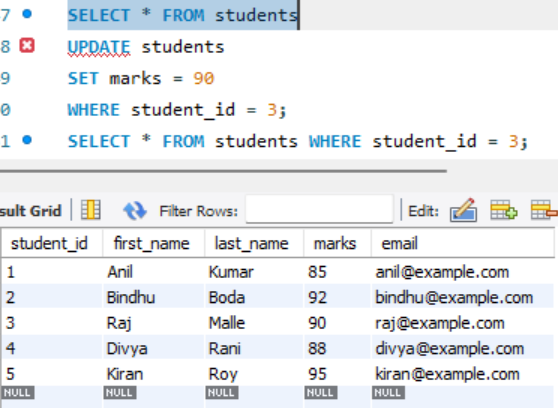
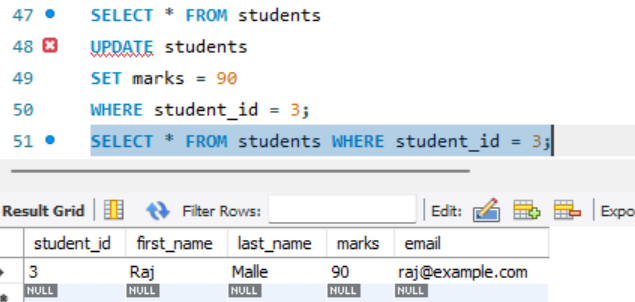
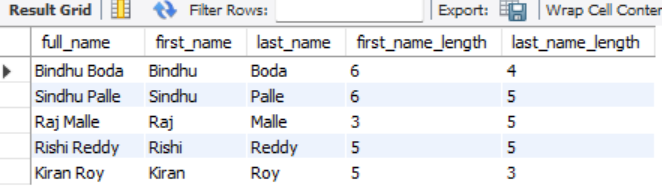
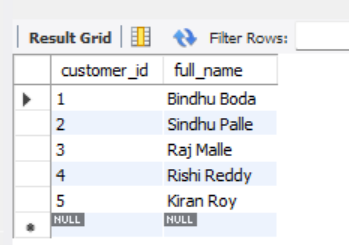
**DATE:12|06|25**

**Medium-Level Practical SQL Questions**

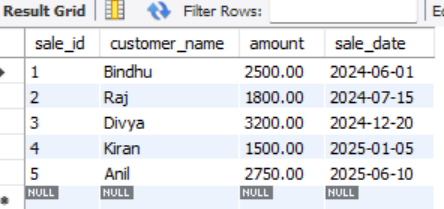
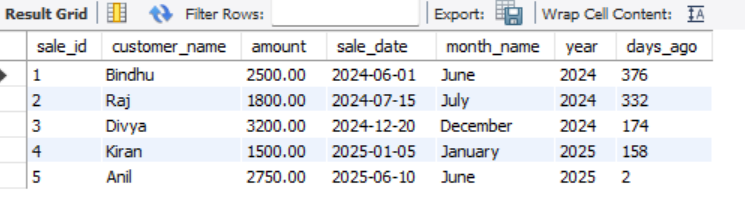
1. Insert and Update with Integrity: Create a 'students' table with constraints (NOT NULL, UNIQUE). Insert 5 records. Then, update a student's marks ensuring data integrity is maintained.



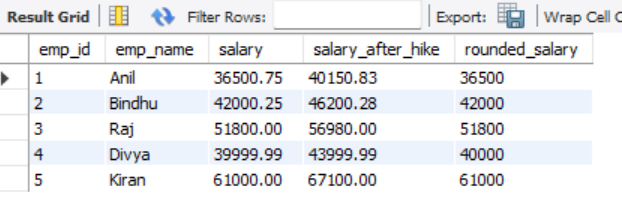
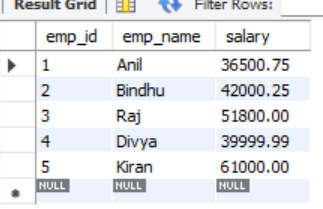
2)String Function Challenge: Given a 'customers' table with a 'full\_name' column, write a query to display: - First name - Last name - Length of each name

****

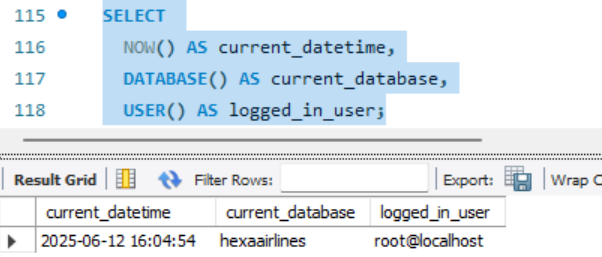
3)Date Function Usage: From a 'sales' table with a 'sale\_date' column, write a query to: - Extract the month name and year - Display how many days ago the sale happened.



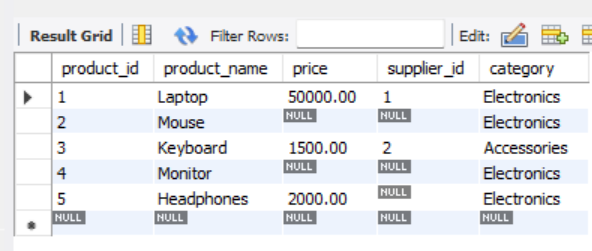
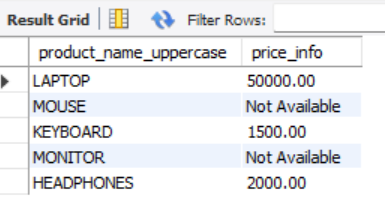
4. Mathematical Functions on Salary: In an 'employees' table, calculate: - Salary after a 10% hike - Round the salary to the nearest hundred



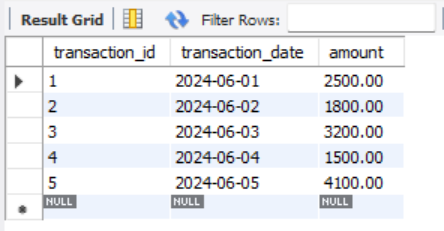
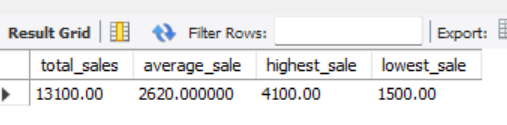
5)System Function Check: Retrieve: - Current date and time - Database name and logged-in user



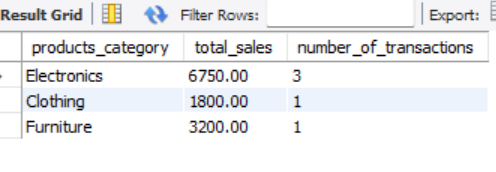
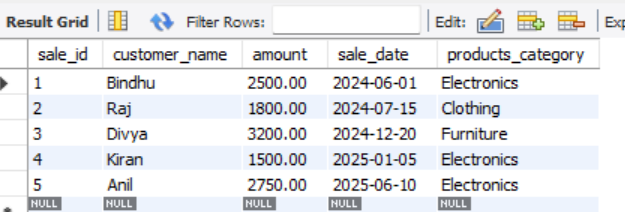
6)Demo: Custom Result Set: From the 'products' table, write a query that: - Returns product name in uppercase - Replaces any NULL prices with 'Not Available'.



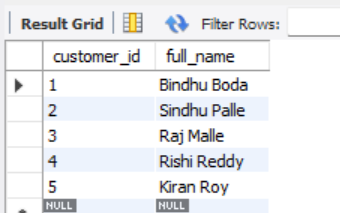
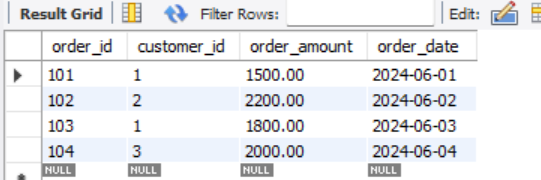
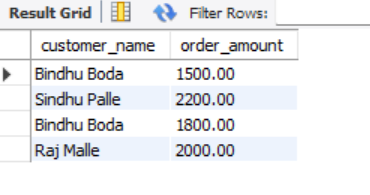
7)Aggregate Functions Practice: From a 'transactions' table, get: - Total sales - Average sale value - Maximum and minimum sale on a single transaction



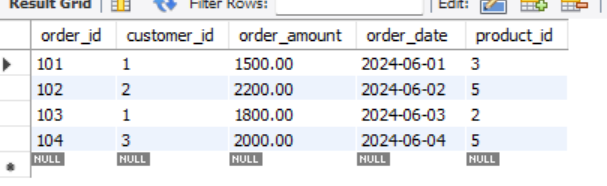
8)Grouping with Aggregation: From a 'sales' table: - Group by product category - Show total sales and number of transactions in each category

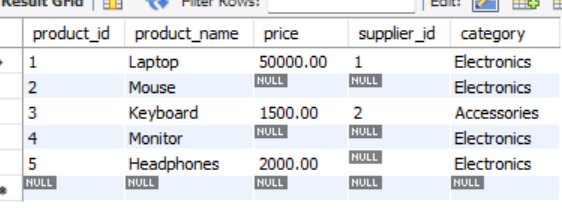


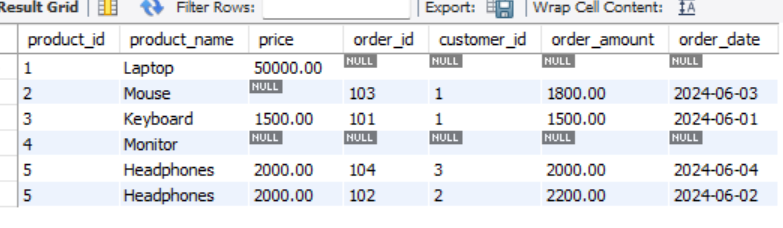
9. Inner Join for Orders and Customers: Join 'orders' and 'customers' to show: - Customer name - Order amount - Only for customers who made orders



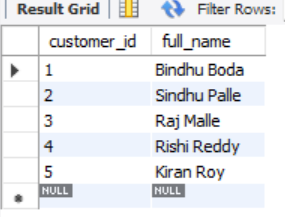
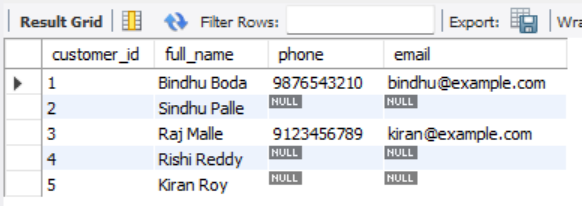
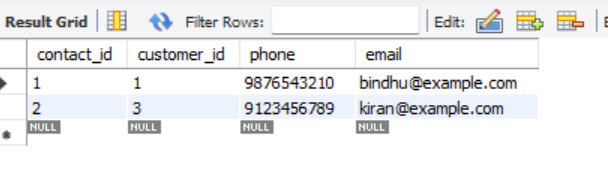
10)Left Join for Products with or without Orders: Show all products with: - Their order details (if available) - Use LEFT JOIN.

Product table Order table

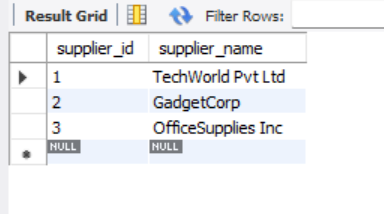
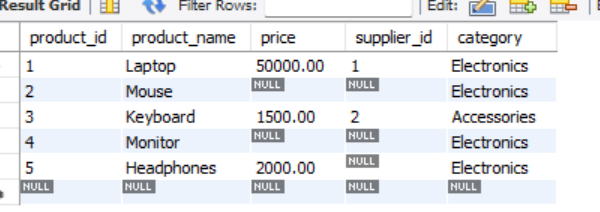


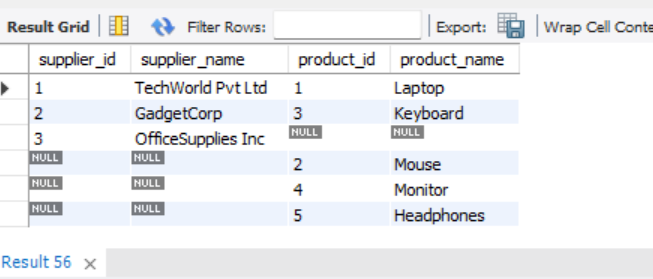


11)Right Join for Customer Contacts: Use a RIGHT JOIN between 'contacts' and 'customers' to display: - All customers, even if they don't have contact info

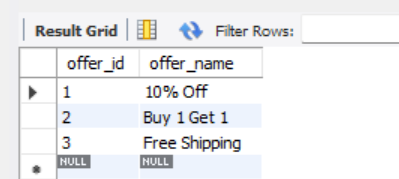
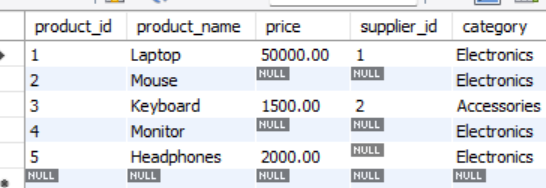


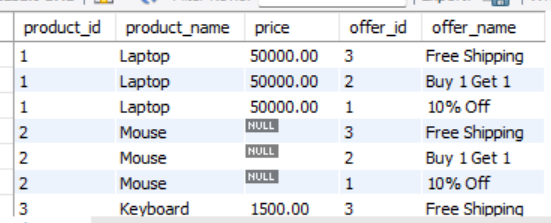
12)Full Outer Join for Suppliers and Products: Use a FULL OUTER JOIN to list: - All suppliers and products - Match supplier to product, or show NULLs where not available

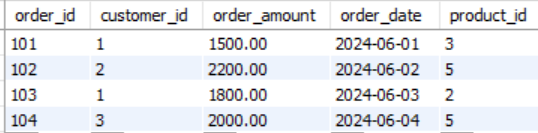
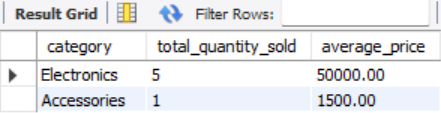
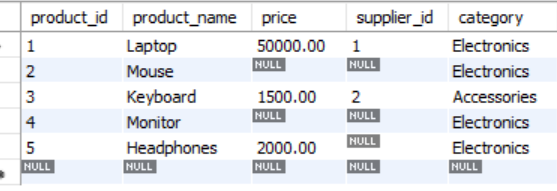




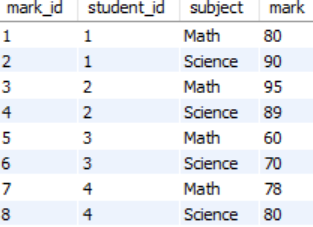
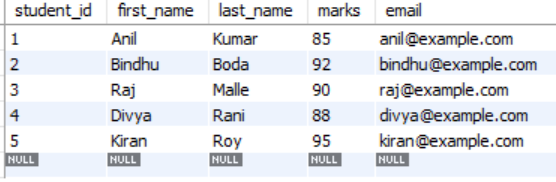
13)Cross Join for Offers: Suppose you have tables 'products' and 'offers'. Write a CROSS JOIN to show: - All possible combinations of products and offers

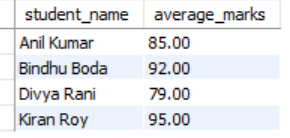




14)Join with Aggregation: Join 'orders' and 'products', then group by product category and: - Show total quantity sold and average price per category

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





**DATE: 13|6|25**

**Subqueries Types along 2 queries for each**

USE BindhuDB1

CREATE TABLE Cartoons (

CartoonID INT PRIMARY KEY,

CartoonName VARCHAR(50),

ChannelID INT

);

INSERT INTO Cartoons (CartoonID, CartoonName, ChannelID) VALUES

(101, 'Tom and Jerry', 1),

(102, 'Heidi', 2),

(103, 'Mickey Mouse', 3),

(104, 'Chhota Bheem', 4),

(105, 'Ben 10', 1);

CREATE TABLE Channels (

ChannelID INT PRIMARY KEY,

ChannelName VARCHAR(50)

);

INSERT INTO Channels (ChannelID, ChannelName) VALUES

(1, 'Cartoon Network'),

(2, 'Kushi'),

(3, 'Disney'),

(4, 'Pogo');

SELECT

c.CartoonName,

ch.ChannelName

FROM

Cartoons c

JOIN

Channels ch ON c.ChannelID = ch.ChannelID;

ALTER TABLE Cartoons ADD Rating INT;

SET SQL\_SAFE\_UPDATES = 0;

UPDATE Cartoons SET Rating =

CASE

WHEN CartoonName = 'Tom and Jerry' THEN 9

WHEN CartoonName = 'Heidi' THEN 6

WHEN CartoonName = 'Mickey Mouse' THEN 8

WHEN CartoonName = 'Chhota Bheem' THEN 7

WHEN CartoonName = 'Ben 10' THEN 8

END;

**--------------------------------------------Single row sub query**

SELECT CartoonName

FROM Cartoons

WHERE ChannelID = (SELECT ChannelID FROM Cartoons WHERE CartoonName = 'Heidi');

SELECT CartoonName, Rating

FROM Cartoons

WHERE Rating = (SELECT MAX(Rating) FROM Cartoons);

**--------------------------------------Multi Row sub query**

SELECT CartoonName

FROM Cartoons

WHERE ChannelID IN (

SELECT ChannelID FROM Channels

WHERE ChannelName IN ('Cartoon Network', 'Disney')

);

SELECT CartoonName, Rating

FROM Cartoons

WHERE Rating IN (

SELECT Rating FROM Cartoons WHERE ChannelID = 1

);

**---------------------------------Nested sub query**

SELECT ch.ChannelName, AVG(c.Rating) AS AvgRating

FROM (

SELECT \* FROM Cartoons

) AS c

JOIN Channels ch ON c.ChannelID = ch.ChannelID

GROUP BY ch.ChannelName;

SELECT ChannelID, CartoonCount

FROM (

SELECT ChannelID, COUNT(\*) AS CartoonCount

FROM Cartoons

GROUP BY ChannelID

) AS Sub

WHERE CartoonCount > 1;

**----------------------------------Correlated sub query**

SELECT CartoonName, Rating

FROM Cartoons c1

WHERE Rating <= (

SELECT AVG(Rating)

FROM Cartoons c2

WHERE c1.ChannelID = c2.ChannelID

);

SELECT CartoonName, Rating

FROM Cartoons c1

WHERE Rating < (

SELECT MAX(Rating)

FROM Cartoons c2

WHERE c1.ChannelID = c2.ChannelID

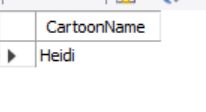
);

**Questions to solve based on Subqueries and keyworks using subqueries**

1. Querying Data by Using Subqueries

SELECT CartoonName

FROM Cartoons

WHERE ChannelID = (SELECT ChannelID FROM Cartoons WHERE CartoonName = 'Heidi');  


1. Querying Data by Using Subqueries Using the EXISTS,

SELECT CartoonName

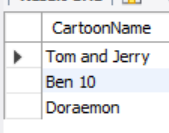
FROM Cartoons c

WHERE EXISTS (

SELECT 1

FROM Channels ch

WHERE ch.ChannelID = c.ChannelID AND ch.ChannelName = 'Cartoon Network'

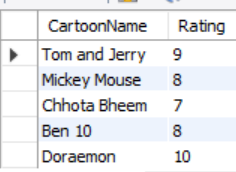
);  


3) Querying Data by Using Subqueries using ANY,  
SELECT CartoonName, Rating

FROM Cartoons

WHERE Rating > ANY (

SELECT Rating FROM Cartoons WHERE ChannelID = 2

);  


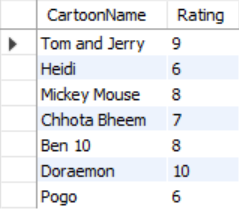
1. Querying Data by Using Subqueries using ALL Keywords

SELECT CartoonName, Rating

FROM Cartoons

WHERE Rating >=ALL (

SELECT Rating FROM Cartoons WHERE ChannelID = 2

);  


1. Querying Data by Using Subqueries using Nested Subqueries

SELECT AVG(Rating)

FROM (

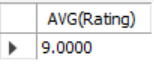
SELECT Rating

FROM Cartoons

WHERE ChannelID = (

SELECT ChannelID FROM Channels WHERE ChannelName = 'Cartoon Network'

)

) AS sub;  


1. Querying Data by Using Subqueries using Correlated

SELECT CartoonName, Rating

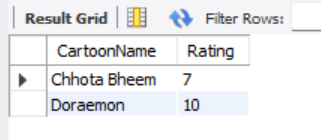
FROM Cartoons c1

WHERE Rating > (

SELECT AVG(Rating)

FROM Cartoons c2

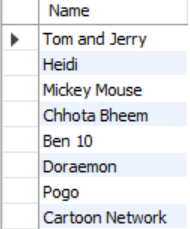
WHERE c1.ChannelID = c2.ChannelID

);  


1. Querying Data by Using Subqueries using Union keyword

SELECT CartoonName AS Name FROM Cartoons

UNION

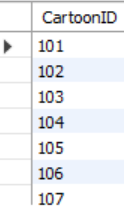
SELECT ChannelName FROM Channels;  


1. Querying Data by Using Subqueries using INTERSECT

SELECT CartoonID FROM Cartoons;

INSERT INTO Cartoons (CartoonID, CartoonName, ChannelID, Rating)

VALUES (107, 'Pogo', 4, 6);

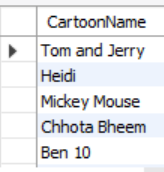


1. Querying Data by Using Subqueries Using Except Keyword

SELECT CartoonName FROM Cartoons

WHERE CartoonName NOT IN (

SELECT ChannelName FROM Channels

);  


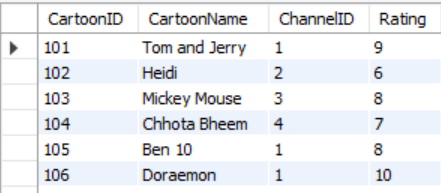
Querying Data by Using Subqueries using Merge Keyword

INSERT INTO Cartoons (CartoonID, CartoonName, ChannelID, Rating)

VALUES (106, 'Doraemon', 1, 10)

ON DUPLICATE KEY UPDATE

Rating = 10;



**SQL Practical Question Paper-1**

**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 (Structured Query Language)** is a standard language used to store, retrieve, and manage data in **relational databases,** where data is organized into **tables** with rows and columns.

**NoSQL (Not Only SQL)** **is a category of non-relational databases designed to store and manage unstructured or semi-structured data, often using flexible formats like JSON, key-value pairs, or graphs.**

| **Feature** | **SQL** | **NoSQL** |
| --- | --- | --- |
| **Type** | Relational Database | Non-Relational Database |
| **Schema** | Fixed, predefined schema | Dynamic, flexible schema |
| **Data Format** | Tables with rows and columns | Document, Key-Value, Column-family, Graph |
| **Scalability** | Vertically scalable | Horizontally scalable |
| **ACID Compliance** | Fully ACID-compliant | Not always ACID-compliant, often BASE model |
| **Query Language** | Structured Query Language (SQL) | Varies (e.g., JSON queries, CQL, etc.) |
| **Best For** | Structured data with complex joins | Semi-structured/unstructured data, high velocity |
| **Examples** | MySQL, Oracle, PostgreSQL, MS SQL Server | MongoDB, Cassandra, Redis, Neo4j |
| **Transactions** | Strong support for multi-row transactions | Limited transaction support (improving gradually) |
| **Use Cases** | Banking, ERP, CRM | Real-time analytics, IoT, Big Data, Social Media |

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

**Rule:**  
Each column should contain only **atomic** values, and repeating groups must be eliminated.

**1NF Table:**

| **StudentID** | **StudentName** | **SubjectID** | **SubjectName** | **TeacherName** | **TeacherEmail** |
| --- | --- | --- | --- | --- | --- |
| 201 | Rahul | ENG101 | English | Mr. Kapoor | kapoor@edu.in |
| 201 | Rahul | MTH102 | Math | Ms. Sharma | sharma@edu.in |
| 202 | Sneha | ENG101 | English | Mr. Kapoor | kapoor@edu.in |

## **Second Normal Form (2NF)**

## **Rule:** The table must be in 1NF, and all non-key attributes should be fully functionally dependent on the **entire composite key** (StudentID, SubjectID). We must remove **partial dependencies**.

| **StudentID** | **StudentName** |
| --- | --- |
| 201 | Rahul |
| 202 | Sneha |

## **Third Normal Form (3NF)**

## **Rule:** A table is in 3NF if it is in 2NF and contains **no transitive dependencies** (non-key attributes depending on other non-key attributes).

**Identified Transitive Dependency:**

* TeacherEmail depends on TeacherName, not directly on SubjectID.

| **StudentID** | **StudentName** |
| --- | --- |
| 201 | Rahul |
| 202 | Sneha |

**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

**ANSWERS:**  
a)

CREATE DATABASE StudentDB;

USE StudentDB;

-----------------------------------

b)CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100)

);

--------------------------------

c) RENAME TABLE Students TO Student\_Info;

---------------------------------------

d)ALTER TABLE Student\_Info

ADD PhoneNumber VARCHAR(15);

------------------------------------

d)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.

**ANSWERS:**

a) USE StudentDB;

CREATE TABLE Student\_Info (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100),

PhoneNumber VARCHAR(15)

);

INSERT INTO Student\_Info (StudentID, Name, DOB, Email, PhoneNumber) VALUES

(1, 'Ram', '2001-05-20', 'ram@gmail.com', '9876543210'),

(2, 'Priya', '1999-08-15', 'priya@yahoo.com', '9123456780'),

(3, 'Anil', '2002-12-10', 'anil@gmail.com', '7890123456');

----------------------------------------

b) UPDATE Student\_Info

SET PhoneNumber = '9999999999'

WHERE StudentID = 2;

-------------------

c) DELETE FROM Student\_Info

WHERE Email LIKE 'priya@gmail.com'

LIMIT 1;

------------------------

d) SELECT Name, Email

FROM Student\_Info

WHERE YEAR(DOB) > 2000;

------------------------

e) 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.

**ANSWERS:**

1. SELECT \*

FROM Student\_Info

WHERE Name LIKE 'A%';

1. SELECT \* FROM Student\_Info

WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';

------------------

1. ALTER TABLE Student\_Info ADD City VARCHAR(50);

UPDATE Student\_Info SET City = 'Hyderabad' WHERE StudentID = 1;

UPDATE Student\_Info SET City = 'Chennai' WHERE StudentID = 2;

UPDATE Student\_Info SET City = 'Delhi' WHERE StudentID = 3;

SELECT \* FROM Student\_Info

WHERE City IN ( 'Hyderabad', 'Chennai');

-----------------------

1. ALTER TABLE Student\_Info ADD Age INT;

UPDATE Student\_Info

SET Age = TIMESTAMPDIFF(YEAR, DOB, CURDATE());

SELECT

Name,

DOB,

TIMESTAMPDIFF(YEAR, DOB, CURDATE()) AS Age

FROM

Student\_Info;

-------------------------------

e) SELECT \*

FROM Student\_Info

WHERE

TIMESTAMPDIFF(YEAR, DOB, CURDATE()) > 20

AND (

Email LIKE '%@gmail.com'

OR Email LIKE '%@yahoo.com'

);

SELECT

s.Name AS Student\_Name,

s.DOB AS Date\_of\_Birth

FROM

Student\_Info AS s;

**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.

**ANSWERS:**

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks INT

);

INSERT INTO Marks (StudentID, Subject, Marks) VALUES

(1, 'Maths', 85),

(2, 'Science', 75),

(3, 'English', 65),

(1, 'Science', 55),

(2, 'Maths', 95);

1. SELECT StudentID, Subject

FROM Marks

WHERE Marks > 70;

----------------

1. SELECT Subject, AVG(Marks) AS Average\_Marks

FROM Marks

GROUP BY Subject;

---------------

1. SELECT Subject, AVG(Marks) AS Average\_Marks

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.

**ANSWERS:**

1. SELECT DATE\_FORMAT(CURDATE(), '%Y-%m-%d') AS Formatted\_Date;

----------------------

1. SELECT

Name,

MONTH(DOB) AS Birth\_Month,

YEAR(DOB) AS Birth\_Year

FROM

Student\_Info;

-----------------------

1. SELECT

UPPER(Name) AS Uppercase\_Name

FROM

Student\_Info;

--------------------------------

1. SELECT

StudentID,

Subject,

ROUND(CAST(Marks AS DECIMAL(5,2)), 2) AS Rounded\_Marks

FROM

Marks;

---------------------------

1. SELECT DATABASE() AS Current\_Database;

**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

**ANSWERS:**

1. SELECT

StudentID,

SUM(Marks) AS Total\_Marks

FROM

Marks

GROUP BY

StudentID;

------------------------

1. SELECT

Subject,

MAX(Marks) AS Highest\_Mark

FROM

Marks

GROUP BY

Subject;

----------------------

1. SELECT

Subject,

AVG(Marks) AS Average\_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.

**ANSWERS:**

CREATE TABLE Courses (

CourseID INT PRIMARY KEY,

CourseName VARCHAR(100)

);

INSERT INTO Courses (CourseID, CourseName) VALUES

(101, 'Maths'),

(102, 'Science'),

(103, 'English'),

(104, 'History'); -- Unenrolled course for right/full join

CREATE TABLE Enrollments (

StudentID INT,

CourseID INT,

FOREIGN KEY (StudentID) REFERENCES Student\_Info(StudentID),

FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)

);

INSERT INTO Enrollments (StudentID, CourseID) VALUES

(1, 101),

(1, 102),

(2, 101),

(3, 103);

a) SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info s

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

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

---------------------------

b) SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info s

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

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

------------------------

c) SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info s

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

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

-------------------------------

d) SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info s

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

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

UNION

SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info s

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

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

------------------------

e) SELECT

s.Name AS Student\_Name,

c.CourseName AS Course

FROM

Student\_Info 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'.

**ANSWERS:**

1. SELECT s.Name, m.Subject, m.Marks

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject = 'Maths'

AND m.Marks > (

SELECT AVG(Marks)

FROM Marks

WHERE Subject = 'Maths'

);

----------------

1. INSERT INTO Student\_Info (StudentID, Name, DOB, Email, PhoneNumber, City, Age)

VALUES (5, 'Varun', '2004-01-10', 'varun@gmail.com', '9090909090', 'Mumbai', 20);

SELECT \*

FROM Student\_Info

WHERE StudentID NOT IN (

SELECT DISTINCT StudentID FROM Marks

);

---------------

1. SELECT \*

FROM Student\_Info s

WHERE EXISTS (

SELECT 1

FROM Marks m

WHERE m.StudentID = s.StudentID

);

------------------------------

1. SELECT s.Name, m.Subject, m.Marks

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Marks > ALL (

SELECT Marks

FROM Marks

WHERE Subject = 'Science'

);

-----------------------

1. SELECT s.Name, m.Subject, m.Marks

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject <> 'English'

AND m.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

**ANSWER:**

1. SELECT Name FROM Student\_Info

UNION

SELECT DISTINCT s.Name

FROM Marks m

JOIN Student\_Info s ON m.StudentID = s.StudentID;

--------------

1. SELECT DISTINCT s.Name

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID;

-------------

c)

-- Add a student who doesn't exist in Marks table

INSERT INTO Student\_Info (StudentID, Name, DOB, Email, PhoneNumber, City, Age)

VALUES (4, 'Divya', '2003-07-22', 'divya@gmail.com', '9012345678', 'Bangalore', 21);

SELECT s.Name

FROM Student\_Info s

LEFT JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.StudentID IS NULL;

--------------------------

d) -- Update if exists

UPDATE Marks

SET Marks = 88

WHERE StudentID = 3 AND Subject = 'English';

-- Insert if not exists

INSERT INTO Marks (StudentID, Subject, Marks)

SELECT 3, 'English', 88

FROM DUAL

WHERE NOT EXISTS (

SELECT 1 FROM Marks WHERE StudentID = 3 AND Subject = 'English'

);

e) SELECT m.StudentID, s.Name, m.Subject, m.Marks

FROM Marks m

JOIN Student\_Info s ON s.StudentID = m.StudentID

WHERE m.Marks > (

SELECT AVG(m2.Marks)

FROM Marks m2

WHERE m2.Subject = m.Subject

);

**SQL Practical Question Paper-2**

**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.

**When is NoSQL Preferred Over SQL?**

1. **Unstructured or Semi-structured Data**:  
   Example: Social media posts.  
   SQL struggles with unpredictable schemas.
2. **Horizontal Scalability is Required**:  
   Example: Large-scale web apps like Facebook or Amazon.  
   NoSQL scales easily across multiple servers (nodes).
3. **High-Speed Read/Write Performance**:  
   Example: Real-time analytics, IoT apps.
4. NoSQL handles rapid data ingestion better than SQL.
5. **Flexible Schema Requirements**:  
   Example: E-commerce product listings where attributes differ (e.g., shoes vs. laptops).  
   NoSQL allows schema-less or dynamic schema.

| **Type** | **Description** | **Real-time Application Example** |
| --- | --- | --- |
| **1. Document-based** | Stores data as JSON-like documents (key-value pairs) | **MongoDB** in e-commerce for flexible product catalogs |
| **2. Key-Value Store** | Stores data as a collection of key-value pairs | **Redis** for caching user sessions in a web app |
| **3. Column-family** | Data stored in columns instead of rows | **Cassandra** for time-series data in IoT or stock trading |
| **4. Graph-based** | Stores data as nodes and edges (relationships) | **Neo4j** for social networks like LinkedIn or fraud detection |

**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.

**ANSWERS:**

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

Break nested data:

Customer\_Order (  
 CustomerID, Name, OrderID, ProductID, Quantity, ProductName  
)

### 2NF (No partial dependency):

**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.

**ANSWERS:**

a) CREATE DATABASE RetailDB;

USE RetailDB;

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Email VARCHAR(100)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Price DECIMAL(10,2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT,

OrderDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

------------------------

b) ALTER TABLE Orders

ADD CONSTRAINT chk\_quantity\_positive CHECK (Quantity > 0);

------------------------

c) ALTER TABLE Products

ADD Discount DECIMAL(5,2);

UPDATE Products

SET Discount = 10.00

WHERE ProductID = 1;

UPDATE Products

SET Discount = 5.50

WHERE ProductID = 2;

**Q4.** (5 marks) Using the above schema:

a) Insert 3 sample orders per customer.

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

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

**ANSWERS:**

a) INSERT INTO Customers (CustomerID, CustomerName, Email) VALUES

(1, 'Alice', 'alice@example.com'),

(2, 'Bob', 'bob@example.com'),

(3, 'Charlie', 'charlie@example.com');

INSERT INTO Products (ProductID, ProductName, Price, Discount) VALUES

(1, 'Laptop', 50000.00, 10.00),

(2, 'Mouse', 800.00, 5.50),

(3, 'Keyboard', 1500.00, NULL);

INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity, OrderDate) VALUES

(1, 1, 1, 2, '2025-06-10'),

(2, 1, 2, 3, '2025-06-11'),

(3, 1, 3, 1, '2025-06-12'),

(4, 2, 1, 6, '2025-06-10'),

(5, 2, 2, 1, '2025-06-11'),

(6, 2, 3, 2, '2025-06-12'),

(7, 3, 1, 4, '2025-06-10'),

(8, 3, 2, 2, '2025-06-11'),

(9, 3, 3, 1, '2025-06-12');

---------------------------

b) UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID

FROM Orders

GROUP BY ProductID

HAVING SUM(Quantity) > 5

);

-------------------------------

c) SELECT ProductID 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.

b) Products not ordered by any customer.

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

**ANSWERS:**

a) SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

-----------------------------

b) SELECT \*

FROM Products

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID

FROM Orders

);

--------------------------

c) SELECT CustomerID, COUNT(\*) AS OrderCount

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.

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

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

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

**ANSWERS:**

a) SELECT

CustomerName,

Email,

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

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

FROM Customers;

--------------------------------------

b) SELECT

OrderID,

CustomerID,

OrderDate,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

-------------------------------------

c) SELECT

CURRENT\_USER() AS CurrentUser,

USER() AS SessionUser;

----------------------------------

d) SELECT

CONCAT('Hello, ', UPPER(SUBSTRING(CustomerName, 1, 1)),

LOWER(SUBSTRING(CustomerName, 2))) AS Greeting

FROM Customers;

**Q7.** (5 marks)

a) Aggregate total revenue by product category.

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

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

**ANSWER:**

a) SELECT

p.Category,

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

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category;

--------------------------------

b) SELECT

p.Category,

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

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category WITH ROLLUP;

--------------------------------

c) SELECT

p.Category,

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

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category

HAVING SUM(p.Price \* o.Quantity) > 100000;

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

Q8. (5 marks)

a) Self join to list customers referred by other customers.

b) Equi join across Orders and Products.

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

d) LEFT OUTER JOIN with WHERE NULL to identify inactive customers.

e) Cross join for all product combinations in a bundle offer.

**ANSWERS:**

a) SELECT

c.CustomerName AS ReferredCustomer,

r.CustomerName AS Referrer

FROM Customers c

JOIN Customers r ON c.ReferredBy = r.CustomerID;

------------------------------------------

b) SELECT

o.OrderID,

o.ProductID,

p.ProductName,

o.Quantity,

p.Price,

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

FROM Orders o

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

c) SELECT \*

FROM (

SELECT

c.CustomerID,

c.CustomerName,

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

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

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY c.CustomerID, c.CustomerName

) AS RankedSpenders

WHERE SpenderRank <= 3;

----------------------------------------------------------------

d) SELECT

c.CustomerID,

c.CustomerName

FROM Customers c

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

WHERE o.OrderID IS NULL;

-------------------------------------------

e) SELECT

p1.ProductName AS Product1,

p2.ProductName AS Product2

FROM Products p1

CROSS JOIN Products p2

WHERE p1.ProductID < p2.ProductID;

Q9. (5 marks)

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

SELECT o.\*

FROM Orders o

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

(

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

FROM Orders o2

JOIN Products p2 ON o2.ProductID = p2.ProductID

WHERE o2.CustomerID = o.CustomerID

);

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

SELECT DISTINCT c.CustomerID, c.CustomerName

FROM Customers c

WHERE EXISTS (

SELECT 1

FROM Orders o

WHERE o.CustomerID = c.CustomerID

GROUP BY o.CustomerID

HAVING COUNT(DISTINCT o.ProductID) >= 2

);

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

SELECT CustomerID, CustomerName

FROM Customers

WHERE CustomerID IN (

SELECT o.CustomerID

FROM Orders o

GROUP BY o.CustomerID

HAVING COUNT(\*) > ALL (

SELECT COUNT(\*)

FROM Orders

GROUP BY CustomerID

HAVING CustomerID <> o.CustomerID

)

);

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

SELECT \*

FROM Products

WHERE Price > ANY (

SELECT Price

FROM Products

WHERE Category = 'Electronics'

);

e) Nested subquery to list top 3 best-selling products.

SELECT ProductID, ProductName, TotalSold

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, p.ProductName

) AS ranked\_products

WHERE rnk <= 3;

**Q10.** (5 marks)

a) Simulate INTERSECT using INNER JOIN on two customer segments.

SELECT pc.CustomerID, c.CustomerName

FROM (

SELECT o.CustomerID

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY o.CustomerID

HAVING SUM(o.Quantity \* p.Price) > 50000

) AS pc

JOIN (

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(\*) > 2

) AS fb ON pc.CustomerID = fb.CustomerID

JOIN Customers c ON c.CustomerID = pc.CustomerID;

b) Use EXCEPT to find products in inventory not yet ordered.

SELECT ProductID, ProductName

FROM Products

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID FROM Orders

);

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

UPDATE Customers

SET CustomerName = 'Updated Name', Email = 'updated@example.com'

WHERE CustomerID = 5;

-- If no row was updated, insert:

INSERT INTO Customers (CustomerID, CustomerName, Email)

SELECT 5, 'Updated Name', 'updated@example.com'

WHERE NOT EXISTS (

SELECT 1 FROM Customers WHERE CustomerID = 5

);

d) Use UNION to combine two regional customer tables. e) Write a WITH CTE that ranks customers by total spend and filters top 5

WITH RankedCustomers AS (

SELECT

c.CustomerID,

c.CustomerName,

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

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, c.CustomerName

)

SELECT \*

FROM RankedCustomers

WHERE rnk <= 5;