SQL Practical Question

# Section A: Advanced Concepts & Schema Design

## 1. NoSQL vs SQL and Types of NoSQL

NoSQL is preferred over SQL when:  
- The data is unstructured or semi-structured (e.g., JSON, XML).  
- High scalability and availability are required.  
- Rapid development and flexibility in schema design are important.  
  
Types of NoSQL Databases and Real-time Applications:  
1. Document Store (e.g., MongoDB): Used in content management systems like blogging platforms.  
2. Key-Value Store (e.g., Redis): Used in caching user sessions in web applications.  
3. Column-Family Store (e.g., Cassandra): Used in logging and analytics platforms like monitoring tools.  
4. Graph Database (e.g., Neo4j): Used in social networks for relationships and recommendations.

## 2. Normalization to BCNF

Given Unnormalized Table:  
Customer (CustomerID, Name, Orders (OrderID, ProductID, Quantity, ProductName))  
  
1NF:  
Customer (CustomerID, Name)  
Order (OrderID, CustomerID, ProductID, Quantity, ProductName)  
  
2NF:  
Customer (CustomerID, Name)  
Order (OrderID, CustomerID)  
OrderDetails (OrderID, ProductID, Quantity, ProductName)  
  
3NF/BCNF:  
Customer (CustomerID, Name)  
Order (OrderID, CustomerID)  
Product (ProductID, ProductName)  
OrderDetails (OrderID, ProductID, Quantity)

# Section B: Complex DDL and DML

## 3. Schema and Constraints

a) CREATE DATABASE RetailDB;  
  
CREATE TABLE Customers (  
 CustomerID INT PRIMARY KEY,  
 Name 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 CHECK (Quantity > 0),  
 FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),  
 FOREIGN KEY (ProductID) REFERENCES Products(ProductID)  
);  
  
b) Implemented CHECK constraint in Orders table.  
  
c) ALTER TABLE Products ADD Discount DECIMAL(5,2);  
UPDATE Products SET Discount = 5.00 WHERE ProductID = 1;

## 4. Insert/Update/Delete

a) INSERT INTO Orders VALUES (1, 1, 101, 3), (2, 1, 102, 2), (3, 1, 103, 1);  
  
b) UPDATE Products SET Price = Price \* 1.10  
WHERE ProductID IN (SELECT ProductID FROM Orders GROUP BY ProductID HAVING SUM(Quantity) > 5);  
  
c) DELETE FROM Orders WHERE ProductID NOT IN (SELECT ProductID FROM Orders);

## 5. Retrieval Queries

a) SELECT CustomerID FROM Orders GROUP BY CustomerID HAVING COUNT(DISTINCT ProductID) > 3;  
  
b) SELECT ProductName FROM Products WHERE ProductID NOT IN (SELECT DISTINCT ProductID FROM Orders);  
  
c) SELECT CustomerID, COUNT(\*) FROM Orders  
WHERE OrderDate >= CURDATE() - INTERVAL 30 DAY GROUP BY CustomerID;

# Section C: Advanced Functions and Aggregations

## 6. Functions Usage

a) SELECT LOWER(SUBSTRING\_INDEX(email, '@', 1)) AS username FROM Customers;  
  
b) SELECT DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder FROM Orders;  
  
c) SELECT USER(), HOST();  
  
d) SELECT CONCAT('Hello ', UPPER(Name), '!') AS Greeting FROM Customers;

## 7. Aggregation & Grouping

a) SELECT Category, SUM(Price \* Quantity) AS TotalRevenue FROM Products P  
JOIN Orders O ON P.ProductID = O.ProductID GROUP BY Category;  
  
b) SELECT Category, SUM(Price \* Quantity) AS Revenue FROM Products P  
JOIN Orders O ON P.ProductID = O.ProductID GROUP BY Category WITH ROLLUP;  
  
c) SELECT Category, SUM(Price \* Quantity) AS Revenue FROM Products P  
JOIN Orders O ON P.ProductID = O.ProductID GROUP BY Category HAVING Revenue > 100000;

# Section D: Complex Joins, Subqueries, and Set Ops

## 8. Various Joins

a) SELECT C1.Name AS Referrer, C2.Name AS Referred FROM Customers C1  
JOIN Customers C2 ON C1.CustomerID = C2.ReferredBy;  
  
b) SELECT O.OrderID, P.ProductName FROM Orders O JOIN Products P ON O.ProductID = P.ProductID;  
  
c) SELECT Name, SUM(Price \* Quantity) OVER (PARTITION BY CustomerID) AS TotalSpend  
FROM Customers C JOIN Orders O ON C.CustomerID = O.CustomerID  
JOIN Products P ON O.ProductID = P.ProductID ORDER BY TotalSpend DESC LIMIT 3;  
  
d) SELECT C.Name FROM Customers C LEFT JOIN Orders O ON C.CustomerID = O.CustomerID WHERE O.OrderID IS NULL;  
  
e) SELECT P1.ProductName AS A, P2.ProductName AS B FROM Products P1 CROSS JOIN Products P2 WHERE P1.ProductID < P2.ProductID;

## 9. Subqueries

a) SELECT Name FROM Customers C WHERE EXISTS (  
 SELECT 1 FROM Orders O WHERE O.CustomerID = C.CustomerID   
 GROUP BY O.CustomerID HAVING SUM(O.Amount) >   
 (SELECT AVG(Amount) FROM Orders WHERE CustomerID = C.CustomerID)  
);  
  
b) SELECT Name FROM Customers C WHERE EXISTS (  
 SELECT 1 FROM Orders O WHERE O.CustomerID = C.CustomerID GROUP BY ProductID HAVING COUNT(DISTINCT ProductID) >= 2  
);  
  
c) SELECT Name FROM Customers WHERE CustomerID = ALL (  
 SELECT CustomerID FROM Orders GROUP BY CustomerID HAVING COUNT(\*) >= ALL (  
 SELECT COUNT(\*) FROM Orders GROUP BY CustomerID  
 )  
);  
  
d) SELECT ProductName FROM Products WHERE Price > ANY (  
 SELECT Price FROM Products WHERE Category = 'Electronics'  
);  
  
e) SELECT ProductID, SUM(Quantity) AS Sold FROM Orders GROUP BY ProductID ORDER BY Sold DESC LIMIT 3;

## Q10. Set Operations

a) SELECT Name FROM SegmentA INNER JOIN SegmentB USING(CustomerID);  
  
b) SELECT ProductName FROM Products WHERE ProductID NOT IN (SELECT DISTINCT ProductID FROM Orders);  
  
c) INSERT INTO Customers (CustomerID, Name)  
VALUES (101, 'John Doe') ON DUPLICATE KEY UPDATE Name = 'John Doe';  
  
d) SELECT Name FROM Region1 UNION SELECT Name FROM Region2;  
  
e) WITH RankedCustomers AS (  
 SELECT CustomerID, SUM(Amount) AS TotalSpend,  
 RANK() OVER (ORDER BY SUM(Amount) DESC) AS Rank FROM Orders GROUP BY CustomerID  
)  
SELECT \* FROM RankedCustomers WHERE Rank <= 5;

SQL Practical Question -2

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

## 1. Differentiate between SQL and NoSQL

SQL (Structured Query Language):  
- SQL databases are relational, table-based.  
- Examples: MySQL, PostgreSQL.  
  
Advantages:  
1. Structured and predictable schema – ideal for complex queries.  
2. ACID compliance ensures data integrity.  
  
Disadvantages:  
1. Scaling is difficult – usually vertical.  
2. Poor performance with unstructured data.

**Real-World Example:**

**Online Banking System**

* **Use Case**: Banks use **MySQL** to store customer details, transactions, and account balances.
* **Why SQL?**: Data is highly structured and relational. It requires strict integrity and consistency (ACID compliance).

**Example Tables**:

* Customers(CustomerID, Name, Address)
* Accounts(AccountID, CustomerID, Balance)
* Transactions(TransactionID, AccountID, Date, Amount)

**Sample Query**:

SELECT Name, Balance

FROM Customers

JOIN Accounts ON Customers.CustomerID = Accounts.CustomerID

WHERE Balance > 10000;

NoSQL (Not only SQL):  
- Non-relational, document/key-value/graph-based.  
- Examples: MongoDB, Cassandra.  
  
Advantages:  
1. Easily scalable – especially horizontally.  
2. Flexible schema – ideal for big data and real-time applications.  
  
Disadvantages:  
1. Lack of standardization across databases.  
2. No support for complex joins or ACID properties.

**Real-World Example:**

**E-commerce Website (e.g., Amazon)**

* **Use Case**: Product catalogs, customer preferences, and real-time activity logs are stored in MongoDB.
* **Why NoSQL?**: Schema-less, easily handles diverse product structures, and scales horizontally for high traffic.
* **Query Example (MongoDB)**:

db.products.find({ "Price": { "$lt": 30000 } })

## 2. Normalize the data to 1NF, 2NF, 3NF

Unnormalized: Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)  
  
1NF: Remove repeating groups:  
Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)  
  
2NF: Remove partial dependencies:  
Student(StudentID, Name)  
Course(CourseID, CourseName, InstructorName, InstructorPhone)  
  
3NF: Remove transitive dependencies:  
Student(StudentID, Name)  
Course(CourseID, CourseName, InstructorID)  
Instructor(InstructorID, InstructorName, InstructorPhone)

## 3. SQL Commands

a) CREATE DATABASE StudentDB;  
b) CREATE TABLE Students (StudentID INT, Name VARCHAR(50), DOB DATE, Email VARCHAR(100));  
c) ALTER TABLE Students RENAME TO Student\_Info;  
d) ALTER TABLE Student\_Info ADD PhoneNumber VARCHAR(15);  
e) DROP TABLE Student\_Info;

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

## 4. Student\_Info DML Operations

a) INSERT INTO Student\_Info (StudentID, Name, DOB, Email, PhoneNumber) VALUES   
(1, 'Arjun', '2001-06-21', 'arjun@gmail.com', '9876543210'),  
(2, 'Bala', '1999-02-14', 'bala@yahoo.com', '9123456780'),  
(3, 'Charan', '2002-10-30', 'charan@gmail.com', '9988776655');  
  
b) UPDATE Student\_Info SET PhoneNumber = '9000000000' WHERE StudentID = 2;  
  
c) DELETE FROM Student\_Info WHERE Email LIKE '%@gmail.com';  
  
d) SELECT Name, Email FROM Student\_Info WHERE YEAR(DOB) > 2000;  
  
e) SELECT DISTINCT SUBSTRING\_INDEX(Email, '@', -1) AS Domain FROM Student\_Info;

## 5. Filtering Techniques

a) SELECT \* FROM Student\_Info WHERE Name LIKE 'A%';  
  
b) SELECT \* FROM Student\_Info WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';  
  
c) SELECT \* FROM Student\_Info WHERE City IN ('Chennai', 'Delhi', 'Mumbai');  
  
d) SELECT \* FROM Student\_Info WHERE (YEAR(CURDATE()) - YEAR(DOB)) > 22 AND Email LIKE '%@yahoo.com';  
  
e) SELECT SI.Name AS StudentName, SI.DOB AS DateOfBirth FROM Student\_Info AS SI;

## 6. Marks Table Creation and Queries

CREATE TABLE Marks (StudentID INT, Subject VARCHAR(50), Marks INT);  
  
INSERT INTO Marks VALUES   
(1, 'Maths', 85),  
(2, 'Science', 78),  
(3, 'English', 65);  
  
a) SELECT StudentID, Subject FROM Marks WHERE Marks > 70;  
  
b) SELECT Subject, AVG(Marks) AS AvgMarks FROM Marks GROUP BY Subject;  
  
c) SELECT Subject FROM Marks GROUP BY Subject HAVING AVG(Marks) BETWEEN 60 AND 90;

# Section C: Functions & Grouping (10 Marks)

## 7. SQL Functions

a) SELECT DATE\_FORMAT(CURDATE(), '%Y-%m-%d') AS TodayDate;  
  
b) SELECT MONTH(DOB) AS BirthMonth, YEAR(DOB) AS BirthYear FROM Student\_Info;  
  
c) SELECT UPPER(Name) AS UpperName FROM Student\_Info;  
  
d) SELECT ROUND(Marks, 2) AS RoundedMarks FROM Marks;  
  
e) SELECT USER() AS CurrentUser, DATABASE() AS CurrentDatabase;

## Q8. Aggregate Functions and Grouping

a) SELECT StudentID, SUM(Marks) AS TotalMarks FROM Marks GROUP BY StudentID;  
  
b) SELECT Subject, MAX(Marks) AS HighestMark FROM Marks GROUP BY Subject;  
  
c) SELECT Subject, AVG(Marks) AS AvgMarks FROM Marks GROUP BY Subject HAVING AVG(Marks) > 75;

# Section D: Joins and Subqueries (25 Marks)9. Joins

Assuming tables: Students(StudentID, Name), Courses(CourseID, CourseName), Enrollment(StudentID, CourseID)  
  
a) SELECT S.StudentID, S.Name, C.CourseName  
 FROM Students S  
 INNER JOIN Enrollment E ON S.StudentID = E.StudentID  
 INNER JOIN Courses C ON E.CourseID = C.CourseID;  
  
b) SELECT S.StudentID, S.Name, C.CourseName  
 FROM Students S  
 LEFT JOIN Enrollment E ON S.StudentID = E.StudentID  
 LEFT JOIN Courses C ON E.CourseID = C.CourseID;  
  
c) SELECT S.StudentID, S.Name, C.CourseName  
 FROM Courses C  
 RIGHT JOIN Enrollment E ON C.CourseID = E.CourseID  
 RIGHT JOIN Students S ON E.StudentID = S.StudentID;  
  
d) -- FULL OUTER JOIN using UNION  
 SELECT S.StudentID, S.Name, C.CourseName  
 FROM Students S  
 LEFT JOIN Enrollment E ON S.StudentID = E.StudentID  
 LEFT JOIN Courses C ON E.CourseID = C.CourseID  
 UNION  
 SELECT S.StudentID, S.Name, C.CourseName  
 FROM Courses C  
 LEFT JOIN Enrollment E ON C.CourseID = E.CourseID  
 LEFT JOIN Students S ON E.StudentID = S.StudentID;  
  
e) SELECT S.StudentID, C.CourseID  
 FROM Students S  
 CROSS JOIN Courses C;

## 10. Subqueries

a) SELECT \* FROM Marks WHERE Subject = 'Maths' AND Marks > (SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths');  
  
b) SELECT \* FROM Students WHERE StudentID NOT IN (SELECT DISTINCT StudentID FROM Marks);  
  
c) SELECT \* FROM Students S WHERE EXISTS (SELECT 1 FROM Marks M WHERE M.StudentID = S.StudentID);  
  
d) SELECT \* FROM Marks WHERE Subject = 'Science' AND Marks > ALL (SELECT Marks FROM Marks WHERE Subject = 'Science');  
  
e) SELECT \* FROM Marks WHERE Subject = 'English' AND Marks > ANY (SELECT Marks FROM Marks WHERE Subject = 'English');

## 11. Set Operators and Correlated Subquery

a) SELECT Name FROM Students  
 UNION  
 SELECT Name FROM Alumni;  
  
b) SELECT Name FROM Students  
 INTERSECT  
 SELECT Name FROM Marks;  
  
c) SELECT Name FROM Students  
 EXCEPT  
 SELECT Name FROM Marks;  
  
d) -- Simulating MERGE with UPDATE and INSERT  
 UPDATE Marks SET Marks = 90 WHERE StudentID = 2 AND Subject = 'Science';  
 INSERT INTO Marks (StudentID, Subject, Marks)  
 SELECT 4, 'Science', 88 FROM DUAL WHERE NOT EXISTS (  
 SELECT \* FROM Marks WHERE StudentID = 4 AND Subject = 'Science');  
  
e) SELECT M1.StudentID, M1.Subject, M1.Marks  
 FROM Marks M1  
 WHERE Marks > (SELECT AVG(Marks) FROM Marks M2 WHERE M2.Subject = M1.Subject);