Prevent invalid data (e.g., letters in phone number)
- Avoid missing or duplicate values
- Maintain relationships (Foreign Key)
- Make databases cleaner, safer, and easier to manage

Types of SQL Constraints

Constraint	Purpose
NOT NULL	Value cannot be NULL
UNIQUE	All values must be different
PRIMARY KEY	Unique + Not Null; identifies each row
FOREIGN KEY	Links one table to another
CHECK	Ensures values follow a condition
DEFAULT	Assigns default value if none provided

Defining Constraints (Syntax)

During table creation

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

Adding constraints later

```
ALTER TABLE table_name ADD CONSTRAINT ...
```

---

1. NOT NULL Constraint

Ensures a column **must have a value**.

Example

```
CREATE TABLE CUSTOMERS (
   ID INT NOT NULL,
   NAME VARCHAR(50) NOT NULL
);
```

Trying to insert NULL in NAME:

```
INSERT INTO CUSTOMERS (ID) VALUES (101);
```

Error:

```
ERROR 1364: Field 'NAME' doesn't have a default value
```

---

2. UNIQUE Constraint

Ensures all values in a column are **unique**.

Example

```
CREATE TABLE CUSTOMERS (
   ID INT NOT NULL,
   NAME VARCHAR(50),
   EMAIL VARCHAR(100) UNIQUE
);
```

Attempting to insert duplicate email:

```
ERROR 1062: Duplicate entry for key 'EMAIL'
```

---

3. PRIMARY KEY Constraint

Uniquely identifies each row.
Automatically **NOT NULL + UNIQUE**.

Example

```
CREATE TABLE CUSTOMERS (
   ID INT PRIMARY KEY,
   NAME VARCHAR(50)
);
```

Inserting duplicate ID:

```
ERROR 1062: Duplicate entry for key 'PRIMARY'
```

---

4. FOREIGN KEY Constraint

Maintains **referential integrity** between tables.

Example

```
CREATE TABLE DEPARTMENTS (
   DEPT_ID INT PRIMARY KEY,
   DEPT_NAME VARCHAR(100)
);

CREATE TABLE CUSTOMERS (
   ID INT PRIMARY KEY,
   NAME VARCHAR(50),
   DEPT_ID INT,
   FOREIGN KEY (DEPT_ID) REFERENCES DEPARTMENTS(DEPT_ID)
);
```

Inserting a non-existing DEPT_ID:

```
ERROR 1452: Cannot add or update a child row
```

---

5. CHECK Constraint

Ensures values satisfy a condition.

Example

```
CREATE TABLE CUSTOMERS (
   ID INT PRIMARY KEY,
   NAME VARCHAR(50),
   AGE INT CHECK (AGE >= 18)
);
```

Inserting age below 18:

```
ERROR 3819: Check constraint violated
```

---

6. DEFAULT Constraint

Automatically provides a **default value** when the user does not.

Example

```
CREATE TABLE CUSTOMERS (
   ID INT PRIMARY KEY,
   NAME VARCHAR(50),
   SALARY DECIMAL(10,2) DEFAULT 5000.00
);
```

Insert without salary:

```
INSERT INTO CUSTOMERS (ID, NAME) VALUES (1, 'Komal');
```

Result → SALARY will be `5000.00`.

Using Multiple Constraints Together

Example

```
CREATE TABLE CUSTOMERS (
   ID INT PRIMARY KEY,
   NAME VARCHAR(50) NOT NULL,
   EMAIL VARCHAR(100) UNIQUE,
   AGE INT CHECK (AGE >= 18),
   SALARY DECIMAL(10,2) DEFAULT 5000.00
);
```

This enforces:

- Unique ID
- Name required
- Unique email
- Age >= 18
- Default salary

Adding Constraints to Existing Tables

1. Add NOT NULL

```
ALTER TABLE CUSTOMERS
MODIFY NAME VARCHAR(100) NOT NULL;
```

2. Add CHECK

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT chk_salary CHECK (SALARY > 0);
```

Violations will produce:

```
ERROR 1048: Column 'NAME' cannot be null
ERROR 3819: Check constraint 'chk_salary' is violated
```

DML Queries (INSERT, SELECT, UPDATE, DELETE) & ORDER BY

SQL Server – INSERT Query

What is INSERT?

`INSERT INTO` is a **DML command** used to add new rows into a table.

Important Rules

- Values must match column **datatypes**
- Must satisfy **constraints** (NOT NULL, PRIMARY KEY, UNIQUE)
- Number of values must match number of columns

Syntax

With column names (Recommended)

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

Without column names

```
INSERT INTO table_name
VALUES (val1, val2, val3);
```

⚠ Values must follow the **table column order**

Insert Single Row

```
INSERT INTO Customers (ID, Name, Age, Address, Salary)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000);
```

Insert Multiple Rows (SQL Server supports this)

```
INSERT INTO Customers (ID, Name, Age, Address, Salary)
VALUES
(2, 'Khilan', 25, 'Delhi', 1500),
(3, 'Kaushik', 23, 'Kota', 2000),
(4, 'Chaitali', 25, 'Mumbai', 6500);
```

Insert Data from Another Table (INSERT…SELECT)

```
INSERT INTO Buyers (ID, Name, Age)
SELECT ID, Name, Age
FROM Customers;
```

✓ Target table must already exist
✓ Column count & datatypes must match

SQL Server – SELECT Query

What is SELECT?

Used to **retrieve data** from a table.
The output is called a **result set**.

Basic Syntax

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

Select All Columns

```
SELECT * FROM Customers;
```

Select Specific Columns

```
SELECT ID, Name, Salary
FROM Customers;
```

Using Expressions

```
SELECT 56 * 65 AS Result;
```

Column Alias

```
SELECT Name AS CustomerName, Salary AS MonthlySalary
FROM Customers;
```

SQL Server – SELECT INTO

What is SELECT INTO?

Creates a **new table** and copies data from an existing table.

⚠ Does **not** copy:

- Indexes
- Constraints
- Keys

Syntax

```
SELECT *
INTO new_table
FROM existing_table;
```

Example

```
SELECT *
INTO Customers_Backup
FROM Customers;
```

Copy Specific Columns

```
SELECT Name, Age, Address
INTO Customer_Details
FROM Customers;
```

Copy with Condition

```
SELECT *
INTO Customers_AgeAbove25
```

```
FROM Customers
WHERE Age > 25;
```

SQL Server – UPDATE Query

What is UPDATE?

Used to **modify existing records** in a table.

⚠ Without `WHERE`, **all rows are updated**

Syntax

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

Update Single Row

```
UPDATE Customers
SET Address = 'Pune'
WHERE ID = 6;
```

Update Multiple Columns

```
UPDATE Customers
SET Address = 'Pune', Salary = 1000
WHERE Name = 'Ramesh';
```

Update Multiple Rows

```
UPDATE Customers
SET Age = Age + 5, Salary = Salary + 3000;
```

SQL Server – DELETE Query

What is DELETE?

Deletes **rows only**, not the table structure.

Syntax

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

Delete Single Row

```
DELETE FROM Customers
WHERE ID = 6;
```

Delete Multiple Rows

```
DELETE FROM Customers
WHERE Age > 25;
```

Delete All Rows (Table remains)

```
DELETE FROM Customers;
```

DELETE with JOIN (SQL Server style)

```
DELETE C
FROM Customers C
INNER JOIN Orders O ON O.Customer_ID = C.ID
WHERE C.Salary > 2000;
```

SQL Server – ORDER BY (Sorting)

What is ORDER BY?

Used to **sort query results**.

- Default order → **ASC**
- Supports **ASC** and **DESC**

Syntax

```
SELECT columns
FROM table
ORDER BY column [ASC | DESC];
```

Ascending Order

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

Descending Order

```
SELECT * FROM Customers
ORDER BY Name DESC;
```

Multiple Column Sorting

```
SELECT * FROM Customers
ORDER BY Age ASC, Salary DESC;
```

Custom Order using CASE

```
SELECT * FROM Customers
ORDER BY CASE Address
    WHEN 'Delhi' THEN 1
    WHEN 'Bhopal' THEN 2
    WHEN 'Kota' THEN 3
    ELSE 100
END;
```

SQL Views

1. What is a View?

- A **view** is a **virtual table** based on a query.
- Does **not store data** (unless indexed).
- Purposes:
    1. Simplify complex queries.
    2. Restrict access to certain rows/columns.
    3. Summarize or combine data for reports.

---

2. Create a View

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example

```
CREATE VIEW Customers_View AS
SELECT * FROM Customers;
```

With WHERE clause

```
CREATE VIEW Buyers_View AS
SELECT * FROM Customers
WHERE Salary > 3000;
```

WITH CHECK OPTION

- Ensures **INSERT/UPDATE through view** obeys the view condition:

```
CREATE VIEW My_View AS
SELECT Name, Age
FROM Customers
WHERE Age >= 25
WITH CHECK OPTION;
```

---

3. Update Through a View

Syntax

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

Examples

- Update single row:

```
UPDATE Customers_View
SET Age = 35
WHERE Name = 'Ramesh';
```

- Update multiple columns:

```
UPDATE Customers_View
SET Name = 'Kaushik Ramanujan', Age = 24
WHERE ID = 3;
```

- Update all rows:

```
UPDATE Customers_View
SET Age = Age + 6;
```

---

4. Delete Through a View

Delete rows

```
DELETE FROM view_name
WHERE condition;
```

Deletes records from the **base table**.

Example

```
DELETE FROM Customers_View
WHERE Age = 22;
```

---

5. Drop a View

Syntax

```
DROP VIEW view_name;
DROP VIEW IF EXISTS view_name;  -- safer
```

Example

```
DROP VIEW Customers_View3;
DROP VIEW IF EXISTS Demo_View;
```

---

6. Rename a View

In SQL Server

```
EXEC sp_rename 'old_view_name', 'new_view_name';
```

Example

```
EXEC sp_rename 'Customers_View', 'View_Customers';
```

In MySQL

```
RENAME TABLE old_view_name TO new_view_name;
```

SQL Operators and Clauses

1. WHERE Clause

- Filters rows based on a condition.
- Works with `SELECT`, `UPDATE`, `DELETE`.
- Operators:
  - Comparison: `=, <, >, <=, >=, <>`
  - Logical: `AND, OR, NOT`
  - Pattern: `LIKE`, `IN`, `NOT IN`

1. WHERE with SELECT

Retrieve specific rows from a table based on a condition.

Example:

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

Output:

ID	NAME	SALARY
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00

---

2. WHERE with UPDATE

Update specific records in a table using a condition.

Example:

```
UPDATE CUSTOMERS
SET SALARY = SALARY + 10000
WHERE NAME = 'Ramesh';
```

- If you don’t use `WHERE`, all rows will be updated.
- Verification:

```
SELECT * FROM CUSTOMERS WHERE NAME = 'Ramesh';
```

---

3. WHERE with IN

Filter rows matching **any value from a list**.

Syntax:

```
WHERE column_name IN (value1, value2, ...);
```

Example:

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

Output:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

---

4. WHERE with NOT IN

Filter rows **excluding** a specific list of values.

Syntax:

```
WHERE column_name NOT IN (value1, value2, ...);
```

Example:

```
SELECT * FROM CUSTOMERS
WHERE AGE NOT IN (25, 23, 22);
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	12000.00
5	Hardik	27	Bhopal	8500.00

ID	NAME	AGE	ADDRESS	SALARY
7	Muffy	24	Indore	10000.00

---

5. WHERE with LIKE

Filter rows based on **pattern matching** using wildcards:

- `%` → any number of characters
- `_` → exactly one character

Syntax:

```
WHERE column_name LIKE pattern;
```

Example:

```
SELECT * FROM CUSTOMERS
WHERE NAME LIKE 'K___%';
```

Output: Names starting with `K` and at least 4 characters long.

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
6	Komal	22	Hyderabad	4500.00

---

6. WHERE with AND / OR Operators

Combine multiple conditions to filter rows.

Syntax:

```
WHERE (condition1 OR condition2) AND condition3;
```

Example:

```
SELECT * FROM CUSTOMERS
WHERE (AGE = 25 OR SALARY < 4500)
AND (NAME = 'Komal' OR NAME = 'Kaushik');
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
3	Kaushik	23	Kota	2000.00

SQL - TOP Clause

SQL TOP Clause

The **`TOP` clause** in SQL Server is used to **limit the number of rows returned** by a query. It can also be used with `UPDATE` and `DELETE` to restrict the number of affected rows.

Note:

- MySQL uses `LIMIT` instead of `TOP`.
- Oracle uses `ROWNUM`.
- `TOP` can accept both a **number of rows** or a **percentage**.

---

1. Basic Syntax

```
SELECT TOP value column_name(s)
FROM table_name
WHERE condition;
```

- `value`: Number or percentage of rows to return.
- `condition`: Optional, filters rows using `WHERE`.

---

2. Example Table

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

INSERT INTO CUSTOMERS VALUES
(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);
```

---

3. TOP without conditions

Fetch the **first N rows**:

```
SELECT TOP 4 * FROM CUSTOMERS;
```

Output:

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

---

4. TOP with ORDER BY

Retrieve **top rows in sorted order**:

```
SELECT TOP 4 * FROM CUSTOMERS ORDER BY SALARY DESC;
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
7	Muffy	24	Indore	10000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00
6	Komal	22	Hyderabad	4500.00

Default `ORDER BY` is ascending. Use `DESC` for descending.

---

5. TOP with PERCENT

Fetch a **percentage of rows**:

```
SELECT TOP 40 PERCENT *
FROM CUSTOMERS
ORDER BY SALARY;
```

- Table has 7 rows → 40% ≈ 2.8 → rounded to **3 rows**.

Output:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00

---

6. TOP with WHERE

Limit **rows satisfying a condition**:

```
SELECT TOP 2 *
FROM CUSTOMERS
WHERE NAME LIKE 'K%';
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00

---

7. TOP with DELETE

Delete **specific number of rows**:

```
DELETE TOP(2)
FROM CUSTOMERS
WHERE NAME LIKE 'K%';
```

- Deletes **top 2 customers** whose name starts with 'K'.
- Verify deletion:

```
SELECT * FROM CUSTOMERS;
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	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

---

8. TOP with WITH TIES

Include **all tied rows** when multiple rows share the same ORDER BY value:

```
SELECT TOP 2 WITH TIES *
```

```
FROM CUSTOMERS
ORDER BY SALARY;
```

Output:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
1	Ramesh	32	Ahmedabad	2000.00

SQL - DISTINCT Keyword

What is DISTINCT?

- `DISTINCT` is used with `SELECT` to remove duplicate rows.
- It returns only **unique values**.
- Useful when working with large tables containing repeated data.

Syntax

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

DISTINCT on a Single Column

- Removes duplicates from one column.

Example:

```
SELECT DISTINCT SALARY FROM CUSTOMERS;
```

→ Returns only unique salary values.

DISTINCT on Multiple Columns

- Checks **unique combinations** across multiple columns.
- A row is considered duplicate only if **all selected columns match**.

Example:

```
SELECT DISTINCT AGE, SALARY FROM CUSTOMERS;
```

DISTINCT with ORDER BY

- First removes duplicates, then sorts the result.

Example:

```
SELECT DISTINCT SALARY
FROM CUSTOMERS
ORDER BY SALARY DESC;
```

DISTINCT with COUNT()

- Counts the **number of unique values** in a column.

Example:

```
SELECT COUNT(DISTINCT AGE) AS UniqueAge FROM CUSTOMERS;
```

DISTINCT with NULL values

- SQL treats **NULL as a unique value** (even if multiple NULLs exist).
- So `DISTINCT` includes only one NULL.

Example:

```
SELECT DISTINCT SALARY FROM CUSTOMERS;
```

SQL - ORDER BY Clause

What is ORDER BY?

- Used to **sort query results**.
- Default order = **ASC (ascending)**.
- Use **DESC** for descending order.
- Appears at the **end of the SELECT query**.

Syntax

```
SELECT column_list
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

---

1. ORDER BY with ASC (Default)

Sorts results in ascending order.

Example:

```
SELECT * FROM CUSTOMERS ORDER BY NAME ASC;
```

---

2. ORDER BY with DESC

Sorts results in descending order.

Example:

```
SELECT * FROM CUSTOMERS ORDER BY NAME DESC;
```

---

3. Using ASC and DESC Together

You can sort each column differently.

Example:

```
SELECT * FROM CUSTOMERS
ORDER BY AGE ASC, SALARY DESC;
```

---

4. ORDER BY on a Single Column

Most basic usage.

Example:

```
SELECT * FROM CUSTOMERS ORDER BY AGE;
```

---

5. ORDER BY on Multiple Columns

Sorting follows the order of columns in the clause.

Example:

```
SELECT * FROM CUSTOMERS
```

```
ORDER BY AGE ASC, SALARY DESC;
```

---

6. ORDER BY with WHERE

Sort only the rows that meet a condition.

Example:

```
SELECT * FROM CUSTOMERS
WHERE AGE = 25
ORDER BY NAME DESC;
```

---

7. ORDER BY with LIMIT

Used to get **top N** sorted rows.

Example:

```
SELECT SALARY
FROM CUSTOMERS
ORDER BY NAME
LIMIT 4;
```

---

8. Custom Sorting with CASE

Used to sort in **your own preferred order**.

Example:

```
SELECT * FROM CUSTOMERS
ORDER BY (
  CASE ADDRESS
    WHEN 'MUMBAI' THEN 1
    WHEN 'DELHI' THEN 2
    WHEN 'HYDERABAD' THEN 3
    WHEN 'AHMEDABAD' THEN 4
    WHEN 'INDORE' THEN 5
    WHEN 'BHOPAL' THEN 6
    WHEN 'KOTA' THEN 7
    ELSE 100
  END
);
```

SQL - Group By Clause

What is GROUP BY?

- Used to **group rows** that have the same values in one or more columns.
- Works mainly with **aggregate functions**:
  - `COUNT()`
  - `SUM()`
  - `AVG()`
  - `MIN()`
  - `MAX()`
- Used for **summarizing data**, like totals or averages.
- Appears **after WHERE** and **before HAVING / ORDER BY**.

Syntax

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

GROUP BY with Aggregate Functions

Used to perform calculations for each group.

Example: Count customers by age

```
SELECT AGE, COUNT(NAME)
FROM CUSTOMERS
GROUP BY AGE;
```

Example: Max salary by age

```
SELECT AGE, MAX(SALARY) AS MAX_SALARY
FROM CUSTOMERS
GROUP BY AGE;
```

GROUP BY on a Single Column

Groups data based on one column.

Example: Average salary by city

```
SELECT ADDRESS, AVG(SALARY) AS AVG_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS;
```

GROUP BY on Multiple Columns

Groups rows using more than one column (unique combinations).

Example:

```
SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS, AGE;
```

GROUP BY with ORDER BY

Sort the grouped results.

Example: Sort minimum salaries high → low

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

GROUP BY with HAVING

- `HAVING` filters **after grouping**
- `WHERE` filters **before grouping**

Example: Groups where age > 24

```
SELECT ADDRESS, AGE, MIN(SALARY) AS MIN_SUM
FROM CUSTOMERS
GROUP BY ADDRESS, AGE
HAVING AGE > 24;
```

GROUP BY with JOIN

Used to summarize data from **multiple tables**.

General syntax:

```
SELECT t1.col, t2.col, aggregate_function(col)
FROM table1 t1
JOIN table2 t2 ON t1.common = t2.common
GROUP BY t1.col, t2.col;
```

Example: Total order amount per customer

```
SELECT C.NAME, SUM(O.AMOUNT) AS TOTAL_ORDER_AMOUNT
FROM CUSTOMERS C
JOIN ORDERS O ON C.ID = O.CUSTOMER_ID
GROUP BY C.NAME;
```

SQL - Having Clause

What is HAVING?

- The **HAVING** clause is used to filter **groups** after applying aggregate functions like:
  - `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`
- It works **after GROUP BY**.
- **WHERE** filters **rows** before grouping.
  HAVING filters **groups** after grouping.
- We need HAVING because **WHERE cannot use aggregate functions**.

Syntax

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

Order of SQL Query:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

HAVING with GROUP BY

Used to filter grouped results.

Example:

```
SELECT ADDRESS, AGE, MIN(SALARY) AS MIN_SUM
FROM CUSTOMERS
GROUP BY ADDRESS, AGE
HAVING AGE > 25;
```

HAVING with ORDER BY

You can sort the filtered groups.

Example:

```
SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS, AGE
HAVING SUM(SALARY) >= 5000
ORDER BY TOTAL_SALARY DESC;
```

HAVING with COUNT()

Filter groups based on number of rows.

```
SELECT AGE, COUNT(AGE)
FROM CUSTOMERS
GROUP BY AGE
HAVING COUNT(AGE) > 2;
```

HAVING with AVG()

Filter groups based on average value.

```
SELECT ADDRESS, AVG(SALARY) AS AVG_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS
HAVING AVG(SALARY) > 5240;
```

HAVING with MAX()

Filter groups using maximum value.

```
SELECT ADDRESS, MAX(SALARY) AS MAX_SALARY
FROM CUSTOMERS
GROUP BY ADDRESS
HAVING MAX(SALARY) > 5240;
```

HAVING with Multiple Conditions

Use AND / OR with aggregate functions.

```
SELECT ADDRESS, AGE, SUM(SALARY), MIN(SALARY)
FROM CUSTOMERS
GROUP BY ADDRESS, AGE
HAVING SUM(SALARY) >= 5000 AND MIN(SALARY) > 1500;
```

HAVING with Joins

HAVING can be used after JOIN + GROUP BY.

```
SELECT C.ADDRESS, SUM(O.AMOUNT) AS TOTAL_ORDERS
FROM CUSTOMERS C
JOIN ORDERS O ON C.ID = O.CUSTOMER_ID
GROUP BY C.ADDRESS
HAVING SUM(O.AMOUNT) > 3000;
```

HAVING vs WHERE (Simple Difference)

WHERE	HAVING
Filters individual rows **before** grouping	Filters groups **after** grouping
Cannot use aggregate functions	Can use aggregate functions
Works in SELECT/UPDATE/DELETE	Mostly used in SELECT with GROUP BY
Checks each row	Checks each group

Example using both:

```
SELECT ADDRESS, COUNT(*) AS TOTAL_CUSTOMERS
FROM CUSTOMERS
WHERE SALARY > 2000
GROUP BY ADDRESS
HAVING COUNT(*) >= 1;
```

SQL - AND and OR Operators

1. What are AND & OR Operators?

- Used in **WHERE** or **HAVING** clause.
- Help to filter rows using **multiple conditions**.

AND

✓ Returns TRUE only if **all conditions** are TRUE.
Example:

```
WHERE Age > 25 AND Salary > 5000
```

OR

✓ Returns TRUE if **any one condition** is TRUE.
Example:

```
WHERE City = 'London' OR City = 'Paris'
```

Truth Table

Condition1	Condition2	AND	OR
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

---

2. AND Operator

Used to filter rows that satisfy **all** given conditions.

Syntax

```
SELECT columns
FROM table
WHERE condition1 AND condition2;
```

Example

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND AGE < 25;
```

Using Multiple AND Conditions

```
WHERE NAME LIKE 'K%' AND AGE >= 22 AND SALARY < 3742;
```

Combining AND with NOT

`NOT` reverses the result of the condition.

```
WHERE NOT (SALARY > 4500 AND AGE < 26);
```

AND with UPDATE

Updates only rows that satisfy **all** conditions.

```
UPDATE CUSTOMERS
SET SALARY = 55000
WHERE AGE > 27;
```

---

3. OR Operator

Returns rows where **at least one** condition is TRUE.

Syntax

```
SELECT columns
FROM table
WHERE condition1 OR condition2;
```

Example

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR AGE < 25;
```

Multiple OR Conditions

```
WHERE NAME LIKE '%l' OR SALARY > 10560 OR AGE < 25;
```

OR with DELETE

Deletes rows that meet **any one** of the conditions.

```
DELETE FROM CUSTOMERS
WHERE AGE = 25 OR SALARY < 2000;
```

---

4. Using AND & OR Together

Use **parentheses** to control the condition order.

Syntax

```
WHERE (condition1 OR condition2) AND condition3;
```

Example

```
SELECT *
FROM CUSTOMERS
WHERE (AGE = 25 OR SALARY < 4500)
AND (NAME = 'Komal' OR NAME = 'Kaushik');
```

SQL - BOOLEAN

1. What is a Boolean?

- A **Boolean** stores **TRUE** or **FALSE** values.
- Useful when we need columns like:
  - IsRed
  - IsAvailable
  - IsActive
- Example: To find all **red cars**, we check a Boolean column `IsRed`.

1. Boolean Representation in SQL Server

- No direct `BOOLEAN` type.
- Use `BIT` instead:
  - `1` → TRUE
  - `0` → FALSE
  - `NULL` → unknown / not specified

Example table:

```
CREATE TABLE CUSTOMERS (
   ID INT,
   Name VARCHAR(150),
   IsAvailable BIT
);
```

---

2. Filtering Boolean Data

- **TRUE condition:**

```
SELECT * FROM CUSTOMERS WHERE IsAvailable = 1;
```

- **FALSE condition:**

```
SELECT * FROM CUSTOMERS WHERE IsAvailable = 0;
```

---

3. Negating Boolean Conditions

- Using `NOT` or `FALSE`:

```
-- Option 1
SELECT * FROM CUSTOMERS WHERE NOT IsAvailable = 1;

-- Option 2 (direct)
SELECT * FROM CUSTOMERS WHERE IsAvailable = 0;
```

4. Handling NULL values

- NULL represents “unknown” in BIT columns.

```
-- Find unknown status
SELECT * FROM CUSTOMERS WHERE IsAvailable IS NULL;

-- Find known (not NULL) status
SELECT * FROM CUSTOMERS WHERE IsAvailable IS NOT NULL;
```

---

5. Updating Boolean Values

```
-- Set to TRUE
UPDATE CUSTOMERS SET IsAvailable = 1 WHERE ID = 123;

-- Set to FALSE
UPDATE CUSTOMERS SET IsAvailable = 0 WHERE ID = 124;
```

SQL - Like Operator

1. What is LIKE Operator?

- Used to search for a **pattern** in a column.
- Mostly used with **WHERE** in SELECT, UPDATE, DELETE.
- Example: Names starting with **K**
- `WHERE student_name LIKE 'K%';`

Wildcards used:

- **%** → zero or more characters
- **_** → exactly one character

---

2. Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE pattern;
```

---

3. Common SQL Wildcards

Wildcard	Meaning
%	0, 1, or many characters
_	Exactly 1 character

---

4. Examples of LIKE Patterns

Pattern	Meaning
`'200%'`	Starts with 200
`'%200%'`	Contains 200 anywhere
`'_00%'`	2nd & 3rd char = 00
`'2_%_%'`	Starts with 2, at least 3 chars
`'%2'`	Ends with 2
`'_2%3'`	2 in second position and ends with 3
`'2___3'`	5 characters: first 2, last 3

---

5. Using % Wildcard

Example: Salary starting with 200

```
SELECT * FROM CUSTOMERS WHERE SALARY LIKE '200%';
```

Example: Names ending with “sh”

```
SELECT * FROM CUSTOMERS WHERE NAME LIKE '%sh';
```

Example: Names containing “al”

```
SELECT * FROM CUSTOMERS WHERE NAME LIKE '%al%';
```

---

6. Using _ Wildcard

Example: Names starting with K, 4+ characters

```
WHERE NAME LIKE 'K___%';
```

Example: “m” in 3rd position

```
WHERE NAME LIKE '__m%';
```

---

7. Combining Wildcards

Pattern	Meaning
`'K___%'`	Starts with K, at least 4 characters
`'_2%3'`	Any 1st char, second is 2, ends with 3
`'2___3'`	5-digit value starting with 2 and ending with 3

---

8. LIKE with OR Operator

```
SELECT *
FROM CUSTOMERS
WHERE NAME LIKE 'C%i' OR NAME LIKE '%k';
```

---

9. SQL NOT LIKE

Used to exclude patterns.

Example: Names NOT starting with K

```
WHERE NAME NOT LIKE 'K%';
```

---

10. Escape Characters in LIKE

Used when searching for literal **%**, **_**, or other special symbols.

Syntax:

```
WHERE column LIKE 'pattern ESCAPE escape_char';
```

Example: Find rows containing literal %

```
SELECT *
FROM EMPLOYEE
WHERE BONUS_PERCENT LIKE '%!%%' ESCAPE '!';
```

---

11. LIKE Without Wildcards

Works like an exact match (=).

Example:

```
WHERE NAME LIKE 'Komal';
```

SQL - IN Operator

1. What is the IN Operator?

- Used in the **WHERE** clause.
- Checks if a column's value **matches any value in a list**.
- Works like multiple OR conditions, but shorter.

Example:

```
WHERE NAME IN ('A', 'B', 'C');
```

This is the same as:

```
WHERE NAME = 'A' OR NAME = 'B' OR NAME = 'C';
```

IN works with:
✓ Numbers
✓ Strings
✓ Dates
✓ Subqueries

---

2. Syntax

```
WHERE column_name IN (value1, value2, value3, ...);
```

---

3. IN with SELECT

To fetch rows where NAME is 'Khilan', 'Hardik', or 'Muffy':

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

---

4. IN vs OR

IN:

- Shorter
- Easier to read
- Better for long lists

OR:

```
WHERE NAME = 'Khilan' OR NAME = 'Hardik' OR NAME = 'Muffy';
```

Produces the same result.

---

5. IN with UPDATE

Update AGE to 30 for customers whose AGE is 25 or 27:

```
UPDATE CUSTOMERS
SET AGE = 30
WHERE AGE IN (25, 27);
```

---

6. NOT IN Operator

Opposite of IN → selects rows **NOT** in the list.

Example: AGE not equal to 25, 23, or 22:

```
SELECT * FROM CUSTOMERS
WHERE AGE NOT IN (25, 23, 22);
```

---

7. IN with Column Name

Checks if a value exists in a column.

Example: Find rows where SALARY contains 2000:

```
SELECT * FROM CUSTOMERS
WHERE 2000 IN (SALARY);
```

---

8. IN with Subquery

Select customers whose SALARY > 2000:

```
SELECT *
FROM CUSTOMERS
WHERE NAME IN (
    SELECT NAME
    FROM CUSTOMERS
    WHERE SALARY > 2000
);
```

---

9. NOT IN with Subquery

Select customers whose salary is **NOT** greater than 2000:

```
SELECT *
FROM CUSTOMERS
WHERE NAME NOT IN (
    SELECT NAME
    FROM CUSTOMERS WHERE SALARY > 2000
);
```

SQL - ANY, ALL Operators

1. What Are ANY and ALL?

Both operators compare **one value** with a set of values returned by a **subquery**.

✓ ANY (or SOME)

- Returns **TRUE** if **at least one** value in the subquery satisfies the comparison
- Works like "**if any one of them is true**"

✓ ALL

- Returns **TRUE** only if **every** value in the subquery satisfies the comparison
- Works like "**must be true for all values**"

Must be used with:

```
=, >, <, >=, <=, <>, !=
```

---

2. Syntax

```
column_name operator ANY (subquery);
column_name operator ALL (subquery);
```

---

3. SQL ANY Operator

Used when you want the condition to be true for **any one** value from the subquery.

✓ ANY with '>'

Find customers whose salary is greater than *any* salary of customers aged 32:

```
SELECT * FROM CUSTOMERS
WHERE SALARY > ANY (
  SELECT SALARY FROM CUSTOMERS WHERE AGE = 32
);
```

✓ ANY with '<'

Find distinct ages where salary is less than *any* average salary:

```
SELECT DISTINCT AGE
FROM CUSTOMERS
WHERE SALARY < ANY (SELECT AVG(SALARY) FROM CUSTOMERS);
```

✓ ANY with '='

Find customers whose age matches **any** age of customers whose name starts with 'K':

```
SELECT *
FROM CUSTOMERS
WHERE AGE = ANY (
  SELECT AGE FROM CUSTOMERS WHERE NAME LIKE 'K%'
);
```

4. SQL ALL Operator

Condition must be true for **all** values in the subquery.

Syntax:

```
column_name operator ALL (subquery);
```

ALL with WHERE

Find customers whose salary is **not equal to any** salary of customers aged 25:

```
SELECT *
FROM CUSTOMERS
WHERE SALARY <> ALL (
  SELECT SALARY FROM CUSTOMERS WHERE AGE = 25
);
```

ALL with HAVING

Groups whose salary is less than the **average salary**:

```
SELECT NAME, AGE, ADDRESS, SALARY
```

```
FROM CUSTOMERS
GROUP BY AGE, SALARY
HAVING SALARY < ALL (
  SELECT AVG(SALARY) FROM CUSTOMERS
);
```

5. Using ANY & ALL with Other SQL Clauses

With JOIN

Compare values from joined tables:

```
SELECT DISTINCT c1.NAME, c1.SALARY, c2.ADDRESS AS ComparedAddress
FROM CUSTOMERS c1
JOIN CUSTOMERS c2 ON c1.ID <> c2.ID AND c2.ADDRESS = 'Delhi'
WHERE c1.SALARY > ANY (
  SELECT SALARY FROM CUSTOMERS WHERE ADDRESS='Delhi'
);
```

With GROUP BY

```
SELECT AGE, COUNT(*) AS CountCustomers
FROM CUSTOMERS
GROUP BY AGE
HAVING COUNT(*) > (
  SELECT AVG(age_count)
  FROM (
     SELECT COUNT(*) AS age_count
     FROM CUSTOMERS
     GROUP BY AGE
  ) AS subquery
);
```

With ORDER BY

```
SELECT *
FROM CUSTOMERS
WHERE SALARY < ANY (
  SELECT SALARY FROM CUSTOMERS WHERE AGE > 30
)
ORDER BY SALARY DESC;
```

---

6. Difference Between ANY and ALL — Quick Table

Feature	ANY	ALL
Meaning	TRUE if **at least one** value satisfies condition	TRUE only if **every** value satisfies condition

Feature	ANY	ALL
Works like	“OR inside subquery”	“AND inside subquery”
Returns TRUE when	One comparison is true	All comparisons are true
Use case	Check if value matches **any one** in list	Check if value satisfies condition for **all values**
Example	`salary > ANY(subquery)`	`salary > ALL(subquery)`

SQL - EXISTS Operator

SQL EXISTS Operator

The `EXISTS` operator is used to **check if a subquery returns any records**.
It returns a **Boolean value**: `TRUE` if at least one record exists, otherwise `FALSE`.

- Usually used in **WHERE** clauses.
- Can also be used with **SELECT, UPDATE, DELETE, INSERT**.
- **Fast** for correlated subqueries because it stops searching after the first match.

---

1. Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE EXISTS (
    subquery
);
```

- The **subquery** can reference the outer query (correlated subquery).
- Returns rows from the main query only if the subquery returns at least one row.

---

2. Examples

a) EXISTS with SELECT

Find customers who have bought cars costing more than 2,000,000:

```
SELECT * FROM CUSTOMERS
WHERE EXISTS (
    SELECT PRICE FROM CARS
    WHERE CARS.ID = CUSTOMERS.ID AND PRICE > 2000000
);
```

Result:
Only customers who satisfy the condition are returned.

---

b) EXISTS with UPDATE

Update customers who have a car:

```
UPDATE CUSTOMERS
SET NAME = 'Kushal'
WHERE EXISTS (
    SELECT 1 FROM CARS WHERE CARS.ID = CUSTOMERS.ID
);
```

- Updates **only those customers** who exist in the CARS table.

---

c) EXISTS with DELETE

Delete customers who bought a car priced 2,250,000:

```
DELETE FROM CUSTOMERS
WHERE EXISTS (
    SELECT * FROM CARS
    WHERE CARS.ID = CUSTOMERS.ID AND PRICE = 2250000
);
```

---

d) EXISTS with INSERT

Insert a car only if the customer exists:

```
INSERT INTO CARS (ID, NAME, PRICE)
SELECT 5, 'Honda Civic', 1800000
WHERE EXISTS (
    SELECT 1 FROM CUSTOMERS WHERE ID = 5
);
```

---

3. Correlated Subqueries with EXISTS

A **correlated subquery** references the outer query and is evaluated for **each row**:

```
SELECT * FROM CUSTOMERS C
WHERE EXISTS (
```

```
    SELECT 1 FROM CARS CA
    WHERE CA.ID = C.ID AND CA.PRICE > C.SALARY
);
```

- Finds all customers who have bought cars costing more than their salary.

---

4. Nested EXISTS Queries

EXISTS queries can be **nested** to check multiple conditions:

```
SELECT * FROM CUSTOMERS C
WHERE EXISTS (
    SELECT 1 FROM CARS CA
    WHERE CA.ID = C.ID AND EXISTS (
        SELECT 1 FROM DEALERS D
        WHERE D.CUSTOMER_ID = C.ID AND D.DEALER_NAME = 'AutoHub'
    )
);
```

- Checks if the customer bought a car **and** through a dealer named `'AutoHub'`.

---

5. NOT EXISTS Operator

Select records that **do not have matching rows** in another table:

```
SELECT * FROM CUSTOMERS
WHERE NOT EXISTS (
    SELECT * FROM CARS
    WHERE CARS.ID = CUSTOMERS.ID
);
```

- Returns customers who **have not bought any cars**.

---

6. Differences Between EXISTS and IN

Aspect	EXISTS	IN
Purpose	Checks if a subquery returns **any rows**	Checks if a value matches **any value** in a list/subquery
Returns	TRUE/FALSE	Matches values

Aspect	EXISTS	IN
Performance	Faster with correlated subqueries, stops at first match	Can be slower with large lists
Subquery	Usually **correlated**	Non-correlated or fixed list
NULL Handling	Ignores NULLs	NULLs can affect results
Use Cases	Existence checks, conditional updates/deletes	Fixed lists, simple comparisons

SQL - CASE Statement

SQL CASE Statement

The `CASE` statement allows **conditional logic in SQL queries**, similar to an IF-THEN-ELSE construct.
It **returns a value** based on conditions you define.

- Can be used in **SELECT, UPDATE, DELETE, INSERT, ORDER BY, GROUP BY**.
- Stops checking once a condition is true.
- If no condition matches:
  - Returns **ELSE value** if provided.
  - Returns **NULL** if no ELSE clause exists.

---

1. Syntax

```
CASE
   WHEN condition1 THEN result1
   WHEN condition2 THEN result2
   ...
   ELSE resultN
END
```

- `condition1, condition2...` → Boolean expressions.
- `result1, result2...` → Values returned for the matching condition.

---

2. Types of CASE

a) Simple CASE

- Compares **one expression** to multiple possible values.
- Stops at the first match.

```
SELECT NAME, AGE,
CASE AGE
   WHEN 22 THEN 'Trainee'
   WHEN 23 THEN 'Junior Engineer'
   WHEN 25 THEN 'Software Engineer'
   ELSE 'Senior Staff'
END AS Title
FROM CUSTOMERS;
```

---

b) Searched CASE

- Each WHEN uses its **own Boolean condition**.
- Can use operators like >, <, `BETWEEN`, `LIKE`.

```
SELECT NAME, SALARY,
CASE
   WHEN SALARY < 3000 THEN 'Low Income'
   WHEN SALARY BETWEEN 3000 AND 7000 THEN 'Middle Income'
   WHEN SALARY > 7000 THEN 'High Income'
END AS Income_Level
FROM CUSTOMERS;
```

---

c) Nested CASE

- A CASE statement **inside another CASE**.
- Useful for **multi-level conditional logic**.

```
SELECT NAME, SALARY,
CASE
   WHEN SALARY < 5000 THEN
      CASE
         WHEN SALARY < 3000 THEN 'Bonus 20%'
         ELSE 'Bonus 10%'
      END
   ELSE 'No Bonus'
END AS Bonus_Plan
FROM CUSTOMERS;
```

---

3. CASE in Different SQL Clauses

a) SELECT

- Used to create **custom columns or formatted output**.

```
SELECT NAME, AGE,
CASE
   WHEN AGE < 25 THEN 'Youth'
   WHEN AGE BETWEEN 25 AND 30 THEN 'Adult'
   ELSE 'Senior'
END AS Age_Group
FROM CUSTOMERS;
```

---

b) ORDER BY

- Sort rows **conditionally**.

```
SELECT * FROM CUSTOMERS
ORDER BY
CASE
    WHEN NAME LIKE 'K%' THEN NAME
    ELSE ADDRESS
END;
```

---

c) GROUP BY

- Group rows into **custom categories**.

```
SELECT
   CASE
      WHEN SALARY <= 4000 THEN 'Lowest paid'
      WHEN SALARY > 4000 AND SALARY <= 6500 THEN 'Average paid'
      ELSE 'Highest paid'
   END AS SALARY_STATUS,
   SUM(SALARY) AS Total
FROM CUSTOMERS
GROUP BY
   CASE
      WHEN SALARY <= 4000 THEN 'Lowest paid'
      WHEN SALARY > 4000 AND SALARY <= 6500 THEN 'Average paid'
      ELSE 'Highest paid'
   END;
```

---

d) WHERE

- Can use CASE with logical operators for **conditional filtering**.

```
SELECT NAME, ADDRESS,
CASE
   WHEN AGE < 25 THEN 'Intern'
   WHEN AGE BETWEEN 25 AND 27 THEN 'Associate Engineer'
   ELSE 'Senior Developer'
END AS Designation
FROM CUSTOMERS
WHERE SALARY >= 2000;
```

---

e) UPDATE

- Update column values **conditionally**.

```
UPDATE CUSTOMERS
SET SALARY =
CASE AGE
   WHEN 25 THEN 17000
   WHEN 32 THEN 25000
   ELSE 12000
END;
```

---

f) INSERT

- Assign values conditionally during **row insertion**.

```
INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (10, 'Viren', 28, 'Varanasi',
   CASE
      WHEN AGE >= 25 THEN 23000
      ELSE 14000
   END
);
```

Differences Between CASE and IF-ELSE

Aspect	CASE Statement	IF-ELSE
Usage	Inline in **SELECT, UPDATE, ORDER BY, GROUP BY**	Procedural SQL blocks or stored programs
Output	Returns a **value** for each row	Executes procedural logic, no direct value in SELECT
Conditions	Multiple conditions in **compact syntax**	Requires separate control flow statements
Portability	Most SQL dialects support	Syntax varies by RDBMS

SQL - NOT Operator

1. NOT Operator

- **Purpose:** Negates a condition; returns rows where the condition is false.
- **Syntax:**

```
SELECT columns
FROM table
WHERE NOT condition;
```

- **Common uses:**
  - `NOT IN` → exclude values from a list
  - `NOT LIKE` → exclude patterns
  - `NOT BETWEEN` → exclude a range
  - `NOT EXISTS` → exclude rows in a subquery
  - `IS NOT NULL` → filter out NULL values

Examples:

```
-- Exclude salaries > 2000
SELECT * FROM CUSTOMERS WHERE NOT (SALARY > 2000);

-- Names not starting with 'K'
SELECT * FROM CUSTOMERS WHERE NAME NOT LIKE 'K%';

-- Age not in list
SELECT * FROM CUSTOMERS WHERE AGE NOT IN (25, 26, 32);

-- Customers with no orders
SELECT * FROM CUSTOMERS
WHERE NOT EXISTS (
    SELECT CUSTOMER_ID FROM ORDERS
    WHERE ORDERS.CUSTOMER_ID = CUSTOMERS.ID
);
```

---

2. NOT EQUAL Operator

- **Purpose:** Returns TRUE if two values are not equal.
- **Syntax:**

```
-- ANSI standard
column_name <> value

-- Alternative
column_name != value
```

- **Examples:**

```
-- Name is not 'Ramesh'
SELECT * FROM CUSTOMERS WHERE NAME <> 'Ramesh';

-- Salary ≥ 2000 and not from 'Bhopal'
SELECT * FROM CUSTOMERS
WHERE ADDRESS <> 'Bhopal' AND SALARY >= 2000;

-- Negate NOT EQUAL to get equality
SELECT * FROM CUSTOMERS WHERE NOT SALARY != 2000;
```

---

3. IS NULL Operator

- **Purpose:** Check for missing/unknown data.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column IS NULL;
```

- **Examples:**

```
-- Find rows with missing address
SELECT * FROM CUSTOMERS WHERE ADDRESS IS NULL;

-- Count rows with missing salary
SELECT COUNT(*) FROM CUSTOMERS WHERE SALARY IS NULL;

-- Update missing ages
UPDATE CUSTOMERS SET AGE = 48 WHERE AGE IS NULL;

-- Delete rows with missing salary
DELETE FROM CUSTOMERS WHERE SALARY IS NULL;
```

- **Important:** `NULL` ≠ `0` and `NULL` ≠ `''`.

---

4. IS NOT NULL Operator

- **Purpose:** Filter rows where a column has actual (non-NULL) data.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column IS NOT NULL;
```

- **Examples:**

```
-- Retrieve rows with an address
SELECT * FROM CUSTOMERS WHERE ADDRESS IS NOT NULL;

-- Count rows with salary
SELECT COUNT(*) FROM CUSTOMERS WHERE SALARY IS NOT NULL;

-- Increase salaries that exist
UPDATE CUSTOMERS SET SALARY = SALARY + 5000 WHERE SALARY IS NOT NULL;

-- Delete rows with salary
DELETE FROM CUSTOMERS WHERE SALARY IS NOT NULL;

-- Join only where amount exists
SELECT c.NAME, o.AMOUNT
FROM CUSTOMERS c
JOIN ORDERS o ON c.ID = o.CUSTOMER_ID
WHERE o.AMOUNT IS NOT NULL;
```

SQL - NOT NULL Constraint

1. NOT NULL

- **Purpose:** Ensures a column **cannot have NULL values**.
- **Behavior:** Trying to insert/update a NULL into a NOT NULL column → **error**.

CREATE TABLE with NOT NULL:

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

- ID, NAME, AGE → cannot be NULL
- ADDRESS, SALARY → can be NULL

ALTER TABLE examples:

```
-- Add NOT NULL
ALTER TABLE CUSTOMERS MODIFY COLUMN ADDRESS CHAR(25) NOT NULL;

-- Remove NOT NULL
ALTER TABLE CUSTOMERS MODIFY COLUMN NAME VARCHAR(20) NULL;
```

---

2. BETWEEN / NOT BETWEEN

- **BETWEEN:** Selects rows within a range (inclusive).
- **NOT BETWEEN:** Selects rows outside a range.

Syntax:

```
SELECT columns FROM table
WHERE column BETWEEN value1 AND value2;

SELECT columns FROM table
WHERE column NOT BETWEEN value1 AND value2;
```

Examples:

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

-- Text
```

```
SELECT * FROM CUSTOMERS WHERE NAME BETWEEN 'A' AND 'L';

-- Update within range
UPDATE CUSTOMERS SET SALARY = 10000 WHERE AGE BETWEEN 25 AND 30;

-- Delete outside range
DELETE FROM CUSTOMERS WHERE AGE NOT BETWEEN 20 AND 24;
```

---

3. UNION / UNION ALL

- **UNION:** Combines results of multiple queries → **removes duplicates**
- **UNION ALL:** Combines results → **keeps duplicates**, faster

Requirements:

- Same number of columns
- Compatible data types
- Same order of columns

Examples:

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

-- UNION ALL
SELECT SALARY FROM CUSTOMERS
UNION ALL
SELECT AMOUNT FROM ORDERS;
```

Key difference:

Feature	UNION	UNION ALL
Duplicates	Removed	Kept
Performance	Slower	Faster
Result Size	Unique rows	All rows

---

4. INTERSECT

- Returns **common rows** from two SELECT queries.
- Requirements: same number of columns, same order, compatible data types.
- Not supported in MySQL (use JOIN + DISTINCT instead).

Example:

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

With BETWEEN or IN:

```
SELECT NAME, AGE, HOBBY FROM STUDENTS
WHERE AGE BETWEEN 20 AND 30
INTERSECT
SELECT NAME, AGE, HOBBY FROM STUDENTS_HOBBY
WHERE AGE BETWEEN 25 AND 30;
```

---

5. EXCEPT

- Returns **rows from the first query that do NOT exist in the second query**.
- Removes duplicates automatically.
- Not supported in MySQL (use LEFT JOIN + DISTINCT instead).

Example:

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

With BETWEEN / IN / LIKE:

```
SELECT NAME, HOBBY, AGE FROM STUDENTS
WHERE AGE BETWEEN 20 AND 30
EXCEPT
SELECT NAME, HOBBY, AGE FROM STUDENTS_HOBBY
WHERE AGE BETWEEN 20 AND 30;
```

---

6. Aliases (Column & Table)

- **Column alias:** Temporary name for output column.
- **Table alias:** Short name for table (useful in joins, self-joins).
- **AS** keyword is optional.

Examples:

```
-- Column alias
SELECT NAME AS CUSTOMER_NAME, SALARY AS MONTHLY_INCOME FROM CUSTOMERS;

-- Table alias
SELECT C.ID, C.NAME, O.AMOUNT
FROM CUSTOMERS AS C
JOIN ORDERS AS O
ON C.ID = O.CUSTOMER_ID;

-- Self join with aliases
```

```
SELECT a.ID, b.NAME AS EARNS_HIGHER, a.NAME AS EARNS_LESS, a.SALARY AS
LOWER_SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;

-- Concatenated column alias
SELECT CONCAT(NAME, ' - ', ADDRESS) AS CUSTOMER_DETAILS FROM CUSTOMERS;
```

SQL Joins

SQL - JOINS

What is a JOIN?

A SQL JOIN is used to combine rows from two or more tables based on a related/common column.
It helps us fetch data from multiple tables as if they were one table.

Basic Syntax

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

- **table1, table2** → tables you are joining
- **column** → common column used for matching

Types of SQL JOINs

1. INNER JOIN

- Returns **only matching rows** from both tables.
- No match = row not included.

Syntax

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

---

2. LEFT JOIN

- Returns **all rows from the left table**, and matching rows from the right.
- No match → right table values become **NULL**.

Syntax

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

---

3. RIGHT JOIN

- Returns **all rows from the right table**, and matching rows from the left.
- No match → left table values become **NULL**.

Syntax

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

---

4. FULL JOIN

- Returns **all rows from both tables**.
- No match → missing side shows **NULL**.
- *Not supported in MySQL directly*.

Syntax

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.col = table2.col;
```

---

5. SELF JOIN

- A table joined **with itself**.
- Useful for comparing rows in the same table.

Syntax

```
SELECT A.col, B.col
FROM table A, table B
WHERE condition;
```

---

6. CROSS JOIN

- Returns **Cartesian product** of two tables.
- Every row of table1 combines with every row of table2.
- If table1 has *n* rows and table2 has *m* rows → result has **n × m rows**.

Syntax

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

Joining Multiple Tables

You can join more than two tables by using multiple JOIN statements.

Syntax

```
SELECT columns
FROM table1
JOIN table2 ON table1.col = table2.col
JOIN table3 ON table2.col = table3.col;
```

SQL - Inner Join

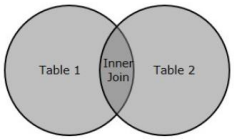
1. What is a SQL Join?

- A SQL Join is used to **combine two or more related tables** using a **common column**.
- Types of joins:
  - **Inner Join**
  - **Outer Join**
    - Left Join
    - Right Join
    - Full Join
      (Here we focus only on Inner Join.)

---

2. Inner Join – Definition

- Returns **only the rows that match** in both tables using a common column.
- Compares rows from both tables and combines only the matched ones.
- It is the **default join** (even if you write "JOIN", it acts as Inner Join).
- Also called **Equi Join**.



---

3. Example Understanding

EmpDetails

ID	Name	Salary
1	John	40000
2	Alex	25000
3	Simon	43000

MaritalStatus

ID	Name	Status
1	John	Married
3	Simon	Married
4	Stella	Unmarried

ID	Name	Salary	Status
1	John	40000	Married
3	Simon	43000	Married

Join Condition

```
EmpDetails.EmpID = MaritalStatus.EmpID
```

Result Contains

Only matched records → ID, Name, Salary, Age, Marital Status

---

4. Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

---

5. Example (CUSTOMERS + ORDERS)

CUSTOMERS table

- ID, Name, Age, Address, Salary

ORDERS table

- OID, Date, Customer_ID, Amount

Inner Join Query

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

Output (only matching IDs)

- Kaushik – 3000
- Kaushik – 1500
- Khilan – 1560
- Chaitali – 2060

---

6. Joining Multiple Tables

Syntax:

```
SELECT column1, column2...
FROM table1
INNER JOIN table2 ON condition1
INNER JOIN table3 ON condition2
...
```

Example (CUSTOMERS + ORDERS + EMPLOYEE)

Join conditions:

- `CUSTOMERS.ID = ORDERS.CUSTOMER_ID`
- `ORDERS.OID = EMPLOYEE.EID`

Result: OID, Date, Amount, Employee_Name

---

7. Inner Join with WHERE Clause

Purpose: Apply additional filters.

Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

Example (Amount > 2000)

```
SELECT ID, NAME, DATE, AMOUNT
```

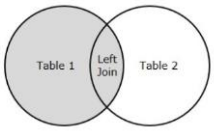
```
FROM CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
WHERE AMOUNT > 2000;
```

Output: Only rows where amount > 2000.

SQL - Left Join

1. What is a LEFT JOIN?

- LEFT JOIN (or LEFT OUTER JOIN) returns:
  - **All rows from the left table**
  - **Only matching rows from the right table**
- If there is **no match** in the right table → right table columns become **NULL**.



---

2. LEFT JOIN – Simple Meaning

- LEFT table = Always fully included
- RIGHT table = Only matched data is included
- Unmatched right-table rows → **NULL**

---

3. LEFT JOIN Venn Diagram Concept

- Shows the **left table completely**
- Only the overlapping/matching part from the right table is added
- Right-table rows without matches are **not shown**

---

4. Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

---

5. Example Tables

CUSTOMERS Table

ID, Name, Age, Address, Salary

ORDERS Table

OID, Date, Customer_ID, Amount

Query:

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

Output Meaning:

- Customers **with orders** → show order details
- Customers **without orders** → show NULL for Amount, Date

Example:

- Ramesh → NULL, NULL
- Kaushik → 1500, 3000
- Hardik → NULL

---

6. LEFT JOIN with Multiple Tables

You can chain multiple LEFT JOINs like this:

```
SELECT C.ID, C.NAME, O.DATE, E.EMPLOYEE_NAME
FROM CUSTOMERS C
LEFT JOIN ORDERS O ON C.ID = O.CUSTOMER_ID
LEFT JOIN EMPLOYEE E ON O.OID = E.EID;
```

Output Meaning:

- All customers shown
- If a customer has no order → Date = NULL, Employee_Name = NULL
- If order has no employee → Employee_Name = NULL

---

7. LEFT JOIN with WHERE Clause

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.col = table2.col
WHERE condition;
```

Warning:

- If the WHERE clause filters right-table columns (example: `WHERE table2.column IS NOT NULL`),
  it **behaves like an INNER JOIN**.

Example (Amount > 2000):

```
SELECT ID, NAME, DATE, AMOUNT
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
WHERE AMOUNT > 2000;
```

Result only shows rows where Amount > 2000.

---

8. Using Aliases with LEFT JOIN

Syntax:

```
SELECT c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS AS c
LEFT JOIN ORDERS AS o
ON c.ID = o.CUSTOMER_ID;
```

- `c` → alias for CUSTOMERS
- `o` → alias for ORDERS
- Makes long queries clean and easy to read

---

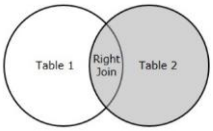
9. Important Points About LEFT JOIN

Returns **ALL rows from left table**
Unmatched right-table values become **NULL**
You can join **multiple tables**
Aliases make queries cleaner
WHERE clause on right table may behave like INNER JOIN
Order matters → first table is always **left table**
LEFT JOIN keeps **unmatched left rows** (INNER JOIN does not)

SQL - Right Join

1. What is a RIGHT JOIN?

- RIGHT JOIN (or RIGHT OUTER JOIN) returns:
  - **All rows from the RIGHT table**
  - **Matching rows from the LEFT table**
- If no match is found → **LEFT table columns become NULL**.



---

2. RIGHT JOIN – Simple Meaning

Right table = always fully included
Left table = included only if matching data exists
Unmatched left-table rows → **NULL values**

---

3. RIGHT JOIN Venn Diagram Concept

- Shows **all rows from the right table**
- Adds only matching data from the left table
- Left-table rows without matches are not shown

---

4. Syntax

```
SELECT left_table.column1, right_table.column2
FROM left_table
RIGHT JOIN right_table
ON left_table.common_column = right_table.common_column;
```

Even if a right-table row has **zero matches**, it still appears with **NULLs** for the left-table columns.

---

5. Example (CUSTOMERS + ORDERS)

Query:

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

Output Meaning:

- Orders table = **fully included**
- Only customers who have matching order records appear
- Example output:
  - Kaushik → 3000, 1500
  - Khilan → 1560
  - Chaitali → 2060

(No customer without orders appears.)

---

6. RIGHT JOIN with Multiple Tables

You can chain multiple RIGHT JOINs:

```
SELECT C.ID, C.NAME, O.DATE, E.EMPLOYEE_NAME
FROM CUSTOMERS C
RIGHT JOIN ORDERS O ON C.ID = O.CUSTOMER_ID
RIGHT JOIN EMPLOYEE E ON O.OID = E.EID;
```

Output Meaning:

- EMPLOYEE table (rightmost table) → always fully included
- ORDERS matched with EMPLOYEE
- CUSTOMERS matched with ORDERS
- If a match fails → left table fields become **NULL**

---

7. RIGHT JOIN with WHERE Clause

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.col = table2.col
WHERE condition;
```

Warning:

- If you put conditions on left-table columns (e.g., `table1.col > 0`), rows with NULL values get removed → behaves like **INNER JOIN**.

Example:

```
SELECT ID, NAME, DATE, AMOUNT
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
WHERE AMOUNT > 1000;
```

Result → Only orders with amount > 1000.

---

8. Using Aliases with RIGHT JOIN

Syntax:

```
SELECT t1.col, t2.col
FROM table1 AS t1
RIGHT JOIN table2 AS t2
ON t1.col = t2.col;
```

Example:

```
SELECT c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS AS c
RIGHT JOIN ORDERS AS o
ON c.ID = o.CUSTOMER_ID;
```

Makes queries easier to read.

SQL - Cross Join

1. What is a CROSS JOIN?

A **CROSS JOIN** returns the **Cartesian product** of two tables.
This means:

➞ Every row in **Table 1** is combined with **every row in Table 2**.
➞ It produces **all possible combinations** of rows.

Mathematically:
If Table1 has *m* rows and Table2 has *n* rows → Result has **m × n rows**.

---

2. Simple Meaning

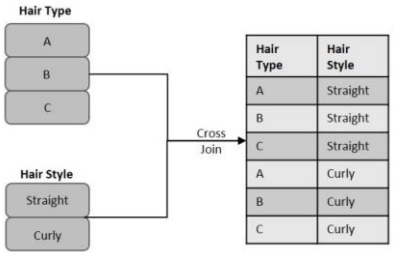
- No matching condition is needed.
- It just pairs each row from table A with every row from table B.
- Output grows fast when tables get large.

---

3. Venn Diagram Concept

The diagram shows **all possible combinations** of values from two sets.
Example:

- Hair Style × Hair Type → all possible hairstyle–hair type pairs.



---

4. Syntax

```
SELECT table1.column1, table2.column2
FROM table1
CROSS JOIN table2;
```

No **ON** condition is required.

---

5. Example (CUSTOMERS × ORDERS)

Query:

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
CROSS JOIN ORDERS;
```

What happens?

- Each customer is combined with each order.
- Result = 2 customers × 2 orders = **4 rows**

---

6. Joining Multiple Tables with CROSS JOIN

You can join 3 or more tables:

```
SELECT columns
FROM table1
CROSS JOIN table2
CROSS JOIN table3
...
```

⚠ **Warning:**
The output size becomes:
rows(table1) × rows(table2) × rows(table3) × …
This can become **very large**.

Example with 3 tables (CUSTOMERS × ORDERS × ORDER_RANGE)

```
SELECT ID, NAME, AMOUNT, DATE, ORDER_RANGE
FROM CUSTOMERS
CROSS JOIN ORDERS
CROSS JOIN ORDER_RANGE;
```

If table sizes are:

- CUSTOMERS = 2 rows
- ORDERS = 2 rows
- ORDER_RANGE = 3 rows

Total rows = **2 × 2 × 3 = 12 rows**

---

7. CROSS JOIN with Inline Tables

You can CROSS JOIN temporary/inline tables inside a query:

```
SELECT A.Name, B.Product
FROM (SELECT 'Ramesh' AS Name UNION ALL SELECT 'Khilan') AS A
CROSS JOIN (SELECT 'Laptop' AS Product UNION ALL SELECT 'Mobile') AS B;
```

This produces:

Name	Product
Khilan	Laptop
Ramesh	Laptop

Name	Product
Khilan	Mobile
Ramesh	Mobile

SQL - Full Join

1. What is a FULL JOIN?

A **FULL JOIN** (or FULL OUTER JOIN) returns:

All rows from both tables
Matching rows where available
NULL for missing matches on either side

It is basically:
LEFT JOIN + RIGHT JOIN combined together

---

2. Support in Databases

- Some SQL databases (e.g., SQL Server, PostgreSQL) support **FULL JOIN** directly.
- **MySQL does NOT support FULL JOIN**.
  In MySQL, you must simulate it using:

```
SELECT ... FROM table1 LEFT JOIN table2 ...
UNION
SELECT ... FROM table1 RIGHT JOIN table2 ...
```

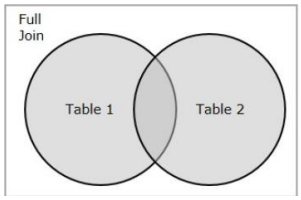
---

3. Venn Diagram Concept

FULL JOIN returns:

- All rows that match
- All rows that only exist in the left table
- All rows that only exist in the right table

Anything missing on one side is filled with **NULL**.



---

4. Syntax

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

---

5. Example (CUSTOMERS + ORDERS)

Query:

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

Explanation:

- Customers with orders → matched rows
- Customers without orders → NULL for order columns
- Orders without matching customers → NULL for customer columns

Result example:

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20
3	Kaushik	3000	2009-10-08
3	Kaushik	1500	2009-10-08
4	Chaitali	2060	2008-05-20
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

---

6. Joining Multiple Tables with FULL JOIN

You can chain multiple FULL JOINs:

```
SELECT columns
FROM table1
FULL JOIN table2 ON condition_1
FULL JOIN table3 ON condition_2
...
```

In MySQL:

You must simulate each FULL JOIN using:

- LEFT JOIN
- UNION
- RIGHT JOIN

Example with 3 tables (CUSTOMERS, ORDERS, EMPLOYEE)

```
SELECT CUSTOMERS.ID, CUSTOMERS.NAME,
       ORDERS.DATE, EMPLOYEE.EMPLOYEE_NAME
FROM CUSTOMERS
FULL JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
FULL JOIN EMPLOYEE ON ORDERS.OID = EMPLOYEE.EID;
```

Meaning:

- All customers
- Matched orders where available
- Matched employee records where available
- Missing matches → NULL

Output sample:

ID	NAME	DATE	EMPLOYEE_NAME
1	Ramesh	NULL	NULL
2	Khilan	2009-11-20	REVATHI
3	Kaushik	2009-10-08	ALEKHYA
3	Kaushik	2009-10-08	SARIKA
4	Chaitali	2008-05-20	VIVEK
5	Hardik	NULL	NULL
6	Komal	NULL	NULL

ID	NAME	DATE	EMPLOYEE_NAME
7	Muffy	NULL	NULL

---

7. FULL JOIN with WHERE Clause

You can apply extra conditions after the join using **WHERE**.

Syntax:

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.col = table2.col
WHERE condition;
```

Example:

```
SELECT ID, NAME, DATE, AMOUNT
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
WHERE ORDERS.AMOUNT > 2000;
```

Result:

Only rows where **amount > 2000** appear:

ID	NAME	DATE	AMOUNT
3	Kaushik	2009-10-08	3000
4	Chaitali	2008-05-20	2060

SQL - Self Join

1. What is a Self Join?

A **SELF JOIN** is when a table is **joined with itself**.

You use it when:

- You want to **compare rows within the same table**, or
- You want to **find relationships stored in one table**, such as manager–employee structure.

Since we use the same table twice, we must use **aliases (A, B)** to differentiate the two copies of the table.

---

2. Syntax

```
SELECT A.column1, B.column2
FROM table_name A
JOIN table_name B
ON A.common_column = B.common_column;
```

Real-Life Example (Secret Santa)

Imagine employees get:

- A **color assigned**
- A **color picked**

To find who is the Secret Santa of whom, we match:
assigned_color = picked_color

This requires comparing rows inside the same table → **Self Join**.

---

3. Example Table (CUSTOMERS)

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

---

1. Self Join Example — Compare Salaries

Goal: List customers who earn **less** than others.

Query:

```
SELECT a.ID, b.NAME AS EARNS_HIGHER, a.NAME AS EARNS_LESS, a.SALARY AS
LOWER_SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;
```

Meaning:

- `a` = person earning less
- `b` = person earning more

This produces **all comparisons** where one person earns less than another.

---

5. Self Join with ORDER BY

You can sort results after joining.

Example (sort by lowest salary first):

```
SELECT a.ID, b.NAME AS EARNS_HIGHER, a.NAME AS EARNS_LESS, a.SALARY AS
LOWER_SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY
ORDER BY a.SALARY;
```

---

6. Self Join for Hierarchical Data (Manager–Employee)

Self Join is commonly used for **hierarchical data** stored in the same table.

Example: Add `MANAGER_ID` to the CUSTOMERS table.

Then you can find each employee's manager:

```
SELECT a.NAME AS EMPLOYEE, b.NAME AS MANAGER
FROM CUSTOMERS a
LEFT JOIN CUSTOMERS b
ON a.MANAGER_ID = b.ID;
```

Output:

EMPLOYEE	MANAGER
Ramesh	NULL
Khilan	Ramesh
Kaushik	Ramesh

EMPLOYEE	MANAGER
Chaitali	Khilan
Hardik	Khilan
Komal	Kaushik
Muffy	Chaitali

---

7. Self Join vs Subquery — What's the Difference?

✓ Self Join

Used to **compare rows directly** with each other.
Useful for:

- Salary comparisons
- Manager–employee relationships
- Row-to-row analysis

✓ Subquery

Used to **filter** or **aggregate** results.
It does not compare rows to each other.

Example: Using Self Join

Find all customers earning less than others:

```
SELECT a.NAME AS EARNS_LESS, a.SALARY, b.NAME AS EARNS_MORE
FROM CUSTOMERS a
JOIN CUSTOMERS b
ON a.SALARY < b.SALARY;
```

Using Subquery (no row comparison)

```
SELECT NAME AS EARNS_LESS, SALARY
FROM CUSTOMERS
WHERE SALARY < (SELECT MAX(SALARY) FROM CUSTOMERS);
```

Self Join → detailed comparisons
Subquery → filtered list only

SQL - DELETE JOIN

1. What is a DELETE JOIN?

A **DELETE JOIN** is used to **delete rows from a table based on a condition that involves another table**.

It combines the `DELETE` statement with a `JOIN`, so you can target rows that **match between two or more tables**.

✅ Useful when you want to delete **related records in one table** that depend on another table.

---

2. Syntax

Basic syntax:

```
DELETE table1
FROM table1
JOIN table2
ON table1.common_field = table2.common_field
WHERE table2.some_column = 'value';
```

- `JOIN` can be any type: `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, etc.
- Use **table aliases** for easier reference.

---

3. Example Tables

CUSTOMERS

```
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)
);

INSERT INTO CUSTOMERS VALUES
(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);
```

ORDERS

```
CREATE TABLE ORDERS (
   OID INT NOT NULL,
   DATE VARCHAR(20) NOT NULL,
```

```
   CUSTOMER_ID INT NOT NULL,
   AMOUNT DECIMAL(18,2)
);

INSERT INTO ORDERS VALUES
(102, '2009-10-08', 3, 3000.00),
(100, '2009-10-08', 3, 1500.00),
(101, '2009-11-20', 2, 1560.00),
(103, '2008-05-20', 4, 2060.00);
```

---

4. Delete Rows with DELETE JOIN

Delete all customers who have orders:

```
DELETE a
FROM CUSTOMERS AS a
INNER JOIN ORDERS AS b
ON a.ID = b.CUSTOMER_ID;
```

Result: Only customers **without orders** remain:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

✓ Orders table remains unchanged.

---

5. Delete with a WHERE Clause

Delete customers **with salary < 2000** who also have orders:

```
DELETE a
FROM CUSTOMERS AS a
INNER JOIN ORDERS AS b
ON a.ID = b.CUSTOMER_ID
WHERE a.SALARY < 2000.00;
```

After deletion, CUSTOMERS table:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
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

ORDERS table remains the same.

SQL - UPDATE JOIN Statement

1. What is an UPDATE JOIN?

An **UPDATE JOIN** is used to **update records in one table based on matching values from another table**.

It's useful when you want to **synchronize or adjust data across multiple related tables**.

---

2. Basic Syntax

```
UPDATE table1
JOIN table2 ON table1.common_field = table2.common_field
SET table1.column_to_update = table2.column_value
WHERE table2.some_column = 'value';
```

- `JOIN` can be any type: `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, etc.
- Use **table aliases** for convenience.

---

3. Example Tables

CUSTOMERS

```
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)
);

INSERT INTO CUSTOMERS VALUES
(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);
```

ORDERS

```
CREATE TABLE ORDERS (
   OID INT NOT NULL,
   DATE VARCHAR(20) NOT NULL,
   CUSTOMER_ID INT NOT NULL,
   AMOUNT DECIMAL(18,2)
);

INSERT INTO ORDERS VALUES
(102, '2009-10-08', 3, 3000.00),
(100, '2009-10-08', 3, 1500.00),
(101, '2009-11-20', 2, 1560.00),
(103, '2008-05-20', 4, 2060.00);
```

---

4. UPDATE JOIN Example (MySQL)

Increase **customer salary by 1000** and **order amount by 500**:

```
UPDATE CUSTOMERS
JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
SET CUSTOMERS.SALARY = CUSTOMERS.SALARY + 1000,
    ORDERS.AMOUNT = ORDERS.AMOUNT + 500;
```

- Updates **both tables** in MySQL.
- Verification:

```
SELECT * FROM CUSTOMERS;
SELECT * FROM ORDERS;
```

---

5. UPDATE JOIN with WHERE Clause

Update **only specific rows**:

```
UPDATE CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
SET CUSTOMERS.SALARY = CUSTOMERS.SALARY + 1000
WHERE ORDERS.CUSTOMER_ID = 3;
```

- Only customer with `ID = 3` is updated.
- Use **WHERE clause** to control which rows are modified.

---

6. SQL Server Syntax

In SQL Server, the `SET` clause comes **before the JOIN**:

```
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
FROM CUSTOMERS
JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

---

7. LEFT JOIN vs INNER JOIN

LEFT JOIN

- Updates all rows from the left table, even if no match in the right table.
- Example: Increase salary **only if an order exists**:

```
UPDATE CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
SET CUSTOMERS.SALARY = CUSTOMERS.SALARY + 500
WHERE ORDERS.CUSTOMER_ID IS NOT NULL;
```

INNER JOIN

- Updates **only matching rows** in both tables.
- Example: Increase salary for orders > 2000:

```
UPDATE CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
SET CUSTOMERS.SALARY = CUSTOMERS.SALARY + 1000
WHERE ORDERS.AMOUNT > 2000;
```

---

8. Important Points

1. **Test with SELECT first** to avoid accidental updates:

```
SELECT *
FROM CUSTOMERS
JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

2. Without a **WHERE clause**, all matching rows may be updated.
3. **MySQL** can update columns in both tables; **SQL Server** usually updates one table at a time.

4. **INNER JOIN** updates only matching rows; **LEFT JOIN** can include non-matching rows.
5. Ensure **indexes** exist on join columns for better performance.

Difference Between JOIN and UNION in SQL

JOIN vs UNION in SQL

Both **JOIN** and **UNION** combine data from multiple tables, but **how they combine it is fundamentally different**.

---

1. JOIN in SQL

Definition: Combines columns from two or more tables based on a **related column**.

Orientation: Horizontal (adds columns).

Purpose: Retrieve related data from multiple tables in a single query.

Types of JOIN:

- **INNER JOIN:** Returns only matching rows.
- **LEFT JOIN:** Returns all rows from the left table; unmatched right rows appear as NULL.
- **RIGHT JOIN:** Returns all rows from the right table; unmatched left rows appear as NULL.
- **FULL OUTER JOIN:** Returns all rows from both tables; unmatched rows filled with NULL.
- **CROSS JOIN:** Cartesian product of the two tables.

Syntax:

```
SELECT table1.column1, table2.column2, ...
FROM table1
JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

```
SELECT c.STUDENT_ID, c.STUDENT_NAME, c.COURSE_NAME, e.EXTRA_COURSE_NAME
FROM COURSES_PICKED c
JOIN EXTRA_COURSES_PICKED e
ON c.STUDENT_ID = e.STUDENT_ID;
```

Result:

STUDENT_ID	STUDENT_NAME	COURSE_NAME	EXTRA_COURSE_NAME
1	JOHN	ENGLISH	PHYSICAL EDUCATION
2	ROBERT	COMPUTER SCIENCE	GYM
3	SASHA	COMMUNICATIONS	FILM
4	JULIAN	MATHEMATICS	PHOTOGRAPHY

---

2. UNION in SQL

Definition: Combines **rows** from two or more `SELECT` queries into a single result set.

Orientation: Vertical (adds rows).

Purpose: Merge results from multiple queries with the same number of columns and compatible data types.

Rules:

- Each `SELECT` must have the same number of columns.
- Column types must be compatible.
- `UNION` removes duplicates by default; `UNION ALL` keeps duplicates.
- Column names come from the **first SELECT**.

Syntax:

```
SELECT column1, column2 FROM table1
UNION
SELECT column1, column2 FROM table2;
```

Example:

```
SELECT STUDENT_ID, STUDENT_NAME, COURSE_NAME FROM COURSES_PICKED
UNION
SELECT STUDENT_ID, STUDENT_NAME, EXTRA_COURSE_NAME FROM
EXTRA_COURSES_PICKED;
```

Result:

STUDENT_ID	STUDENT_NAME	COURSE_NAME
1	JOHN	ENGLISH
2	ROBERT	COMPUTER SCIENCE
3	SASHA	COMMUNICATIONS
4	JULIAN	MATHEMATICS
1	JOHN	PHYSICAL EDUCATION
2	ROBERT	GYM
3	SASHA	FILM
4	JULIAN	PHOTOGRAPHY