LECTURE ONE

UNDERSTANDING MySQL & MySQL WORKBENCH

* **What is SQL?**

SQL (Structured Query Language) is a language used to “**communicate**” (ask questions and give instructions) to a database.

* **What is a Database?**

A database is like a “**storage** **room”** (more like an organized storage room) for information. Inside the room, there are “**tables**”-like shelves- that store related data.

For example:

* One table can have the list of staff.
* Another table can have the list of students.

So, each table is made up of “**columns**” (type of information like Name, Age, Gender) and “**rows**” (each row is a full record of a thing or person)

* **What is MySQL Workbench?**

MySQL Workbench is a tool that makes it easy to write SQL queries and manage databases.

* Click on “**Schemas**” to see existing databases.
* **Opening SQL Scripts in MySQL Workbench**

In this step, you’ll open and run the SQL script to create and load a database:

* **Open SQL Workbench**: Open your MySQL Workbench and go to **File** → **Open SQL Script**.
* **Locate Your File**: Navigate to the folder where your SQL script is stored. You’ll see files like “northwind.sql” or a similar script. Select the script and click **Open**.
* **Run the Script**: Once the script is loaded, you can run it. Press **Ctrl + A** (to select all) and then click the **Execute** button. This will execute all commands in the script. The script will run and show progress in the **Action Output** window. If there are no errors, it means your database was successfully created.
* **Refreshing the Schema:** After running the script, you may need to refresh the schema to see the changes: In your MySQL Workbench, look for the **Schemas** section. Right-click and select **Refresh**. This will show your newly created database, such as “northwind” or any other database defined in your script.
* **Loading Data via Command Prompt**

**For larger datasets**, you might need to load data using the command prompt:

* **Extract the test\_db-master file. Copy and paste it in C:\Program Files\MySQL\MySQL Server 8.0\bin**
* **Navigate to MySQL**: Type Windows + R and type cmd. In the command prompt, navigate to **C:\Program Files\MySQL\MySQL Server 8.0\bin\** or clear the file path in the file exploree and type cmd and hit **Enter**. This will automatically navigate to C:\Program Files\MySQL\MySQL Server 8.0\bin\
* **Commands**:
* C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u root -t -p
* Enter password: \*\*\*\*\*\*\*\*\*\*\*\*\*\*
* Welcome to the MySQL monitor. Commands end with ; or \g.
* Your MySQL connection id is 24
* Server version: 8.0.42 MySQL Community Server - GPL
* Copyright (c) 2000, 2025, Oracle and/or its affiliates.
* Oracle is a registered trademark of Oracle Corporation and/or its
* affiliates. Other names may be trademarks of their respective
* owners.
* Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
* mysql> source employees.sql

This will execute the SQL script from the command line.

* **Creating a Database**

Think of database as the container that holds all your tables.

* **Commands: CREATE** **DATABASE** appclick;
* This creates a new database named “appclick”
* Always end each query with a “**semicolon (;)”**
* After running the query, click the “**refresh** **icon**” in Schemas to see your new database.
* **Using the Database**

You must tell MySQL which database you are working in or just double-click the database name from the sidebar.

* **Commands: USE** appclick;
* **Creating a Table: Staff**

Tables are like Excel sheets with columns and rows.

* **Commands: CREATE TABLE** Staff (

Staff\_id INT,

Staff\_name VARCHAR (50),

Gender CHAR (6),

Department VARCHAR(20),

Hire\_date DATE,

Salary DECIMAL(10,2),

PRIMARY KEY (Staff\_id)

);

**Explanation:**

* **Data types:** define the kind of value a column in a table can hold. They help MySQL understand:
* How much space to reserve for the data
* What kind of operations are allowed (e.g. Math, Sorting)
* How to protect the data from errors (Like adding words together)

|  |  |  |  |
| --- | --- | --- | --- |
| **Categories of Data Types** | **Data Type** | **What it Means** | **When to Use** |
| **NUMERIC TYPE** | **INT** | **Whole numbers only** | **Age, ID, Quantity, Scores** |
| **DECIMAL (X, Y)** | **Number with decimal X=total, Y=decimals** | **Money, Salary, Price** |
| **FLOAT** | **Decimal with floating precision** | **Height, Weight, Scientific values** |
| **DOUBLE** | **Like FLOAT but with more precision** | **Advance science or financial calculations** |
| **TINYINT** | **Very small integers (0-255)** | **Yes/No values (e.g. Is Active =0 or 1)** |
| **STRING TYPE** | **CHAR(N)** | **Fixed-length text (N-exactness)** | **Codes like Y/N, Gender** |
| **VARCHAR(N)** | **Variable- Length text (max N characters)** | **Names, Addresses, Titles** |
| **TEXT** | **Large text blocks** | **Comments, Descriptions, Paragraphs** |
| **DATE AND TIME TYPE** | **DATE** | **Stores only the date (YYYY-MM\_DD)** | **Birthdays, Hire dates** |
| **TIME** | **Stores only time**  **(HH:MM:SS)** | **Event time, Shift time** |
| **DATETIME** | **Stores date and time together** | **Timestamp of a transaction or update** |
| **YEAR** | **Stores only year (YYYY)** | **Year of Admission** |
| **OTHER USEFUL TYPE** | **BOOLEAN** | **True/ False (in MySQL its actually TINYINT)** | **Yes/No status (1=True, 0=False)** |
| **ENUM** | **List of allowed values** | **Gender (‘Male’, ‘Female’), Roles (‘Admin’, ‘User’)** |

* **Verifying Table Structure**

To check what is inside the table:

* **Commands**: **SELECT** \***FROM** Staff;
* **Inserting** **Data**

To add information (a row) into the staff table:

* **Commands**: **INSERT INTO** Staff **VALUES** (

1,

‘John Paul’

‘Male’

‘Developer’

‘2022-11-25’

95000.50

);

* **Creating another Table: Students**
* **Commands: CREATE TABLE** Students (

Student\_id INT,

Student\_name VARCHAR (30),

Gender CHAR (6),

Course VARCHAR (30),

Age INT,

Height FLOAT,

Study\_mode VARCHAR (20),

Staff\_id INT,

PRIMARY KEY (Student\_id),

FOREIGN KEY (Staff\_id) REFERENCES Staff(Staff\_id)

);

* Foreign key is a way to connect tables
* **Adding** **Students**
* **Commands: INSERT INTO** Students **VALUES (**

1,

‘Mary Jane’,

‘Female’,

‘Data Analysis’,

22,

55,

‘Virtual’,

1

);

LECTURE TWO

UNDERSTANDING DATABASE CONSTRAINTS

* **What are Constraints?**

In databases, constraints are like rules that help ensure the “**quality**” **and** ‘**structure**’ of the data. They make sure that the information entered into the database is ‘**valid’** and ‘**consistent**’.

* **Why are Constraints Important?**

If you don’t set rules, people can add data however they like. **For example**, imagine someone says their age is -5 or enters abc for phone number!

By using constraints, we can ‘**restrict**’ what kind of data can be entered and ensure good data quality.

* **Types of Constraints**

1. **Auto** **Increment**

When you need a column (like an ID) to ‘**increase automatically**’ every time a new entry is added. **For example,** you don’t need to manually enter Employee IDs; MySQL will do it for you.

* **Why Use It?**

Help in creating ‘**unique’ IDs** automatically.

1. **Not** **Null**

This constraint ensures that ‘**a column cannot have empty values’.** Think of it as a required field in a form. If a field is marked ‘**Not** **Null’** you can’t leave it blank.

* **Why Use It?**

Ensures important data (like name, email, age) is ‘**always filled in’**.

1. **Unique**

Ensure that all values in a column are ‘**different**’. **For example,** if you’re tracking email addresses, you wouldn’t want two users to have the same email.

* **Why Use It?**

Guarantees that no two rows in a table have the same value for the column.

1. **Check**

This constraint allows you to set ‘**a condition on**’ the values in a column. **For example,** you might want to ensure that only people over 18 can sign up.

* **Why Use It?**

Help enforce rules, like age limits or value ranges.

1. **Default**

This allows you to set a default value for a column if no value is provided. **For example,** if you don’t specify a city, MySQL might automatically set it to ‘**Ibadan**’.

* **Why Use It?**

Ensure columns have default values when no input is given.

1. **Enum**

You can restrict a column to only accept certain ‘**specific values**’. **For example,** you might want to allow only ‘**Male**’ or ‘**Female**’ for a gender column.

* **Why Use It?**

Helps prevent invalid data from entering by limiting the options.

* **Commands: CREATE TABLE** Employee (

Emp\_ID INT AUTO\_INCREMENT,

Emp\_name VARCHAR(50) NOT NULL,

Email VARCHAR(60) UNIQUE,

Age INT CHECK (Age >= 18),

City VARCHAR(50) DEFAULT ‘Ibadan’,

Gender ENUM (‘Male’, ‘Female’)

);

* **Modifying Your Table**

You can modify the structure of your table by adding, changing, or removing columns.

* **Renaming a Table**
* **Commands: ALTER TABLE** Employees

**RENAME TO** Staff;

* **Adding a New Column**
* **Commands: ALTER TABLE** Employee

**ADD COLUMN** Email VARCHAR (100);

* **DELETE FUNCTION removes data, row by row.**
* **Commands: DELETE FROM** Employee

**WHERE** age<18;

* This command deletes only employee whose age is less than 18
* **TRUNCATE FUNCTION deletes all rows in the table and can’t be used with ‘WHERE’ clause.**
* **Commands: TRUNCATE TABLE** Employee;
* This deletes all rows from the employee.
* **NOTE:**
* MySQL doesn't support directly **modifying or dropping** CHECK constraints (at least not in the way it handles other constraints like primary keys).
* If a column in a table is **not a PRIMARY KEY or UNIQUE**, MySQL cannot use it for a foreign key reference.
* **Adding a foreign key does *not* copy or import data** from one table to another. It only sets a **rule** to link them — to make sure values in Employees.DepartmentID match valid IDs in Departments.DepartmentID.
* To **add a foreign key to an already created table** in **MySQL Workbench**
* = (equals) cannot be used with NULL in SQL.
* **IS** **NULL** is the correct way to check for **NULL** values in SQL.
* **Commands: ALTER TABLE** Employee

**ADD COLUMN** Department\_Name

**ADD CONSTRAINT FOREIGN KEY(**Department\_Name)

**REFERENCES** Department **(**DepartmentName**);**

**OR**

**ALTER TABLE** Employee

**ADD CONSTRAINT** Department\_Name /\* Name of the new column\*/

**FOREIGN KEY(**Department\_Name)

**REFERENCES** Department **(**DepartmentName**);**

* **Filtering Data:**

Sometimes, we don’t want everything—just specific information. For example, if you're looking for a particular person, you can use a **filter**. Let's say we want to find the contact name and address of a customer named "Maria Anders." Here’s how you do it:

* **Commands: SELECT** ContactName**,** Address **FROM** Customers

**WHERE** ContactName **=** 'MariaAnders'**;**

Here**, WHERE FUNCTION** helps us narrow down our search. It’s like saying**, "Show me only the books that are written by Shakespeare."**

LECTURE THREE

UNDERSTANDING LOGICAL OPERATORS

In SQL, logical operators help **refine** your search and make your queries more powerful. They allow you to combine multiple conditions in a single query. This is especially useful when you need to filter data based on more than one condition. Logical operators work like the "**rules**" that guide SQL on how to choose the right data. The three most common logical operators in SQL are **AND**, **OR**, and **NOT**.

1. **The AND Operator**

The **AND** operator is used to combine multiple conditions in a query. It ensures that all the conditions must be true for the result to be returned.

**Example:**

Imagine you are looking for a product that is named **"Chai"** and has a **ProductID of 2**. You want both conditions to be true at the same time. The query would look like this:

* **Commands: SELECT** ProductID, ProductName

**FROM** Products

**WHERE** ProductName = 'Chai' **AND** ProductID = 2;

**In this case**:

* The query checks if the product name is **"Chai"**.
* It also checks if the product ID is **2**.

Only products that meet **both** conditions will be returned.

* **Key Point**: The **AND** operator will only return results if **all conditions** are true.

1. **The OR Operator**

The **OR** operator is used when you want any of the conditions to be true. It gives more flexibility than **AND** because only one of the conditions needs to be true for a result to be returned.

**Example:**

Let’s say you want to find products that are either **"Chai"** or have a **ProductID of 2**. The query would look like this:

* **Commands: SELECT** ProductID, ProductName

**FROM** Products

**WHERE** ProductName = 'Chai' **OR** ProductID = 2;

**In this case:**

* The query will return all products where the product name is **"Chai"**.
* It will also return products where the product ID is **2**, even if they don’t have the name "Chai".
* **Key Point**: The **OR** operator will return results if **either condition** is true.

1. **The NOT Operator**

The **NOT** operator is used to exclude certain conditions from the results. It helps us filter out records that do not match the specified condition.

**Example:**

Let’s say you want to find all products **except** those named **"Chai"**. You would use the **NOT** operator to exclude "Chai" from your search:

* **Commands: SELECT** ProductID, ProductName

**FROM** Products

**WHERE** ProductName **NOT** 'Chai';

**OR**

**SELECT** ProductID, ProductName

**FROM** Products

**WHERE** ProductName != 'Chai' ;

In this case:

* The query will return all products except those with the name **"Chai"**.
* **Key Point**: The **NOT** operator helps you **exclude** specific conditions from your results.
* **OTHER EXAMPLES:**
* /\* Write a query that returns regiondescription and regionid that are not in 2 and 4\*/
* **Commands: SELECT** regiondescription, regionid

**FROM** region

**WHERE** regionid **NOT** **IN** (2,4);

LECTURE FOUR

Using Comparison Operators

In addition to logical operators, SQL also uses comparison operators to filter data based on values.

**Comparison Operators:**

* **=** (Equals)
* **<>** or **!=** (Not equals)
* **>** (Greater than)
* **<** (Less than)
* **>=** (Greater than or equal to)
* **<=** (Less than or equal to)

**Example:** If we want to find employees with a salary greater than or equal to $40,000, we can write:

* **Commands: SELECT** Name, Salary

**FROM** Employees

**WHERE** Salary >= 40000;

* This query will return all employees whose salary is **greater than or equal to $40,000**.
* **Using the IN Operator**

The **IN** operator helps you match a value against a list of values, making your query cleaner when you want to check multiple values.

**Example:** Let’s say we want to find employees who live in **London**, **Berlin**, or **New York**:

* **Commands: SELECT** Name, City

**FROM** Employees

**WHERE** City **IN** ('London', 'Berlin', 'New York');

* This query will return employees who live in **any** of the three cities listed.
* **Using the BETWEEN Operator**

The **BETWEEN** operator allows you to filter results based on a range of values.

**Example:** If we want to find employees who have worked between 2015 and 2020, we can write:

* **Commands: SELECT** Name, HireDate

**FROM** Employees

**WHERE** HireDate **BETWEEN** '2015-01-01' **AND** '2020-12-31';

* This query will return employees whose **hire date** falls between January 1, 2015, and December 31, 2020.
* **Combining Logical Operators**

You can combine **AND**, **OR**, and **NOT** to create more complex queries.

**Example:** If we want to find employees who either work in the "Sales" department **and** have a salary greater than $50,000, **or** work in the "Marketing" department:

* **Commands: SELECT** Name, Department, Salary

**FROM** Employees

**WHERE** (Department = 'Sales' **AND** Salary > 50000)

**OR** Department = 'Marketing';

* This query will return employees who meet either of the two conditions:
  + They work in "Sales" and earn more than $50,000.
  + They work in "Marketing."
* **Using Aggregate Functions in SQL**

SQL also provides aggregate functions to perform calculations on data across multiple rows. The most commonly used aggregate functions are:

* **COUNT()**: Returns the number of rows.
* **SUM()**: Returns the total sum of a numeric column.
* **ROUND ():** Returns a number that is rounded to the number of decimal places you specify**.**
* **AVG()**: Returns the average of a numeric column.
* **MAX()**: Returns the maximum value in a column.
* **MIN()**: Returns the minimum value in a column.

**Example:** If we want to count how many employees there are in total:

* **Commands: SELECT** **COUNT**(\*)

**FROM** Employees;

* This query will return the total number of employees in the "Employees" table.
* **Using Modulo in SQL**

Modulo, represented by the **%** operator, returns the remainder of a division operation. It is often used to filter results based on whether numbers are divisible by a specific value.

**Example:** If we want to find employees whose employee ID is divisible by 2 (even employee IDs):

* **Commands: SELECT** Name, EmployeeID

**FROM** Employees

**WHERE** EmployeeID % 2 = 0;

**OR**

**SELECT** Name, EmployeeID

**FROM** Employees

**WHERE** **MOD (**EmployeeID, 2) = 0;

* This query will return all employees with **even** employee IDs. The modulo operation ensures that only employees with IDs divisible by 2 are included in the results.
* **Concatenating Data in SQL**

Sometimes, you might want to combine two or more columns into a single output. This is called concatenation.

**Example:** To display both the **first name** and **last name** of an employee in one column:

* **Commands: SELECT** **CONCAT** (FirstName, ' ', LastName) **AS** EmployeeName

**FROM** Employees;

**OR**

**SELECT** **CONCAT** \_**WS** (‘ ’, FirstName, LastName) **AS** EmployeeName

**FROM** Employees;

* This query will combine the **first name** and **last name** of each employee, separated by a space, and display it as a new column called "EmployeeName."
* **OTHER EXAMPLES**:
* Write a query that returns full information about employees
* **Commands: SELECT** **CONCAT** (firstname, ' ', lastname, ' is a ', title, ' and was hired on ', hiredate) **AS** Info

**FROM** employees;

* You can also use **TRUNCATE** to **kind of** round off values:
* **Commands: SELECT** **ROUND**(SUM(Unitprice \* Quantity), 2) **AS** Totalsales

**FROM** `order details`;

**OR**

**SELECT** **TRUNCATE**(**SUM**(Unitprice \* Quantity),2) **AS** Totalsales

**FROM** `order details`;

LECTURE FIVE

GROUP BY AND AGGREGATION

In SQL, when you're working with large amounts of data, you often need to **group similar values** and **perform calculations** on those groups. This is where **GROUP BY** and **aggregate functions** like COUNT, SUM, AVG, MIN, and MAX come in.

Imagine you're working with an employee database and you want to find:

* How many employees work in each department
* The average salary per department
* The total revenue generated by each city

To do all these, you need to group the data logically and apply some form of calculation—that’s what aggregation is all about.

1. **What is GROUP BY in SQL?**

**GROUP BY** is a clause used to **group rows that have the same values** in specified columns into summary rows. You use it with **aggregate functions** to perform calculations for each group.

* **Syntax:**

SELECT column\_name, AGGREGATE\_FUNCTION(column\_name)

FROM table\_name

GROUP BY column\_name;

* **Example:**

SELECT department, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department;

**This query shows the number of employees in each department**.

1. **Common Aggregate Functions and How They Differ**

**Aggregate functions** perform calculations on a set of values and return a single value.

**a. COUNT()**

* **Use**: Counts the number of rows or non-null values.
* **Example**: COUNT(\*) counts all rows, while COUNT(salary) counts only non-null salaries.
* **Syntax:**

SELECT department, COUNT(\*)

FROM employees

GROUP BY department;

**b. SUM()**

* **Use**: Adds up numeric values in a column.
* **Example**: Total salary paid per department.
* **Syntax:**

SELECT department, SUM(salary)

FROM employees

GROUP BY department;

**c. AVG()**

* **Use**: Returns the average (mean) of numeric values.
* **Example**: Average salary per department.
* **Syntax:**

SELECT department, AVG(salary)

FROM employees

GROUP BY department;

**d. MIN() and MAX()**

* **Use**: Find the minimum or maximum value in a column.
* **Example**: Find the highest salary per department.
* **Syntax:**

SELECT department, MAX(salary) FROM employees GROUP BY department;

1. **WHERE vs HAVING**

* **WHERE Clause**

Used to filter **rows before** grouping happens.

Cannot be used with aggregate functions.

* **Syntax:**

SELECT \* FROM employees

WHERE salary > 50000;

* **HAVING Clause**

Used to filter **groups after** aggregation.

Works with aggregate functions.

* **Syntax:**

SELECT department, AVG(salary)

FROM employees

GROUP BY department

HAVING AVG(salary) > 60000;

1. **ORDER BY and LIMIT with GROUP BY**

You can sort and limit your grouped results for better insights.

* **ORDER BY**

Sorts results either in ascending (default) or descending order.

* **Syntax:**

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

ORDER BY total\_salary DESC;

* **LIMIT**

Restricts the number of results returned.

* **Syntax:**

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

ORDER BY total\_salary DESC

LIMIT 3;

This gives you the top 3 departments with the highest salary totals.

1. **DISTINCT Values with GROUP BY**

You can use DISTINCT to remove duplicates before or after grouping if needed.

* **Syntax:**

SELECT DISTINCT department, salary

FROM employees

WHERE salary > 50000;

Or combine GROUP BY to find unique patterns in grouped data.

LECTURE SIX

JOIN

In SQL, a **JOIN** is used to combine rows from two or more tables based on a related column between them. It's extremely useful when your data is split across multiple tables and you need to pull it together for analysis or reporting.

Imagine two spreadsheets:

* One has **employee details** (name, ID, department ID)
* The other has **department details** (department ID, department name)

To see which employee belongs to which department, you need to "join" these tables using their common field — the **department ID**.

TYPES OF JOIN

1. **INNER JOIN**

* **What it does:**  
  Returns only the rows where there is a match in both tables.
* **Example Scenario:**  
  Imagine you have two tables:

**Customers**

|  |  |
| --- | --- |
| **CustomerID** | **Name** |
| 1 | James |
| 2 | Ada |
| 3 | Musa |

**Orders**

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerID** | **Product** |
| 101 | 1 | Tyres |
| 102 | 3 | Engine Oil |
| 103 | 4 | Battery |

**Query:**

SELECT Customers.Name, Orders.Product

FROM Customers

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

**Result:**

|  |  |
| --- | --- |
| **Name** | **Product** |
| James | Tyres |
| Musa | Engine Oil |

* **Explanation:**  
  Only customers with matching orders appear. Ada is left out because she has no order. Order 103 is also excluded because there is no customer with ID 4.

1. **LEFT JOIN (or LEFT OUTER JOIN)**

* **What it does:**  
  Returns all rows from the left (first) table, and matched rows from the right table. If there’s no match, you get NULL.

**Query:**

SELECT Customers.Name, Orders.Product

FROM Customers

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

**Result:**

|  |  |
| --- | --- |
| **Name** | **Product** |
| James | Tyres |
| Ada | NULL |
| Musa | Engine Oil |

* **Explanation:**  
  Everyone from the Customers table shows up. Ada is included, even though she didn’t order anything.

1. **RIGHT JOIN (or RIGHT OUTER JOIN)**

* **What it does:**  
  Returns all rows from the right (second) table, and the matching rows from the left table.

**Query:**

SELECT Customers.Name, Orders.Product

FROM Customers

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

**Result:**

|  |  |
| --- | --- |
| **Name** | **Product** |
| James | Tyres |
| Musa | Engine Oil |
| NULL | Battery |

* **Explanation:**  
  Everyone from the Orders table shows up. The order for customer ID 4 doesn’t match anyone, so the Name is NULL.

1. **CROSS JOIN**

* **What it does:**  
  Returns every combination of rows between two tables. This is called a **Cartesian product**.
* **Example Scenario:**  
  **Colors**

|  |  |
| --- | --- |
| **ColorID** | **Color** |
| 1 | Red |
| 2 | Blue |

**Sizes**

|  |  |
| --- | --- |
| **SizeID** | **Size** |
| 1 | Small |
| 2 | Large |

**Query:**

SELECT Colors.Color, Sizes.Size

FROM Colors

CROSS JOIN Sizes;

**Result:**

|  |  |
| --- | --- |
| **Color** | **Size** |
| Red | Small |
| Red | Large |
| Blue | Small |
| Blue | Large |

* **Explanation:**  
  Every color is paired with every size.

1. **SELF JOIN**

* **What it does:**  
  A table joins with itself. Useful when comparing rows within the same table.
* **Example Scenario:**  
  **Employees**

| EmpID | Name | ManagerID |
| --- | --- | --- |
| 1 | John | NULL |
| 2 | Sarah | 1 |
| 3 | David | 1 |

**Query:**

SELECT E.Name AS Employee, M.Name AS Manager

FROM Employees E

LEFT JOIN Employees M ON E.ManagerID = M.EmpID;

**Result:**

| **Employee** | **Manager** |
| --- | --- |
| John | NULL |
| Sarah | John |
| David | John |

* **Explanation:**  
  We are comparing each employee with their manager in the same table. A self join helps visualize internal relationships like mentorship or reporting lines.

1. **FULL OUTER JOIN:**

A **FULL OUTER JOIN** combines the results of:

* **LEFT JOIN** → all rows from the left table, even if no match
* **RIGHT JOIN** → all rows from the right table, even if no match

In short, it returns:

* All matching rows
* All unmatched rows from the left table (with NULLs on the right side)
* All unmatched rows from the right table (with NULLs on the left side)
* **Example Scenario**

Let’s say we have:

**Customers**

| **CustomerID** | **Name** |
| --- | --- |
| 1 | James |
| 2 | Ada |
| 3 | Musa |

**Orders**

| **OrderID** | **CustomerID** | **Product** |
| --- | --- | --- |
| 101 | 1 | Tyres |
| 102 | 3 | Engine Oil |
| 103 | 4 | Battery |

**Query:**

SELECT c.Name, o.Product

FROM Customers c

FULL OUTER JOIN Orders o ON c.CustomerID = o.CustomerID;

**Result:**

| **Name** | **Product** |
| --- | --- |
| James | Tyres |
| Ada | NULL |
| Musa | Engine Oil |
| NULL | Battery |

* James and Musa → matched
* Ada → no orders → Product = NULL
* OrderID 103 (CustomerID 4) → no matching customer → Name = NULL

Not supported in some databases → Some systems (like MySQL before version 8) don’t have FULL OUTER JOIN. You may have to **simulate it** using UNION:

SELECT ... **FROM A** LEFT JOIN B ...

UNION

SELECT ... **FROM A** RIGHT JOIN B ...

LECTURE SEVEN

SUBQUERIES, CASE STATEMENTS, TEXT FUNCTIONS, AND VIEWS IN MySQL

1. **SUBQUERIES**

A subquery is a query inside another SQL query. It's used to fetch data that will be used by the main (outer) query.

* **When to Use:**
* When you need to **compare each row** in a table with a calculated value.
* When filtering rows **based on data in another table**.
* When you're performing **aggregations** or calculations that need to be reused immediately

.

* **Example:**

Find employees who earn more than the average salary.

* **Query:**

SELECT name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

Here, the subquery (SELECT AVG(salary)...) gets the average salary. The main query uses it to filter.

* **Where Subqueries Can Be Used:**
* **In the WHERE clause**: To filter rows.
* **In the SELECT clause**: To show extra calculated values.
* **In the FROM clause**: As a temporary table.
* **Take Note:**
* Subqueries should return a **single value** when used with comparison operators (=, >, <, etc.).
* For multiple results, use IN, ANY, or EXISTS.
* They can **slow down performance** in large databases—so use them wisely.

1. **CASE STATEMENTS**

CASE is SQL's way of saying "if-else." It helps you **group, label, or conditionally transform data** in a column.

* **When to Use:**
* When you want to **categorize numeric or text data**.
* When you need to **assign new values** based on existing column data.
* **Example:**

Categorize customers by the number of items they ordered.

* **Query:**

SELECT customer\_name, quantity\_ordered,

CASE

WHEN quantity\_ordered > 100 THEN 'VIP'

WHEN quantity\_ordered BETWEEN 50 AND 100 THEN 'Regular'

ELSE 'New'

END AS customer\_type

FROM orders;

* **Where You Can Use It:**
* In the **SELECT clause**: To show custom labels.
* In **ORDER BY**: To sort based on conditions.
* In **WHERE** (less common but possible).
* **Take Note:**
* CASE is checked **top to bottom**, and stops when it finds a match.
* Always include an ELSE clause to **handle unmatched cases**.
* It can be nested inside other functions or used in calculations.

1. **TEXT FUNCTIONS (String Functions)**

These are built-in tools in SQL to **manipulate, format, or extract parts of text** (also called "strings").

* **Common Examples:**
* UPPER(text): Converts text to **all uppercase**  
  UPPER('hello') → 'HELLO'
* LOWER(text): Converts text to **all lowercase**  
  LOWER('HELLO') → 'hello'
* SUBSTRING(text, start, length): Extracts part of a string  
  SUBSTRING('Customer2025', 1, 8) → 'Customer'
* **When to Use:**
* To **standardize data** (e.g., making names or emails lowercase).
* When performing **searches or comparisons** (e.g., case-insensitive matches).
* When you need to **extract parts** of text (like a product code, year, etc.).
* **Take Note:**
* UPPER and LOWER do not affect numbers or special characters.
* SUBSTRING starts counting from **1**, not 0.
* Useful when working with **usernames, email fields, product names, or formatted data**.

1. **VIEWS**

A view is a **stored SQL query**. It behaves like a virtual table that shows data based on a defined query. Think of a view like a **shortcut** to a frequently used or complex query.

* **When to Use:**
* When you're **repeating the same query** many times.
* When you want to **hide some columns or simplify** access for users.
* When you want to create a **readable version of a complex join or filter**.
* **Example:**

Create a view of high earners:

* **Query:**

CREATE VIEW high\_earners AS

SELECT name, salary

FROM employees

WHERE salary > 50000;

Now instead of writing that query every time, just do:

* **Query:**

SELECT \* FROM high\_earners;

* **Take Note:**
* A **view does not store actual data**—it shows data live from the original tables.
* If the underlying data changes, the view reflects it.
* **You can’t always edit** data through a view (especially if it involves joins or calculations).
* Use clear, meaningful names for views.

LECTURE EIGHT

Understanding CTEs, Views, and Window Functions

* **What is a Common Table Expression (CTE)?**

A **CTE** is a temporary, named result set (like a virtual table) used within a SQL query. Think of it as a way to **break a complex query into simpler parts**.

* **Why Use CTEs?**
* Makes queries **easier to read and maintain**.
* Helps you **organize logic** when you’re working with multiple layers of calculations.
* **Does not consume permanent storage space**—it exists only while the query runs.
* **Syntax of a CTE:**

WITH cte\_name AS (

SELECT columns

FROM table

WHERE condition

)

SELECT \* FROM cte\_name;

* **Practical Example of a CTE**

**Get customer details and calculate total price per order.**

* **Query:**

WITH CustomerOrder AS (

SELECT

customer\_id,

customer\_name,

order\_id,

quantity,

unit\_price,

quantity \* unit\_price AS total\_price

FROM orders

)

SELECT \* FROM CustomerOrder;

* **Explanation:**
* CustomerOrder is the CTE.
* It selects data from the orders table, including a calculated column total\_price.
* You can then use CustomerOrder in your main query like a normal table.
* **Note: The reason you're required to use aliases for your columns in the CTE is not because the CTE *must* have aliases per se, but because your main query depends on those column names.**
* **Query:**

WITH PriceInfo AS(

SELECT O.Customerid AS CustomerID, O.Orderid AS OrderID, OD.UnitPrice AS Price,

OD.Quantity AS Quantity, (OD.UnitPrice \* OD.Quantity) AS TotalAmount

FROM Orders O

JOIN `Order Details` OD ON O.OrderID = OD.OrderID)

SELECT CustomerID, OrderID, TotalAmount,

CASE

WHEN TotalAmount > 150 THEN 'Better Person'

WHEN TotalAmount < 100 THEN 'You no try at all'

ELSE 'You dey try'

END AS CustomerDescription

FROM PriceInfo;

* **How is a CTE Different from a View?**

| **Feature** | **CTE** | **View** |
| --- | --- | --- |
| **Storage** | Temporary, in-memory | Stored permanently in the database |
| **Reusability** | One-time use in a single query | Can be reused in multiple queries |
| **Performance** | Efficient for complex logic once | Better for repeated logic or dashboards |
| **Editing** | Cannot be indexed or updated | Can sometimes be indexed/updated depending on DBMS |

* **Introduction to Window Functions**

**Window functions** perform calculations across a **set of rows related to the current row**, without collapsing rows like GROUP BY does.

* **Syntax:**

Function\_name(column) OVER (

PARTITION BY column\_to\_group

ORDER BY column\_to\_sort

)

* **Types of Window Functions (with Examples)**

1. **ROW\_NUMBER()**

Gives a unique row number to each row within a partition.

* **Syntax:**

SELECT

customer\_id,

order\_id,

ROW\_NUMBER() OVER (PARTITION BY customer\_id ORDER BY order\_date) AS row\_num

FROM orders;

* **Use Case:** Find the first order of every customer.

1. **RANK()**

Ranks rows within a partition, with **gaps** if there are ties.

* **Syntax:**

SELECT

product\_id,

price,

RANK() OVER (PARTITION BY category\_id ORDER BY price DESC) AS price\_rank

FROM products;

* **Use Case:** Rank products by price in each category.

1. **DENSE\_RANK()**

Same as RANK() but **no gaps** in ranking.

* **Syntax:**

SELECT

employee\_id,

department\_id,

salary,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_rank

FROM employees;

* **Use Case:** Rank employees' salaries per department, without skipping ranks.

1. **NTILE(n)**

Divides the result set into n **equal parts or tiles**.

* **Syntax:**

SELECT

student\_id,

score,

NTILE(4) OVER (ORDER BY score DESC) AS performance\_quartile

FROM students;

* **Use Case:** Group students into 4 performance bands (top 25%, next 25%, etc.).

1. **LAG() and LEAD()**

These retrieve values from previous or next rows.

* **LAG Example:**
* **Syntax:**

SELECT

employee\_id,

hire\_date,

LAG(hire\_date) OVER (ORDER BY hire\_date) AS previous\_hire

FROM employees;

* **Use Case:** Find the hire date of the employee just before the current one.
* **LEAD Example:**
* **Syntax:**

SELECT

employee\_id,

hire\_date,

LEAD(hire\_date) OVER (ORDER BY hire\_date) AS next\_hire

FROM employees;

* **Use Case:** Find the next hire after each employee.

1. **SUM() as a Window Function**

Calculates running totals or totals over a window without collapsing rows.

* **Syntax:**

SELECT

customer\_id,

order\_id,

order\_amount,

SUM(order\_amount) OVER (PARTITION BY customer\_id ORDER BY order\_date) AS running\_total

FROM orders;

* **Use Case:** Track cumulative spending per customer over time.

1. **AVG(), MIN(), MAX() as Window Functions**

* just like aggregate functions but maintains the row structure.
* **Syntax:**

SELECT

department\_id,

employee\_id,

salary,

AVG(salary) OVER (PARTITION BY department\_id) AS avg\_salary

FROM employees;

* **Use Case:** Show each employee’s salary and the average salary in their department.
* **Total Sales Example Using Window Function**
* **Syntax:**

SELECT

region,

salesperson\_id,

sales\_amount,

SUM(sales\_amount) OVER (PARTITION BY region) AS total\_region\_sales

FROM sales\_data;

* **Explanation:**
* This shows each salesperson’s sale.
* It also shows total sales for the region they belong to—on every row.

LECTURE NINE

STORED PROCEDURE

* **What Are Stored Procedures?**

A **stored procedure** is a precompiled collection of SQL statements stored in your database. Think of it like a reusable script or a "function" that performs specific tasks (e.g., calculating data, updating records). Once created, you can call it anytime without rewriting the code.

* **Why Use Stored Procedures?**
* **Reusability**: Run the same logic across multiple queries/applications.
* **Efficiency**: Reduces network traffic (send one command instead of multiple queries).
* **Security**: Control data access by restricting direct table access.
* **Maintenance**: Update the procedure once to apply changes everywhere.
* **Create a stored procedure to get all customers**
* **QUERY:**

DELIMITER //

CREATE PROCEDURE PrintCustomerList()

BEGIN

SELECT \*FROM Customers;

END //

DELIMITER ;

* **TO USE:**

CALL PrintCustomerList();

* **Create a stored procedure that takes a customerid as input and retrieves all details about that customer from the customers table**
* **QUERY:**

DELIMITER //

CREATE PROCEDURE CustomerInfo( IN ID VARCHAR(10))

BEGIN

SELECT \*FROM Customers

WHERE CustomerID = ID;

END //

DELIMITER ;

* **TO USE:**

CALL CustomerInfo('AROUT');

CALL CustomerInfo('QUICK');

* **Create a stored procedure that calculates the total number of orders placed by a specific customer and returns it via an output parameter**
* **QUERY:**

DELIMITER //

CREATE PROCEDURE OrderInfo(IN Customer\_ID VARCHAR(10), OUT Order\_Count INT)

BEGIN

SELECT COUNT(\*) INTO Order\_Count

FROM Orders

WHERE CustomerID = Customer\_ID;

END //

DELIMITER;

* **TO USE**:

CALL OrderInfo('QUICK', @Order\_Count);

SELECT @order\_Count;