13/6/25 Question Paper

1. Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.

SQL (Relational)

- Table with rows & columns
- Strict schema — structure cannot change easily
- Safe and reliable (ACID) — perfect for banking, payments
- Scales vertically (increase server power)
- Examples: MySQL, Oracle, SQL Server

➥ Advantages: Reliable, Standard
➥ Disadvantages: Less flexible, Hard to scale

Real-Life Example: Bank System

NoSQL (Non-Relational)

- Stores JSON, documents, graphs, or key-value pairs
- Schema-less — easy to modify
- Eventually consistent — fine for social media, blogs
- Scales horizontally (add more nodes)
- Examples: MongoDB, Neo4j, Redis

➥ Advantages: Flexible, Easily scalable
➥ Disadvantages: Less standard, Weaker consistency

Real-Life Example: Instagram or Facebook: Instagram

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

1NF (First Normal Form) :

Each cell contains **atomic values**.
Primary Key = (StudentID, CourseID)

CREATE TABLE StudentCourse (StudentID INT,Name VARCHAR(100),CourseID INT,CourseNameVARCHAR(100),InstructorName VARCHAR(100), InstructorPhone VARCHAR(15), PRIMARY KEY (StudentID, CourseID));

2NF (remove partial dependency — Course details separate)

CREATE TABLE Student (StudentID INT PRIMARY KEY, Name VARCHAR(100));

CREATE TABLE Course (CourseID INT PRIMARY KEY, CourseName VARCHAR(100), InstructorName VARCHAR(100), InstructorPhone VARCHAR(15));

CREATE TABLE StudentCourse (StudentID INT, CourseID INT, PRIMARY KEY (StudentID, CourseID), FOREIGN KEY (StudentID) REFERENCES Student(StudentID), FOREIGN KEY (CourseID) REFERENCES Course(CourseID));

3NF (remove transitive dependency — Instructor separate):

CREATE TABLE Student (StudentID INT PRIMARY KEY, Name VARCHAR(100));

CREATE TABLE Instructor (InstructorID INT PRIMARY KEY, InstructorName VARCHAR(100), InstructorPhone VARCHAR(15));

CREATE TABLE Course (CourseID INT PRIMARY KEY, CourseName VARCHAR(100), InstructorID INT, FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID));

CREATE TABLE StudentCourse (StudentID INT, CourseID INT, PRIMARY KEY (StudentID, CourseID), FOREIGN KEY (StudentID) REFERENCES Student(StudentID), FOREIGN KEY (CourseID) REFERENCES Course(CourseID));

3.a) Create a database named StudentDB.

CREATE DATABASE StudentDB;

b) Create a table Students with fields: StudentID, Name, DOB, Email.

CREATE TABLE Students (StudentID INT, Name VARCHAR(100), DOB DATE, Email VARCHAR(100));

c) Rename the table to Student_Info.

```
ALTER TABLE Students RENAME TO Student_Info;
```

d) Add a column PhoneNumber.

```
ALTER TABLE Student_Info ADD COLUMN PhoneNumber VARCHAR(15);
```

e) Drop the table.

```
DROP TABLE Student_Info;
```

Section B: DML & Filtering Data (15 Marks)

4.a) Insert 3 student records into Student_Info.

```
INSERT INTO Student_Info (StudentID, Name, DOB, Email, PhoneNumber) VALUES
(1, 'Harci Niha Jhara', '2001-05-21', 'harci@example.com', '555-1234'),
(2, 'Jhara Harci Niha', '1999-04-12', 'jhara@gmail.com', '555-5678'),
(3, 'Niha Harci Jhara', '2002-09-30', 'niha@example.com', '555-8765');
```

b) Update one student's phone number.

```
UPDATE Student_Info SET PhoneNumber = '555-9999' WHERE StudentID = 1;
```

c) Delete one student whose email ends with @gmail.com.

```
DELETE FROM Student_Info WHERE Email LIKE '%@gmail.com';
```

d) Retrieve only names and emails of students born after the year 2000.

```
SELECT Name, Email FROM Student_Info WHERE DOB > '2000-01-01';
```

e) Retrieve distinct domain names from the email column.

```
SELECT DISTINCT SUBSTRING(Email, INSTR(Email, '@') + 1) AS DomainName FROM Student_Info;
```

5. a) Retrieve students with names starting with 'A'.

```
SELECT * FROM Student_Info WHERE Name LIKE 'A%';
```

b) Retrieve students with phone number between 9000000000 and 9999999999.

```
SELECT * FROM Student_Info WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';
```

c) Retrieve students using IN operator on city names.

```
SELECT * FROM Student_Info WHERE City IN ('Chennai', 'Bangalore', 'Hyderabad');
```

d) Use AND, OR to filter students based on age and email provider.

```
SELECT * FROM Student_Info
WHERE (DATEDIFF(CURDATE(), DOB)/365 > 21 AND Email LIKE '%@example.com')
   OR (DATEDIFF(CURDATE(), DOB)/365 < 18);
```

e) Use table and column aliasing in a query to get all student names and DOBs.

```
SELECT S.Name AS StudentName, S.DOB AS DateOfBirth FROM Student_Info AS S;
```

6. Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows.

```
CREATE TABLE Marks (StudentID INT, Subject VARCHAR(100), Marks INT);

INSERT INTO Marks (StudentID, Subject, Marks) VALUES

(1, 'Mathematics', 85),

(2, 'Physics', 72),

(3, 'Chemistry', 65);
```

a) Display student IDs and their subjects where marks > 70

```
SELECT StudentID, Subject FROM Marks WHERE Marks > 70;
```

b) Display subjects with average marks.

```
SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks GROUP BY Subject;
```

c) Filter subjects with average marks between 60 and 90.

```
SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks GROUP BY Subject HAVING AverageMarks
BETWEEN 60 AND 90;
```

Section C: Functions & Grouping

7. a) Get the current date and format it as "YYYY-MM-DD"

```
SELECT DATE_FORMAT(CURDATE(), '%Y-%m-%d') AS FormattedDate;
```

b) Extract month and year from a DOB column

```
SELECT StudentID, Name, YEAR(DOB) AS Year, MONTH(DOB) AS Month FROM
Student_Info;
```

c) Convert a student's name to uppercase

```
SELECT StudentID, UPPER(Name) AS NameInUpperCase FROM Student_Info;
```

d) Round off marks to 2 decimal places

```
SELECT StudentID, Subject, ROUND(Marks, 2) AS RoundedMarks FROM Marks;
```

e) Use system function to return user name or current database.

```
SELECT USER() AS CurrentUser, DATABASE() AS CurrentDatabase;
```

8. a) Display total marks of each student.

```
SELECT StudentID, SUM(Marks) AS TotalMarks FROM Marks GROUP BY StudentID;
```

b) Display subject-wise highest mark.

```
SELECT Subject, MAX(Marks) AS HighestMark FROM Marks GROUP BY Subject;
```

c) Use GROUP BY and HAVING to display subjects with average marks > 75

```
SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks GROUP BY Subject HAVING AverageMarks
>75;
```

Section D: Joins and Subqueries

9.a) Inner Join to retrieve students and their courses.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
INNER JOIN Marks ON Student_Info.StudentID = Marks.StudentID;
```

b) Left Join to get all students even if not enrolled.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
LEFT JOIN Marks ON Student_Info.StudentID = Marks.StudentID;
```

c) Right Join to get all courses even if no students.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
RIGHT JOIN Marks ON Student_Info.StudentID = Marks.StudentID;
```

e) Full Outer Join equivalent using UNION.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
LEFT JOIN Marks ON Student_Info.StudentID = Marks.StudentID
UNION
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
RIGHT JOIN Marks ON Student_Info.StudentID = Marks.StudentID;
```

f) Cross Join to show all combinations.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject FROM Student_Info
CROSS JOIN Marks;
```

10. a) Students who scored more than average in 'Maths'.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Marks
FROM Student_Info
JOIN Marks ON Student_Info.StudentID = Marks.StudentID
WHERE Marks.Subject = 'Maths'
  AND Marks.Marks > (SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths');
```

b) Students not in the Marks table.

```
SELECT Student_Info.StudentID, Student_Info.Name
FROM Student_Info
LEFT JOIN Marks ON Student_Info.StudentID = Marks.StudentID
WHERE Marks.StudentID IS NULL;
```

c) Use EXISTS to get students with at least one subject.

```
SELECT Student_Info.StudentID, Student_Info.Name
FROM Student_Info
WHERE EXISTS (SELECT 1 FROM Marks WHERE Marks.StudentID = Student_Info.StudentID);
```

d) Use ALL to find those scoring more than all in 'Science'.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Marks
FROM Student_Info
JOIN Marks ON Student_Info.StudentID = Marks.StudentID
```

```
WHERE Marks.Marks > ALL (SELECT Marks.Marks FROM Marks WHERE Subject = 'Science');
```

e) Use ANY for students scoring better than some in 'English'.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Marks
FROM Student_Info
JOIN Marks ON Student_Info.StudentID = Marks.StudentID
WHERE Marks.Marks > ANY (SELECT Marks.Marks FROM Marks WHERE Subject = 'English');
```

11. a) UNION of student names from two tables.

```
SELECT Name FROM Student_Info
UNION
SELECT Name FROM Marks;
```

b) INTERSECT to find common students.

```
SELECT Student_Info.StudentID, Student_Info.Name
FROM Student_Info
INNER JOIN Marks ON Student_Info.StudentID = Marks.StudentID;
```

c) EXCEPT to list students in Students but not in Marks.

```
SELECT Student_Info.StudentID, Student_Info.Name
FROM Student_Info
LEFT JOIN Marks ON Student_Info.StudentID = Marks.StudentID
WHERE Marks.StudentID IS NULL;
```

d) MERGE concept or simulate with UPDATE and INSERT.

```
UPDATE Student_Info SET PhoneNumber = '555-0000' WHERE StudentID = 1;
INSERT INTO Student_Info (StudentID, Name, DOB, Email, PhoneNumber)
SELECT 4, 'New Student', '2000-04-04', 'new@example.com', '555-1111'
FROM DUAL
WHERE NOT EXISTS (SELECT 1 FROM Student_Info WHERE StudentID = 4);
```

e) Correlated subquery to list students with above average per subject.

```
SELECT Student_Info.StudentID, Student_Info.Name, Marks.Subject, Marks.Marks
FROM Student_Info
JOIN Marks ON Student_Info.StudentID = Marks.StudentID
WHERE Marks.Marks > (
   SELECT AVG(Marks.Marks)
   FROM Marks
   WHERE Marks.Subject = Marks.Subject );
```

SQL Practical Question Paper
Section A: Advanced Concepts & Schema Design

1. 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 you have large volumes of unstructured or semi-structured data (like social media messages, blogs, reviews).
- When your application requires high scalability and flexible schema (schema may frequently change).
- When you need real-time performance and low latency with massive amounts of data.
- When you want horizontal scaling across multiple nodes instead of growing vertically.

Types of NoSQL Databases with Real-world Applications:

Type	Description	Real-world Application
Document Store	Stores data in documents (JSON, BSON)	Blog platform (Wordpress), Product catalog
Key-Value Store	Stores data as key-value pairs	Caching (Redis), User sessions
Column Store	Stores data by column instead of row	Large-scale analytical platforms (Cassandra, HBase)
Graph Database	Stores data in nodes and relationships	Social networks (Instagram, Facebook), Recommendation engines

2.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.

Customer (CustomerID PK, Name)

Product (ProductID PK, ProductName)

Order (OrderID PK, CustomerID, ProductID, Quantity)

Section B: Complex DDL and DML

3. a) Create a database RetailDB and design a schema for Customers, Orders, and Products with primary and foreign keys.

```
CREATE DATABASE RetailDB;
USE RetailDB;
CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  Name VARCHAR(100)
);
CREATE TABLE Products (
```

```
  ProductID INT PRIMARY KEY,
  ProductName VARCHAR(100)
);
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  CustomerID INT,
  ProductID INT,
  Quantity INT CHECK (Quantity > 0),
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
  FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

b) Implement a check constraint on Quantity (>0) in Orders.

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  CustomerID INT,
  ProductID INT,
  Quantity INT CHECK (Quantity > 0),
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
  FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

c) Alter the Products table to add 'Discount' column and update some values

```
ALTER TABLE Products ADD COLUMN Discount DECIMAL(5, 2);
UPDATE Products SET Discount = 10.00 WHERE ProductID = 1;
UPDATE Products SET Discount = 5.00 WHERE ProductID = 2;
```

4. **Using the above schema:**

a) Insert 3 sample orders per customer.

```
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (1, 1, 1, 2);
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (2, 1, 2, 5);
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (3, 1, 3, 7);
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (4, 2, 1, 1);
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (5, 2, 2, 6);
INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity) VALUES (6, 2, 3, 4);
```

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

```
UPDATE Products
SET Discount = Discount + (Discount * 0.10)
WHERE ProductID IN (SELECT ProductID FROM Orders WHERE Quantity > 5);
```

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

```
DELETE FROM Orders
WHERE ProductID NOT IN (SELECT ProductID FROM Orders);
```

5. Retrieve the following:

a) Customers who ordered more than 3 different products.

```
SELECT CustomerID, Name
FROM Customers
JOIN Orders USING (CustomerID)
GROUP BY CustomerID, Name
HAVING COUNT(DISTINCT ProductID) > 3;
```

b) Products not ordered by any customer.

```
SELECT ProductID, ProductName
FROM Products
LEFT JOIN Orders USING (ProductID)
WHERE Orders.ProductID IS NULL;
```

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

```
SELECT CustomerID, Name, COUNT(OrderID) AS OrdersLast30
FROM Customers
LEFT JOIN Orders USING (CustomerID)
WHERE OrderDate >= DATEADD(day, -30, GETDATE())
GROUP BY CustomerID, Name;
```

Section C: Advanced Functions and Aggregations

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

```
SELECT CustomerID,
LOWER(Email) AS StandardizedEmail,
RIGHT(Email, CHARINDEX('@', Email) - 1) AS Username,
SUBSTRING(Email, CHARINDEX('@', Email) + 1, LEN(Email)) AS Domain
FROM Customers;
```

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

```
SELECT OrderID,
DATEDIFF(day, OrderDate, GETDATE()) AS DaysDiff
FROM Orders;
```

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

```
SELECT SYSTEM_USER AS CurrentUser, HOST_NAME() AS CurrentHost;
```

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

```
SELECT CustomerID,
```

```
CONCAT('Hello, ',
UPPER(Name),
'!') AS Greeting
FROM Customers;
```

7. a) Aggregate total revenue by product category.

```
SELECT ProductCategory, SUM(Quantity * Price) AS TotalRevenue
FROM Orders
JOIN Products USING (ProductID)
GROUP BY ProductCategory;
```

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

```
SELECT ProductCategory, ProductName, SUM(Quantity * Price) AS Revenue
FROM Orders
JOIN Products USING (ProductID)
GROUP BY ROLLUP (ProductCategory, ProductName);
```

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

```
SELECT ProductCategory, SUM(Quantity * Price) AS TotalRevenue
FROM Orders
JOIN Products USING (ProductID)
GROUP BY ProductCategory
HAVING SUM(Quantity * Price) > 100000;
```

Section D: Complex Joins, Subqueries, and Set Ops

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

```
SELECT C1.CustomerID, C1.Name AS Customer, C2.CustomerID AS ReferredBy, C2.Name AS ReferredByName
FROM Customers C1
JOIN Customers C2 ON C1.ReferredBy = C2.CustomerID;
```

b) Equi join across Orders and Products.

```
SELECT Orders.OrderID, Customers.Name, Products.ProductName, Orders.Quantity
FROM Orders
JOIN Customers USING (CustomerID)
JOIN Products USING (ProductID);
```

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

```
SELECT CustomerID, Name, SUM(Quantity * Price) AS TotalSpent,
RANK() OVER (ORDER BY SUM(Quantity * Price) DESC) AS Rank
FROM Customers
JOIN Orders USING (CustomerID)
JOIN Products USING (ProductID)
```

```
GROUP BY CustomerID, Name
HAVING Rank <= 3;
```

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

```
SELECT Customers.CustomerID, Customers.Name
FROM Customers
LEFT JOIN Orders USING (CustomerID)
WHERE Orders.CustomerID IS NULL;
```

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

```
SELECT P1.ProductName AS Product1, P2.ProductName AS Product2
FROM Products P1
CROSS JOIN Products P2
WHERE P1.ProductID <> P2.ProductID;
```

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

```
SELECT CustomerID, Name
FROM Customers C
WHERE EXISTS (
  SELECT 1 FROM Orders O
  WHERE O.CustomerID = C.CustomerID
    AND (O.Quantity * P.Price) > (
    SELECT AVG(Quantity * Price)
      FROM Orders O2
      JOIN Products P USING (ProductID)
      WHERE O2.CustomerID = C.CustomerID
    )
);
```

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

```
SELECT CustomerID, Name
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, Name, SUM(Quantity) AS QuantityOrdered
FROM Customers C
JOIN Orders O USING (CustomerID)
```

```
GROUP BY CustomerID, Name
HAVING SUM(Quantity) > ALL (
  SELECT SUM(Quantity)
  FROM Orders
  GROUP BY CustomerID
);
```

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

```
SELECT ProductID, ProductName, Price
FROM Products
WHERE Price > ANY (
  SELECT Price FROM Products WHERE ProductCategory = 'Electronics'
);
```

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

```
SELECT ProductID, ProductName, SUM(Quantity) AS QuantitySold
FROM Orders
JOIN Products USING (ProductID)
GROUP BY ProductID, ProductName
ORDER BY QuantitySold DESC
FETCH FIRST 3 ROWS ONLY;
```

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

```
SELECT C1.CustomerID, C1.Name
FROM CustomerSegment1 C1
INNER JOIN CustomerSegment2 C2 ON C1.CustomerID = C2.CustomerID;
```

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

```
SELECT ProductID, ProductName FROM Products
EXCEPT
SELECT ProductID, ProductName FROM Orders JOIN Products USING (ProductID);
```

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

```
MERGE INTO Customers AS C USING StagingTable AS S
ON (C.CustomerID = S.CustomerID)
WHEN MATCHED THEN
UPDATE SET C.Name = S.Name
WHEN NOT MATCHED THEN
  INSERT (CustomerID, Name) VALUES (S.CustomerID, S.Name);
```

d) Use UNION to combine two regional customer tables.

```
SELECT CustomerID, Name FROM Customers_RegionA
UNION
SELECT CustomerID, Name FROM Customers_RegionB;
```

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

```
WITH CustomerSpend AS (
  SELECT CustomerID, Name, SUM(Quantity * Price) AS TotalSpend,
  RANK() OVER (ORDER BY SUM(Quantity * Price) DESC) AS Rank
  FROM Customers
  JOIN Orders USING (CustomerID)
  JOIN Products USING (ProductID)
  GROUP BY CustomerID, Name
)
SELECT CustomerID, Name, TotalSpend
FROM CustomerSpend
WHERE Rank <= 5;
```

13/6/25-Task Assignment

1) Querying Data by Using Subqueries

```
SELECT Name FROM Customers
WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

2) Querying Data by Using Subqueries Using the EXISTS

```
SELECT Name FROM Customers C
WHERE EXISTS (SELECT 1 FROM Orders O WHERE O.CustomerID = C.CustomerID);
```

3) Querying Data by Using Subqueries using ANY,

```
SELECT ProductName FROM Products
  WHERE Price > ANY (SELECT Price FROM Products WHERE ProductCategory = 'Electronics');
```

4) Querying Data by Using Subqueries using ALL Keywords

```
SELECT ProductName FROM Products
WHERE Price > ALL (SELECT Price FROM Products WHERE ProductCategory = 'Accessories');
```

5) Querying Data by Using Subqueries using Using Nested Subqueries

```
SELECT Name FROM Customers
WHERE CustomerID IN (
 SELECT CustomerID FROM Orders
 WHERE ProductID IN (
  SELECT ProductID FROM Products WHERE ProductCategory = 'Books'
 )
);
```

6) Querying Data by Using Subqueries Using Correlated Subqueries

```
SELECT C.CustomerID, C.Name
FROM Customers C
WHERE (SELECT SUM(Quantity) FROM Orders O WHERE O.CustomerID = C.CustomerID) > 10;
```

7) Querying Data by Using Subqueries Using UNION,

```
SELECT CustomerID, Name FROM Customers_1
UNION
SELECT CustomerID, Name FROM Customers_2;
```

8) Querying Data by Using Subqueries using INTERSECT,

```
SELECT CustomerID FROM Customers_1
INTERSECT
SELECT CustomerID FROM Customers_2;
```

9) Querying Data by Using Subqueries using EXCEPT,

```
SELECT CustomerID FROM Customers_1
```

```
EXCEPT
 SELECT CustomerID FROM Customers_2;
```

10) Querying Data by Using Subqueries using MERGE"

```
MERGE INTO Customers AS C USING StagingTable AS S
ON (C.CustomerID = S.CustomerID)
WHEN MATCHED THEN
  UPDATE SET C.Name = S.Name
WHEN NOT MATCHED THEN
  INSERT (CustomerID, Name) VALUES (S.CustomerID, S.Name);
```

11/6/25 Assignment

Section A: Managing Databases (10 mins)

1. List all system databases in SQL Server.

```
SELECT schema_name
FROM information_schema.schemata
WHERE schema_name IN ('mysql', 'information_schema',   'performance_schema', 'sys');
```

2. List physical file paths for all databases.

```
SELECT table_schema, table_name, engine
FROM information_schema.tables
WHERE table_schema NOT IN ('mysql', 'information_schema', 'performance_schema', 'sys');
```

3. Create a new user-defined database named TeamDB

```
CREATE DATABASE TeamDB;
USE TeamDB;
```

4. Rename the database TeamDB to ProjectDB.

```
ALTER DATABASE TeamDB MODIFY NAME = ProjectDB;
```

5. Drop the ProjectDB database.

```
DROP DATABASE ProjectDB;
```

Section B: Managing Tables (10 mins)

1. Create a table Employees with the following columns: EmpID INT (Primary Key) Name VARCHAR(50) Department VARCHAR(30) JoiningDate DATE IsActive BIT Salary DECIMAL(10,2)

```
CREATE TABLE Employees ( EmpID INT PRIMARY KEY, Name VARCHAR(50), Department
  VARCHAR(30), JoiningDate DATE, IsActive BIT, Salary DECIMAL(10,2) );
```

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

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

3. Rename table Employees to TeamMembers.

```
EXEC sp_rename 'Employees', 'TeamMembers';
```

4. Drop the table TeamMembers.

```
DROP TABLE TeamMembers;
```

Section C: DML Operations (10 mins)

1. Insert three rows into Employees.

```
INSERT INTO Employees VALUES (1, 'Amit', 'HR', '2022-01-01', 1, 50000), (2,
'Sneha', 'IT', '2021-06-15', 1, 75000), (3, 'John', 'Finance', '2020-10-10', 0, 65000);
```

2. **Update salary of 'Sneha' to 80000**.

```
UPDATE Employees SET Salary = 80000 WHERE Name = 'Sneha';
```

3. **Delete employee with IsActive = 0.**

```
DELETE FROM Employees WHERE IsActive = 0;
```

4. **Retrieve names and departments of all employees.**

```
SELECT Name, Department FROM Employees;
```

5. **Fetch employees from 'IT' department with salary above 70000.**

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

6. **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');
```

12/6/25 Assignment

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.

Ans:
```
CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE,
  marks INT CHECK (marks >= 0 AND marks <= 100)
);
INSERT INTO students (student_id, name, email, marks) VALUES
(1, 'Anjali', 'anjali@example.com', 88),
(2, 'Rahul', 'rahul@example.com', 75),
(3, 'Sneha', 'sneha@example.com', 92),
(4, 'Vikram', 'vikram@example.com', 65),
(5, 'Divya', 'divya@example.com', 80);
SET SQL_SAFE_UPDATES = 0;
UPDATE students SET marks = 95 WHERE name = 'Sneha';
SET SQL_SAFE_UPDATES = 1;
```

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

Ans:
```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  full_name VARCHAR(100)
);
INSERT INTO customers (customer_id, full_name) VALUES
(1, 'Anjali Sharma'),
(2, 'Rahul Mehra'),
(3, 'Sneha Kapoor'),
(4, 'Vikram Singh'),
(5, 'Divya Joshi');
SELECT
  full_name,
  SUBSTRING_INDEX(full_name, ' ', 1) AS first_name
FROM customers;
SELECT
  full_name,
  SUBSTRING_INDEX(full_name, ' ', -1) AS last_name
FROM customers;
SELECT
  full_name,
  LENGTH(SUBSTRING_INDEX(full_name, ' ', 1)) AS first_name_length,
  LENGTH(SUBSTRING_INDEX(full_name, ' ', -1)) AS last_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

Ans:
```
CREATE TABLE sales (
```

```
 sale_id INT PRIMARY KEY,
 sale_date DATE
);
INSERT INTO sales (sale_id, sale_date) VALUES
(1, '2024-12-15'),
(2, '2025-01-10'),
(3, '2025-05-01'),
(4, '2025-06-10');
SELECT sale_id,sale_date,MONTHNAME(sale_date) AS month_name FROM sales;
SELECT sale_id,sale_date,YEAR(sale_date) AS year FROM sales;
SELECT sale_id,sale_date,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

Ans:

```
CREATE TABLE employees (
 emp_id INT PRIMARY KEY,
 name VARCHAR(50),
 salary DECIMAL(10,2)
);
INSERT INTO employees (emp_id, name, salary) VALUES
(1, 'Anjali', 28500),
(2, 'Rahul', 36750),
(3, 'Sneha', 42320),
(4, 'Vikram', 49999);
SELECT name,salary,salary * 1.10 AS salary_after_hike FROM employees;
SELECT name,salary,ROUND(salary, -2) AS rounded_salary FROM employees;
```

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

Ans:

```
SELECT NOW() AS current_datetime;
SELECT DATABASE() AS current_database;
SELECT USER() AS 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'

Ans:

```
CREATE TABLE products (
product_id INT PRIMARY KEY,
product_name VARCHAR(100),
price DECIMAL(10,2)
 );
 INSERT INTO products (product_id, product_name, price) VALUES
 (1, 'Laptop', 45000),
 (2, 'Mouse', NULL),
 (3, 'Keyboard', 1500),
 (4, 'Monitor', NULL);
 SELECT UPPER(product_name) AS product_name_upper,IFNULL(CAST(price AS
 CHAR), '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 (
 txn_id INT PRIMARY KEY,
 customer_name VARCHAR(50),
 amount DECIMAL(10,2)
);
INSERT INTO transactions (txn_id, customer_name, amount) VALUES
(1, 'Anjali', 1500),
(2, 'Rahul', 3200),
(3, 'Sneha', 2800),
(4, 'Vikram', 500),
(5, 'Neha', 4500);
SELECT SUM(amount) AS total_sales FROM transactions;
SELECT AVG(amount) AS average_sale FROM transactions;
SELECT MAX(amount) AS max_sale FROM transactions;
SELECT MIN(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 sale (
 sale_id INT PRIMARY KEY,
 category VARCHAR(50),
 amount DECIMAL(10,2)
);
INSERT INTO sale (sale_id, category, amount) VALUES
(1, 'Electronics', 2500),
(2, 'Clothing', 1200),
(3, 'Electronics', 4000),
(4, 'Groceries', 900),
(5, 'Clothing', 1800),
(6, 'Groceries', 600);
SELECT category,SUM(amount) AS total_sales FROM sale GROUP BY category;
SELECT category,COUNT(*) AS number_of_transactions FROM sale GROUP BY
category;
```

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

```
CREATE TABLE orders (
 order_id INT PRIMARY KEY,
 customer_id INT,
 amount DECIMAL(10,2)
);
SELECT c.customer_id,o.amount FROM customers c INNER JOIN orders o ON
c.customer_id = o.customer_id;
```

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

```
SELECT p.product_name,o.order_id,o.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

```
CREATE TABLE contacts (
  contact_id INT PRIMARY KEY,
  customer_id INT,
  phone VARCHAR(15),
  email VARCHAR(100)
);
INSERT INTO contacts (contact_id, customer_id, phone, email) VALUES
(1, 1, '9876543210', 'sneha@mail.com'),
(2, 3, '9123456780', 'vikram@mail.com');
SELECT c.customer_id,c.full_name AS customer_name,ct.phone,ct.email FROM contacts ct
RIGHT JOIN customers c ON ct.customer_id = c.customer_id;
```

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(100)
);
INSERT INTO suppliers VALUES
(1, 'Tata Supplies'),
(2, 'Nova Traders'),
(3, 'Eco Goods');
ALTER TABLE products ADD supplier_id INT;
UPDATE products SET supplier_id = 1 WHERE product_id = 101;
SELECT s.supplier_id,s.supplier_name,p.product_id,p.product_name FROM suppliers s
LEFT JOIN products p ON s.supplier_id = p.supplier_id
UNION
SELECT s.supplier_id,s.supplier_name,p.product_id,p.product_name FROM suppliers s
RIGHT JOIN products p ON s.supplier_id = p.supplier_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(100)
);
INSERT INTO offers VALUES
(1, '10% Discount'),
(2, 'Free Shipping'),
(3, 'Buy 1 Get 1');
SELECT p.product_id,p.product_name,o.offer_id,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

```
SELECT
  p.product_name,
  SUM(o.amount) AS total_quantity_sold,
  AVG(p.price) AS average_price
FROM orders o
JOIN products p ON o.product_id = p.product_id
GROUP BY p.product_name;
```

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 AUTO_INCREMENT,
  student_id INT,
  subject VARCHAR(50),
  marks INT,
  FOREIGN KEY (student_id) REFERENCES students(student_id)
);
INSERT INTO marks (student_id, subject, marks) VALUES
(1, 'Math', 88),
(1, 'Science', 92),
(2, 'Math', 76),
(2, 'Science', 72),
(3, 'Math', 60);
SELECT s.name AS student_name,AVG(m.marks) AS average_mark FROM students s
JOIN marks m ON s.student_id = m.student_id
GROUP BY s.name HAVING AVG(m.marks) > 75;
```