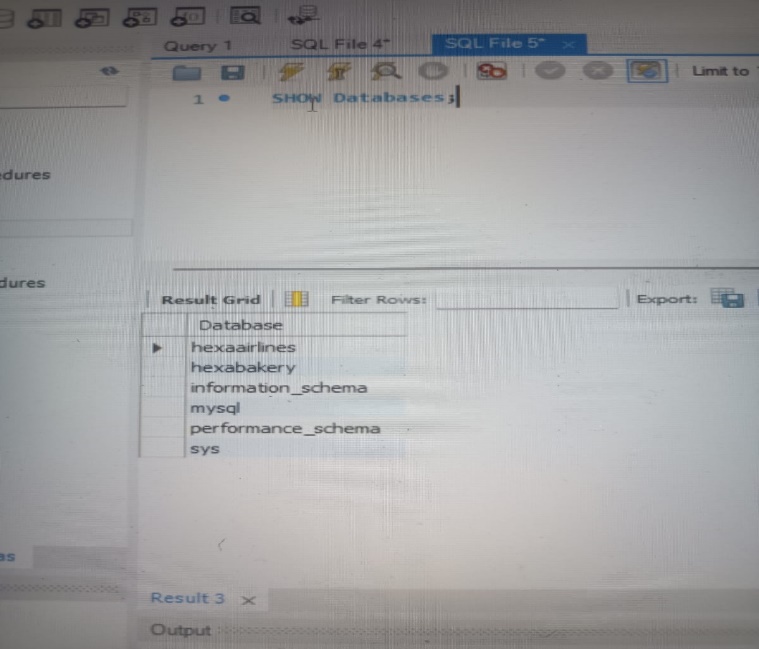
## ASSIGNMENT 1 – DATE: 11-06-2025

### SECTION A

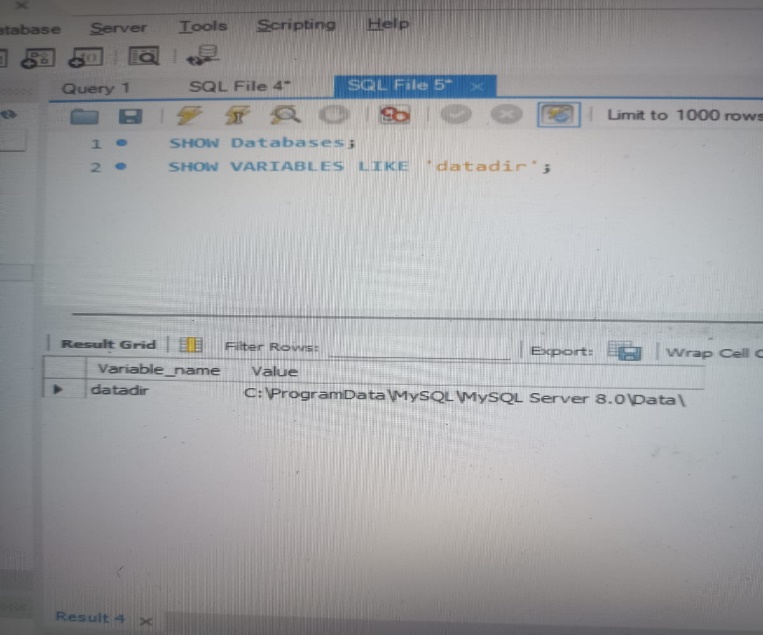
1. List all system databases in SQL Server

SHOW Databases;



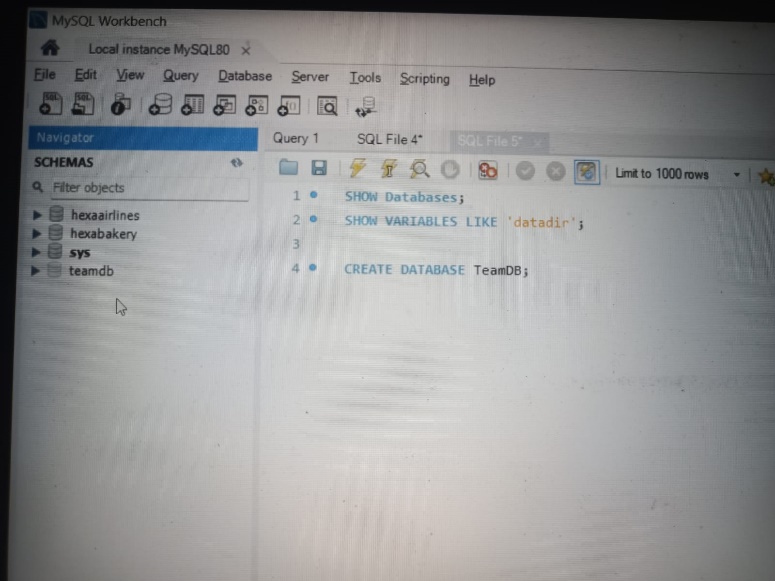
1. List physical file paths for all databases

SHOW VARIABLES LIKE 'datadir';



1. Create a new user-defined database named TeamDB

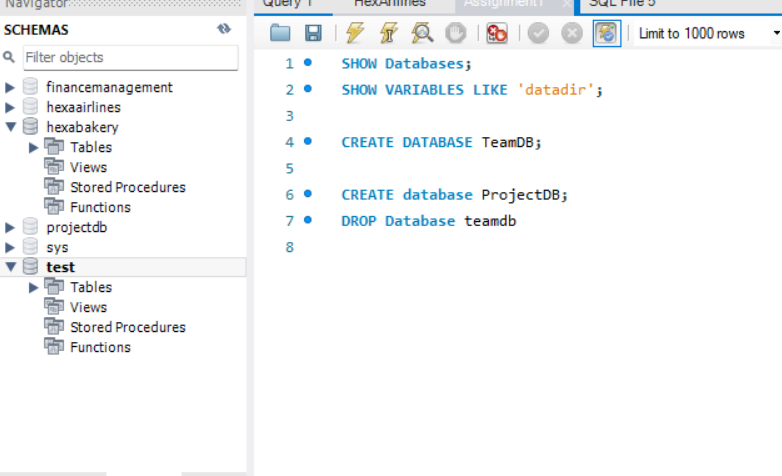
CREATE DATABASE TeamDB;



1. Rename the database TeamDB to ProjectDB.

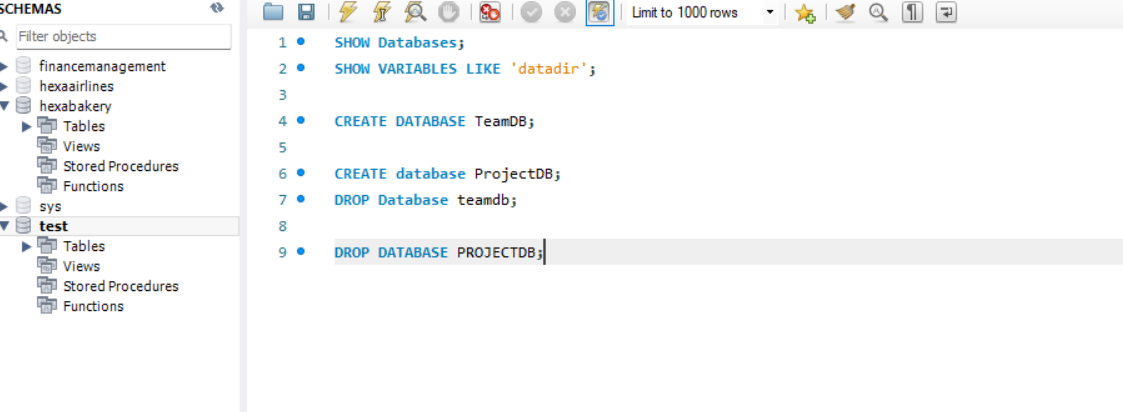
CREATE database ProjectDB;

DROP Database teamdb;



1. Drop the ProjectDB database

DROP DATABASE PROJECTDB;



### MANAGING TABLES (SECTION B)

1. Create a table Employees with the following columns

CREATE DATABASE ASSIGNMENT1;

USE ASSIGNMENT1;

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

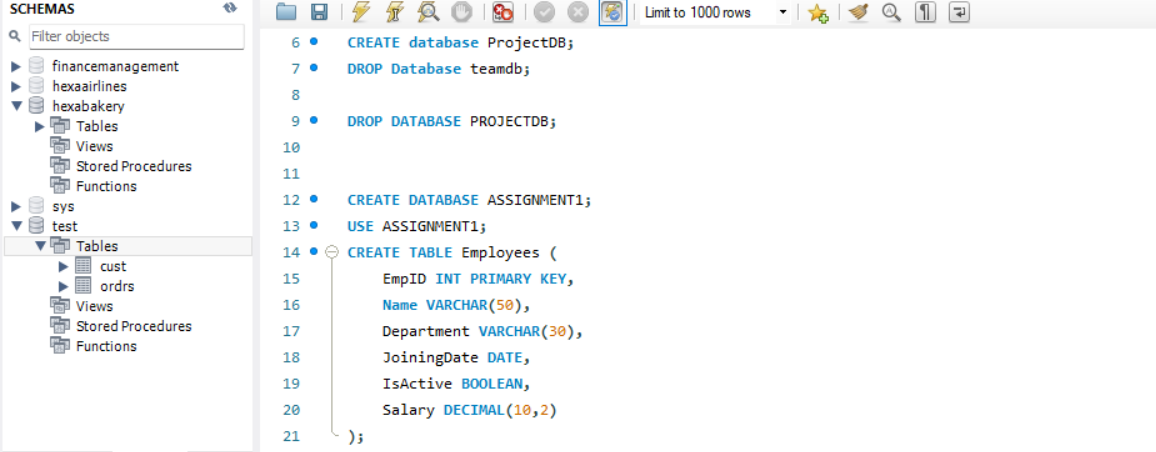
Department VARCHAR(30),

JoiningDate DATE,

IsActive BOOLEAN,

Salary DECIMAL(10,2)

);



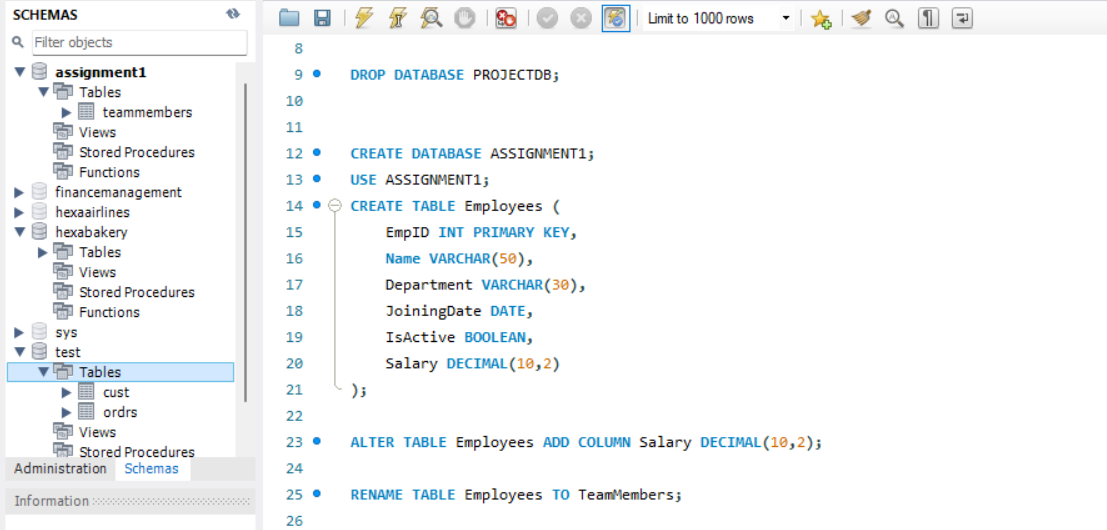
1. Add a column Salary (DECIMAL) to the table.

ALTER TABLE Employees ADD COLUMN Salary DECIMAL(10,2);



1. Rename table Employees to TeamMembers.

RENAME TABLE Employees TO TeamMembers;



1. Drop the table TeamMembers

DROP TABLE TeamMembers;



### DML OPERATIONS (SECTIONS C)

1. Insert three rows into Employees.

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Department VARCHAR(30),

JoiningDate DATE,

IsActive BOOLEAN,

Salary DECIMAL(10,2)

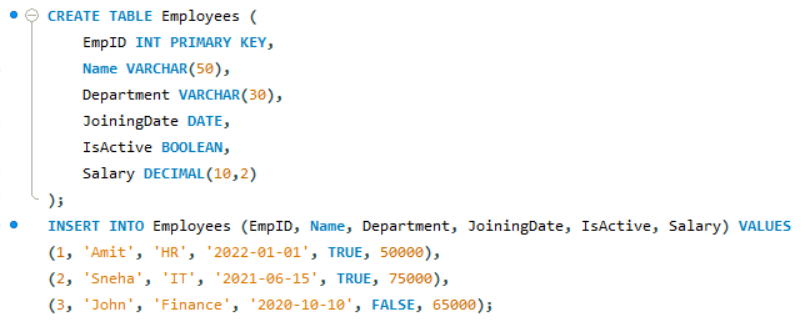
);

INSERT INTO Employees (EmpID, Name, Department, JoiningDate, IsActive, Salary) VALUES

(1, 'Amit', 'HR', '2022-01-01', TRUE, 50000),

(2, 'Sneha', 'IT', '2021-06-15', TRUE, 75000),

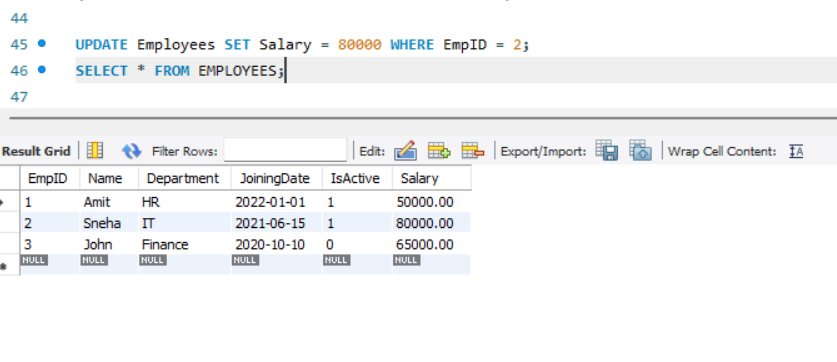
(3, 'John', 'Finance', '2020-10-10', FALSE, 65000);



1. Update salary of 'Sneha' to 80000

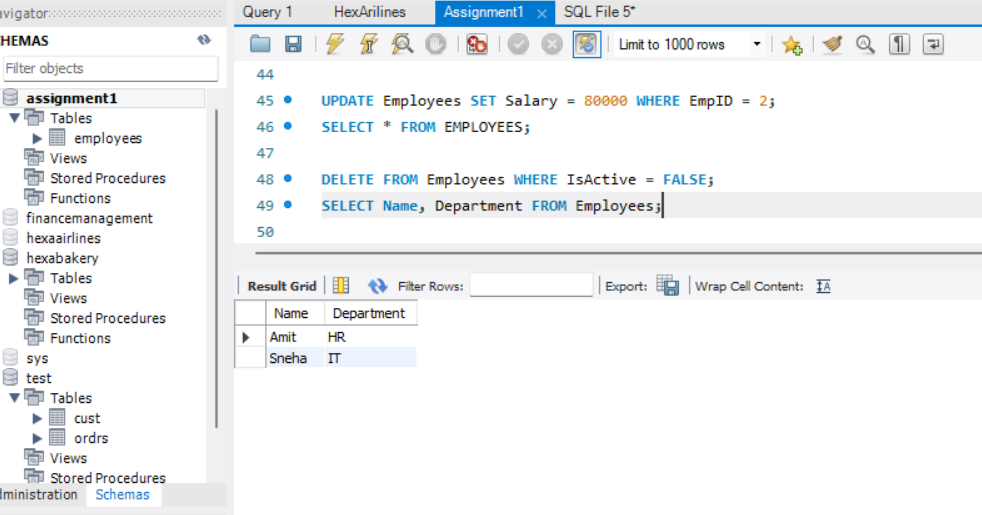
UPDATE Employees SET Salary = 80000 WHERE EmpID = 2;

SELECT \* FROM EMPLOYEES;



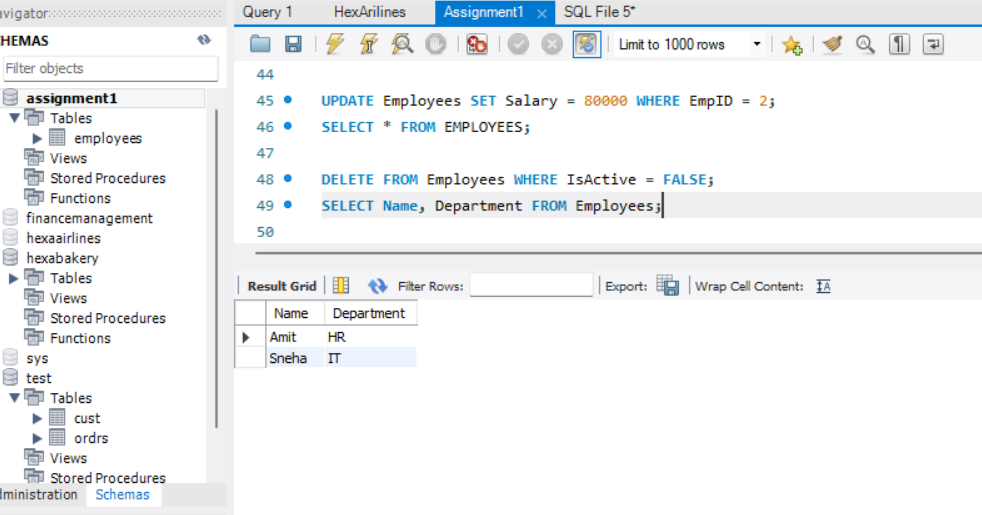
1. Delete employee with IsActive = 0

DELETE FROM Employees WHERE IsActive = FALSE;



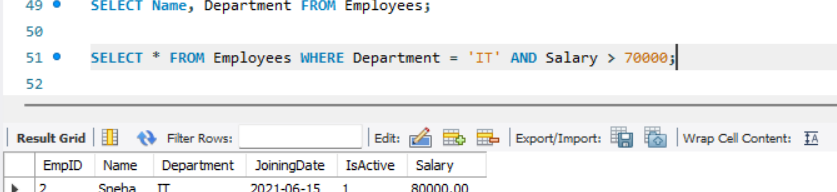
1. Retrieve the name and department of all employee’s

SELECT Name, Department FROM Employees;



1. Fetch employees from 'IT' department with salary above 70000

SELECT \* FROM Employees WHERE Department = 'IT' AND Salary > 70000;

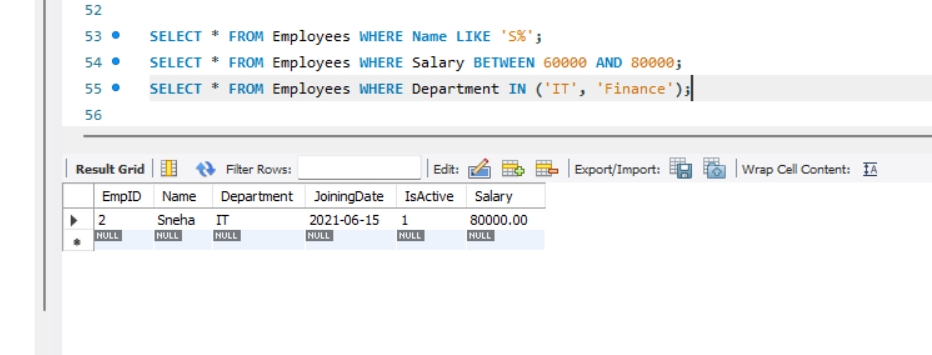


1. Apply filtering using LIKE, BETWEEN, and IN.

SELECT \* FROM Employees WHERE Name LIKE 'S%';

SELECT \* FROM Employees WHERE Salary BETWEEN 60000 AND 80000;

SELECT \* FROM Employees WHERE Department IN ('IT', 'Finance');



# ASSIGNMENT 2 – DATE: 12-06-2025

1. Inset and Update with Integrity: Create a 'students' table with constraints (NOT NULL, UNIQUE). Insert 5 records. Then, update a student's marks ensuring data integrity is maintained.

CREATE DATABASE ASSIGNMENT2;

USE ASSIGNMENT2;

CREATE TABLE STUDENTS(

studentID INT PRIMARY KEY AUTO\_INCREMENT,

Name varchar(30) NOT NULL,

Marks INT NOT NULL,

email VARCHAR(100) UNIQUE

);

INSERT INTO STUDENTS VALUES

(1,'Anas',90, 'anas@gmail.com'),

(2,'Jothi',80,'jothi@gmail.com'),

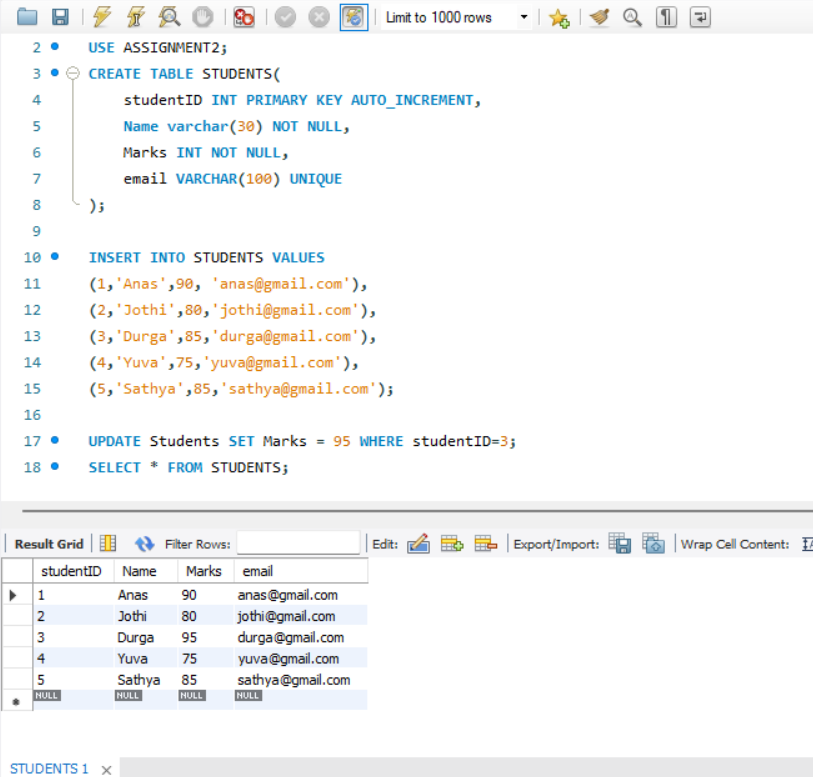
(3,'Durga',85,'durga@gmail.com'),

(4,'Yuva',75,'yuva@gmail.com'),

(5,'Sathya',85,'sathya@gmail.com');

UPDATE Students SET Marks = 95 WHERE studentID=3;

SELECT \* FROM STUDENTS;



2. String Function Challenge: Given a 'customers' table with a 'full\_name' column, write a query to display: - First name - Last name - Length of each name

CREATE TABLE CUSTOMERS(

fullname varchar(50));

INSERT INTO CUSTOMERS VALUES

('Anas Muhammad'),

('Abay Ragav'),

('Afeef Abdulla'),

('Deepak t'),

('Prem Kumar'),

('Loganathan');

SELECT

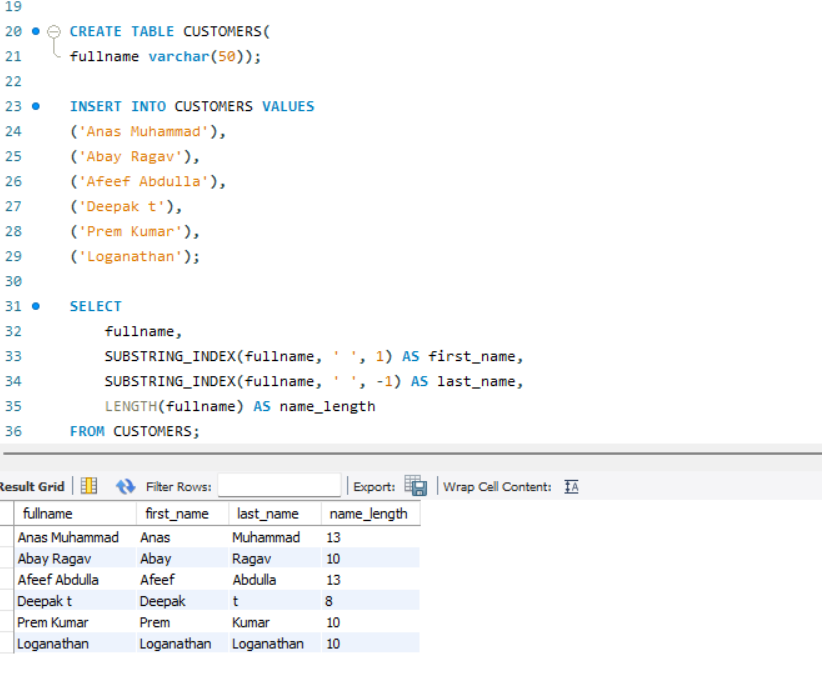
fullname,

SUBSTRING\_INDEX(fullname, ' ', 1) AS first\_name,

SUBSTRING\_INDEX(fullname, ' ', -1) AS last\_name,

LENGTH(fullname) AS name\_length

FROM CUSTOMERS;



3. Date Function Usage: From a 'sales' table with a 'sale\_date' column, write a query to: - Extract the month name and year - Display how many days ago the sale happened

CREATE TABLE sales (

sale\_id INT PRIMARY KEY,

sale\_date DATE

);

INSERT INTO sales VALUES

(1, '2024-05-10'),

(2, '2023-11-15'),

(3, '2025-01-01'),

(4, CURDATE());

SELECT

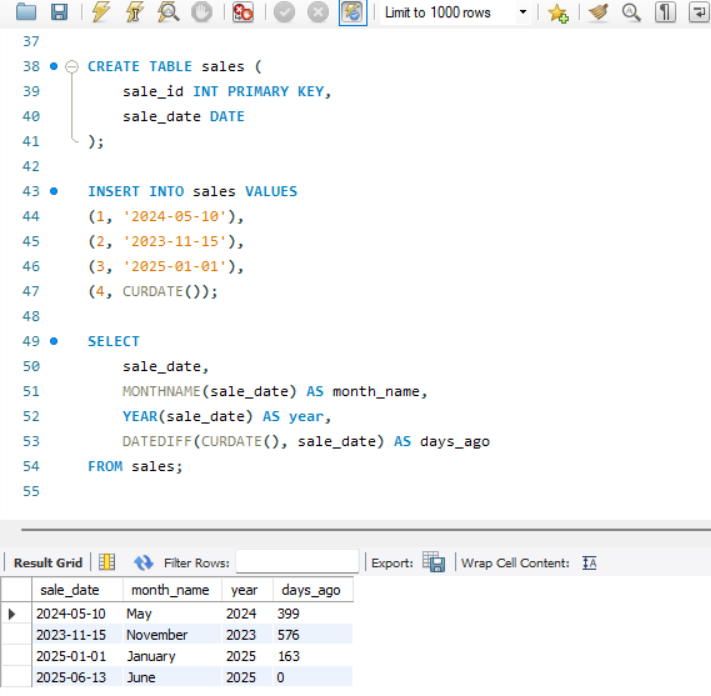
sale\_date,

MONTHNAME(sale\_date) AS month\_name,

YEAR(sale\_date) AS year,

DATEDIFF(CURDATE(), sale\_date) AS days\_ago

FROM sales;



4. Mathematical Functions on Salary: In an 'employees' table, calculate: - Salary after a 10% hike - Round the salary to the nearest hundred

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

name VARCHAR(50),

salary DECIMAL(10,2)

);

INSERT INTO employees VALUES

(1, 'Anas', 52000),

(2, 'Sneha', 65000),

(3, 'Ravi', 45000);

SELECT \* FROM EMPLOYEES;

SELECT

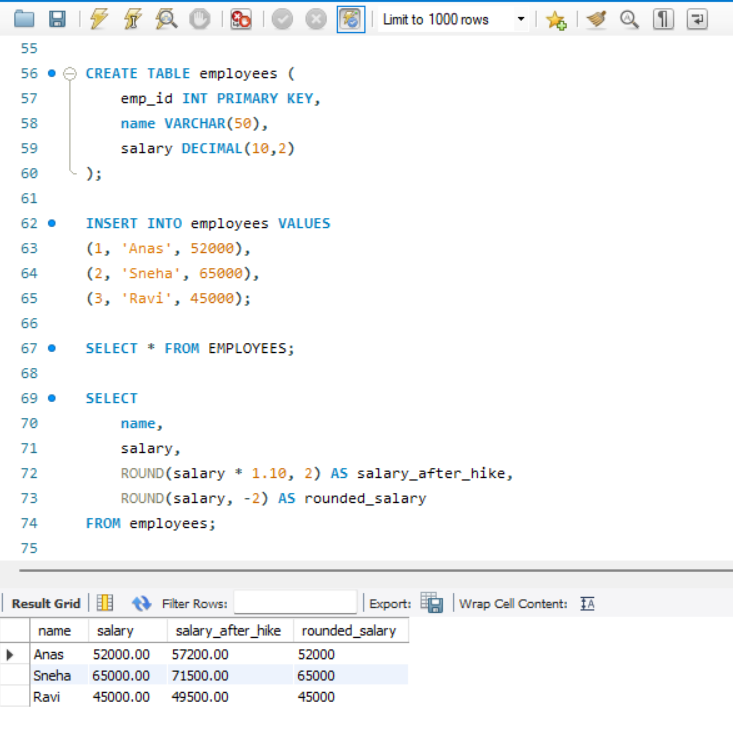
name,

salary,

ROUND(salary \* 1.10, 2) AS salary\_after\_hike,

ROUND(salary, -2) AS rounded\_salary

FROM employees;



5. System Function Check: Retrieve: - Current date and time - Database name and logged-in user

SELECT

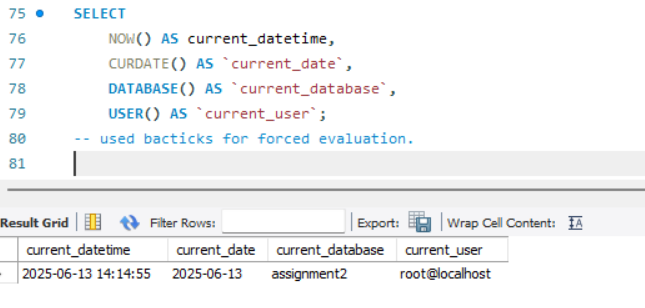
NOW() AS current\_datetime,

CURDATE() AS `current\_date`,

DATABASE() AS `current\_database`,

USER() AS `current\_user`;

-- used bacticks for forced evaluation.



6. Demo: Custom Result Set: From the 'products' table, write a query that: - Returns product name in uppercase - Replaces any NULL prices with 'Not Available'

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

price DECIMAL(10,2)

);

-- inserting values without null

INSERT INTO products VALUES

(1, 'Laptop', 50000),

(2, 'Tablet', 18000);

-- inserting value with null

INSERT INTO products (product\_id,product\_name) values (3,'Phone');

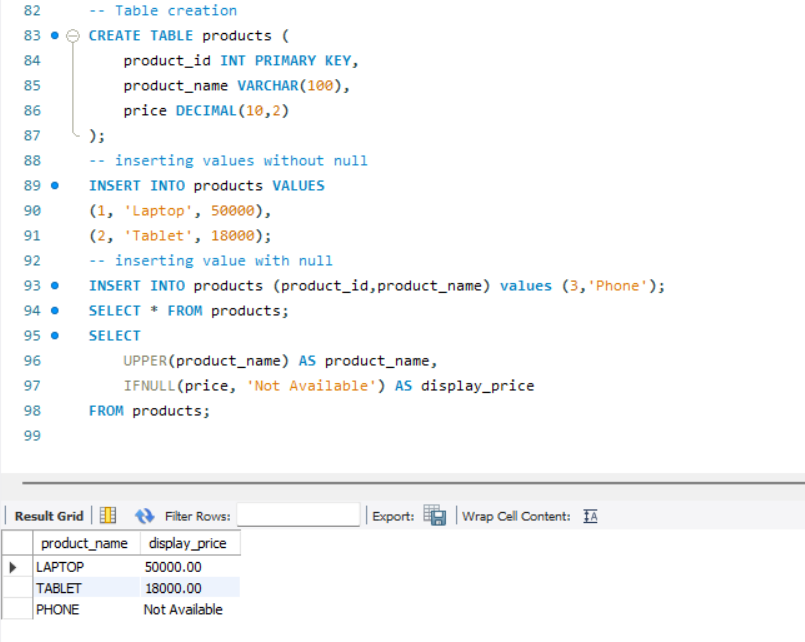
SELECT \* FROM products;

SELECT

UPPER(product\_name) AS product\_name,

IFNULL(price, 'Not Available') AS display\_price

FROM products;



7. Aggregate Functions Practice: From a 'transactions' table, get: - Total sales - Average sale value - Maximum and minimum sale on a single transaction

CREATE TABLE transactions (

trans\_id INT PRIMARY KEY,

sale\_amount DECIMAL(10,2)

);

INSERT INTO transactions VALUES

(1, 200),

(2, 450),

(3, 150),

(4, 700);

SELECT

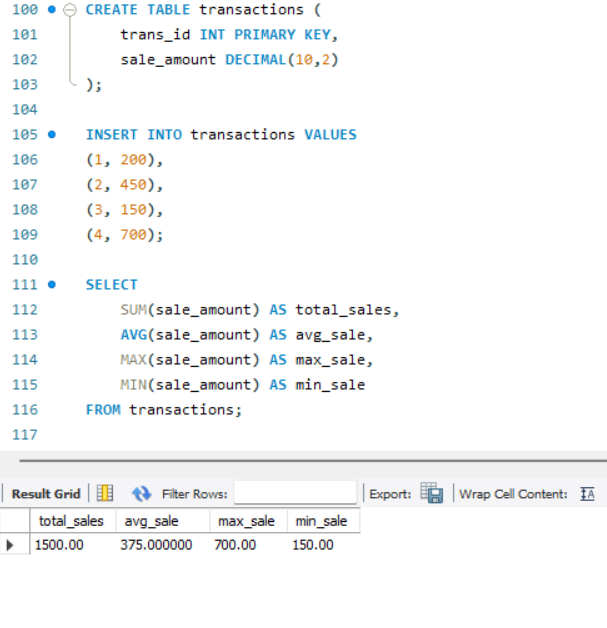
SUM(sale\_amount) AS total\_sales,

AVG(sale\_amount) AS avg\_sale,

MAX(sale\_amount) AS max\_sale,

MIN(sale\_amount) AS min\_sale

FROM transactions;



8. Grouping with Aggregation: From a 'sales' table: - Group by product category - Show total sales and number of transactions in each category

CREATE TABLE sales2 (

sale2\_id INT PRIMARY KEY,

product\_category VARCHAR(50),

sale\_amount DECIMAL(10,2)

);

INSERT INTO sales2 VALUES

(1, 'Electronics', 1000),

(2, 'Electronics', 2000),

(3, 'Clothing', 800),

(4, 'Clothing', 1200);

SELECT

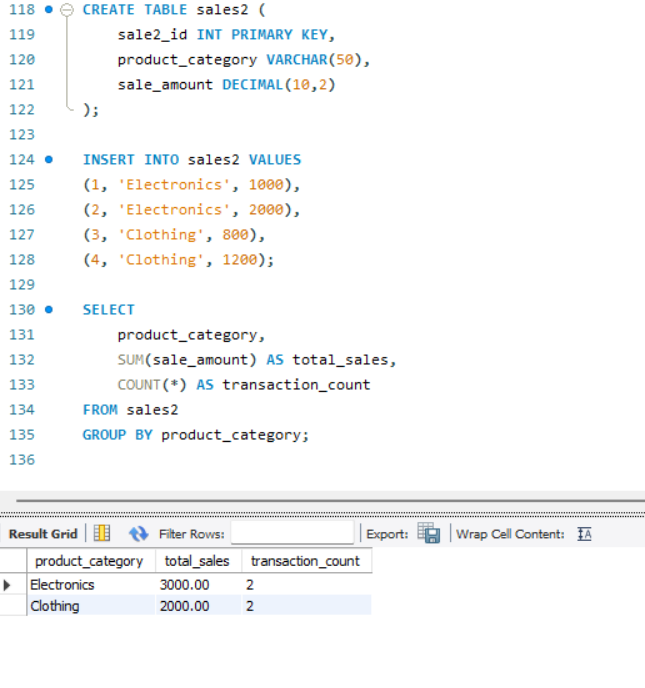
product\_category,

SUM(sale\_amount) AS total\_sales,

COUNT(\*) AS transaction\_count

FROM sales2

GROUP BY product\_category;



9. Inner Join for Orders and Customers: Join 'orders' and 'customers' to show: - Customer name - Order amount - Only for customers who made orders

ALTER TABLE CUSTOMERS

ADD COLUMN CustomerId INT PRIMARY KEY AUTO\_INCREMENT;

SELECT \* FROM CUSTOMERS;

ALTER TABLE CUSTOMERS

MODIFY CustomerId INT FIRST;

SELECT\*FROM CUSTOMERS;

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customerId INT,

order\_amount DECIMAL(10,2),

FOREIGN KEY (customerId) REFERENCES CUSTOMERS(CustomerId)

);

INSERT INTO orders VALUES

(101,1,1200),

(102,2,800),

(103,3,600),

(104,4,1200),

(105,5,1800),

(106,6,1200)

;

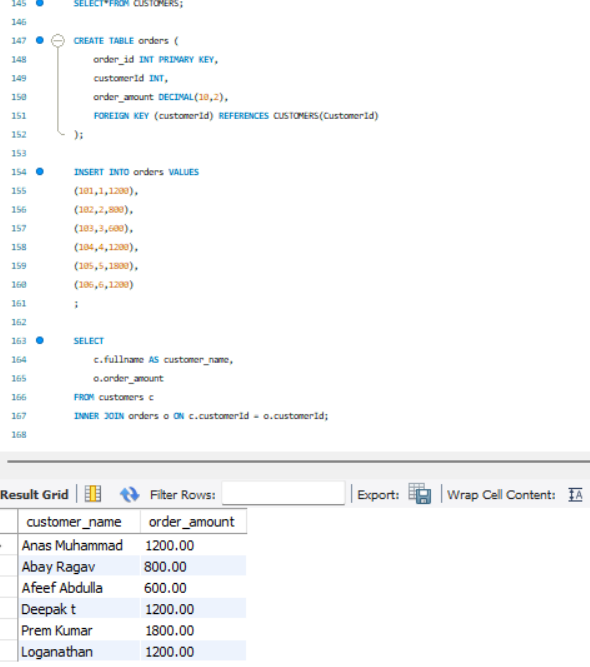
SELECT

c.fullname AS customer\_name,

o.order\_amount

FROM customers c

INNER JOIN orders o ON c.customerId = o.customerId;



10. Left Join for Products with or without Orders: Show all products with: - Their order details (if available) - Use LEFT JOIN

SELECT \* FROM PRODUCTS;

ALTER TABLE ORDERS

ADD COLUMN Product\_id INT;

ALTER TABLE ORDERS

ADD FOREIGN KEY (product\_id) REFERENCES PRODUCTS(product\_id);

SELECT

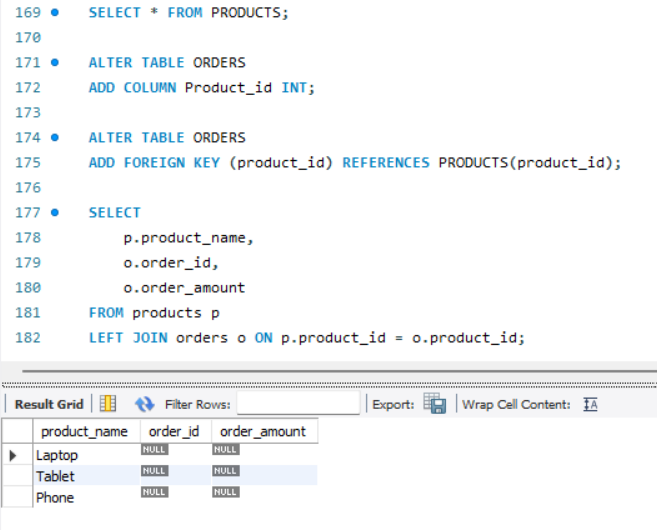
p.product\_name,

o.order\_id,

o.order\_amount

FROM products p

LEFT JOIN orders o ON p.product\_id = o.product\_id;



11. Right Join for Customer Contacts: Use a RIGHT JOIN between 'contacts' and 'customers' to display: - All customers, even if they don't have contact info

SELECT \* FROM CUSTOMERS;

CREATE TABLE contacts (

contact\_id INT PRIMARY KEY,

customerId INT,

phone VARCHAR(15),

email VARCHAR(50),

FOREIGN KEY (customerId) REFERENCES customers(customerId)

);

INSERT INTO contacts VALUES

(1, 1, '9999999999', 'anas@gmail.com'),

(2,2,'2020202020','abay@gmail.com'),

(3,3,'303030303030','afeef@gmail.com'),

(4,4,'4040404040','deepak@gmail.com');

SELECT

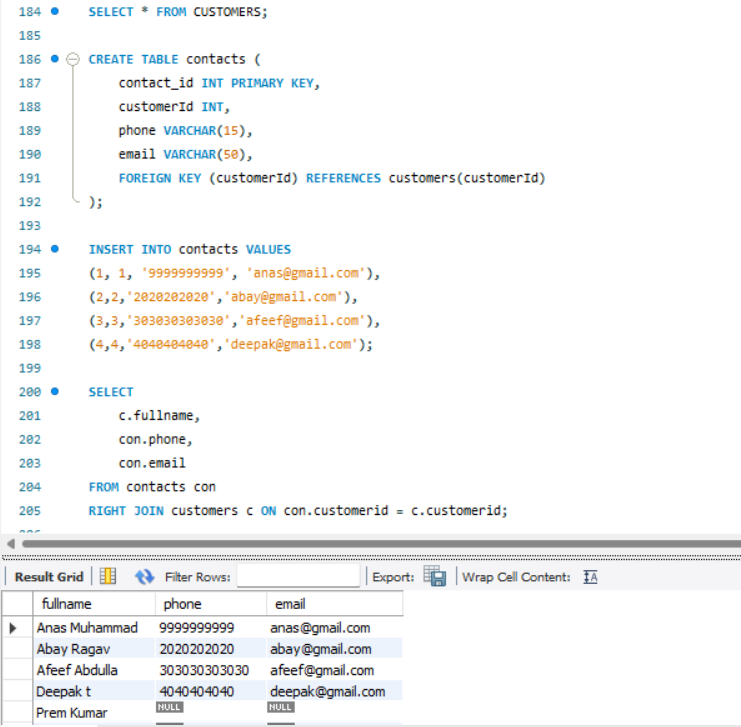
c.fullname,

con.phone,

con.email

FROM contacts con

RIGHT JOIN customers c ON con.customerid = c.customerid;



12. Full Outer Join for Suppliers and Products: Use a FULL OUTER JOIN to list: - All suppliers and products - Match supplier to product, or show NULLs where not available

CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

supplier\_name VARCHAR(50)

);

INSERT INTO suppliers VALUES

(1, 'Dell'),

(2, 'HP');

SELECT s.supplier\_name, p.product\_name

FROM suppliers s

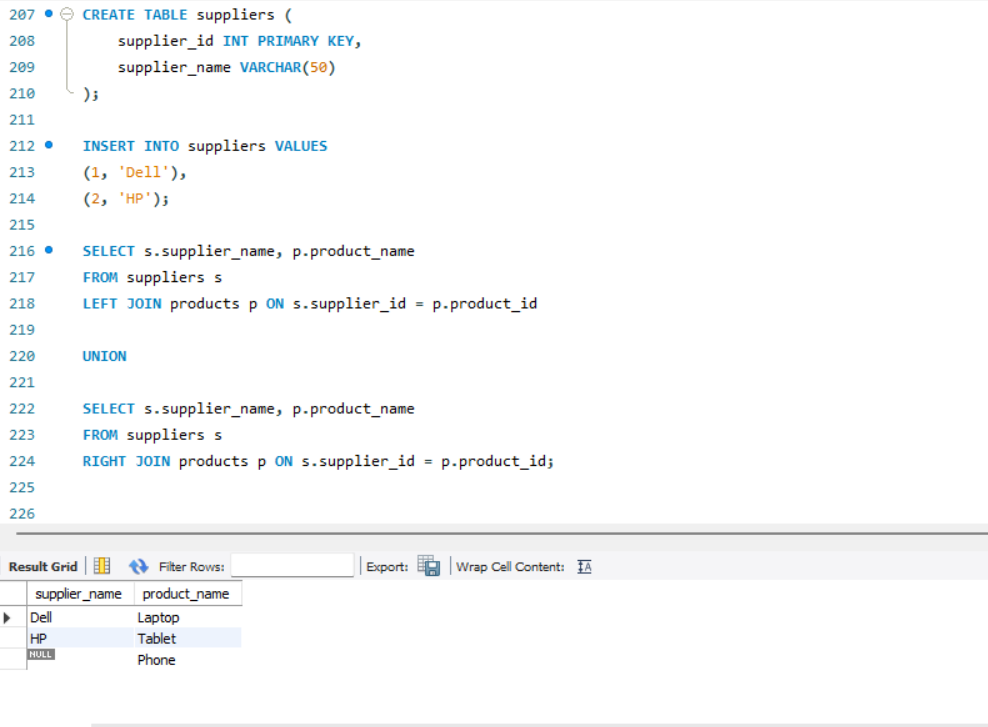
LEFT JOIN products p ON s.supplier\_id = p.product\_id

UNION

SELECT s.supplier\_name, p.product\_name

FROM suppliers s

RIGHT JOIN products p ON s.supplier\_id = p.product\_id;



13. Cross Join for Offers: Suppose you have tables 'products' and 'offers'. Write a CROSS JOIN to show: - All possible combinations of products and offers

CREATE TABLE offers (

offer\_id INT PRIMARY KEY,

offer\_name VARCHAR(50)

);

INSERT INTO offers VALUES

(1, 'Summer Sale'),

(2, 'New Year Offer');

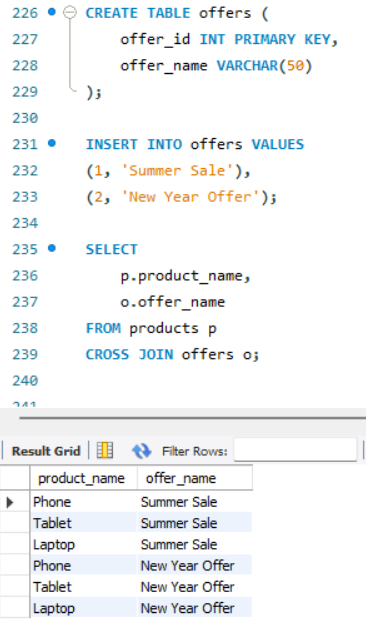
SELECT

p.product\_name,

o.offer\_name

FROM products p

CROSS JOIN offers o;



14. Join with Aggregation: Join 'orders' and 'products', then group by product category and: - Show total quantity sold and average price per category

ALTER TABLE products ADD COLUMN category VARCHAR(50);

UPDATE products SET category = 'Electronics' WHERE product\_id IN (1,3);

UPDATE products SET category = 'Gadgets' WHERE product\_id = 2;

ALTER TABLE orders ADD COLUMN quantity INT;

UPDATE orders SET quantity = 2 WHERE order\_id = 1;

UPDATE orders SET quantity = 1 WHERE order\_id = 2;

SELECT

p.category,

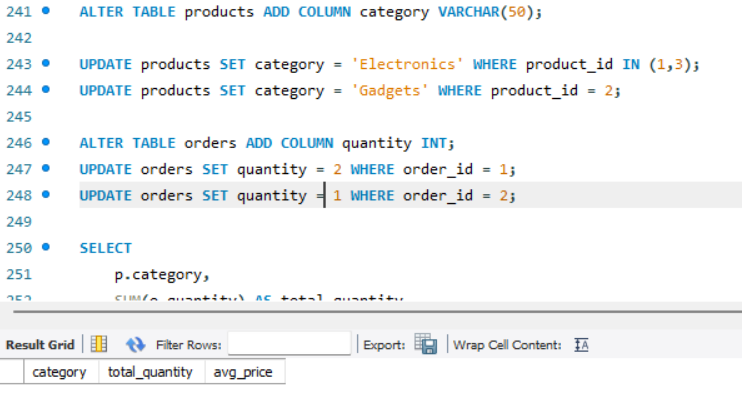
SUM(o.quantity) AS total\_quantity,

AVG(p.price) AS avg\_price

FROM orders o

JOIN products p ON o.product\_id = p.product\_id

GROUP BY p.category;



15. Demo: Join with Grouping and Filter: Join 'students' and 'marks' tables. Display: - Student name - Average marks - Filter to show only students with average marks > 75

CREATE TABLE marks (

mark\_id INT PRIMARY KEY,

student\_id INT,

marks INT

);

INSERT INTO marks VALUES (1, 1, 85), (2, 2, 75), (3, 3, 95), (4, 3, 90), (5, 4, 60);

SELECT \* FROM STUDENTS;

SELECT \* FROM MARKS;

ALTER TABLE MARKS

CHANGE student\_id studentid int;

SELECT

s.name,

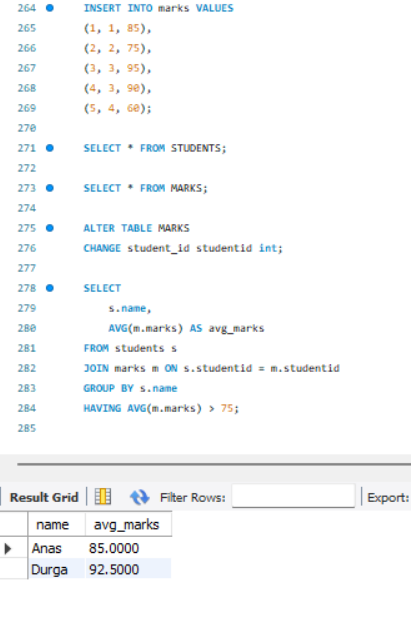
AVG(m.marks) AS avg\_marks

FROM students s

JOIN marks m ON s.studentid = m.studentid

GROUP BY s.name

HAVING AVG(m.marks) > 75;



## QUERYING USING SUBQUERIES – DATE: 13-06-2025

1) Querying Data by Using Subqueries

2) Querying Data by Using Subqueries Using the EXISTS,

3) Querying Data by Using Subqueries using ANY,

4) Querying Data by Using Subqueries using ALL Keywords

5) Querying Data by Using Subqueries using Using Nested Subqueries

6) Querying Data by Using Subqueries Using Correlated Subqueries

7) Querying Data by Using Subqueries Using UNION,

8) Querying Data by Using Subqueries using INTERSECT,

9) Querying Data by Using Subqueries using EXCEPT,

10)Querying Data by Using Subqueries using MERGE.

**Sample Table: Employees**

CREATE TABLE Employees (EmpID INT, Name VARCHAR(50), Department VARCHAR(50), Salary INT);

INSERT INTO Employees VALUES (1, 'Alice', 'HR', 5000);

INSERT INTO Employees VALUES (2, 'Bob', 'IT', 7000);

INSERT INTO Employees VALUES (3, 'Charlie', 'Finance', 6000);

INSERT INTO Employees VALUES (4, 'David', 'IT', 8000);

INSERT INTO Employees VALUES (5, 'Eva', 'HR', 5500);

INSERT INTO Employees VALUES (6, 'Frank', 'Finance', 6200);

**1. Querying Data by Using Subqueries**

Query:

SELECT Name FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);

**2. Querying Data by Using Subqueries Using the EXISTS**

Query:

SELECT Name FROM Employees e WHERE EXISTS (SELECT 1 FROM Employees WHERE Department = 'IT' AND e.Department = Department);

**3. Querying Data by Using Subqueries using ANY**

Query:

SELECT Name FROM Employees WHERE Salary > ANY (SELECT Salary FROM Employees WHERE Department = 'HR');

**4. Querying Data by Using Subqueries using ALL Keywords**

Query:

SELECT Name FROM Employees WHERE Salary > ALL (SELECT Salary FROM Employees WHERE Department = 'HR');

**5. Querying Data by Using Subqueries using Nested Subqueries**

Query:

SELECT Name FROM Employees WHERE Salary = (SELECT MAX(Salary) FROM Employees WHERE Department = (SELECT Department FROM Employees WHERE Name = 'Charlie'));

**6. Querying Data by Using Subqueries Using Correlated Subqueries**

Query:

SELECT Name FROM Employees e1 WHERE Salary > (SELECT AVG(Salary) FROM Employees e2 WHERE e1.Department = e2.Department);

**7. Querying Data by Using Subqueries Using UNION**

Query:

SELECT Name FROM Employees WHERE Department = 'HR' UNION SELECT Name FROM Employees WHERE Salary > 7000;

**8. Querying Data by Using Subqueries using INTERSECT**

Query:

SELECT Name FROM Employees WHERE Department = 'IT' INTERSECT SELECT Name FROM Employees WHERE Salary > 7000;

**9. Querying Data by Using Subqueries using EXCEPT**

Query:

SELECT Name FROM Employees WHERE Department = 'IT' EXCEPT SELECT Name FROM Employees WHERE Salary > 7000;

**10. Querying Data by Using Subqueries using MERGE**

Query:

MERGE INTO Employees AS target USING (SELECT 2 AS EmpID, 'Bob' AS Name) AS source ON target.EmpID = source.EmpID WHEN MATCHED THEN UPDATE SET Salary = 7500;

## QUESTION PAPER 1 - DATE: 13-06-2025

## Section A: Basics & Data Definition (10 Marks)

Q1. (3 marks) Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.

**SQL**

SQL is a relational database which means that the representation is in Rows and Columns. SQL stands for Structured Query Language and it has a fixed schema. Examples of SQL are MySQL and Postgre.

ADVANTAGES AND DISADVANTAGES – ACID Compliance (adv), Less Scalable (disadv).

REAL TIME EXAMPLE – Employee database.

**NoSQL**

It is a non-relational database hence making it more flexible and dynamic than SQL. NoSQL has key- value pair as its data structure. Examples of NoSQL are MongoDB and Firebase.

ADVANTAGES AND DISADVANTAGES – Handles large volume easily (adv), No standard query language (disadv)

REAL TIME EXAMPLE – Netflix.

Q2. (2 marks) Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF: Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone).

Unnormalized data - Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

1NF data – Split into repeating groups

2NF data – Split into 2 tables

Student(StudentID, Name)

Enrollment(StudentID, CourseID)

Course(CourseID, CourseName, InstructorName, InstructorPhone)

3NF data - Split Instructor into new table

Instructor(InstructorName, InstructorPhone)

Course(CourseID, CourseName, InstructorName)

Q3. (5 marks) a) Create a database named StudentDB. b) Create a table Students with fields: StudentID, Name, DOB, Email. c) Rename the table to Student\_Info. d) Add a column PhoneNumber. e) Drop the table.

1. CREATE DATABASE StudentDB;

USE StudentDB;

1. CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(50),

DOB DATE,

Email VARCHAR(100));

1. RENAME TABLE Students TO Student\_Info;
2. ALTER TABLE Student\_Info ADD PhoneNumber Varchar(15);
3. DROP TABLE STUDENT INFO;

## Section B: DML & Filtering Data (15 Marks)

Q4. (5 marks) a) Insert 3 student records into Student\_Info. b) Update one student's phone number. c) Delete one student whose email ends with @gmail.com. d) Retrieve only names and emails of students born after the year 2000. e) Retrieve distinct domain names from the email column.

1. INSERT INTO Student\_Info VALUES

(1, 'Anas', '2002-05-10', 'anas@gmail.com', 9876543210),

(2, 'Priya', '1999-12-01', 'priya@yahoo.com', 8765432109),

(3, 'Ravi', '2003-08-20', 'ravi@gmail.com', 9988776655);

1. UPDATE Student\_Info SET PhoneNumber = 9090909090 WHERE StudentID = 2;
2. DELETE FROM Student\_Info WHERE Email LIKE '%@gmail.com';
3. SELECT Name, Email FROM Student\_Info WHERE YEAR(DOB) > 2000;
4. SELECT DISTINCT SUBSTRING\_INDEX(Email, '@', -1) AS domain FROM Student\_Info;

Q5. (5 marks) a) Retrieve students with names starting with 'A'. b) Retrieve students with phone number between 9000000000 and 9999999999. c) Retrieve students using IN operator on city names. d) Use AND, OR to filter students based on age and email provider. e) Use table and column aliasing in a query to get all student names and DOBs.

1. SELECT \* FROM Student\_Info WHERE Name LIKE 'A%';

b. SELECT \* FROM Student\_Info WHERE PhoneNumber BETWEEN 9000000000 AND 9999999999;

c. SELECT \* FROM Student\_Info WHERE City IN ('Chennai', 'Delhi', 'Mumbai');

d. SELECT \* FROM Student\_Info WHERE YEAR(CURDATE()) - YEAR(DOB) < 25 AND Email LIKE '%@gmail.com';

e. SELECT Name AS StudentName, DOB AS BirthDate FROM Student\_Info;

Q6. (5 marks) Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows. a) Display student IDs and their subjects where marks > 70. b) Display subjects with average marks. c) Filter subjects with average marks between 60 and 90.

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks INT

);

INSERT INTO Marks VALUES

(1, 'Maths', 85),

(1, 'English', 78),

(2, 'Maths', 92);

a. SELECT StudentID, Subject FROM Marks WHERE Marks > 70;

b. SELECT Subject, AVG(Marks) AS avg\_marks FROM Marks GROUP BY Subject;

c. SELECT Subject FROM Marks GROUP BY Subject

HAVING AVG(Marks) BETWEEN 60 AND 90;

## SECTION C: Functions & Grouping (10 Marks)

Q7. (5 marks) a) Get the current date and format it as "YYYY-MM-DD". b) Extract month and year from a DOB column. c) Convert a student's name to uppercase. d) Round off marks to 2 decimal places. e) Use system function to return user name or current database.

a. SELECT DATE\_FORMAT(CURDATE(), '%Y-%m-%d') AS today;

b. SELECT MONTH(DOB) AS birth\_month, YEAR(DOB) AS birth\_year FROM Student\_Info;

c. SELECT UPPER(Name) FROM Student\_Info;

d. SELECT ROUND(Marks, 2) FROM Marks;

e. SELECT DATABASE() AS db\_name, USER() AS user;

Q8. (5 marks) a) Display total marks of each student. b) Display subject-wise highest mark. c) Use GROUP BY and HAVING to display subjects with average marks > 75.

a. SELECT StudentID, SUM(Marks) AS total\_marks FROM Marks GROUP BY StudentID;

b. SELECT Subject, MAX(Marks) AS highest FROM Marks GROUP BY Subject;

c. SELECT Subject, AVG(Marks) AS avg\_marks

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) > 75;

## Section D: Joins and Subqueries (25 Marks)

Q9. (5 marks) a) Inner Join to retrieve students and their courses. b) Left Join to get all students even if not enrolled. c) Right Join to get all courses even if no students. d) Full Outer Join equivalent using UNION. e) Cross Join to show all combinations.

Assume tables: Students, Courses, Enrollments

a. Inner Join

SELECT s.Name, c.CourseName

FROM Students s

JOIN Enrollments e ON s.StudentID = e.StudentID

JOIN Courses c ON e.CourseID = c.CourseID;

b. Left Join

SELECT s.Name, c.CourseName

FROM Students s

LEFT JOIN Enrollments e ON s.StudentID = e.StudentID

LEFT JOIN Courses c ON e.CourseID = c.CourseID;

c.Right Join

SELECT s.Name, c.CourseName

FROM Courses c

RIGHT JOIN Enrollments e ON c.CourseID = e.CourseID

RIGHT JOIN Students s ON e.StudentID = s.StudentID;

d.Full Outer Join (using UNION)

SELECT s.Name, c.CourseName

FROM Students s

LEFT JOIN Enrollments e ON s.StudentID = e.StudentID

LEFT JOIN Courses c ON e.CourseID = c.CourseID

UNION

SELECT s.Name, c.CourseName

FROM Courses c

LEFT JOIN Enrollments e ON c.CourseID = e.CourseID

LEFT JOIN Students s ON e.StudentID = s.StudentID;

e.Cross Join

SELECT s.Name, c.CourseName

FROM Students s

CROSS JOIN Courses c;

Q10. (5 marks) a) Students who scored more than average in 'Maths'. b) Students not in the Marks table. c) Use EXISTS to get students with at least one subject. d) Use ALL to find those scoring more than all in 'Science'. e) Use ANY for students scoring better than some in 'English'.

a. SELECT StudentID FROM Marks

WHERE Subject = 'Maths' AND Marks > (

SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths'

);

b. SELECT \* FROM Students

WHERE StudentID NOT IN (SELECT DISTINCT StudentID FROM Marks);

c. SELECT \* FROM Students s

WHERE EXISTS (

SELECT 1 FROM Marks m WHERE m.StudentID = s.StudentID

);

d.

SELECT StudentID FROM Marks

WHERE Subject = 'Science' AND Marks > ALL (

SELECT Marks FROM Marks WHERE Subject = 'Science'

);

e.SELECT StudentID FROM Marks

WHERE Subject = 'English' AND Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English'

);

Q11. (5 marks) a) UNION of student names from two tables. b) INTERSECT to find common students. c) EXCEPT to list students in Students but not in Marks. d) MERGE concept or simulate with UPDATE and INSERT. e) Correlated subquery to list students with above average per subject

a. SELECT Name FROM Students

UNION

SELECT Name FROM Student\_Info;

b. SELECT s.Name

FROM Students s

JOIN Student\_Info si ON s.Name = si.Name;

c. SELECT Name FROM Students

WHERE Name NOT IN (SELECT Name FROM Student\_Info);

d. INSERT INTO Students (StudentID, Name)

VALUES (4, 'Riya')

ON DUPLICATE KEY UPDATE Name = 'Riya';

e. SELECT s.Name

FROM Students s

WHERE EXISTS (

SELECT 1 FROM Marks m

WHERE m.StudentID = s.StudentID

GROUP BY m.Subject

HAVING AVG(m.Marks) > (

SELECT AVG(Marks) FROM Marks WHERE Subject = m.Subject

)

);

## QUESTION PAPER 2 - DATE: 13-06-2025

## Section A: Advanced Concepts & Schema Design (10 Marks)

Q1. (4 marks) Explain with examples the scenarios where NoSQL is preferred over SQL. Discuss types of NoSQL databases and suggest a real-time application for each.

**Scenarios where NoSQL is preferred over SQL:**

**High scalability & performance:** NoSQL databases like MongoDB are used in systems handling massive real-time data (e.g., sensor networks, social media). **Unstructured or semi-structured data:** When data formats are dynamic, such as JSON logs or user-generated content. **Flexible schema requirements:** Useful in fast-paced development where schemas evolve rapidly (e.g., content platforms). **High availability over consistency (BASE model):** In globally distributed apps like e-commerce and messaging apps.

Example: Content Management System (CMS) for blogs storing articles as documents and Session management in web applications.

Q2. (6 marks) A retail store keeps the following unnormalized record: Customer (CustomerID, Name, Orders (OrderID, ProductID, Quantity, ProductName)) Normalize the data up to BCNF with appropriate table structures.

**Unnormalized Data:**

Customer (CustomerID, Name, Orders (OrderID, ProductID, Quantity, ProductName))

**1NF (Atomic values, remove repeating groups):**

Break nested data:

Customer\_Order (

CustomerID, Name, OrderID, ProductID, Quantity, ProductName

)

**2NF (No partial dependency):**

Remove dependency of Product info on part of primary key

Tables:

Customer (CustomerID, Name)

OrderDetails (OrderID, CustomerID)

Product (ProductID, ProductName)

OrderItems (OrderID, ProductID, Quantity)

**3NF (No transitive dependency):**

Already achieved since all non-key columns depend only on keys.

**BCNF (All determinants are candidate keys):**

Already satisfied. All functional dependencies have determinant as candidate key.

## Section B: Complex DDL and DML (15 Marks)

Q3. (5 marks) a) Create a database RetailDB and design a schema for Customers, Orders, and Products with primary and foreign keys. b) Implement a check constraint on Quantity (>0) in Orders. c) Alter the Products table to add 'Discount' column and update some values.

a) CREATE DATABASE RetailDB; USE RetailDB;

b) CREATE TABLE Customers (

CustomerID INT PRIMARY KEY AUTO\_INCREMENT,

Name VARCHAR(100)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY AUTO\_INCREMENT,

ProductName VARCHAR(100),

Price DECIMAL(10, 2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY AUTO\_INCREMENT,

CustomerID INT,

ProductID INT,

Quantity INT CHECK (Quantity > 0),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

c) ALTER TABLE Products ADD COLUMN Discount DECIMAL(5,2);

Q4. (5 marks) Using the above schema:

a) Insert 3 sample orders per customer.

INSERT INTO Customers (Name) VALUES ('Alice'), ('Bob');

INSERT INTO Products (ProductName, Price) VALUES ('Laptop', 50000), ('Mouse', 500), ('Keyboard', 1200);

INSERT INTO Orders (CustomerID, ProductID, Quantity) VALUES (1, 1, 2), (1, 2, 6), (1, 3, 1);

INSERT INTO Orders (CustomerID, ProductID, Quantity) VALUES (2, 2, 3), (2, 1, 1), (2, 3, 10);

b) Update prices with 10% increase where quantity sold > 5.

UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID FROM Orders GROUP BY ProductID HAVING SUM(Quantity) > 5);

c) Delete orders where the product has never been sold.

DELETE FROM Products

WHERE ProductID NOT IN ( SELECT DISTINCT ProductID FROM Orders);

Q5. (5 marks) Retrieve the following:

a) Customers who ordered more than 3 different products.

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

b) Products not ordered by any customer.

SELECT \* FROM Products

WHERE ProductID NOT IN ( SELECT DISTINCT ProductID FROM Orders);

c) Count of orders placed by each customer in the last 30 days

ALTER TABLE Orders ADD COLUMN OrderDate DATE DEFAULT CURDATE();

SELECT CustomerID, COUNT(\*) AS RecentOrders FROM Orders WHERE OrderDate >= CURDATE() - INTERVAL 30 DAY GROUP BY CustomerID;

## Section C: Advanced Functions and Aggregations (10 Marks)

Q6. (5 marks)

a) Use string functions to standardize and extract parts from customer email IDs.

SELECT

LOWER(Email) AS StandardizedEmail,

SUBSTRING\_INDEX(Email, '@', 1) AS Username,

SUBSTRING\_INDEX(Email, '@', -1) AS Domain

FROM Customers;

b) Use date functions to compute days between order date and today

SELECT OrderID,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

c) Use system functions to return current user and host.

SELECT CURRENT\_USER() AS User,

HOST\_NAME() AS Host;

SELECT SYSTEM\_USER();

d) Use nested functions to format a customer greeting string.

SELECT

CONCAT('Hello ', UPPER(Name), '! Your email is ', Email) AS Greeting

FROM Customers;

Q7. (5 marks)

a) Aggregate total revenue by product category.

SELECT p.Category,

SUM(o.Quantity \* p.Price) AS TotalRevenue

FROM Orders o JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category;

b) Use GROUP BY with ROLLUP to compute subtotal and grand total sales.

SELECT product\_category,region,

SUM(sale\_amount) AS total\_sales

FROM sales GROUP BY product\_category, region WITH ROLLUP;

c) Use HAVING clause to filter categories with revenue > 100000.

SELECT p.Category,

SUM(o.Quantity \* p.Price) AS Revenue

FROM Orders o JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category

HAVING Revenue > 100000;

## Section D: Complex Joins, Subqueries, and Set Ops (25 Marks)

Q8. (5 marks)

1. Self join to list customers referred by other customers.

SELECT c1.Name AS Customer, c2.Name AS ReferredBy

FROM Customers c1

JOIN Customers c2 ON c1.ReferredByID = c2.CustomerID;

1. Equi join across Orders and Products.

SELECT o.OrderID, p.ProductName, o.Quantity

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID;

1. Join Customers and Orders to display top 3 spenders using window function.

SELECT \* FROM (

SELECT

c.CustomerID, c.Name,

SUM(o.Quantity \* p.Price) AS TotalSpent,

RANK() OVER (ORDER BY SUM(o.Quantity \* p.Price) DESC) AS Rank

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY c.CustomerID

) AS Ranked

WHERE Rank <= 3;

1. LEFT OUTER JOIN with WHERE NULL to identify inactive customers

SELECT c.CustomerID, c.Name

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

WHERE o.OrderID IS NULL;

1. Cross join for all product combinations in a bundle offer.

SELECT

p1.ProductName AS ProductA,

p2.ProductName AS ProductB

FROM Products p1

CROSS JOIN Products p2

WHERE p1.ProductID < p2.ProductID;

Q9. (5 marks)

1. Correlated subquery to get customers whose order amount exceeds their average.

SELECT \* FROM Orders o1

WHERE o1.Quantity \* (SELECT Price FROM Products WHERE ProductID = o1.ProductID) >

(SELECT AVG(o2.Quantity \* p2.Price)

FROM Orders o2

JOIN Products p2 ON o2.ProductID = p2.ProductID

WHERE o2.CustomerID = o1.CustomerID);

1. Subquery using EXISTS to find customers with at least 2 different products.

SELECT \* FROM Customers c

WHERE EXISTS (

SELECT 1

FROM Orders o

WHERE o.CustomerID = c.CustomerID

GROUP BY o.ProductID

HAVING COUNT(DISTINCT o.ProductID) >= 2);

1. Use ALL to find customers who ordered more than every other customer.

SELECT c.CustomerID, c.Name

FROM Customers c

WHERE (

SELECT COUNT(\*) FROM Orders o WHERE o.CustomerID = c.CustomerID

) > ALL (

SELECT COUNT(\*) FROM Orders o2 WHERE o2.CustomerID != c.CustomerID GROUP BY o2.CustomerID

);

1. Use ANY to find products costlier than some in category 'Electronics'.

SELECT \* FROM Products

WHERE Price > ANY (

SELECT Price

FROM Products

WHERE Category = 'Electronics');

1. Nested subquery to list top 3 best-selling products.

SELECT \* FROM (

SELECT p.ProductID, p.ProductName, SUM(o.Quantity) AS TotalSold,

RANK() OVER (ORDER BY SUM(o.Quantity) DESC) AS rnk

FROM Products p

JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY p.ProductID

) AS RankedProducts

WHERE rnk <= 3;

Q10. (5 marks)

1. Simulate INTERSECT using INNER JOIN on two customer segments.

SELECT pc.CustomerID, pc.Name

FROM PremiumCustomers pc

INNER JOIN LoyalCustomers lc ON pc.CustomerID = lc.CustomerID;

1. Use EXCEPT to find products in inventory not yet ordered.

SELECT p.ProductID, p.ProductName

FROM Products p

LEFT JOIN Orders o ON p.ProductID = o.ProductID

WHERE o.ProductID IS NULL;

1. Simulate MERGE: If customer exists, update; else insert.

INSERT INTO Customers (CustomerID, Name, Email)

VALUES (101, 'John Doe', '[john@example.com](mailto:john@example.com)')

ON DUPLICATE KEY UPDATE

Name = VALUES(Name),

Email = VALUES(Email);

1. Use UNION to combine two regional customer tables.

SELECT CustomerID, Name, Email FROM EastRegionCustomers

UNION

SELECT CustomerID, Name, Email FROM WestRegionCustomers;

1. Write a WITH CTE that ranks customers by total spend and filters top 5.

WITH CustomerSpending AS (

SELECT c.CustomerID, c.Name,

SUM(o.Quantity \* p.Price) AS TotalSpent,

RANK() OVER (ORDER BY SUM(o.Quantity \* p.Price) DESC) AS rnk

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY c.CustomerID

)

SELECT CustomerID, Name, TotalSpent

FROM CustomerSpending

WHERE rnk <= 5;