INTRODUCTION TO DATABASES

Understanding the Basics of Data Storage

WHAT IS A DATABASE

- ➢A database is an organized collection of data.
- ➢It allows for easy access, management, and updating of data.

WHY USE A DATABASE?

- Stores data in a structured format
- Ensures data consistency and integrity
- Allows quick retrieval and manipulation of data
- Scalable for large volumes of data

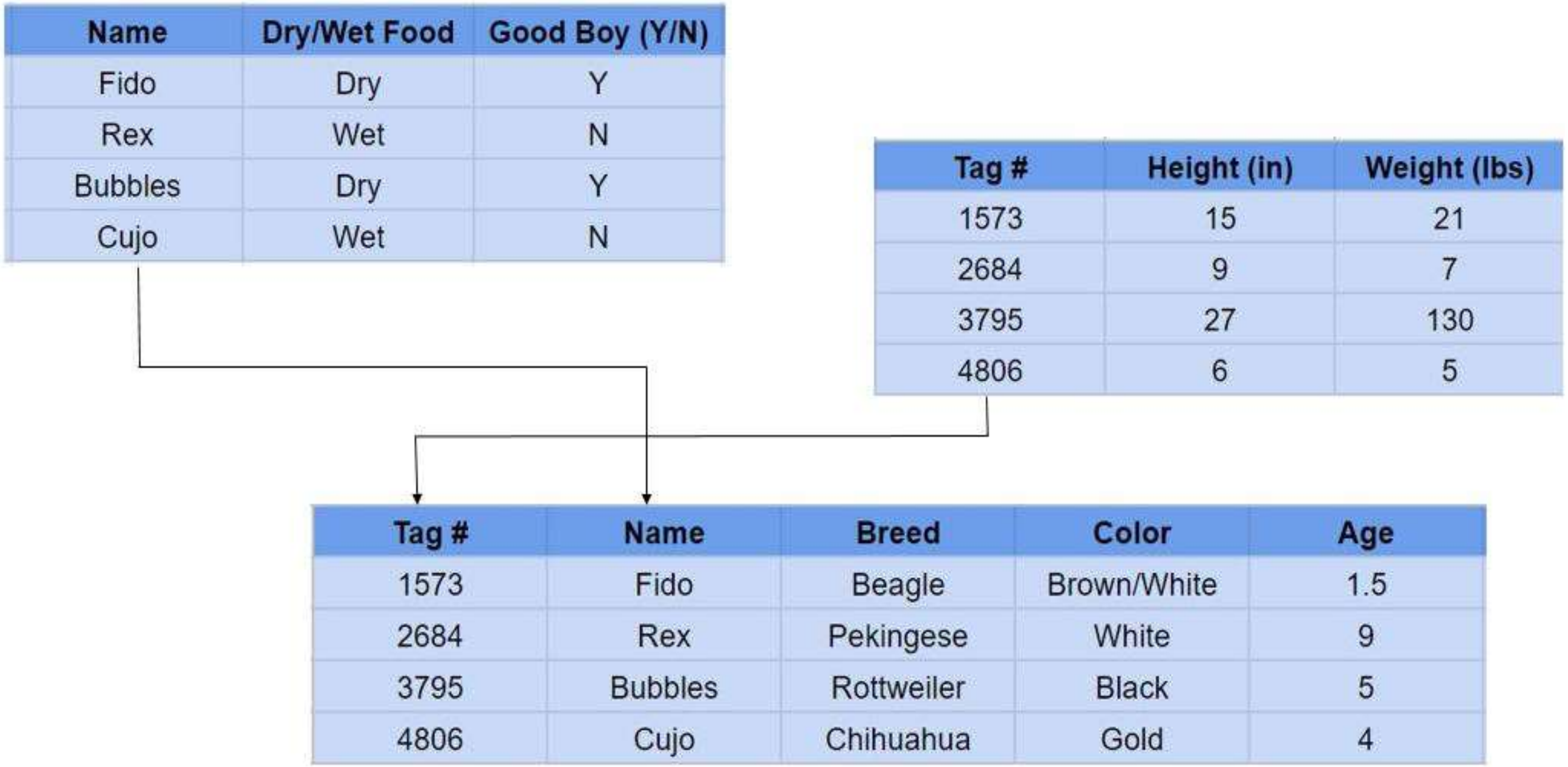
REAL-LIFE EXAMPLES

- Student records in a college
- Customer information in an online store
- Patient records in a hospital

TYPES OF DATABASES

- **Relational Database (RDBMS):** Data in tables (e.g., MySQL, PostgreSQL)
- **NoSQL Database:** Handles unstructured data (e.g., MongoDB)
- **Cloud Database:** Hosted on cloud platforms (e.g., Google Cloud SQL)

RELATIONAL DATABASE EXAMPLE



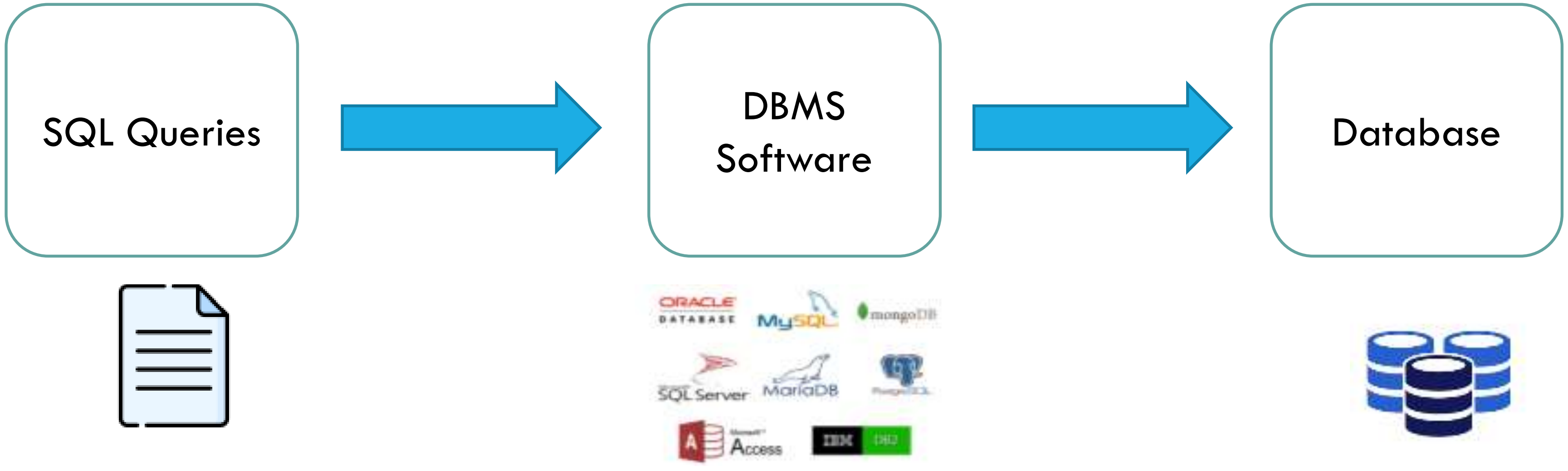
Name	Dry/Wet Food	Good Boy (Y/N)
Fido	Dry	Y
Rex	Wet	N
Bubbles	Dry	Y
Cujo	Wet	N

Tag #	Height (in)	Weight (lbs)
1573	15	21
2684	9	7
3795	27	130
4806	6	5

Tag #	Name	Breed	Color	Age
1573	Fido	Beagle	Brown/White	1.5
2684	Rex	Pekingese	White	9
3795	Bubbles	Rottweiler	Black	5
4806	Cujo	Chihuahua	Gold	4

DATABASE MANAGEMENT SYSTEM (DBMS)

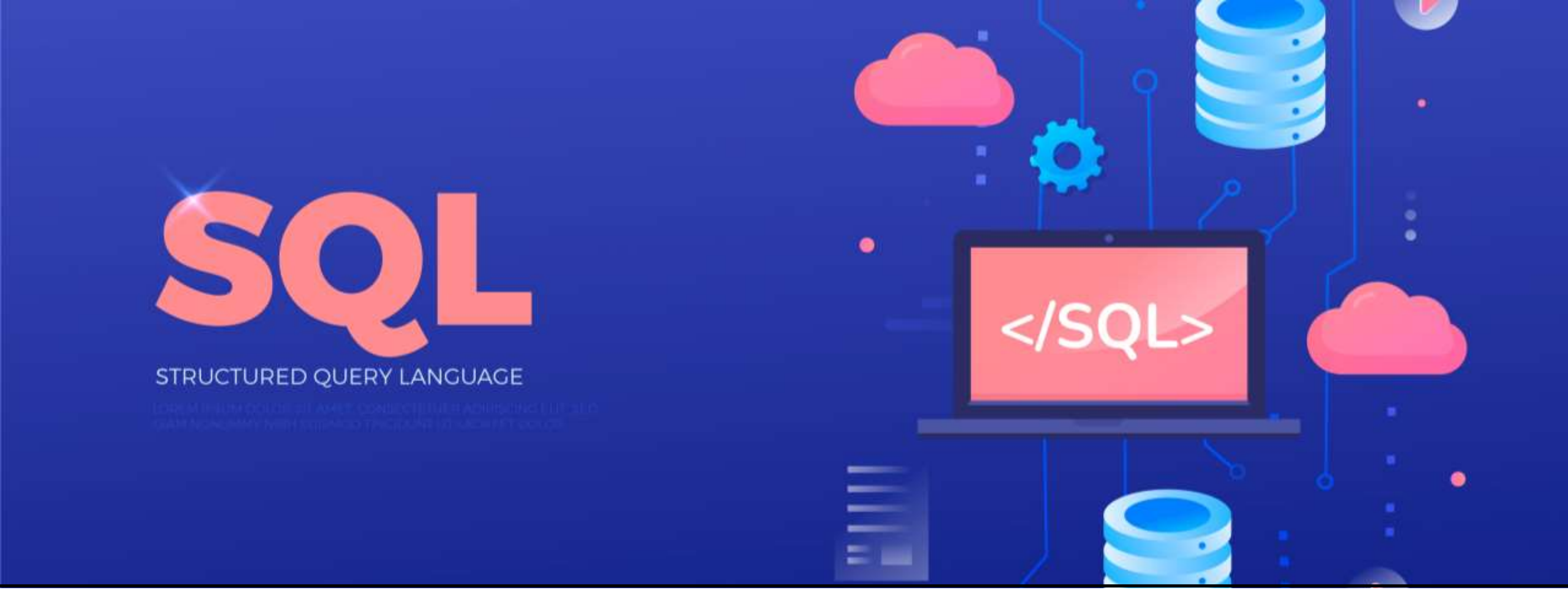
- Software to manage and interact with databases
- Examples: MySQL, SQL Server, Oracle, SQLite
- Handles queries, updates, and security



SQL Queries

DBMS Software

Database



INTRODUCTION TO SQL

WHAT IS SQL?

- **SQL** stands for **Structured Query Language.**
- It is a **standard language** used to communicate with **relational databases.**
- Developed by IBM in the 1970s and adopted as a **standard by ANSI and ISO**.
- SQL is a widely used relational database management system (RDBMS).
- SQL is free and open-source.
- SQL is ideal for both small and large applications.

WHY IS SQL IMPORTANT?

- **Universal Language:** SQL is used in almost every modern RDBMS (MySQL, PostgreSQL, etc.).
- **Data Management:** Essential for managing structured data in business, healthcare, education, etc.
- **Career Opportunities:** Widely used in data analysis, software development, and database admin jobs.
- **Foundation for Data Science & BI:** Key skill for working with tools like Power BI, Tableau, and Python.
- **Easy to Learn and Read:** SQL has a **declarative syntax** that's close to English, making it accessible for beginners.

TYPES OF SQL DATABASES:

Database	Description
MySQL	Open-source, widely used in web applications (WordPress, PHP-based websites)
PostgreSQL	Advanced open-source database with strong support for complex queries
SQLite	Lightweight, serverless database embedded in mobile and desktop applications
SQL Server	Microsoft's enterprise-grade RDBMS, integrated with .NET applications
Oracle DB	Powerful, secure database used in large-scale enterprise systems

SQL BASICS

SQL SYNTAX RULES

- SQL is **not case-sensitive** (but best practice is to use uppercase for commands)
- Each SQL statement ends with a **semicolon (;)**
- **Keywords** must be used correctly
- Strings are enclosed in **single quotes (' ')**
- Statements are usually written as:

```
CREATE database database_name;
```

DATABASES & TABLES

- **Database:** A collection of organized data stored electronically.
- **Table:** A structure inside a database that holds rows (records) and columns (fields).

```
-- Create a database
CREATE DATABASE SchoolDB;

-- Use a database
USE SchoolDB;

-- Create a table
CREATE TABLE Students (  StudentID INT
PRIMARY KEY, Name VARCHAR(50),
Age INT, Grade VARCHAR(5) );
```

CRUD OPERATIONS

Operation	SQL Command	Purpose
Create	INSERT	Add new rows
Read	SELECT	Retrieve data
Update	UPDATE	Modify existing rows
Delete	DELETE	Remove rows

WHAT ARE DATA TYPES IN SQL?

- **Data types** define the **kind of data** that can be stored in a column — like text, numbers, or dates.
- Each column in a table must be assigned a data type.

1. NUMERIC DATA TYPES

Data Type	Description	Example
INT	Integer number	100, -25
DECIMAL(p,s)	Fixed-point number (precision, scale)	10.99
FLOAT	Floating-point number (less precise)	3.14159
BIGINT	Very large integers	9000000000
SMALLINT	Small integer	100

2. STRING/TEXT DATA TYPES

Data Type	Description	Example
CHAR(n)	Fixed-length text	'Yes', 'No'
VARCHAR(n)	Variable-length text	'Hello world'
TEXT	Large text	paragraphs

3. DATE AND TIME DATA TYPES

Data Type	Description	Example
DATE	Stores date only	'2025-05-13'
TIME	Stores time only	'14:30:00'
DATETIME	Stores date and time	'2025-05-13 14:30:00'
TIMESTAMP	Stores UNIX timestamp	'2025-05-13 14:30:00'
YEAR	Stores a year	2025

4. BOOLEAN DATA TYPE

➢In MySQL, BOOLEAN is treated as TINYINT(1) internally.

Data Type	Description	Example
BOOLEAN	True or False	TRUE / FALSE

5. OTHER USEFUL DATA TYPES

Data Type	Description	Example
ENUM	Set of predefined string values	'Male', 'Female'
SET	One or more values from a list	'Math', 'Science'
BLOB	Binary data (images, files)	(used for media storage)

CONSTRAINTS IN SQL

Enforcing rules on data integrity and validation

WHAT ARE CONSTRAINTS?

- In SQL, **constraints** are **rules** you apply to columns in a table to **control what kind of data** can be inserted, updated, or stored.
- They help **protect the accuracy and integrity** of the data in your database.
- **Why Are Constraints Important?**
  - Prevent invalid or duplicate data
  - Ensure relationships between tables are valid
  - Automatically enforce business rules

TYPES OF CONSTRAINTS:

Constraint	Description
NOT NULL	Makes sure a column cannot be empty
UNIQUE	Ensures all values in a column are different
PRIMARY KEY	Uniquely identifies each row in a table (not null + unique)
FOREIGN KEY	Connects a column in one table to the primary key of another table
CHECK	Ensures that values meet a specific condition (e.g. age > 18)
DEFAULT	Sets a default value for a column if none is provided

1. NOT NULL & UNIQUE

NOT NULL:

- Ensures that a column **cannot have NULL values**

```
Name VARCHAR(50) NOT NULL
```

UNIQUE:

- Ensures that all values in a column are **different**

```
Email VARCHAR(100) UNIQUE
```

2. PRIMARY KEY

- Uniquely identifies each record in a table
- Cannot be NULL or duplicate

```
ID INT PRIMARY KEY
```

3. FOREIGN KEY

- Links records between tables
- Enforces **referential integrity**

```
StudentID INT, FOREIGN KEY (StudentID) REFERENCES Students(ID)
```

4. CHECK & DEFAULT

CHECK:

- Validates that data **meets a condition**

```
Age INT CHECK (Age >= 18)
```

DEFAULT:

- Sets a **default value** if none is provided

```
Status VARCHAR(10) DEFAULT 'Active'
```

COMMAND

Single Line Command

```
--single line command
```

Multi line Command

```
/*
multiline command
*/
```

OPERATORS IN SQL

COMPARISON OPERATORS

Operator	Description	Example
=	Equal to	Age = 18
>	Greater than	Age > 18
<	Less than	Age < 25
LIKE	Pattern matching	Name LIKE 'A%' (starts with A)
BETWEEN	Within a range	Age BETWEEN 18 AND 25
IN	Matches multiple	Age IN (18, 20, 22)

LOGICAL OPERATORS

- Used to combine multiple conditions in a WHERE clause.

Operator	Description	Example
AND	Both conditions must be true	Age > 18 AND Name = 'Anjali'
OR	At least one condition is true	Age < 18 OR Name = 'Ravi'
NOT	Negates a condition	NOT Age = 20

SQL STATEMENTS

SQL STATEMENTS CLASSIFICATION

➢ SQL statements are grouped into five major categories based on their function:

- DDL – Data Definition Language
- DML – Data Manipulation Language
- DCL – Data Control Language
- TCL – Transaction Control Language
- DQL – Data Query Language

1. DDL – DATA DEFINITION LANGUAGE

- Used to **define or modify** the structure of database objects.

Command	Purpose
CREATE	Creates tables, databases, etc.
ALTER	Modifies an existing table
DROP	Deletes a table or database

CREATING A DATABASE

- This command creates a new database named SchoolDB

```
CREATE DATABASE SchoolDB;
```

- Sets SchoolDB as the current working database

```
USE SchoolDB;
```

- This command delete a database SchoolDB

```
DROP DATABASE SchoolDB;
```

CREATING TABLES

- Defines a new table called Students with specified columns and data types

```
CREATE TABLE table_name(column1 datatype, column2 datatype,
column3 datatype ,….);
```

DESCRIBE OR DESC IN SQL

- Shows the structure/schema of a table.
- It tells you:
  - Column names
  - Data types
  - NULL allowed or not
  - Keys (like PRIMARY KEY)
- **Syntax:**

```
DESCRIBE table_name;
-- or
DESC table_name
```

- **Example:**

```
DESC employees;
```

DELETING TABLES

- Used to **permanently remove** a table and its data.
- Drop a Table:

```
DROP TABLE Students;
```

- **Warning:** This action cannot be undone

ALTERING TABLES

USED TO **ADD, MODIFY, OR DELETE COLUMNS** IN AN EXISTING TABLE.

- 1. **Add a New Column**

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

- 2. **Add Multiple Columns**

```sql
ALTER TABLE table_name ADD (col1 datatype, col2 datatype);
```

- 3. **Modify Column Type:**

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

- 4. Change Column Name and Type

```
ALTER TABLE table_name
CHANGE old_column_name new_column_name datatype;
```

- 5. Rename Table

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

- 6. Rename a Column (MySQL 8.0+)

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

- 7. Drop a Column

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

8. ADD A CONSTRAINT

- Add Primary Key:

```
ALTER TABLE table_name
ADD PRIMARY KEY (column_name);
```

- Add Unique Key:

```
ALTER TABLE table_name
ADD UNIQUE (column_name);
```

- Add Foreign Key:

```
ALTER TABLE second_tablename ADD CONSTRAINT fk_second_tablename
FOREIGN KEY (column_name) REFERENCES first_tablename(column_name);
```

9. DROP A CONSTRAINT

- **Drop Primary Key:**

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

- **Drop Foreign Key:**

```
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;
```

10. Set Default Value

```
ALTER TABLE table_name
ALTER column_name SET DEFAULT 'value';
```

11. Drop Default Value

```
ALTER TABLE table_name
ALTER column_name DROP DEFAULT;
```

12. Add AUTO_INCREMENT

```
ALTER TABLE table_name
MODIFY id INT AUTO_INCREMENT;
```

TRUNCATE IN SQL

- It deletes all rows from a table.
- Much faster than DELETE because it doesn’t log individual row deletions.
- Resets the table like it's brand new — but keeps the table structure.
- **Syntax:**

```
TRUNCATE TABLE table_name;
```

2. DML – DATA MANIPULATION LANGUAGE

- Used to **manipulate data** in existing tables.

Command	Purpose
SELECT	Retrieves data
INSERT	Adds new data
UPDATE	Modifies existing data
DELETE	Removes data from a table

INSERT INTO

- Adds a new row into the Students table
- Column names and values must match in order

```
INSERT INTO Students (StudentID, Name, Age, JoinDate)
VALUES (1, 'Anjali', 20, '2024-06-01');
```

SELECT STATEMENT

- Retrieves specified columns from a table
- Use SELECT * to retrieve all columns

```
SELECT Name, Age FROM Students;
```

UPDATE STATEMENT

- Used to **modify existing records** in a table.
- Always use WHERE to avoid updating all rows.
- **SET SQL_SAFE_UPDATES = 0;**

```
UPDATE Students
SET Age = 21
WHERE StudentID = 1;
```

DELETE STATEMENT

- Used to **remove records** from a table.
- Use WHERE to target specific rows.
- **SET SQL_SAFE_UPDATES = 0;**

```
DELETE FROM Students WHERE StudentID = 1;
```

3. DCL – DATA CONTROL LANGUAGE

- Used to **control access** to data.

Command	Purpose
GRANT	Gives user access privileges
REVOKE	Removes access privileges

4. TCL – TRANSACTION CONTROL LANGUAGE

- Used to manage **transactions** in a database.

Command	Purpose
COMMIT	Saves the transaction
ROLLBACK	Undoes changes made in a transaction

5. DQL – DATA QUERY LANGUAGE

- Used to **query and retrieve data.**

Command	Purpose
SELECT	Extracts data from a database

5.1. SELECT, FROM, WHERE

- **Syntax:**

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

- **Example:**

```
SELECT Name, Age FROM Students WHERE Age > 18;
```

- SELECT – defines which columns to retrieve.
- FROM – specifies the table.
- WHERE – filters rows based on conditions.

5.2. DISTINCT KEYWORD

- Used to return only **unique values,** removing duplicates.
- **Syntax:**

```
SELECT DISTINCT column_name
FROM table_name;
```

- **Example:**

```
SELECT DISTINCT Grade
FROM Students;
```

5.3. ORDER BY

- Sorts the result set by one or more columns.
- **Syntax:**

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

- **Example:**

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

5.4. **LIMIT** (USED IN MYSQL/POSTGRESQL)

- Restricts the number of rows returned.
- **Syntax:**

```
SELECT * FROM table_name LIMIT number;
SELECT * FROM table_name LIMIT initial_position,number_of_row;
```

- **Example:**

```
SELECT * FROM Students LIMIT 5;
SELECT * FROM Students LIMIT 3,5;
```

5.5 BETWEEN

- Filters rows within a range (inclusive).
- **Syntax:**

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

- **Example:**

```
SELECT * FROM Students
WHERE Age BETWEEN 18 AND 25;
```

5.6. IN & NOT IN

- Checks if a column value exists in a given list.
- **Syntax:**

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

- **Example:**

```
SELECT * FROM Students WHERE Grade IN ('A', 'B');
SELECT * FROM Students WHERE Grade NOT IN ('A', 'B');
```

5.7. LIKE

- Used for pattern matching with wildcards (%, _).
- **Syntax:**

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

EXAMPLE (LIKE):

```
SELECT * FROM Students
WHERE Name LIKE 'A%';     -- Starts with A

SELECT * FROM Students
WHERE Name LIKE '%n';     -- Ends with n

SELECT * FROM Students
WHERE Name LIKE '_l%';    -- Second letter is 'l'
```

5.8. IS NULL

- Checks for NULL (missing) values.
- **Syntax:**

```
SELECT * FROM table_name
WHERE column_name IS NULL;
```

- **Example:**

```
SELECT * FROM Students
WHERE Grade IS NULL;
```

```sql
SELECT * FROM students;

SELECT * FROM  students WHERE Age=20;

SELECT  Name FROM  students WHERE Age=20;

SELECT Name,Age FROM  students WHERE Age>20;

SELECT Name FROM  students WHERE Age>=23;

SELECT Name FROM  students WHERE Age<=23;
SELECT * FROM `tab1` WHERE fees<=20000 and course='python';

SELECT * FROM `tab1` WHERE fees=20000 or course='python';

SELECT * FROM `tab1` WHERE not course='python';

SELECT * FROM  table_name WHERE fees BETWEEN 20000 and 30000;

SELECT * FROM  table_name WHERE course in('python','java');
```

LIKE

```
SELECT * FROM table_name WHERE course LIKE `100`;
SELECT * FROM  table_name WHERE course LIKE '%t';
SELECT * FROM  table_name WHERE course LIKE 'a%';
SELECT * FROM  table_name WHERE course LIKE `_ _a%`;
SELECT * FROM  table_name WHERE course LIKE `_ _a`;
```

CASE COMPARISON

- Case in-sensitive comparison:

```
Select * from table_name where upper(column_name)=upper(“Value”)
```

- Case sensitive comparison:

```
Select * from table_name where BINARY column_name=“Value”
```

ALIAS NAMES(AS)

```
Select user_id as “User Id” ,full_name as “Full Name” from table_name;
```

FILTERING & CONDITIONS

FILTERING & CONDITIONS

- Filtering is key to retrieving specific rows based on one or more conditions.
- You’ll use logical operators like AND, OR, NOT, and nested conditions with parentheses to build complex queries.

1. AND OPERATOR

- Returns rows **only if all conditions** are true.
- **Syntax:**

```
SELECT * FROM table_name
WHERE condition1  AND condition2;
```

- **Example:**

```
SELECT * FROM Students
WHERE Age > 18 AND Grade = 'A';
```

2. OR OPERATOR

- Returns rows if **any one** of the conditions is true.
- **Syntax:**

```
SELECT * FROM table_name
WHERE condition1 OR condition2;
```

- **Example:**

```
SELECT * FROM Students
WHERE Grade=‘A’ OR Grade = ‘B';
```

3. NOT OPERATOR

- Reverses the result of a condition — returns rows **that do not match.**
- **Syntax:**

```
SELECT * FROM table_name
WHERE NOT condition;
```

- **Example:**

```
SELECT * FROM Students
WHERE NOT Grade = 'F';
```

4. NESTED CONDITIONS (USING PARENTHESES)

- Used to group conditions and define logical precedence.
- **Syntax:**

```
SELECT * FROM table_name
WHERE (condition1 OR condition2) AND condition3;
```

- **Example:**

```
SELECT * FROM Students
WHERE (Grade = 'A' OR Grade = 'B') AND Age >= 18;
```

JOINS IN SQL

Combining data across multiple tables

WHAT IS A JOIN?

- A **JOIN** combines rows from two or more tables based on a **related column**
- Used when data is split across tables (normalized database structure)

Syntax:

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

TYPES OF JOINS

Type	Description
INNER JOIN	Returns records with matching values in both tables
LEFT JOIN	Returns all records from the left table and matching ones from the right
RIGHT JOIN	Returns all records from the right table and matching ones from the left
FULL OUTER JOIN	Returns all records when there is a match in either left or right table

CREATE 2 TABLES

Table 1: Student

```
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT
);
```

Table 2:

```
CREATE TABLE Marks (
    StudentID INT,
    Subject VARCHAR(50),
    Score INT,
    FOREIGN KEY (StudentID) REFERENCES Students(ID)
);
```

INSERT SAMPLE DATA INTO BOTH TABLES

Table 1: Student

```
INSERT INTO Students (ID, Name, Age)
VALUES

(1, 'Arjun', 20),

(2, 'Sneha', 21),

(3, 'Rahul', 22);
```

Table 2:

```
INSERT INTO Marks (StudentID, Subject,
Score) VALUES

(1, 'Math', 80),

(1, 'English', 70),

(2, 'Math', 90);
```

1.INNER JOIN EXAMPLE

- Returns only records with **matching values** in both tables
- Only students **with marks** will be shown

```
SELECT Students.Name, Marks.Subject, Marks.Score
FROM Students
INNER JOIN Marks ON Students.ID = Marks.StudentID;
```

2. LEFT JOIN (LEFT OUTER JOIN)

- Returns **all records from the left table,** and matched records from the right
- Students **without marks** will appear with Null

```
SELECT Students.Name, Marks.Score
FROM Students
LEFT JOIN Marks ON Students.ID = Marks.StudentID;
```

3. RIGHT JOIN (RIGHT OUTER JOIN)

- Returns **all records from the right table**, and matched records from the left
- Shows all marks, even if no student info is found

```
SELECT Students.Name, Marks.Score
FROM Students
RIGHT JOIN Marks ON Students.ID = Marks.StudentID;
```

4. FULL JOIN (FULL OUTER JOIN)

- Returns **all records when there is a match in either** left or right table
- Shows all students and all marks, matched and unmatched

```
SELECT Students.Name, Marks.Score
FROM Students
FULL OUTER JOIN Marks ON Students.ID = Marks.StudentID;
```

- MySQL (and MariaDB) does not support FULL OUTER JOIN directly.

ALTERNATIVE SOLUTION IN MYSQL:

USE UNION OF LEFT JOIN AND RIGHT JOIN

```
-- LEFT JOIN part
SELECT Students.Name, Marks.Score
FROM Students
LEFT JOIN Marks ON Students.ID = Marks.StudentID
UNION
-- RIGHT JOIN part
SELECT Students.Name, Marks.Score
FROM Students
RIGHT JOIN Marks ON Students.ID = Marks.StudentID
LIMIT 0, 25;
```

5. SELF JOIN

- A table is joined with **itself**, useful for **comparing rows** in the same table
- Example: Finding students and their mentors

```
SELECT A.Name AS Student1, B.Name AS Student2
FROM Students A, Students B
WHERE A.MentorID = B.ID;
```

GROUPING & AGGREGATION

summarize large datasets

GROUPING & AGGREGATION

- **Grouping and aggregation** allow you to summarize large datasets — like counting rows, finding averages, or grouping results based on specific columns.

1. GROUP BY

- Used to group rows that have the same values in specified columns, often used with aggregate functions.
- **Syntax:**

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

- **Example:**

```
SELECT Grade, COUNT(*) AS StudentCount
FROM Students GROUP BY Grade;
```

2. AGGREGATE FUNCTIONS

Function	Description
COUNT()	Counts number of rows
SUM()	Adds values of a column
AVG()	Calculates average value
MIN()	Finds the minimum value
MAX()	Finds the maximum value

EXAMPLES:

```
-- Total number of students
SELECT COUNT(*) FROM Students;
-- Total salary of employees
SELECT SUM(Salary) FROM Employees;
-- Average age of students
SELECT AVG(Age) FROM Students;
-- Youngest student
SELECT MIN(Age) FROM Students;
-- Oldest student
SELECT MAX(Age) FROM Students;
```

3. GROUP BY WITH MULTIPLE COLUMNS

- You can group by more than one column to get more detailed summaries.
- **Example:**

```
SELECT Grade, Age,COUNT(*) AS CountByGroup
FROM Students GROUP BY Grade,Age;
```

4. HAVING CLAUSE

- Used to filter aggregated results. It's like WHERE, but for groups.
- **Syntax:**

```
SELECT column_name, AGG_FUNC(column)
FROM table  GROUP BY column_name HAVING condition;
```

- **Example:**

```
-- Show grades with more than 2 students
SELECT Grade, COUNT(*) AS StudentCount
FROM Students GROUP BY Grade HAVING COUNT(*) > 2;
```

- **⚠️ You cannot use WHERE with aggregate functions. Use HAVING.**

SUBQUERIES & NESTED SELECTS

Writing queries inside queries to solve complex problems

WHAT IS A SUBQUERY?

- A **subquery** is a query inside another query
- Used in SELECT, FROM, or WHERE clauses
- Helps solve **step-by-step logic** in one query

SINGLE-ROW SUBQUERY

- Returns **only one row**
- Finds the oldest student

```
SELECT Name
FROM Students
WHERE Age = (SELECT MAX(Age) FROM Students);
```

MULTI-ROW SUBQUERY

- Returns **multiple rows**
- Matches many ages returned from subquery

```
SELECT Name
FROM Students
WHERE Age IN (SELECT Age FROM Students WHERE Age >
20);
```

USING IN WITH SUBQUERIES

- Use when matching **any value from a list**

```
SELECT Name
FROM Students
WHERE ID IN (SELECT StudentID FROM Marks WHERE Score
> 80);
```

USING EXISTS WITH SUBQUERIES

- Checks if the **subquery returns any rows** (true/false)

```
SELECT Name FROM Students S
WHERE EXISTS (
    SELECT 1 FROM Marks M WHERE M.StudentID = S.ID
);
```

USING ANY

- ANY Example:

True if **greater than at least one** returned value

```
SELECT Name FROM Students
WHERE Age > ANY (SELECT Age FROM Students WHERE Age < 20);
```

- The outer query selects names of students **whose age is greater than *any one* of the students who are younger than 20.**

USING ALL

- ANY Example:

True if **greater than all** returned values

```
SELECT Name FROM Students
WHERE Age > ALL (SELECT Age FROM Students WHERE Age < 20);
```

- The outer query selects names of students **whose age is greater than *every* student whose age is less than 20.**

CORRELATED SUBQUERIES

- A **correlated subquery** refers to **columns from the outer query.** It is executed **once for every row** in the outer query.

- **Syntax:**

```
SELECT column1 FROM table1 outer WHERE column2 = (
SELECT MAX(column2) FROM table2 inner  WHERE inner.column3 =
outer.column3);
```

- **Example:**

```
-- Get students whose age is the highest in their department
SELECT Name, Age, DepartmentID FROM Students S1 WHERE Age = (SELECT
MAX(Age) FROM Students S2 WHERE S1.DepartmentID = S2.DepartmentID);
```

ADVANCED SQL

ADVANCED SQL

- This section introduces powerful SQL expressions for handling conditional logic, null values, and existence checks.

1. CONDITIONAL STATEMENT

- **Example:**

```
SELECT full_name AS name,
IF (gender=“Male”, “He is a man”, “She is a woman”)
AS Gender
FROM user;
```

2. CASE STATEMENT

- Performs **if-else logic** inside SQL queries.
- **Syntax:**

```
SELECT column,
  CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default_result
  END AS alias_name
FROM table;
```

EXAMPLE:

```
SELECT Name, Grade,
  CASE
    WHEN Grade = 'A' THEN 'Excellent'
    WHEN Grade = 'B' THEN 'Good'
    ELSE 'Needs Improvement'
  END AS Performance
FROM Students;
```

3. COALESCE()

- Returns the **first non-NULL** value in a list of expressions.
- **Syntax:**

```
COALESCE (value1, value2, ..., default_value)
```

- **Example:**

```
-- Show student's grade or 'Not Assigned' if NULL
SELECT Name, COALESCE (Grade, 'Not Assigned') AS
FinalGrade
FROM Students;
```

4. NULLIF()

- Compares two values. If they're equal, returns **NULL**; otherwise returns the **first value.**
- **Syntax:**

```
NULLIF (value1, value2)
```

- **Example:**

```
-- Avoid division by zero
SELECT StudentID, Marks, NULLIF (TotalSubjects, 0) AS
SafeSubjects
FROM Scores;
```

5. EXISTS / NOT EXISTS

- Used to test if a **subquery returns any rows.** Returns TRUE/FALSE.
- **Syntax:**

```
SELECT column FROM table
WHERE EXISTS (
  SELECT 1 FROM other_table WHERE condition);
```

EXAMPLE:

```sql
-- Show students who belong to existing departments
SELECT Name
FROM Students S
WHERE EXISTS (
  SELECT 1
  FROM Departments D
  WHERE D.DepartmentID = S.DepartmentID
);
```

NOT EXISTS EXAMPLE:

```
-- Departments with no students
SELECT DepartmentName
FROM Departments D
WHERE NOT EXISTS (
  SELECT 1
  FROM Students S
  WHERE S.DepartmentID = D.DepartmentID
);
```

WINDOW FUNCTIONS

CREATE TABLE

```
-- Create the Students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50),
    DepartmentID INT,
    Age INT,
    Marks INT
);
```

INSERT DATASET SQL

```
-- Insert sample data
INSERT INTO Students (StudentID, Name, DepartmentID, Age, Marks) VALUES
(1, 'Alice', 101, 21, 88),
(2, 'Bob', 101, 20, 92),
(3, 'Charlie', 102, 23, 88),
(4, 'David', 101, 22, 92),
(5, 'Eve', 102, 21, 78),
(6, 'Frank', 101, 20, 92),
(7, 'Grace', 103, 22, 85),
(8, 'Helen', 103, 20, 88);
```

WINDOW FUNCTIONS

- Window functions perform calculations across a set of table rows related to the current row — without collapsing them into groups (unlike GROUP BY).

1. ROW_NUMBER()

- Gives a **unique sequential number** to each row in the result set **within a partition.**
- **Syntax:**

```
SELECT column1,
       ROW_NUMBER() OVER (PARTITION BY col2 ORDER BY col3) AS row_num
FROM table;
```

- **Example:**

```
SELECT Name, DepartmentID,
       ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Age DESC) AS RankInDept
FROM Students;
```

2. RANK()

- Assigns a rank to each row within a partition. **Ties get the same rank,** and the next rank is skipped.
- **Example:**

```
SELECT Name, Marks,
       RANK() OVER (ORDER BY Marks DESC) AS RankByMarks
FROM Students;
```

3. DENSE_RANK()

- Same as RANK() but **does not skip ranks** after ties.
- **Example:**

```
SELECT Name, Marks,
       DENSE_RANK() OVER (ORDER BY Marks DESC) AS DenseRankByMarks
FROM Students;
```

4. LEAD() AND LAG()

- LEAD() – fetches next row's value
- LAG() – fetches previous row's value
- **Example:**

```
SELECT Name, Marks,
       LAG(Marks) OVER (ORDER BY Marks DESC) AS PreviousMarks,
       LEAD(Marks) OVER (ORDER BY Marks DESC) AS NextMarks
FROM Students;
```

5. PARTITION BY

- Divides result set into partitions (groups), and window functions are applied to each partition separately.
- **Example:**

```
SELECT Name, DepartmentID, Age,
       RANK() OVER (PARTITION BY DepartmentID ORDER BY Age DESC) AS AgeRankInDept
FROM Students;
```

6. OVER() CLAUSE

- **The OVER() clause defines:**
    - Partitioning (PARTITION BY)
    - Ordering (ORDER BY) for window functions.

COMMON TABLE EXPRESSIONS (CTES)

COMMON TABLE EXPRESSIONS (CTES)

- A Common Table Expression (CTE) is a temporary result set (like a named subquery) defined using the WITH clause.
- It helps break down complex queries into readable parts.

1. SIMPLE (NON-RECURSIVE) CTE

- Used to create an alias for a SELECT query that can be reused in the main query.
- **Syntax:**

```
WITH cte_name AS (
  SELECT column1, column2
  FROM table_name
  WHERE condition
)
SELECT *
FROM cte_name;
```

EXAMPLE:

```
WITH HighScorers AS (
  SELECT Name, Marks
  FROM Students
  WHERE Marks > 80
)
SELECT * FROM HighScorers;
```

2. RECURSIVE CTE

- Used when you need to work with hierarchical or tree-structured data (like an employee-manager chain or folder structure).
- **Syntax:**

```
WITH RECURSIVE cte_name AS (
  -- Anchor Member (base case)
  SELECT column1, column2 FROM table WHERE condition
  UNION ALL
  -- Recursive Member
  SELECT t.column1, t.column2  FROM table t JOIN cte_name c ON t.parent_column = c.column)
SELECT * FROM cte_name;
```

EXAMPLE:

```
WITH RECURSIVE EmployeeHierarchy AS (
  SELECT EmployeeID, Name, ManagerID FROM Employees
  WHERE ManagerID IS NULL  -- top-level boss
  UNION ALL
  SELECT e.EmployeeID, e.Name, e.ManagerID FROM Employees e
  JOIN EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID
)
SELECT * FROM EmployeeHierarchy;
```

WHEN TO USE CTES

- To simplify complex joins or nested queries
- To improve query readability
- For hierarchical data processing (via recursion)

TRANSACTIONS AND LOCKING

Ensuring data integrity and consistency in SQL operations

1. TRANSACTION

- A **transaction** is a sequence of one or more SQL statements that are executed as a **single unit of work.** A transaction must follow the **ACID properties:**
- **Common Transaction Command:**

Keyword	Description
START TRANSACTION or BEGIN	Starts a new transaction
COMMIT	Saves the changes made in the transaction
ROLLBACK	Undoes the changes if something goes wrong

EXAMPLE TABLE

```
CREATE TABLE Accounts (
    account_id INT PRIMARY KEY,
    account_holder VARCHAR(100),
    balance DECIMAL(10,2)
);

INSERT INTO Accounts VALUES (1, 'Alice', 1000.00), (2, 'Bob', 500.00);
```

EXAMPLE: TRANSACTION (MONEY TRANSFER)

```
-- Start transaction

START TRANSACTION;

-- Step 1: Deduct from Alice's account

UPDATE Accounts SET balance = balance - 200 WHERE account_id = 1;

-- Step 2: Add to Bob's account

UPDATE Accounts SET balance = balance + 200 WHERE account_id = 2;

-- If everything is okay, commit the transaction

COMMIT;

-- If any issue, use ROLLBACK;

-- ROLLBACK;
```

COMMIT

- Saves all changes made in the current transaction.

```
START TRANSACTION;
UPDATE BankAccounts SET balance = balance - 100 WHERE
account_id = 1;
UPDATE BankAccounts SET balance = balance + 100 WHERE
account_id = 2;
COMMIT;
```

ROLLBACK

Cancels changes made in the current transaction and returns the database to the last committed state.

```
START TRANSACTION;
DELETE FROM Students WHERE StudentID = 101;

-- Mistake found
ROLLBACK;
```

ACID PROPERTIES

A **good transaction system** follows **ACID** principles:

Property	Meaning
Atomicity	All changes in a transaction happen or none do
Consistency	DB moves from one valid state to another
Isolation	Transactions don’t interfere with each other
Durability	Committed changes persist even after failure

2. TRANSACTION ISOLATION LEVELS

- Control **how/when changes made by one transaction become visible** to others.

Level	Description	Issues Prevented
READ UNCOMMITTED	Can read uncommitted changes (dirty reads)	None
READ COMMITTED	Reads only committed data	No dirty reads
REPEATABLE READ	Same row gives same result in one transaction	+ no non-repeatable reads
SERIALIZABLE	Fully isolated (like rows are locked)	All anomalies prevented

SET ISOLATION LEVEL:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
```

3. LOCKING IN SQL

- **Locks** prevent multiple users from modifying the same data at the same time — this ensures **data consistency**.

Lock Type	Description
Shared Lock	Multiple users can read, no one can write
Exclusive Lock	Only one user can read/write

EXAMPLE WITH LOCKING

- Let’s say Alice is transferring money, and we don’t want Bob to view or change the data midway.

```
START TRANSACTION;
-- Lock Alice's and Bob's rows
SELECT * FROM Accounts WHERE account_id IN (1,2) FOR UPDATE;
-- Perform transaction safely
UPDATE Accounts SET balance = balance - 200 WHERE account_id = 1;
UPDATE Accounts SET balance = balance + 200 WHERE account_id = 2;
COMMIT;
```

WHAT DOES FOR UPDATE DO?

It **locks** the selected rows so that:

- No other transaction can update or delete them.
- Other users trying to access those rows must wait.

HOW IT ALL WORKS TOGETHER

1. START TRANSACTION begins a block of work.
2. Use locks (FOR UPDATE) to prevent conflicts.
3. Perform all required INSERT/UPDATE/DELETE operations.
4. If successful → COMMIT.
5. If any error → ROLLBACK.

4. DEADLOCKS

- Occurs when **two transactions wait for each other** to release a lock — causing a standstill.

T1	T2
Locks row A	Locks row B
Waits for row B	Waits for row A

- MySQL will **automatically detect** and kill one transaction.

HOW TO VIEW DEADLOCKS:

```
SHOW ENGINE INNODB STATUS;
```

SECURITY & ACCESS CONTROL

SECURITY & ACCESS CONTROL IN MYSQL

- Ensuring proper **user management**, **role-based access**, and **authentication** is crucial to protect your MySQL database from unauthorized access or accidental data loss.
- MySQL security is about managing users, assigning roles, and handling login authentication.
- Purpose: To stop unauthorized access or accidental data loss.
- **Keywords:**
  - **User Management** – Who can log in.
  - **Role-Based Access** – What they are allowed to do.
  - **Authentication** – Making sure only trusted people get in.

CREATE NEW USER:

Creating a new User:

```
create user 'vinthiya'@'localhost' identified by 'vinthiya123';
```

Show the Permission of the user:

```
show grants for 'vinthiya'@'localhost';
```

1. GRANT

Gives specific privileges to a user.

Syntax:

```
GRANT privilege_list ON database.table TO 'username'@'host';
```

Example:

```
GRANT SELECT, INSERT ON school.Students TO 'john'@'localhost';
```

User john (on same computer = localhost) is allowed to read and add data in the Students table of the school database.

Gives Allprivileges to a user.

Syntax:

```
GRANT ALL PRIVILEGES ON database.table TO 'username'@'host';
```

Example:

```
GRANT ALL PRIVILEGES ON school.Students TO 'john'@'localhost';
```

2. REVOKE

Removes privileges previously granted to a user.

Syntax:

```
REVOKE privilege_list ON database.table FROM 'username'@'host';
```

Example:

```
REVOKE INSERT ON school.Students FROM 'john'@'localhost';
```

COMMON PRIVILEGES

Privilege	Description
SELECT	Read data from tables
INSERT	Add new rows
UPDATE	Modify existing rows
DELETE	Remove rows
ALL	All available privileges

3. USER ROLES (MYSQL 8.0+)

Roles are named collections of privileges that can be granted to users — simplifies permission management.

Create Role:

```
CREATE ROLE 'developer';
```

Grant Privileges to Role:

```
GRANT SELECT, INSERT ON school.* TO 'developer';
```

Assign Role to User:

```
GRANT 'developer' TO 'alice'@'localhost';
```

Set Role as Default:

```
SET DEFAULT ROLE 'developer' TO 'alice'@'localhost';
```

DATABASE AUTHENTICATION BASICS

MySQL uses usernames and passwords for login, stored in the mysql.user system table.

Create New User:

```
GRANT 'developer' TO 'alice'@'localhost';
```

Change Password:

```
ALTER USER 'bob'@'localhost' IDENTIFIED BY 'NewPass456!';
```

Delete User:

```sql
DROP USER 'bob'@'localhost';
```

SQL SET OPERATORS

SQL SET OPERATORS

- Set operators are used to **combine the results** of two or more **SELECT** statements.
- **Rules to Remember:**

  Both SELECT statements must have:

  - The same number of columns
  - Same data types
  - Same order

EXAMPLE TABLES:

Table 1: Students_A

```
CREATE TABLE Students_A (
  ID INT,
  Name VARCHAR(50)
);

INSERT INTO Students_A (ID, Name) VALUES
(1, 'Alice'),
(2, 'Bob'),
(3, 'Charlie'),
(4, 'David');
```

Table 2: Students_B

```
CREATE TABLE Students_B (
  ID INT,
  Name VARCHAR(50)
);

INSERT INTO Students_B (ID, Name) VALUES
(3, 'Charlie'),
(4, 'David'),
(5, 'Eve'),
(6, 'Frank');
```

1. UNION

- Combines results from two SELECT queries.
- Removes duplicates.

```
SELECT Name FROM Students_A
UNION
SELECT Name FROM Students_B;
```

- ***Output***: Unique list of cities from both tables.

2. UNION ALL

- Same as UNION, but keeps duplicates.

```
SELECT Name FROM Students_A
UNION ALL
SELECT Name FROM Students_B;
```

- ***Output:*** All cities, including repeated ones.

3. **INTERSECT** (NOT SUPPORTED IN MYSQL DIRECTLY)

- Returns only common rows in both results

```
SELECT Name FROM Students_A
INTERSECT
SELECT Name FROM Students_B;
```

- *Output*: Cities that exist in **both** tables.
- In MySQL, simulate with:

```
SELECT Name FROM Students_A
WHERE Name IN (SELECT Name FROM Students_B);
```

4. EXCEPT / MINUS

Returns rows from the first query that **aren’t in** the second.

In MySQL, use NOT IN.

```
SELECT Name FROM Students_A
EXCEPT
SELECT Name FROM Students_B;
```

Output: Cities in Customers, but not in Suppliers.

MySQL version:

```
SELECT Name FROM Students_A
WHERE Name NOT IN (SELECT Name FROM Students_B);
```

COMBINING SET OPERATORS:

You can chain set operators together for more complex queries.

```
-- Step 1: UNION of both tables
SELECT Name FROM ( SELECT Name FROM Students_A UNION SELECT
Name FROM Students_B ) AS AllNames

-- Step 2: EXCEPT (simulate with NOT IN)
WHERE Name NOT IN (SELECT Name FROM Students_A)

-- Step 3: INTERSECT (simulate with IN)
AND Name IN ( SELECT Name FROM Students_B );
```

STORED PROCEDURES

Reusable blocks of SQL logic

STORED PROCEDURE

- A Stored Procedure is a precompiled collection of SQL statements (like SELECT, INSERT, UPDATE, DELETE) that you can save and reuse.
- Think of it like a function in programming — once created, it can be called with or without parameters whenever needed.

WHY USE STORED PROCEDURES?

- Code Reusability
- Better Performance (compiled once, used many times)
- Reduced Network Traffic (logic runs on server)
- Security (can restrict direct access to tables)

SYNTAX (MYSQL)

```
DELIMITER //

CREATE PROCEDURE procedure_name ([IN | OUT | INOUT] parameter_name datatype, ...)
BEGIN
   -- SQL statements
END //

DELIMITER ;
```

SIMPLE EXAMPLE: NO PARAMETERS

```
DELIMITER //

CREATE PROCEDURE ShowAllEmployees()
BEGIN
   SELECT * FROM Employees;
END //

DELIMITER ;
```

TO CALL THE PROCEDURE:

```
CALL ShowAllEmployees();
```

ALTER PROCEDURE(NOT SUPPORTED INMYSQL):

```
ALTER PROCEDURE GetEmployees
AS
BEGIN
    SELECT emp_id, full_name, salary
    FROM employees
    WHERE salary > 50000;
END;
```

VARIABLE:

```
ALTER PROCEDURE GetEmployees
AS
BEGIN
    DECLARE total INT DEFAULT= 0;
    SELECT COUNT(emp_id)
    INTO total FROM employee;
    SELECT total;
END;
```

EXAMPLE WITH INPUT PARAMETER:

```
DELIMITER //
CREATE PROCEDURE AddEmployee(IN empName VARCHAR(50), IN salary DECIMAL(10,2))
BEGIN
   INSERT INTO Employees (name, salary) VALUES (empName, salary);
END //
DELIMITER ;


-- Call the procedure
CALL AddEmployee('John Doe', 50000);
```

EXAMPLE WITH OUTPUT PARAMETER:

```
DELIMITER //

CREATE PROCEDURE GetEmployeeCount(OUT total INT)
BEGIN
   SELECT COUNT(*) INTO total FROM Employees;
END //

DELIMITER ;
```

CALL AND DISPLAY THE OUTPUT:

```
CALL GetEmployeeCount(@emp_count);
SELECT @emp_count;
```

MODIFY / DROP PROCEDURE

Drop a procedure:

```
DROP PROCEDURE IF EXISTS ShowAllEmployees;
```

View procedures:

```
SHOW PROCEDURE STATUS WHERE Db =
'your_database_name';
```

FUNCTION

FUNCTION:

- A Function in SQL is a stored block of code that performs a calculation or operation and returns a single value.
- It's similar to a stored procedure, but with key differences:
  - A function must return a value. in SELECT
  - It can be used in SQL expressions (e.g.,).
  - It cannot modify database state (no INSERT, UPDATE, etc.).

SAMPLE DATA

```
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    salary DECIMAL(10,2),
    join_date DATE,
    bonus_percent DECIMAL(5,2)
);
```

```
INSERT INTO employees (name, email, salary, join_date, bonus_percent)
VALUES
('John Doe', 'john.doe@example.com', 50000, '2023-01-15', 10.00),
('Jane Smith', 'jane.smith@example.com', 60000, '2022-11-01', 12.50),
('Mike Jordan', 'mike.j@example.com', 45000, '2023-07-01', 8.00),
('Anu Priya', 'anu.priya@example.com', 55000, '2024-03-10', 15.00);
```

WHY USE FUNCTIONS?

- Reuse complex logic in a simple call
- Simplify and clean up queries
- Useful in reporting, validation, and formatting

1. SYSTEM FUNCTIONS (BUILT-IN)

Predefined by SQL to perform operations like:

- String functions:

  UPPER(), LOWER(), SUBSTRING()

- Date functions:

  GETDATE(), DATE_ADD(date, INTERVAL n DAY),DATEDIFF(date1, date2)

- Math functions:

  ROUND(), CEILING(), FLOOR()

- Aggregate functions:

  SUM(), AVG(), COUNT(), MIN(), MAX()

1.1. STRING HANDLING FUNCTIONS

a. **CHAR_LENGTH(string)**
   - Returns the length of a string.
   - Query: SELECT CHAR_LENGTH("I am SCOPE");
   - Result: 10

b. **CHARACTER_LENGTH(string)**
   - Same as CHAR_LENGTH.

c. **FORMAT(number, decimal points)**
   - Format a number to decimal or non-decimal and separate the number by comma.
   - Query: SELECT FORMAT(18976.1234,2);
   - Result: 18,976.12

d. **CONCAT(value1, value2, etc.)**

- Joins multiple values into a single string.
- Query: SELECT CONCAT('i','am','extraordinary');
- Result: iamextraordinary

e. **CONCAT_WS(separator, str1, str2, etc.)**

- Joins multiple values with a separator.
- Query: SELECT CONCAT_WS('-', 'i','am','extraordinary');
- Result: i-am-extraordinary

f. **INSERT(string, start position, number of characters to replace, replace string)**

- Query: SELECT INSERT("hello world", 7, 5, "friends");
- Result: hello friends

g. **LEFT(string, number of characters)**

- Query: SELECT LEFT("hello world", 5);
- Result: hello

h. **RIGHT(string, number of characters)**

- Query: SELECT RIGHT("hello world", 5);
- Result: world

i. **SUBSTR(string, start position, length)**

- Query: SELECT SUBSTR("I am from India", 11, 5);
- Result: India

j. **SUBSTRING(string, start position, length)**

- Same as SUBSTR.

k. **UCASE(string) / UPPER(string)**

- Convert to uppercase.
- Query: SELECT UCASE("india");
- Result: INDIA

l. **LCASE(string) / LOWER(string)**

- Convert to lowercase.
- Query: SELECT LCASE(“INDIA");
- Result: india

TRIM FUNCTIONS

a. **TRIM(string)**
   - Removes whitespace from both ends.
   - Query: SELECT TRIM(" I am from India ");
   - Result: I am from India

b. **LTRIM(string)**
   - Removes whitespace from the left.
   - Query: SELECT LTRIM(" I am from India ");
   - Result: I am from India

c. **RTRIM(string)**

- Removes whitespace from the right.
- Query: SELECT RTRIM(" I am from India ");
- Result: I am from India

d. **REVERSE(string)**

- Reverses a string.
- Query: SELECT REVERSE("SCOPE");
- Result: EPOCS

e. **REPLACE(string, search string, replacing string)**

- Query: SELECT REPLACE("I am sad", "sad", "happy");
- Result: I am happy

1.2. DATE FUNCTIONS

a. **CURDATE()**
   - Returns current date.
   - Query: SELECT CURDATE();
   - Result: 2022-11-07

b. **CURTIME()**
   - Returns current time.
   - Query: SELECT CURTIME();
   - Result: 09:00:00

c. **CURRENT_DATE()**
   - Same as CURDATE()

d. **CURRENT_TIME()**
   - Same as CURTIME()

e. **NOW()**
   - Returns current date and time.
   - Query: SELECT NOW();
   - Result: 2022-11-07 09:00:00

f. **CURRENT_TIMESTAMP()**
   - Returns current standard date and time.

1.3. MATH FUNCTIONS

a. **AVG(column_name)**
   - Returns average value.
   - Query: SELECT AVG(price) FROM products_table WHERE condition;

b. **COUNT(column_name)**
   - Count rows.
   - Query: SELECT COUNT(*) FROM users_table WHERE condition;

c. **GREATEST(value1, value2, etc.)**
   - Returns highest value.
   - Query: SELECT GREATEST(100, 10, 27, 42);

d. **MAX(column_name)**
   - Returns max from column.

e. **LEAST(value1, value2, etc.)**
   - Returns smallest value.
   - Query: SELECT LEAST(40, 20, 30);

f. **MIN(column_name)**
   - Returns min from column.

g. **MOD(x, y)**
   - Returns remainder.
   - Query: SELECT MOD(19, 4);
   - Result: 3

h) **RAND()**

- Random number between 0 and 1.
- Query: SELECT RAND();
- Result: e.g., 0.2567871569253385
- Query: SELECT RAND(10);
- Result: random decimal under 10

i) **ROUND(number, decimal_points)**

- Rounds number.
- Query: SELECT ROUND(100.1234, 2);
- Result: 100.12

j. **SUM(column_name)**
   - Returns total sum.
   - Query: SELECT SUM(price) FROM products_table WHERE condition;

k. **LAST_INSERT_ID()**
   - Returns last AUTO_INCREMENT id.
   - Query: SELECT LAST_INSERT_ID();
   - Result: Example 101

2. USER-DEFINED FUNCTIONS (UDF)

Created by users to perform custom tasks.

SYNTAX (MYSQL)

```
DELIMITER //
CREATE FUNCTION GetDiscount(price DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
   RETURN price * 0.10;
END //
DELIMITER ;
-- Use in a query
SELECT name, salary, GetDiscount(salary) AS Discount FROM Employees;
```

EXAMPLE:

```
DELIMITER //
CREATE FUNCTION GetDiscount(price DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
   RETURN price * 0.10;
END //
DELIMITER ;
-- Use in a query
SELECT name, salary, GetDiscount(salary) AS Discount FROM Employees;
```

KEY DIFFERENCES

Feature	Stored Procedure	Function
Returns a value	Optional (via OUT parameter)	Mandatory (via RETURN)
Used in SQL queries	✖ No	✔ Yes
Modifies database (DML)	✔ Yes (insert, update, delete allowed)	✖ No (read-only)
Syntax usage	CALL ProcedureName()	Used inside queries like SELECT
Output type	Can return multiple values via OUT	Returns only one value

WHEN TO USE?

Use Stored Procedure when:

- You need to **perform operations** like insert, update, or delete
- You want to **group multiple SQL statements**
- You want **modular code** to handle business logic
- You need **input/output parameters**

Use Function when:

- You need to **calculate and return a single value**
- You want to use it **inside a query**
- You are doing **read-only operations**

TRIGGERS & EVENTS

Automating database responses to changes

1. WHAT IS A TRIGGER?

- A **Trigger** is a set of SQL statements that **automatically executes in response to a specific events** like INSERT, UPDATE, or DELETE.
- **Types of Triggers:**

Timing	Event	Description
BEFORE	INSERT	Trigger runs before inserting a row
AFTER	INSERT	Trigger runs after inserting a row
BEFORE	UPDATE	Trigger runs before updating a row
AFTER	UPDATE	Trigger runs after updating a row
BEFORE	DELETE	Trigger runs before deleting a row
AFTER	DELETE	Trigger runs after deleting a row

SYNTAX: CREATE TRIGGER

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
   -- trigger logic here
END;
```

EXAMPLE 1: LOG INSERTIONS INTO ANOTHER TABLE

```
-- Assume we have a table to log inserts
CREATE TABLE student_log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    student_name VARCHAR(100),
    inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Trigger to log student insert
CREATE TRIGGER log_student_insert
AFTER INSERT ON students
FOR EACH ROW
BEGIN
    INSERT INTO student_log (student_name)
    VALUES (NEW.name);
END;
```

EXAMPLE 2: PREVENT SALARY REDUCTION

```
CREATE TRIGGER prevent_salary_cut
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    IF NEW.salary < OLD.salary THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Salary reduction not allowed';
    END IF;
END;
```

USE CASES FOR TRIGGERS

- Automatically update timestamps (created_at, updated_at)
- Validate or modify input data before saving
- Log changes to audit tables
- Enforce business rules (e.g., prevent deletion if balance > 0)

2. EVENT SCHEDULER (MYSQL)

- The **Event Scheduler** lets you **schedule SQL jobs to run automatically** at a given time or interval.
- **Enable Event Scheduler (if not already):**

```
SET GLOBAL event_scheduler = ON;
```

SYNTAX: CREATE EVENT

```
CREATE EVENT event_name

ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 MINUTE

DO

   SQL statement;
```

EXAMPLE: AUTO-CLEAN OLD LOGS EVERY DAY

```
CREATE EVENT delete_old_logs

ON SCHEDULE EVERY 1 DAY

DO

  DELETE FROM student_log

  WHERE inserted_at < NOW() - INTERVAL 30 DAY;
```

TO CHECK EVENTS:

```
SHOW EVENTS;
```

TO DROP AN EVENT:

```
DROP EVENT event_name;
```

DATABASE DESIGN CONCEPTS

DATABASE DESIGN CONCEPTS

Good database design ensures **data integrity**, **efficiency**, and **maintainability.** The key concepts include **normalization** and **keys** (primary & foreign).

1. NORMALIZATION

Normalization is the process of organizing data to reduce **redundancy** and improve **data integrity** by dividing data into related tables.

Normal Forms:

Normal Form	Description
1NF (First Normal Form)	Ensure each column contains atomic (indivisible) values, and each row is unique.
2NF (Second Normal Form)	Be in 1NF, and every non-key column is fully dependent on the primary key (remove partial dependencies).
3NF (Third Normal Form)	Be in 2NF, and no transitive dependencies (non-key columns shouldn’t depend on other non-key columns).

1NF (FIRST NORMAL FORM)

Rule:

Each column should contain **atomic** (indivisible) values.

No **repeating groups or arrays.**

Each row must be **unique.**

StudentID	Name	Course
1	Alice	Math, Physics

→

StudentID	Name	Course
1	Alice	Math
1	Alice	Physics

2NF – SECOND NORMAL FORM

Rule:

- Be in 1NF
- All non-key attributes must depend on the whole primary key (no partial dependency)

Example:

- Imagine a table with a composite primary key (StudentID, CourseID)
- Problem:

StudentID	CourseID	StudentName

Solution (Split tables):

1. Students Table

StudentID	StudentName

2. Enrollments Table

StudentID	Course ID

3NF – THIRD NORMAL FORM

Rule:

Be in **2NF**

No **transitive dependencies** (non-key → non-key)

✗ Problem:

StudentID	Name	DeptID	DeptName

Solution:
Split into:

1. **Students Table**
   | StudentID | Name | DeptID |
2. **Departments Table**
   | DeptID | DeptName |

BENEFITS OF NORMALIZATION:

- Eliminates redundancy
- Improves data consistency
- Makes the database scalable and easier to maintain
- Ensures logical data storage

2. PRIMARY KEY IN SQL

- A Primary Key uniquely identifies each record in a table.
- It cannot be NULL and must be unique.
- A table can have only one primary key (it can consist of one or more columns).

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT
);
```

COMPOSITE PRIMARY KEY

- A composite key uses multiple columns together as a unique identifier.
- It is helpful when one column is not enough to make a row unique.

```
CREATE TABLE StudentCourses
( StudentID INT,
CourseID INT,
 EnrollmentDate DATE,
PRIMARY KEY (StudentID, CourseID));
```

3. FOREIGN KEY IN SQL

- A Foreign Key is a column (or combination of columns) that creates a link between two tables.
- It refers to the Primary Key in another table.
- It helps maintain referential integrity — ensures data consistency between related tables.

```sql
CREATE TABLE enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_name VARCHAR(100),
    FOREIGN KEY (student_id) REFERENCES students(student_id)
);
```

In this example:

- student_id in enrollments is a Foreign Key.
- It references the student_id in the students table.
- This means: A student must exist in the students table before they can be added to enrollments.

DIFFERENCE BETWEEN PRIMARY KEY AND FOREIGN KEY

Feature	Primary Key	Foreign Key
Purpose	Uniquely identifies a record	Links records between two tables
Uniqueness	Must be unique	Can have duplicates
NULL values	Not allowed	Allowed (if not explicitly restricted)
Table	Only one per table	Can have multiple
Data integrity	Ensures row uniqueness	Ensures valid references

VIEWS AND INDEXES

Simplify queries and improve performance

WHAT IS A VIEW?

- A **View** is a **virtual table** based on a SQL query
- It does **not store data** but shows results from other tables
- Useful for **simplifying complex queries** and improving security

CREATING A VIEW

```
CREATE VIEW StudentScores AS
SELECT Students.Name, Marks.Subject, Marks.Score
FROM Students
JOIN Marks ON Students.ID = Marks.StudentID;
```

You can now query this view:

```
SELECT * FROM StudentScores WHERE Score > 80;
```

WHAT IS AN INDEX?

- An **Index** is a **database object** that speeds up data retrieval
- Works like a **book index –** quickly finds rows without scanning the whole table
- Best used on **frequently searched** or **JOINed** columns

```
SHOW INDEX FROM Table_name;
```

CREATING AN INDEX

```
CREATE INDEX idx_student_name ON Students(Name);
```

```
ALTER TABLE table_name DROP INDEX idx_student_name;
```

```
ALTER TABLE table_name ADD INDEX idx_student_name;
```

This index speeds up queries that search by student name:

```
SELECT * FROM Students WHERE Name = 'Ravi';
```

TYPES OF INDEXES & PERFORMANCE IMPACT

Pros: Speeds up SELECT, JOIN, WHERE

Cons: Slows down INSERT, UPDATE, DELETE (because index must be updated too)

Type	Description
Single-Column	Index on one column
Composite	Index on multiple columns

DATA IMPORT/EXPORT & INTEGRATION :

WHAT DOES THIS MEAN?

- **Import:** Bring data *into* your SQL database.
- **Export:** Take data *out* from your SQL database.
- **Integration:** Connect SQL with other tools or systems (like APIs or ETL tools) to automate or manage data flows.

1. CSV / EXCEL IMPORT

What is it?

Sometimes, your data is stored in Excel or CSV files (like .csv files).

MySQL Workbench allows you to import this data into a database table.

HOW TO IMPORT CSV FILE INTO A TABLE:

Example: You have a CSV file students.csv

StudentID	Name	Age
1	Rahul	20
2	Priya	22

Step 1: Create Table (manually or using SQL)

```
CREATE TABLE Students (
StudentID INT,
Name VARCHAR(100),
Age INT);
```

Step 2: Import the CSV file

1. In MySQL Workbench, go to:
    - Server → Data Import
2. Choose "Import from Self-Contained File"
    - Select your .csv file
3. Select your database and table
4. Click "Start Import"

☞ Now, your CSV data is inserted into the Students table.

HOW TO EXPORT MYSQL DATA TO A CSV:

📌 Steps:

1. Right-click the table → "Table Data Export Wizard“
2. Choose export format as CSV
3. Select where to save
4. Click Export

✅ Now you have your data in a .csv file.

PART 2: SQL WITH APIS / ETL TOOLS (BASIC CONCEPT)

What is API Integration?

Sometimes data comes from **web services/APIs** (e.g., weather data, payment info).

You can use **Python** or **ETL tools** to:

- Get API data
- Clean it
- Store it into **MySQL tables**

EXAMPLE WITH PYTHON:

```
import requests
import mysql.connector
# Call API
response = requests.get("https://api.example.com/data")
data = response.json()
# Connect to MySQL
conn = mysql.connector.connect( host="localhost", user="root", password="pass", database="demo")
cursor = conn.cursor()
# Insert API data into MySQL
for item in data:
    cursor.execute("INSERT INTO table_name (col1, col2) VALUES (%s, %s)", (item['val1'], item['val2']))
conn.commit()
```

ETL TOOLS (BASIC MENTION)

ETL = Extract, Transform, Load

- **Extract:** Get data from source (CSV, Excel, API)
- **Transform:** Clean/change format
- **Load:** Insert into MySQL

Popular tools:

- **Talend**
- **Apache Nifi**
- **Microsoft SSIS**
- **Pentaho**
- **Airbyte**

You can use these tools to **automate importing data** from many sources into MySQL.

ADVANCED QUERY PATTERNS

ADVANCED QUERY PATTERNS

These techniques solve complex real-world SQL problems like reshaping data (pivot), generating SQL dynamically, and handling hierarchical/parent-child relationships.

Pattern	Use Case	Supported In
Pivot	Rows → Columns	MySQL, PostgreSQL (manual or built-in)
Unpivot	Columns → Rows	Manual using UNION
Dynamic SQL	Build query strings at runtime	MySQL, SQL Server
Hierarchical Data	Trees (e.g. employee → manager)	Use Recursive CTEs

1. PIVOT

PIVOT: CONVERTS **ROWS INTO COLUMNS**.

Use Case:

You have sales data by month:

Employee	Month	Sales
Alice	Jan	1000
Alice	Feb	1200
Bob	Jan	800
Bob	Feb	950

Pivot to:

Employee	Jan	Feb
Alice	1000	1200
Bob	800	950

IN MYSQL (MANUAL PIVOT USING CASE)

```
SELECT
  Employee,
  SUM(CASE WHEN Month = 'Jan' THEN Sales ELSE 0 END) AS Jan,
  SUM(CASE WHEN Month = 'Feb' THEN Sales ELSE 0 END) AS Feb
FROM sales_data
GROUP BY Employee;
```

2. UNPIVOT: CONVERTS COLUMNS INTO ROWS

SQL doesn't have native UNPIVOT, but you can simulate it using UNION ALL.

Example:

```
SELECT Employee, 'Jan' AS Month, Jan AS Sales FROM
pivot_table

UNION ALL

SELECT Employee, 'Feb', Feb FROM pivot_table;
```

3. DYNAMIC SQL

- Dynamic SQL is SQL code **generated and executed at runtime.**
- It's useful when the exact query depends on dynamic conditions like table names, column filters, etc.
- **Real-time Use Case:**

You want to **search any column** based on user input.

EXAMPLE (MYSQL WITH PREPARE, EXECUTE):

```
SET @col_name = 'Department';

SET @value = 'HR';

SET @query = CONCAT('SELECT * FROM employees WHERE ',
@col_name, ' = ''', @value, '''');

PREPARE stmt FROM @query;

EXECUTE stmt;

DEALLOCATE PREPARE stmt;
```

4. HIERARCHICAL DATA (USING RECURSIVE CTES)

- Recursive CTEs (Common Table Expressions) allow you to query **hierarchical or tree-structured data,** like an employee-manager relationship.

ID	Name	Manager_ID
1	Alice	NULL
2	Bob	1
3	Carol	2
4	Dave	2

- Alice is the top manager (no one manages her).
- Bob reports to Alice.
- Carol and Dave report to Bob.

RECURSIVE CTE QUERY (POSTGRESQL / MYSQL 8+):

```
WITH RECURSIVE emp_tree AS (
    SELECT ID, Name, Manager_ID, 1 AS Level
    FROM employees
    WHERE Manager_ID IS NULL
    UNION ALL
    SELECT e.ID, e.Name, e.Manager_ID, t.Level + 1
    FROM employees e
    JOIN emp_tree t ON e.Manager_ID = t.ID
)
SELECT * FROM emp_tree;
```

TESTING & DEBUGGING QUERIES :

TESTING & DEBUGGING QUERIES

Debugging and testing are essential for writing correct, efficient, and safe SQL queries — especially before applying changes to production data.

1. COMMON SQL DEBUGGING PRACTICES

a. Use LIMIT for Safer Testing

Start with limited rows to avoid long-running or destructive queries:

```
SELECT * FROM employees LIMIT 10;
```

b. Preview with SELECT Before UPDATE or DELETE

Check your conditions:

```
-- Check affected rows before updating
SELECT * FROM students WHERE grade = 'F';
-- Then run UPDATE
UPDATE students SET status = 'Review' WHERE grade = 'F';
```

c. Use EXPLAIN to Understand Query Execution

It shows how MySQL will execute a query (indexes, order, joins):

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 123;
```

d. **Print or Log Values (in Stored Procedures/Triggers)**

In MySQL, use SIGNAL or custom logging table:

```
CREATE TABLE log_table (
    id INT AUTO_INCREMENT PRIMARY KEY, message VARCHAR(255),
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

```
 DELIMITER $$

CREATE TRIGGER after_user_insert

AFTER INSERT ON users

FOR EACH ROW

BEGIN    INSERT INTO log_table (message)

VALUES (CONCAT('User ID is ', NEW.user_id));

END$$

DELIMITER ;
```

2. USING TEMPORARY TABLES FOR TESTING

Temporary tables allow you to **test transformations or logic** without affecting real data.

Syntax: Create Temporary Table

```
CREATE TEMPORARY TABLE temp_employees AS

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

USE CASE: TEST AN UPDATE WITHOUT CHANGING ORIGINAL DATA

Note: Temporary tables **automatically disappear** when your session ends.

```
-- Step 1: Create temp table from real data
CREATE TEMPORARY TABLE temp_students AS
SELECT * FROM students;
-- Step 2: Try your update logic
UPDATE temp_students SET grade = 'A‘ WHERE marks > 90;
-- Step 3: Verify
SELECT * FROM temp_students;
```

RESET AND TRY AGAIN

You can **drop and re-create** the temporary table multiple times as you tweak your logic.

```
DROP TEMPORARY TABLE IF EXISTS temp_students;
```

TOOLS & PLATFORMS (MYSQL, POSTGRESQL, ETC.)

TOOLS & PLATFORMS (MYSQL, POSTGRESQL, ETC.)

- SQL is supported across multiple **database systems (RDBMS)** — each offering its own strengths, syntax variations, and tools.

1. MYSQL

- Type: Open-source
- RDBMSBest For: Web applications, small to medium-scale systems
- Key Features:
  - Widely used with PHP, Python, Django, etc.
  - Community and Enterprise editions
  - Tools: MySQL Workbench, phpMyAdmin
- Syntax Example:

```
SELECT * FROM Students LIMIT 5;
```

2. POSTGRESQL

Type: Open-source, object-relational database

Best For: Complex queries, large data analytics, GIS data

Key Features:

- Strong support for window functions, CTEs, JSON
- ACID-compliant
- Tools: pgAdmin, Dbeaver

Syntax Example:

```
SELECT * FROM Students FETCH FIRST 5 ROWS ONLY;
```

3. SQL SERVER (BY MICROSOFT)

Type: Proprietary RDBMS

Best For: Enterprise systems, .NET integrations

Key Features:

- Advanced transaction handling and reporting
- Integration with Power BI, Azure
- Tools: SQL Server Management Studio (SSMS)

Syntax Example:

```
SELECT TOP 5 * FROM Students;
```

4. ORACLE DATABASE

Type: Commercial RDBMS

Best For: High-performance, mission-critical enterprise systems

Key Features:

- Advanced features: Real Application Clusters, Flashback
- Tools: Oracle SQL Developer

Syntax Example:

```
SELECT * FROM Students WHERE ROWNUM <= 5;
```

5. GOOGLE BIGQUERY

Type: Serverless, cloud-based data warehouse (not a traditional RDBMS)

Best For: Big data analytics and fast SQL querying over huge datasets

Key Features:

- Fully managed
- Pay-per-query model
- Ideal for data lakes, analytics

Syntax Example:

```
SELECT * FROM `project.dataset.Students` LIMIT 5;
```

6. SNOWFLAKE

Type: Cloud-native data warehouse

Best For: Scalable analytics, multi-cloud deployment

Key Features:

- Automatic scaling
- Supports semi-structured data (JSON, Avro)
- Built-in support for CTEs, window functions

Syntax Example:

```
SELECT * FROM Students LIMIT 5;
```

COMPARISON

Platform	Open Source	Best Use Case	UI Tool
MySQL	✓	Web apps	MySQL Workbench, phpMyAdmin
PostgreSQL	✓	Complex queries, GIS, analytics	pgAdmin, DBeaver
SQL Server	✗	Enterprise, .NET apps	SSMS
Oracle	✗	High-performance enterprise use	SQL Developer
BigQuery	✗ (Cloud)	Cloud-scale analytics	Google Cloud Console
Snowflake	✗ (Cloud)	Scalable data warehouse	Snowflake Web UI

SQL PERFORMANCE TUNING

Writing efficient and scalable SQL queries

AVOIDING COMMON PERFORMANCE ISSUES

- ✗ Don't use SELECT * — retrieve only needed columns
- ✗ Avoid unnecessary DISTINCT, ORDER BY, or GROUP BY
- ✗ Minimize nested subqueries when JOINs can be used

- ➢ Use LIMIT to reduce result size
- ➢ Archive old data from large tables

REAL-WORLD PROJECT IDEAS

- Student Course Tracker
- Hospital Database
- Hotel Management System
- E-commerce product catalog

Thank you