# 11-06-2025

**SQL Server Practical Assignment (30 Minutes)**

Section A: Managing Databases (10 mins)

**1. List all system databases in SQL Server.**

SELECT name FROM sys.databases WHERE database\_id < 5;

**2. List physical file paths for all databases.**

SELECT name, physical\_name FROM sys.master\_files;

**3. Create a new user-defined database named TeamDB.**

CREATE DATABASE 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');

# 12-06-2025

**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.

**CREATE TABLE students (**

**student\_id INT PRIMARY KEY,**

**name VARCHAR(100) NOT NULL UNIQUE,**

**marks INT NOT NULL**

**);**

**INSERT INTO students VALUES**

**(1, 'Alice', 85),**

**(2, 'Bob', 78),**

**(3, 'Charlie', 92),**

**(4, 'David', 88),**

**(5, 'Eva', 90);**

**UPDATE students SET marks = 95 WHERE name = 'Bob';**

**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 the customers table**

**CREATE TABLE customers (**

**customer\_id INTEGER PRIMARY KEY AUTOINCREMENT,**

**full\_name TEXT NOT NULL**

**);**

**-- Insert sample data**

**INSERT INTO customers (full\_name) VALUES**

**('John Doe'),**

**('Jane Smith'),**

**('Emily Johnson');**

**-- String function query to extract first and last names and their lengths**

**SELECT**

**full\_name,**

**SUBSTR(full\_name, 1, INSTR(full\_name, ' ') - 1) AS first\_name,**

**SUBSTR(full\_name, INSTR(full\_name, ' ') + 1) AS last\_name,**

**LENGTH(SUBSTR(full\_name, 1, INSTR(full\_name, ' ') - 1)) AS first\_name\_length,**

**LENGTH(SUBSTR(full\_name, INSTR(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

**CREATE TABLE sales (**

**sale\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**sale\_date DATE NOT NULL**

**);**

**INSERT INTO sales (sale\_date) VALUES**

**('2025-06-10'),**

**('2025-05-20'),**

**('2025-04-01');**

**SELECT**

**sale\_id,**

**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 AUTO\_INCREMENT PRIMARY KEY,**

**name VARCHAR(100) NOT NULL,**

**salary DECIMAL(10, 2) NOT NULL**

**);**

**INSERT INTO employees (name, salary) VALUES**

**('Alice', 48350.75),**

**('Bob', 55990.00),**

**('Charlie', 61240.25),**

**('Diana', 70000.00);**

**SELECT**

**emp\_id,**

**name,**

**salary,**

**ROUND(salary \* 1.10, 2) AS salary\_after\_10\_percent\_hike,**

**ROUND(salary, -2) AS rounded\_to\_nearest\_100**

**FROM employees;**

**5. System Function Check:**

Retrieve:

- Current date and time

- Database name and logged-in user

**SELECT**

**NOW() AS current\_datetime,**

**DATABASE() AS current\_database,**

**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'

**CREATE TABLE products (**

**product\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**product\_name VARCHAR(100) NOT NULL,**

**price DECIMAL(10, 2) DEFAULT NULL**

**);**

**INSERT INTO products (product\_name, price) VALUES**

**('Laptop', 75000.00),**

**('Tablet', NULL),**

**('Smartphone', 35000.00),**

**('Headphones', NULL);**

**SELECT**

**UPPER(product\_name) AS product\_name\_upper,**

**IFNULL(CAST(price AS CHAR), 'Not Available') AS price\_display**

**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 (**

**transaction\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**amount DECIMAL(10, 2)**

**);**

**INSERT INTO transactions (amount) VALUES**

**(250.00), (499.99), (120.75), (780.50), (350.25);**

**SELECT**

**SUM(amount) AS total\_sales,**

**AVG(amount) AS average\_sale\_value,**

**MAX(amount) AS max\_sale,**

**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 sales (**

**sale\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**product\_category VARCHAR(50),**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO sales (product\_category, sale\_amount) VALUES**

**('Electronics', 1200.50),**

**('Clothing', 750.00),**

**('Electronics', 500.00),**

**('Clothing', 300.25),**

**('Furniture', 1500.00);**

**SELECT**

**product\_category,**

**SUM(sale\_amount) AS total\_sales,**

**COUNT(\*) AS number\_of\_transactions**

**FROM sales**

**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

**CREATE TABLE customers (**

**customer\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**name VARCHAR(100)**

**);**

**CREATE TABLE orders (**

**order\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**customer\_id INT,**

**order\_amount DECIMAL(10, 2),**

**FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)**

**);**

**INSERT INTO customers (name) VALUES**

**('Alice'), ('Bob'), ('Charlie');**

**INSERT INTO orders (customer\_id, order\_amount) VALUES**

**(1, 500.00),**

**(2, 1200.75),**

**(1, 300.00); -- Charlie didn't order**

**10. Left Join for Products with or without Orders:**

Show all products with:

- Their order details (if available)

- Use LEFT JOIN

**CREATE TABLE products (**

**product\_id INT PRIMARY KEY,**

**product\_name VARCHAR(100)**

**);**

**CREATE TABLE orders (**

**order\_id INT PRIMARY KEY,**

**product\_id INT,**

**quantity INT,**

**FOREIGN KEY (product\_id) REFERENCES products(product\_id)**

**);**

**INSERT INTO products VALUES**

**(1, 'Laptop'), (2, 'Phone'), (3, 'Tablet');**

**INSERT INTO orders VALUES**

**(101, 1, 2), (102, 2, 1); -- No order for Tablet**

**SELECT**

**p.product\_name,**

**o.order\_id,**

**o.quantity**

**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 customers (**

**customer\_id INT PRIMARY KEY,**

**name VARCHAR(100)**

**);**

**CREATE TABLE contacts (**

**contact\_id INT PRIMARY KEY,**

**customer\_id INT,**

**email VARCHAR(100),**

**FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)**

**);**

**INSERT INTO customers VALUES**

**(1, 'Alice'), (2, 'Bob'), (3, 'Charlie');**

**INSERT INTO contacts VALUES**

**(201, 1, 'alice@mail.com'), (202, 2, 'bob@mail.com'); -- Charlie has no contact**

**SELECT**

**c.customer\_id,**

**c.name,**

**ct.email**

**FROM contacts ct**

**RIGHT JOIN customers c ON c.customer\_id = ct.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)**

**);**

**CREATE TABLE products (**

**product\_id INT PRIMARY KEY,**

**product\_name VARCHAR(100),**

**supplier\_id INT**

**);**

**INSERT INTO suppliers VALUES**

**(1, 'Supplier A'), (2, 'Supplier B');**

**INSERT INTO products VALUES**

**(10, 'Laptop', 1), (11, 'Monitor', NULL);**

**-- Left join**

**SELECT**

**s.supplier\_name,**

**p.product\_name**

**FROM suppliers s**

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

**UNION**

**-- Right join**

**SELECT**

**s.supplier\_name,**

**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(50)**

**);**

**-- Assume 'products' table already exists**

**INSERT INTO offers VALUES**

**(1, '10% Off'), (2, 'Buy 1 Get 1');**

**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 = CASE**

**WHEN product\_name = 'Laptop' THEN 'Electronics'**

**WHEN product\_name = 'Phone' THEN 'Electronics'**

**WHEN product\_name = 'Tablet' THEN 'Electronics'**

**ELSE 'General'**

**END;**

**SELECT**

**p.category,**

**SUM(o.quantity) AS total\_quantity\_sold,**

**AVG(p\_price.price) AS average\_price**

**FROM orders o**

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

**JOIN (**

**SELECT product\_id, 50000 AS price FROM products**

**) AS p\_price ON o.product\_id = p\_price.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 students (**

**student\_id INT PRIMARY KEY,**

**name VARCHAR(100)**

**);**

**CREATE TABLE marks (**

**mark\_id INT PRIMARY KEY,**

**student\_id INT,**

**subject VARCHAR(50),**

**score INT,**

**FOREIGN KEY (student\_id) REFERENCES students(student\_id)**

**);**

**INSERT INTO students VALUES**

**(1, 'Ravi'), (2, 'Neha'), (3, 'John');**

**INSERT INTO marks VALUES**

**(1, 1, 'Math', 80), (2, 1, 'Science', 90),**

**(3, 2, 'Math', 70), (4, 2, 'Science', 60),**

**(5, 3, 'Math', 95), (6, 3, 'Science', 85);**

**SELECT**

**s.name AS student\_name,**

**AVG(m.score) AS average\_marks**

**FROM students s**

**JOIN marks m ON s.student\_id = m.student\_id**

**GROUP BY s.student\_id**

**HAVING AVG(m.score) > 75;**

# 13-06-2025

**Querying Data by Using Subqueries - Examples**

**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);

**Querying Data by Using Subqueries**

Query:

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

**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);

**Querying Data by Using Subqueries using ANY**

Query:

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

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

Query:

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

**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'));

**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);

**Querying Data by Using Subqueries Using UNION**

Query:

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

**Querying Data by Using Subqueries using INTERSECT**

Query:

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

**Querying Data by Using Subqueries using EXCEPT**

Query:

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

**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**

# **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.

**Types of NoSQL databases with real-time examples:**

1. **Document-oriented (e.g., MongoDB):**  
    *Example:* Content Management System (CMS) for blogs storing articles as documents.
2. **Key-Value Store (e.g., Redis):**  
    *Example:* Session management in web applications.
3. **Column-family Store (e.g., Apache Cassandra):**  
    *Example:* Time-series data in IoT applications.
4. **Graph Databases (e.g., Neo4j):**  
    *Example:* Social media networks to model users and their relationships.

**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.

**Types of NoSQL databases with real-time examples:**

1. **Document-oriented (e.g., MongoDB):**  
    *Example:* Content Management System (CMS) for blogs storing articles as documents.
2. **Key-Value Store (e.g., Redis):**  
    *Example:* Session management in web applications.
3. **Column-family Store (e.g., Apache Cassandra):**  
    *Example:* Time-series data in IoT applications.
4. **Graph Databases (e.g., Neo4j):**  
    *Example:* Social media networks to model users and their relationships.

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)

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

CREATE DATABASE RetailDB; USE RetailDB;

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

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)

);

1. Alter the Products table to add 'Discount' column and update some values.

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

Q4. (5 marks)

Using the above schema:

1. 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);

1. 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

);

1. 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:

1. Customers who ordered more than 3 different products.

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

1. Products not ordered by any customer.

SELECT \* FROM Products

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID FROM Orders

);

1. 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)

1. 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;

1. Use date functions to compute days between order date and today.

SELECT

OrderID,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

1. Use system functions to return current user and host.

SELECT CURRENT\_USER() AS User,

HOST\_NAME() AS Host;

SELECT SYSTEM\_USER();

1. 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)

1. 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;

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

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;

1. 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;

**Question Paper - 2**

# **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.

|  |  |  |
| --- | --- | --- |
| **Feature** | **SQL (Relational DB)** | **NoSQL (Non-relational DB)** |
| **Data Model** | Structured, tabular with fixed schema | Flexible schema: key-value, document, graph, etc. |
| **Query Language** | Structured Query Language (SQL) | No standard language; uses JSON-like queries |
| **Scalability** | Vertically scalable (scale-up) | Horizontally scalable (scale-out) |
| **Transactions** | ACID-compliant (Atomicity, Consistency, etc.) | Often BASE (Basically Available, Soft state, Eventually consistent) |
| **Best For** | Complex queries, structured data | Big data, unstructured/semi-structured data |
| **Examples** | MySQL, PostgreSQL, Oracle | MongoDB, Cassandra, Redis, Couchbase |

**SQL (Relational Databases)**

**Advantages:**

1. **Structured Data and Relationships:**
2. SQL databases are ideal for storing structured data with defined relationships.
3. *Example:* Banking systems (like ICICI Bank or SBI) use SQL databases (e.g., Oracle, PostgreSQL) to maintain customer-account-transaction relationships.
4. **ACID Compliance:**

SQL databases ensure reliable transactions with properties like atomicity, consistency, isolation, and durability.

*Example:* E-commerce platforms like Flipkart use MySQL to ensure stock is updated reliably when orders are placed.

**Disadvantages:**

1. **Rigid Schema:**

SQL databases require a predefined schema, and changing it can be difficult and disruptive.

*Example:* In a university database, adding a new column like "LinkedIn Profile" to student records may require downtime and significant changes.

1. **Limited Horizontal Scalability:**

SQL databases are not naturally suited for scaling across multiple servers.

*Example:* A rapidly growing startup might struggle to scale PostgreSQL without complex replication setups.

**NoSQL (Non-Relational Databases)**

**Advantages:**

1. **Flexible Schema:**

NoSQL databases allow storing data without a fixed schema, making them adaptable to changing requirements.

*Example:* Social media platforms like Instagram use MongoDB to store diverse content like images, comments, and user metadata.

1. **Horizontal Scalability:**

NoSQL systems are designed to distribute data across multiple servers easily.

*Example:* Netflix uses Apache Cassandra to handle vast amounts of user data globally with high availability.

**Disadvantages:**

1. **Weaker Transaction Support:**

Many NoSQL databases do not support full ACID transactions, leading to possible data inconsistency.

*Example:* In a payment processing system, a failure during transaction recording could result in incorrect balances.

1. **Lack of Standard Query Language:**

NoSQL databases use different and sometimes proprietary query languages, making it harder for developers to learn and switch between systems.*Example:* A developer experienced in SQL would need to learn new syntax to query data in Couchbase or DynamoDB.

Q2. (2 marks)

Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF:

Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

**Unnormalized Form (UNF)**

The given table is:

Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

This structure may contain repeating groups or multi-valued fields. For example, a student may be enrolled in multiple courses with different instructors, which results in repeating groups.

**First Normal Form (1NF)**

**Rule:** Eliminate repeating groups and ensure each attribute contains only atomic (indivisible) values.

After converting to 1NF, the table becomes:

Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

Example rows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **StudentID** | **Name** | **CourseID** | **CourseName** | **InstructorName** | **InstructorPhone** |
| 101 | Alice | CSE101 | DBMS | Dr. Roy | 12345 |
| 101 | Alice | CSE102 | OS | Dr. Sen | 67890 |
| 102 | Bob | CSE101 | DBMS | Dr. Roy | 12345 |

Now, each row contains atomic values and represents a single course enrollment.

**Second Normal Form (2NF)**

**Rule:** The table must be in 1NF and should not contain partial dependencies (i.e., non-key attributes should depend on the whole primary key).

In the 1NF table, the composite primary key is (StudentID, CourseID). The following attributes show partial dependency:

* Name depends only on StudentID
* CourseName, InstructorName, and InstructorPhone depend only on CourseID

To remove these partial dependencies, we decompose the table into:

1. Student table:

Student(StudentID, Name)

1. Course table:

Course(CourseID, CourseName, InstructorName, InstructorPhone)

1. Enrollment table:

Enrollment(StudentID, CourseID)

**Third Normal Form (3NF)**

**Rule:** The table must be in 2NF and should not contain transitive dependencies (i.e., non-key attributes should not depend on other non-key attributes).

In the Course table, InstructorPhone depends on InstructorName, not directly on the primary key CourseID. This is a transitive dependency.

To remove this, we further decompose:

1. Student table:

Student(StudentID, Name)

1. Course table:

Course(CourseID, CourseName, InstructorName)

1. Instructor table:

Instructor(InstructorName, InstructorPhone)

1. Enrollment table:

Enrollment(StudentID, CourseID)

**Summary of Tables in 3NF**

|  |  |
| --- | --- |
| **Table** | **Attributes** |
| Student | StudentID (Primary Key), Name |
| Course | CourseID (Primary Key), CourseName, InstructorName |
| Instructor | InstructorName (Primary Key), InstructorPhone |
| Enrollment | StudentID (Primary Key), CourseID (Primary Key) |

This structure ensures the database design adheres to the rules of 1NF, 2NF, and 3NF.

Q3. (5 marks)

1. Create a database named StudentDB.
   * Create database StudentDB;
2. Create a table Students with fields: StudentID, Name, DOB, Email.
   * Create table Students (StudentID int Primary Key auto\_increment, Name varchar(50), DOB date , Email varchar(100));
3. Rename the table to Student\_Info.
   * Alter table Students rename to Student\_Info;
4. Add a column PhoneNumber.
   * Alter table Student\_Info add column PhoneNumber int(10)
5. Drop the table.
   * Drop table Student\_Info

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

Q4. (5 marks)

1. Insert 3 student records into Student\_Info.

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

VALUES

('Alice Thomas', '2002-03-15', '[alice.thomas@gmail.com](mailto:alice.thomas@gmail.com)', 9876543210),

('Brian Dsouza', '1999-07-20', '[brian.dsouza@yahoo.com](mailto:brian.dsouza@yahoo.com)', 9123456789),

('Catherine Roy', '2001-11-10', '[catherine.roy@outlook.com](mailto:catherine.roy@outlook.com)', 9988776655);

1. Update one student's phone number.

UPDATE Student\_Info

SET PhoneNumber = 9999999999

WHERE Name = 'Alice Thomas';

1. Delete one student whose email ends with @gmail.com.

DELETE FROM Student\_Info WHERE Email LIKE '[%@gmail.com](mailto:%@gmail.com)';

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

SELECT Name, Email

FROM Student\_Info

WHERE YEAR(DOB) > 2000;

1. Retrieve distinct domain names from the email column.

SELECT DISTINCT SUBSTRING\_INDEX(Email, '@', -1) AS Domain FROM Student\_Info;

Q5. (5 marks)

1. Retrieve students with names starting with 'A'.

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

1. Retrieve students with phone number between 9000000000 and 9999999999.

SELECT \* FROM Student\_Info

WHERE PhoneNumber BETWEEN 9000000000 AND 9999999999;

1. Retrieve students using IN operator on city names.

SELECT \* FROM Student\_Info

WHERE City IN ('Mumbai', 'Delhi', 'Chennai');

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

SELECT \* FROM Student\_Info

WHERE

(YEAR(CURDATE()) - YEAR(DOB)) < 25 AND (Email LIKE '[%@yahoo.com](mailto:%@yahoo.com)' OR Email LIKE '[%@outlook.com](mailto:%@outlook.com)');

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

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

Q6. (5 marks)

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

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks DECIMAL(5,2)

);

INSERT INTO Marks (StudentID, Subject, Marks) VALUES

(1, 'Math', 85.5),

(2, 'Science', 67.25),

(1, 'English', 72.00);

1. Display student IDs and their subjects where marks > 70.

SELECT StudentID, Subject

FROM Marks

WHERE Marks > 70;

1. Display subjects with average marks.

SELECT Subject, AVG(Marks) AS AvgMarks

FROM Marks

GROUP BY Subject;

1. Filter subjects with average marks between 60 and 90.

SELECT Subject, AVG(Marks) AS AvgMarks

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) BETWEEN 60 AND 90;

# **Section C: Functions & Grouping (10 Marks)**

Q7. (5 marks)

1. Get the current date and format it as "YYYY-MM-DD".

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

1. Extract month and year from a DOB column.

SELECT Name, MONTH(DOB) AS BirthMonth, YEAR(DOB) AS BirthYear FROM Student\_Info;

1. Convert a student's name to uppercase.

SELECT UPPER(Name) AS UpperCaseName FROM Student\_Info;

1. Round off marks to 2 decimal places.

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

1. Use system function to return user name or current database.

SELECT CURRENT\_USER() AS LoggedInUser, DATABASE() AS CurrentDB;

Q8. (5 marks)

1. Display total marks of each student.

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

1. Display subject-wise highest mark.

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

1. Use GROUP BY and HAVING to display subjects with average marks > 75.

SELECT Subject, AVG(Marks) AS AverageMark

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) > 75;

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

Q9. (5 marks)

1. Inner Join to retrieve students and their courses.

SELECT s.Name, c.CourseName

FROM Student\_Info s

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

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

1. Left Join to get all students even if not enrolled.

SELECT s.Name, c.CourseName

FROM Student\_Info s

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

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

1. Right Join to get all courses even if no students.

SELECT s.Name, c.CourseName

FROM Student\_Info s

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

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

1. Full Outer Join equivalent using UNION.

SELECT s.Name, c.CourseName

FROM Student\_Info s

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

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

UNION

SELECT s.Name, c.CourseName

FROM Student\_Info s

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

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

1. Cross Join to show all combinations.

SELECT s.Name, c.CourseName

FROM Student\_Info s

CROSS JOIN Course c;

Q10. (5 marks)

1. Students who scored more than average in 'Maths'.

SELECT StudentID, Marks

FROM Marks

WHERE Subject = 'Math'

AND Marks > (

SELECT AVG(Marks)

FROM Marks

WHERE Subject = 'Math'

);

1. Students not in the Marks table.

SELECT \*

FROM Student\_Info

WHERE StudentID NOT IN (

SELECT DISTINCT StudentID

FROM Marks

);

1. Use EXISTS to get students with at least one subject.

SELECT \*

FROM Student\_Info s

WHERE EXISTS (

SELECT 1

FROM Marks m

WHERE s.StudentID = m.StudentID

);

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

SELECT \*

FROM Student\_Info s

WHERE EXISTS (

SELECT 1

FROM Marks m

WHERE s.StudentID = m.StudentID

);

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

SELECT \*

FROM Marks

WHERE Marks > ANY (

SELECT Marks

FROM Marks

WHERE Subject = 'English'

);

Q11. (5 marks)

1. UNION of student names from two tables.

SELECT Name FROM Student\_Info

UNION

SELECT DISTINCT Name FROM Student\_Info s

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

1. INTERSECT to find common students.
2. EXCEPT to list students in Students but not in Marks.
3. MERGE concept or simulate with UPDATE and INSERT.
4. Correlated subquery to list students with above average per subject.

SELECT \*

FROM Marks m1

WHERE Marks > (

SELECT AVG(m2.Marks)

FROM Marks m2

WHERE m2.Subject = m1.Subject

);