## **QUIZ - DAY 3 (18-06-25)**

**Section 1: Managing Databases**

Which of the following is NOT a system database in SQL Server?  
 a) master  
 b) model  
 c) tempdb  
 d) userdb

Which system database stores all login accounts and configuration settings?  
 a) tempdb  
 b) model  
 c) master  
 d) msdb

What is the purpose of the model database in SQL Server?  
 a) Backup  
 b) Log storage  
 c) Template for new databases  
 d) System configuration

What are the two main types of database files in SQL Server?  
 a) MDF and NDF  
 b) LDF and MDF  
 c) NDF and BAK  
 d) BAK and TRN

Which SQL command is used to create a new database?  
 a) MAKE DATABASE  
 b) NEW DATABASE  
 c) CREATE DATABASE  
 d) INIT DATABASE

What happens when you execute DROP DATABASE SalesDB?  
 a) SalesDB is backed up  
 b) SalesDB is renamed  
 c) SalesDB is deleted permanently  
 d) SalesDB is restored

Which command renames a database in SQL Server?  
 a) RENAME DATABASE old\_name TO new\_name  
 b) ALTER DATABASE old\_name MODIFY NAME = new\_name  
 c) UPDATE DATABASE NAME  
 d) SET DATABASE NAME

**Section 2: Managing Tables**

Which data type should be used to store a date of birth?  
 a) VARCHAR  
 b) DATE  
 c) INT  
 d) TEXT

What command is used to create a table?  
 a) MAKE TABLE  
 b) INSERT TABLE  
 c) CREATE TABLE  
 d) DEFINE TABLE

How do you add a new column to an existing table?  
 a) ALTER TABLE table\_name ADD column\_name datatype  
 b) MODIFY TABLE table\_name ADD column\_name  
 c) UPDATE TABLE table\_name ADD column\_name  
 d) APPEND column\_name TO table\_name

Which command is used to rename a table?  
 a) RENAME TABLE old\_name TO new\_name  
 b) ALTER TABLE old\_name RENAME TO new\_name  
 c) EXEC sp\_rename 'old\_name', 'new\_name'  
 d) MODIFY TABLE RENAME

What is the command to delete a table permanently?  
 a) DELETE TABLE table\_name  
 b) ERASE TABLE table\_name  
 c) DROP TABLE table\_name  
 d) REMOVE TABLE table\_name

**Section 3: DML - Manipulating Data**

Which command adds data into a table?  
 a) INSERT INTO  
 b) ADD ROW  
 c) CREATE DATA  
 d) APPEND TO

Which clause is used to update data in a table?  
 a) MODIFY  
 b) UPDATE  
 c) CHANGE  
 d) SET TABLE

What does the DELETE statement do?  
 a) Removes a column  
 b) Removes all data from a table  
 c) Removes specific rows  
 d) Deletes the table schema

Which clause is used to filter rows in a SELECT statement?  
 a) HAVING  
 b) SELECT  
 c) WHERE  
 d) ORDER BY

Which keyword ensures no duplicate records are returned?  
 a) UNIQUE  
 b) NO\_REPEAT  
 c) DISTINCT  
 d) ONLY

What does the LIKE keyword do in SQL?  
 a) Finds exact matches  
 b) Finds pattern-based matches  
 c) Sorts records  
 d) Deletes matches

Which operator is used to combine multiple conditions in a WHERE clause?  
 a) TO  
 b) WITH  
 c) AND / OR  
 d) IF / ELSE

What does the BETWEEN operator do?  
 a) Compares text fields  
 b) Finds rows outside a range  
 c) Filters values within a range  
 d) Joins tables

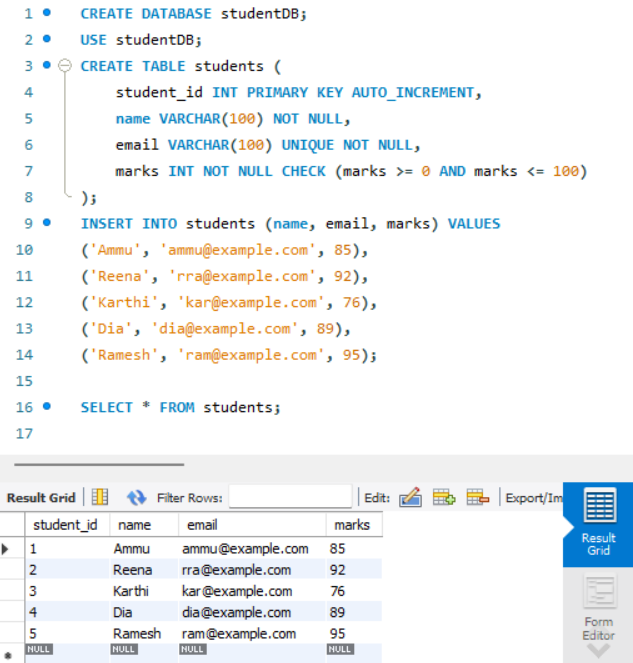
# **Assignment Day 4 (19-06-25)**

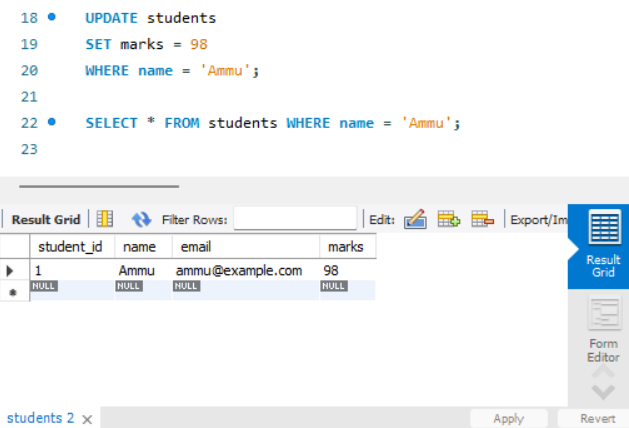
# Medium-Level Practical SQL Questions

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.

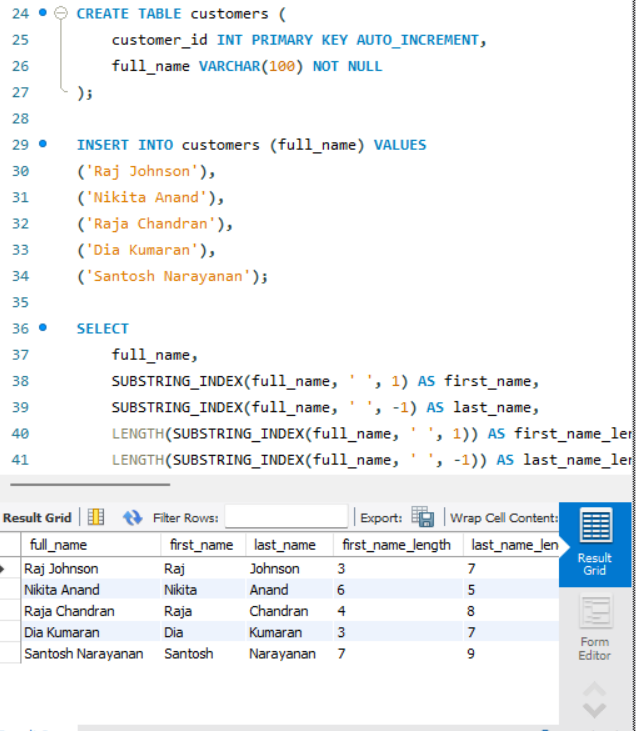
**ANSWER**

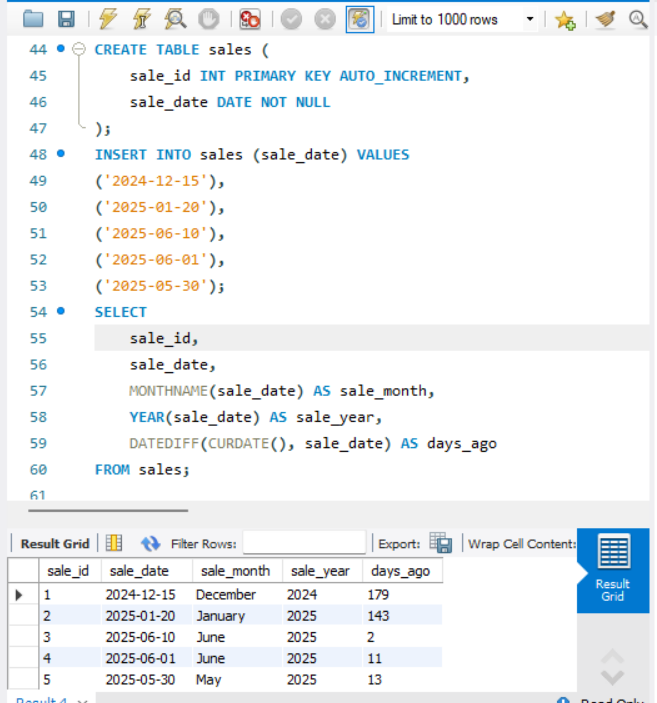
**Create a 'students' table with constraints (NOT NULL, UNIQUE). Insert 5 records.**



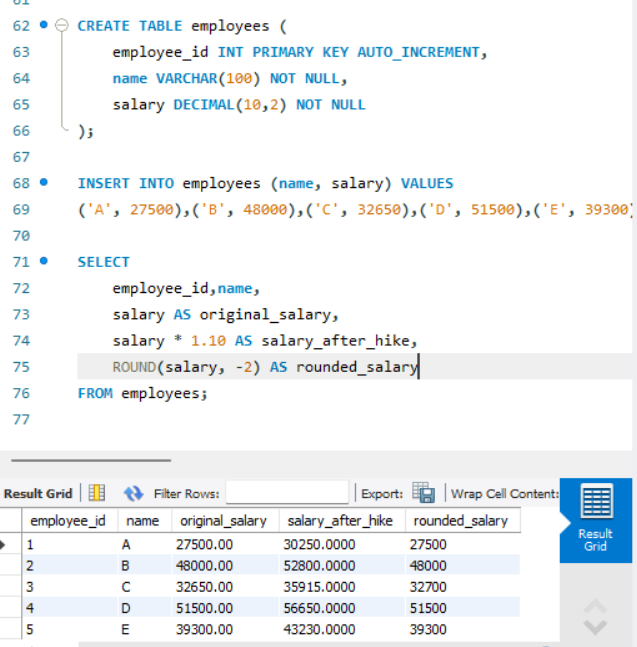
**Update a student's marks ensuring data integrity is maintained.**

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

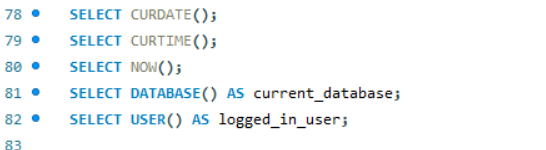


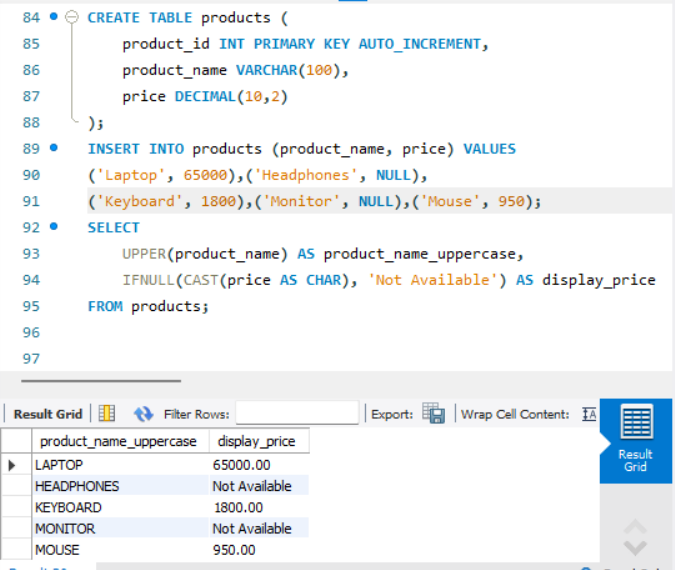
3. Date Function Usage: From a 'sales' table with a 'sale\_date' column, write a query to: - Extract the month name and year - Display how many days ago the sale happened

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

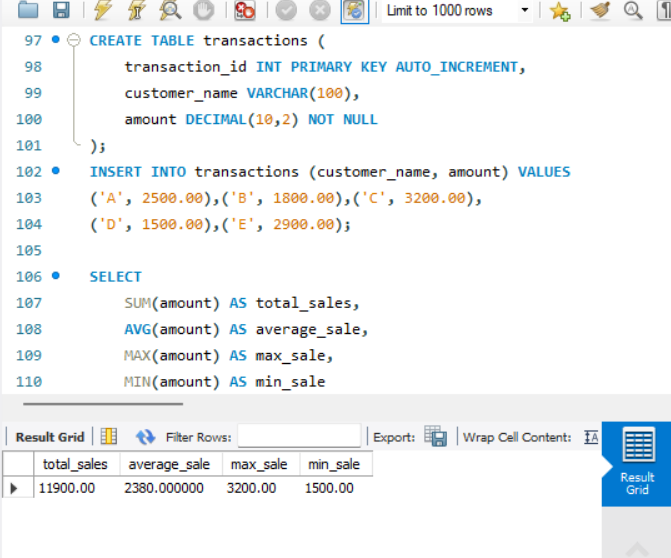


5. System Function Check: Retrieve: - Current date and time - Database name and logged-in user

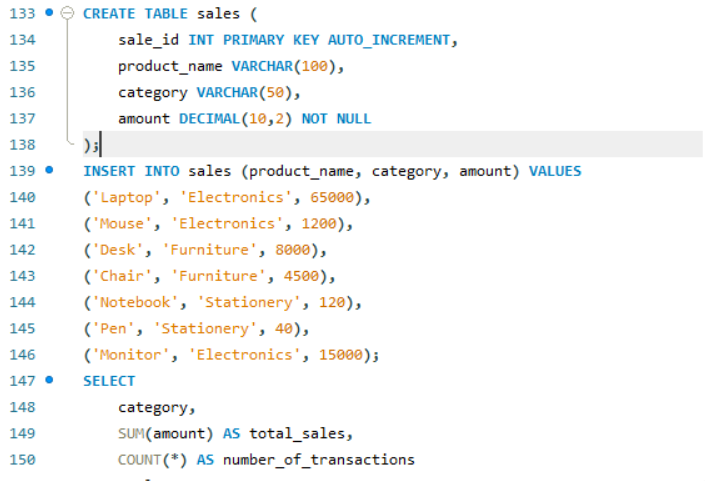


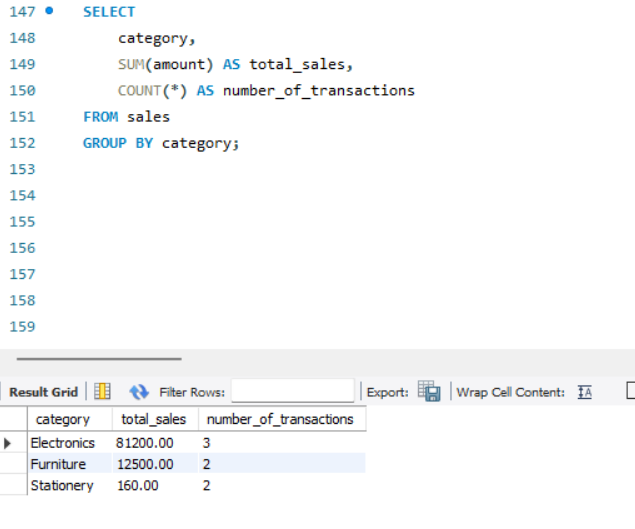
6. Demo: Custom Result Set: From the 'products' table, write a query that: - Returns product name in uppercase - Replaces any NULL prices with 'Not Available'

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

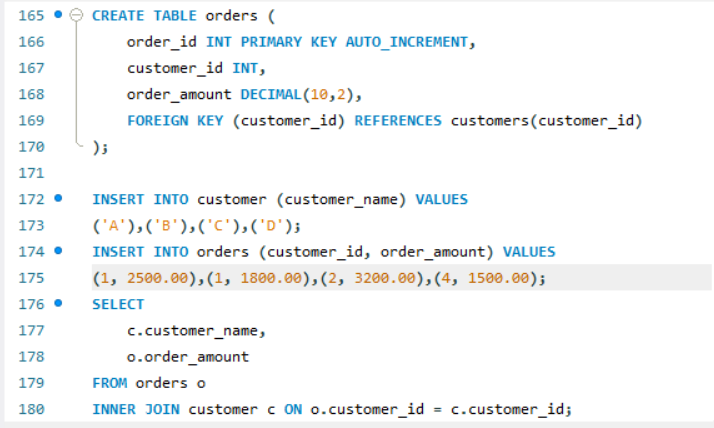


8. Grouping with Aggregation: From a 'sales' table: - Group by product category - Show total sales and number of transactions in each category

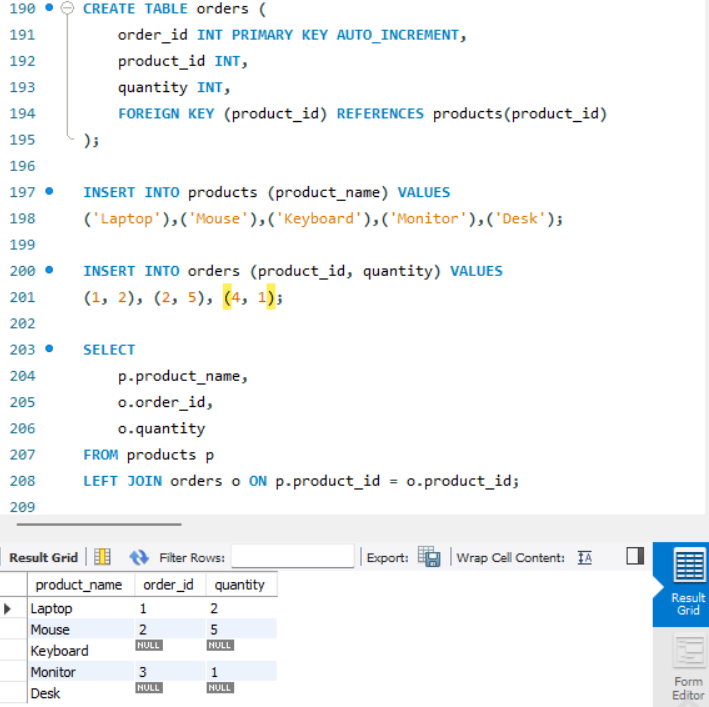




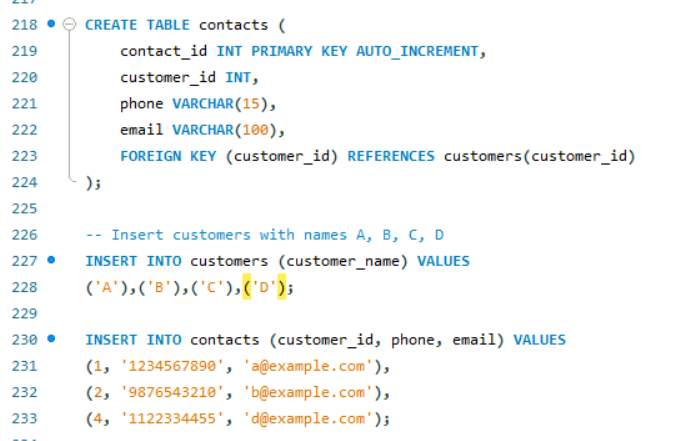
9. Inner Join for Orders and Customers: Join 'orders' and 'customers' to show: - Customer name - Order amount - Only for customers who made orders

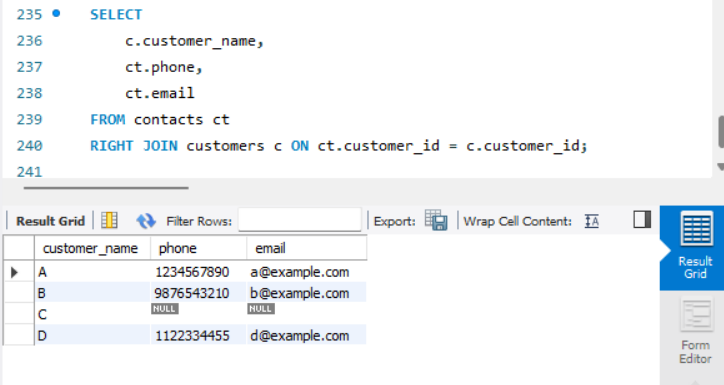


10. Left Join for Products with or without Orders: Show all products with: - Their order details (if available) - Use LEFT JOIN

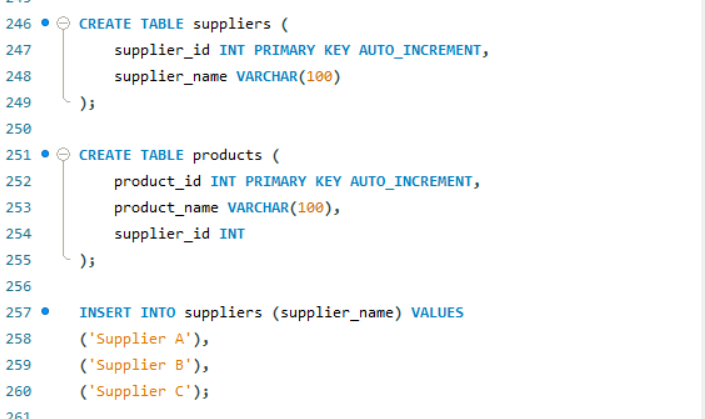


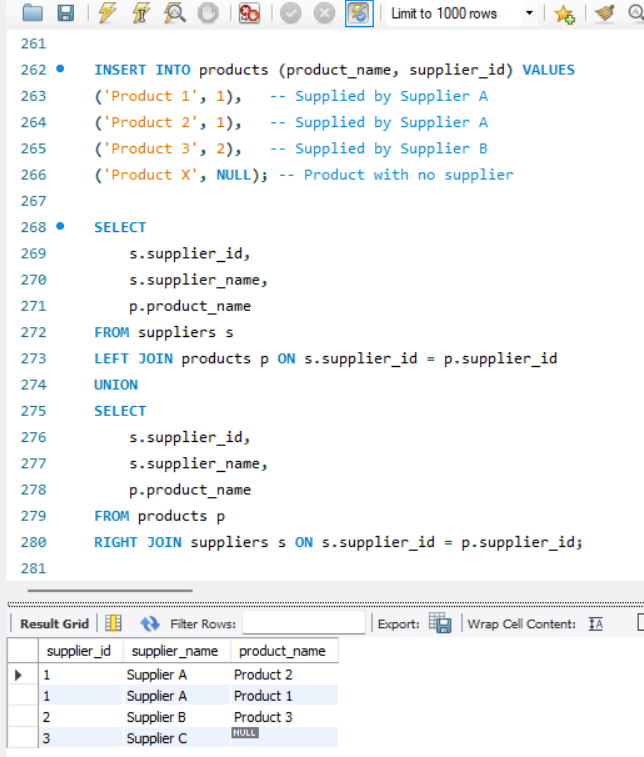
11. Right Join for Customer Contacts: Use a RIGHT JOIN between 'contacts' and 'customers' to display: - All customers, even if they don't have contact info



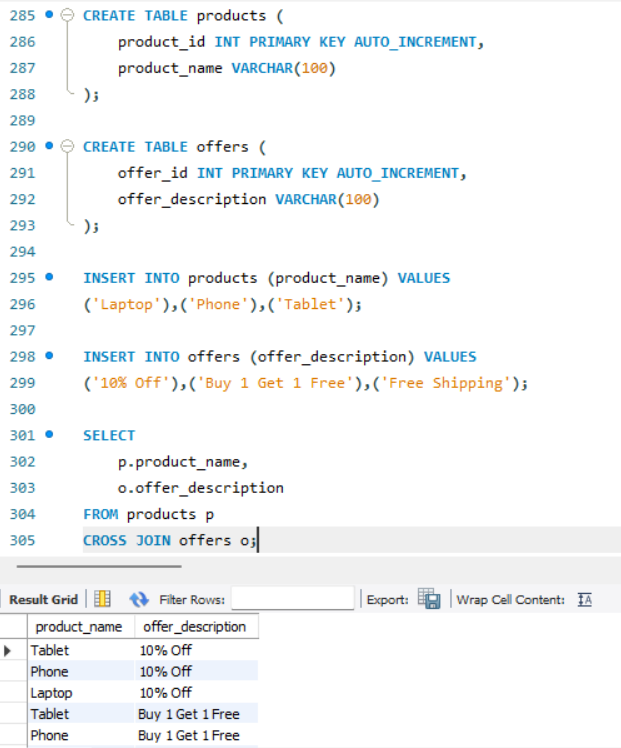


12. Full Outer Join for Suppliers and Products: Use a FULL OUTER JOIN to list: - All suppliers and products - Match supplier to product, or show NULLs where not available

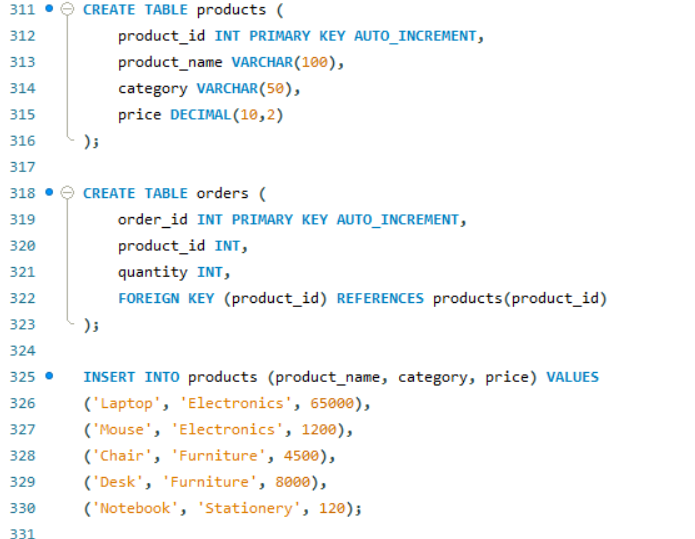


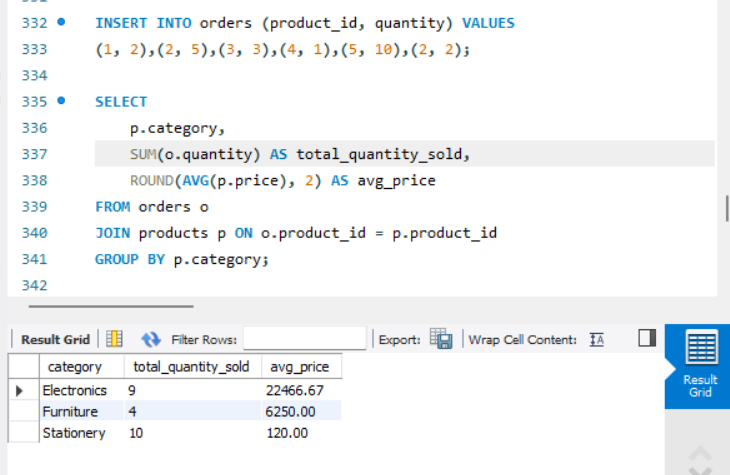


13. Cross Join for Offers: Suppose you have tables 'products' and 'offers'. Write a CROSS JOIN to show: - All possible combinations of products and offers

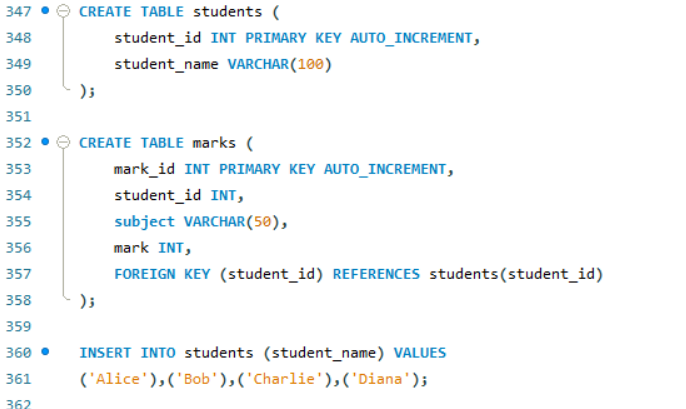


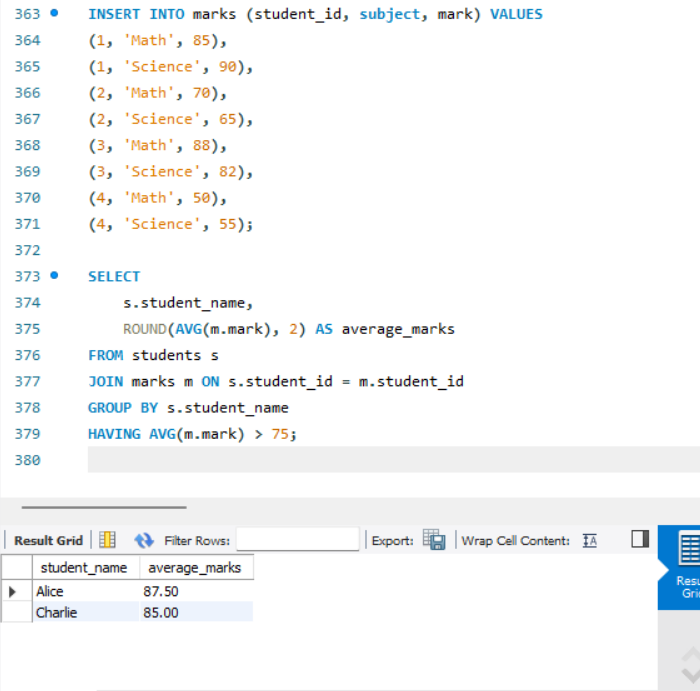
14. Join with Aggregation: Join 'orders' and 'products', then group by product category and: - Show total quantity sold and average price per category





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





# Assignment Day 5 - 20-06-2025

1) Querying Data by Using Subqueries

2) Querying Data by Using Subqueries Using the EXISTS,

3) Querying Data by Using Subqueries using ANY,

4) Querying Data by Using Subqueries using ALL Keywords

5) Querying Data by Using Subqueries using Using Nested Subqueries

6) Querying Data by Using Subqueries Using Correlated Subqueries

7) Querying Data by Using Subqueries Using UNION,

8) Querying Data by Using Subqueries using INTERSECT,

9) Querying Data by Using Subqueries using EXCEPT,

10)Querying Data by Using Subqueries using MERGE

### **1) Querying Data Using Subqueries**

**Table: employees**

CREATE TABLE employees(

id INT, name VARCHAR(10), dept\_id INT, salary INT

);

INSERT INTO employees VALUES

(1,'Alice',10,5000),(2,'Bob',20,6000),

(3,'Carol',10,7000),(4,'Dave',30,4000);

**Query**: Get employees earning more than the overall average.

SELECT \* FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

### **2) Using EXISTS**

**Table: departments**

CREATE TABLE departments(

dept\_id INT, dept\_name VARCHAR(10)

);

INSERT INTO departments VALUES (10,'IT'),(20,'HR'),(30,'Sales');

**Query**: Show departments with at least one employee.

SELECT dept\_name FROM departments d

WHERE EXISTS (

SELECT 1 FROM employees e WHERE e.dept\_id = d.dept\_id

);

### **3) Using ANY**

**Query**: Employees whose salary is higher than any IT employee.

SELECT \* FROM employees

WHERE salary > ANY (

SELECT salary FROM employees WHERE dept\_id = 10

);

### **4) Using ALL**

**Query**: Employees who earn more than all employees in dept 10.

SELECT \* FROM employees

WHERE salary > ALL (

SELECT salary FROM employees WHERE dept\_id = 10

);

### **5) Nested Subqueries**

**Query**: Employees earning more than the minimum salary of the highest-paying department.

SELECT \* FROM employees

WHERE salary > (

SELECT MIN(salary) FROM employees WHERE dept\_id =

(SELECT dept\_id FROM employees GROUP BY dept\_id ORDER BY AVG(salary) DESC LIMIT 1)

);

### **6) Correlated Subqueries**

**Query**: Employees earning above their department’s average.

SELECT \* FROM employees e

WHERE salary > (

SELECT AVG(salary) FROM employees WHERE dept\_id = e.dept\_id

);

### **7) Using UNION**

**Query**: List all names of employees and departments.

SELECT name AS item FROM employees

UNION

SELECT dept\_name AS item FROM departments;

### **8) Using INTERSECT**

**Assume** a table of names test:

CREATE TABLE test\_names(name VARCHAR(10));

INSERT INTO test\_names VALUES ('Alice'),('IT'),('X');

**Query**: Names present in both employees and test\_names.

SELECT name FROM employees

INTERSECT

SELECT name FROM test\_names;

### **9) Using EXCEPT**

**Query**: Employee names not in test\_names.

SELECT name FROM employees

EXCEPT

SELECT name FROM test\_names;

### **10) Using MERGE**

**Assume** a target table emp\_archive(id, name, dept\_id, salary).

**Query**: Sync employees table into emp\_archive.

MERGE INTO emp\_archive AS t

USING employees AS s

ON t.id = s.id

WHEN MATCHED THEN

UPDATE SET name = s.name, dept\_id = s.dept\_id, salary = s.salary

WHEN NOT MATCHED THEN

INSERT (id,name,dept\_id,salary) VALUES (s.id,s.name,s.dept\_id,s.salary);

# **Assignment test - Day 5**

## **SQL Practical Question Paper**

## **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.

**ANSWERS**

**Scenarios Where NoSQL is Preferred Over SQL**

NoSQL is better than SQL in many modern situations. For example, working with data that doesn't fit into neat rows and columns, NoSQL is more flexible. It doesn't need a fixed table structure, so it’s easier to use when data keeps changing or growing.

Also, when an app gets a lot of traffic — like thousands of people shopping online at the same time — NoSQL handles those large numbers better. It's also useful when you need to store data across many computers instead of just one, because it can split and manage the data easily.

Another reason to use NoSQL is if we are building apps quickly and don’t want to define the database structure every time something changes. Startups, real-time apps, and big data systems often use NoSQL for this reason.

**Types of NoSQL and Real-Life Uses**

4 main types of NoSQL databases:

1. **Document-based databases** (MongoDB) store data in flexible formats like JSON. A good example is an e-commerce app, where each product has different features and details.
2. **Key-value databases** (Redis) are simple and fast — they save data as a “key” and a “value.” A real use is in web applications for remembering user sessions
3. **Column-based databases** (Cassandra) store data in columns instead of rows. These are perfect for storing logs or sensor data that come in over time.
4. **Graph databases** (Neo4j) are great when you need to understand relationships, like in social media apps (who follows whom), or for recommendation systems

## 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**

#### **Unnormalized Form (UNF):**

In the given structure, all order details are kept inside a single customer record. This makes the data repetitive and hard to manage. Multiple orders are stored in one row, which is not ideal.

#### **First Normal Form (1NF):**

In 1NF, we remove the repeating group (Orders).  
 We separate the data into individual rows for each order.  
 We create three separate tables:

1. Customer Table: stores CustomerID and Name
2. Orders Table: stores OrderID, CustomerID, ProductID, Quantity
3. Product Table: stores ProductID and ProductName

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

In 2NF, we remove partial dependencies. For example, ProductName depends only on ProductID, not on the full primary key (OrderID + ProductID). So we move ProductName into the Product Table to avoid repeating it for every order.

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

Now we remove transitive dependencies.  
Customer Name depends only on CustomerID.  
Product Name depends only on ProductID.  
We have already separated these, so the design meets 3NF.

#### **Boyce-Codd Normal Form (BCNF):**

In BCNF, we check that every functional dependency has a candidate key on the left-hand side.

**In our design:**

* CustomerID → Name (CustomerID is a key)
* ProductID → ProductName (ProductID is a key)
* (OrderID, ProductID) → Quantity (Composite key is valid)

So, all tables satisfy BCNF.

#### **Final Table Structures (BCNF):**

1. **Customer Table:** Customer(CustomerID PRIMARY KEY, Name)
2. **Product Table:** Product(ProductID PRIMARY KEY, ProductName)
3. **Orders Table:** Orders(OrderID, CustomerID, ProductID, Quantity, PRIMARY KEY(OrderID, ProductID), FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID), FOREIGN KEY (ProductID) REFERENCES Product(ProductID)).

## **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 a 'Discount' column and update some values.

**ANSWERS**

#### **a) Creating the RetailDB Database and Designing the Schema**

CREATE DATABASE RetailDB;

USE RetailDB;

Now, we create three tables: Customers, Products, and Orders.  
The schema includes primary keys and foreign keys

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name 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) Adding a Check Constraint on Quantity**

ALTER TABLE Orders

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

#### **c) Altering Products Table to Add ‘Discount’ Column and Update Values**

ALTER TABLE Products

ADD Discount DECIMAL(5,2);

Then, we update some discount values:

UPDATE Products SET Discount = 5.00 WHERE ProductID = 1;

UPDATE Products SET Discount = 10.00 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) Inserting 3 Sample Orders Per Customer**

INSERT INTO Orders VALUES (101, 1, 1, 2, '2024-06-01');

INSERT INTO Orders VALUES (102, 1, 2, 1, '2024-06-03');

INSERT INTO Orders VALUES (103, 1, 3, 4, '2024-06-05');

INSERT INTO Orders VALUES (104, 2, 1, 6, '2024-06-04');

INSERT INTO Orders VALUES (105, 2, 2, 2, '2024-06-06');

INSERT INTO Orders VALUES (106, 2, 3, 3, '2024-06-08');

#### **b) Updating Prices with a 10% Increase Where Quantity Sold > 5**

UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID FROM Orders WHERE Quantity > 5

);

#### **c) Deleting Orders Where the Product Has Never Been Sold**

DELETE FROM Orders

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.   
**a) Customers Who Ordered More Than 3 Different Products**  
SELECT CustomerID  
FROM Orders  
GROUP BY CustomerID  
HAVING COUNT(DISTINCT ProductID) > 3;  
**b) Products Not Ordered by Any Customer**

## SELECT \*

## FROM Products

## WHERE ProductID NOT IN (

## SELECT DISTINCT ProductID FROM Orders

## );

#### **c) Count of Orders Placed by Each Customer in the Last 30 Days**

## 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 the current user and host. d) Use nested functions to format a customer greeting string.

**ANSWERS**

#### **a) Using string functions to standardize and extract parts from customer email IDs**

Example 1: Convert all email IDs to lowercase for consistency:

SELECT LOWER(Email) AS StandardizedEmail

FROM Customers;

Example 2: Extract the domain part from the email (after '@'):

SELECT Email,

SUBSTRING(Email, LOCATE('@', Email) + 1) AS EmailDomain

FROM Customers;

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

SELECT OrderID, OrderDate,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

#### **c) Use system functions to return current user and host**

SELECT CURRENT\_USER() AS CurrentUser,

USER() AS LoggedInUser,

SESSION\_USER() AS SessionUser;

#### **d) Use nested functions to format a customer greeting string**

SELECT CONCAT('Hello, ', UPPER(Name), '! Welcome back.') AS Greeting

FROM Customers;

SELECT CONCAT('Hello, ', CONCAT(UPPER(LEFT(Name,1)), LOWER(SUBSTRING(Name,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 clauses to filter categories with revenue > 100000. **ANSWERS**

#### **a) Aggregate total revenue by product category**

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

SELECT P.Category,

SUM(P.Price \* O.Quantity) AS Revenue

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

GROUP BY P.Category WITH ROLLUP;

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

SELECT P.Category,

SUM(P.Price \* O.Quantity) AS Revenue

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)**

**ANSWERS**

## 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.

#### **a) Self Join to List Customers Referred by Other Customers**

SELECT C1.Name AS ReferredCustomer, C2.Name AS Referrer

FROM Customers C1

JOIN Customers C2 ON C1.ReferredBy = C2.CustomerID;

#### **b) Equi Join Across Orders and Products**

SELECT O.OrderID, O.Quantity, P.ProductName, P.Price

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID;

#### **c) Join Customers and Orders to Display Top 3 Spenders Using Window Function**

SELECT CustomerID, TotalSpend

FROM (

SELECT O.CustomerID, SUM(P.Price \* O.Quantity) AS TotalSpend,

RANK() OVER (ORDER BY SUM(P.Price \* O.Quantity) DESC) AS SpendRank

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

GROUP BY O.CustomerID

) Ranked

WHERE SpendRank <= 3;

#### **d) 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;

#### **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

## Q9. (5 marks) a) Correlated subquery to get customers whose order amount exceeds their average. b) Subquery using EXISTS to find customers with at least 2 different products. c) Use ALL to find customers who ordered more than every other customer. d) Use ANY to find products costlier than some in category 'Electronics'. e) Nested subquery to list top 3 best-selling products.

**ANSWERS**

#### **a) Correlated Subquery to Get Customers Whose Order Amount Exceeds Their Average**

SELECT CustomerID, OrderID, (Price \* Quantity) AS OrderAmount

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

WHERE (Price \* Quantity) >

(SELECT AVG(P2.Price \* O2.Quantity)

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.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

FROM Orders

GROUP BY CustomerID

HAVING COUNT(\*) > ALL (

SELECT COUNT(\*) FROM Orders GROUP BY CustomerID

);

#### **d) Use ANY to Find Products Costlier Than Some in Category 'Electronics'**

SELECT ProductName

FROM Products

WHERE Price > ANY (

SELECT Price FROM Products WHERE Category = 'Electronics'

);

#### **e) Nested Subquery to List Top 3 Best-Selling Products**

SELECT ProductID, TotalSold

FROM (

SELECT ProductID, SUM(Quantity) AS TotalSold,

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

FROM Orders

GROUP BY ProductID

) Ranked

WHERE rnk <= 3;

## Q10. (5 marks) a) Simulate INTERSECT using INNER JOIN on two customer segments. b) Use EXCEPT to find products in inventory not yet ordered. c) Simulate MERGE: If customer exists, update; else insert. d) Use UNION to combine two regional customer tables. e) Write a WITH CTE that ranks customers by total spend and filters top 5

**ANSWERS**

#### **a) Simulate INTERSECT Using INNER JOIN on Two Customer Segments**

SELECT P.CustomerID

FROM PremiumCustomers P

INNER JOIN LoyalCustomers L ON P.CustomerID = L.CustomerID;

#### **b) Use EXCEPT to Find Products in Inventory Not Yet Ordered**

SELECT ProductID

FROM Products

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

#### **c) Simulate MERGE: If Customer Exists, Update; Else Insert**

UPDATE Customers

SET Email = 'newemail@example.com'

WHERE CustomerID = 5;

INSERT INTO Customers (CustomerID, Name, Email)

SELECT 5, 'John', 'newemail@example.com'

WHERE NOT EXISTS (SELECT 1 FROM Customers WHERE CustomerID = 5);

#### **d) Use UNION to Combine Two Regional Customer Tables**

SELECT \* FROM NorthRegionCustomers

UNION

SELECT \* FROM SouthRegionCustomers;

#### **e) Write a WITH CTE That Ranks Customers by Total Spend and Filters Top 5**

WITH SpendRank AS (

SELECT CustomerID, SUM(P.Price \* O.Quantity) AS TotalSpend,

RANK() OVER (ORDER BY SUM(P.Price \* O.Quantity) DESC) AS rnk

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

GROUP BY CustomerID

)

SELECT \* FROM SpendRank

WHERE rnk <= 5;

## **SQL Practical 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.

**ANSWERS**

#### **SQL (Structured Query Language):**

SQL databases are relational, which means they store data in tables with rows and columns. They require a predefined schema, and relationships between tables are maintained using keys.

**Advantages of SQL:**

1. **Strong Data Integrity:** Data is well-structured using constraints, primary and foreign keys.
2. **ACID Compliance:** Transactions are reliable and consistent, ideal for banking or financial systems.

**Disadvantages of SQL:**

1. **Less Flexible Schema:** Difficult to change structure frequently.
2. **Scales Vertically:** Harder to handle very large data across multiple servers.

**Real-world Example:** Banking systems like SBI or ICICI use SQL databases (e.g., MySQL, PostgreSQL) for secure and structured data handling.

#### **NoSQL (Not Only SQL):**

NoSQL databases are non-relational. They store data in different formats like key-value, document, column, or graph. They are schema-less, which allows for more flexibility.

**Advantages of NoSQL:**

1. **Flexible Schema:** You can store different data formats without changing structure.
2. **Scales Horizontally:** Easily handles large-scale data by adding more servers.

**Disadvantages of NoSQL:**

1. **No Standard Query Language:** Each NoSQL DB has its own syntax.
2. **Weaker Consistency:** Not fully ACID compliant, which can lead to inconsistency in sensitive data.

**Real-world Example:** Companies like Amazon or Netflix use NoSQL (e.g., MongoDB, Cassandra) to manage huge, unstructured datasets like user preferences or product catalogs.

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

**ANSWERS**

**Unnormalized Data:** Student(StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

First Normal Form (1NF):  
**Student Table (1NF):** (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

Second Normal Form (2NF):  
**Student Table:** (StudentID, Name)  
**Course Table:** (CourseID, CourseName, InstructorName, InstructorPhone)  
**Enrollment Table:** (StudentID, CourseID)

Third Normal Form (3NF):  
**Student Table:** (StudentID, Name)  
**Course Table:** (CourseID, CourseName, InstructorID)  
**Instructor Table:** (InstructorID, InstructorName, InstructorPhone)  
**Enrollment Table:** (StudentID, CourseID)

## 

## 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 a database named StudentDB**

CREATE DATABASE StudentDB;

USE StudentDB;

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

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100)

);

#### **c) Rename the table to Student\_Info**

RENAME TABLE Students TO Student\_Info;

#### **d) Add a column PhoneNumber:**

#### ALTER TABLE Student\_Info

ADD PhoneNumber VARCHAR(15);

#### **e) Drop the table**

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) Insert 3 student records into Student\_Info**

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

VALUES (1, 'Ananya Rao', '2002-05-14', 'ananya@gmail.com', '9876543210');

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

VALUES (2, 'Rahul Verma', '1999-10-20', 'rahul@yahoo.com', '9123456789');

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

VALUES (3, 'Sneha Das', '2001-03-25', 'sneha@gmail.com', '9988776655');

#### **b) Update one student's phone number**

UPDATE Student\_Info

SET PhoneNumber = '9000012345'

WHERE StudentID = 1;

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

DELETE FROM Student\_Info

WHERE Email LIKE '%@gmail.com'

LIMIT 1;

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

SELECT Name, Email

FROM Student\_Info

WHERE YEAR(DOB) > 2000;

#### **e) Retrieve distinct domain names from the email column**

SELECT DISTINCT

SUBSTRING(Email, LOCATE('@', Email) + 1) AS EmailDomain

FROM Student\_Info;

## Q5. (5 marks) a) Retrieve students with names starting with 'A'. b) Retrieve students with phone numbers between 9000000000 and 9999999999. c) Retrieve students using the 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**

#### **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 999999999**

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', 'Delhi', 'Mumbai');

#### **d) 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 '%@gmail.com' OR Email LIKE '%@yahoo.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 DateOfBirth

FROM Student\_Info 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 VALUES (1, 'Math', 85);

INSERT INTO Marks VALUES (2, 'Science', 75);

INSERT INTO Marks VALUES (3, 'English', 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 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**

#### **a) Get the current date and format it as "YYYY-MM-DD"**

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

#### **b) Extract month and year from a DOB column**

SELECT MONTH(DOB) AS BirthMonth, YEAR(DOB) AS BirthYear

FROM Student\_Info;

#### **c) Convert a student's name to uppercase**

SELECT UPPER(Name) AS UpperCaseName

FROM Student\_Info;

#### **d) Round off marks to 2 decimal places**

SELECT 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;

## 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**

#### **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 HighestMarks

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 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**

Assume:

* Student\_Info(StudentID, Name)
* Courses(CourseID, CourseName)
* Enrollments(StudentID, CourseID)

#### **a) Inner Join to retrieve students and their courses**

SELECT S.Name, C.CourseName

FROM Student\_Info S

JOIN Enrollments E ON S.StudentID = E.StudentID

JOIN Courses C ON E.CourseID = C.CourseID;

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

SELECT S.Name, C.CourseName

FROM Student\_Info S

LEFT JOIN Enrollments E ON S.StudentID = E.StudentID

LEFT JOIN Courses C ON E.CourseID = C.CourseID;

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

SELECT S.Name, C.CourseName

FROM Student\_Info S

RIGHT JOIN Enrollments E ON S.StudentID = E.StudentID

RIGHT JOIN Courses C ON E.CourseID = C.CourseID;

#### **d) Full Outer Join equivalent using UNION**

SELECT S.Name, C.CourseName

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, C.CourseName

FROM Student\_Info S

RIGHT JOIN Enrollments E ON S.StudentID = E.StudentID

RIGHT JOIN Courses C ON E.CourseID = C.CourseID;

#### **e) Cross Join to show all combinations**

SELECT S.Name, C.CourseName

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**

### **a) Students who scored more than average in 'Maths'**

SELECT StudentID, Marks

FROM Marks

WHERE Subject = 'Maths' AND Marks > (

SELECT AVG(Marks)

FROM Marks

WHERE Subject = 'Maths'

);

### **b) Students not in the Marks table**

SELECT \*

FROM Student\_Info

WHERE StudentID NOT IN (

SELECT DISTINCT StudentID FROM Marks

);

### **c) 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

);

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

SELECT StudentID, Marks

FROM Marks

WHERE Subject = 'Science' AND Marks > ALL (

SELECT Marks FROM Marks WHERE Subject = 'Science'

);

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

SELECT StudentID, Marks

FROM Marks

WHERE Subject = 'English' AND Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English'

);

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

**ANSWERS**

### **a) UNION of student names from two tables**

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Old\_Students;

### **b) INTERSECT to find common students**

SELECT S.Name

FROM Student\_Info S

INNER JOIN Old\_Students O ON S.Name = O.Name;

### **c) EXCEPT to list students in Student\_Info but not in Marks**

SELECT S.\*

FROM Student\_Info S

LEFT JOIN Marks M ON S.StudentID = M.StudentID

WHERE M.StudentID IS NULL;

### **d) MERGE concept: If exists, update; else insert**

UPDATE Student\_Info

SET Email = 'newemail@example.com'

WHERE StudentID = 1;

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

SELECT 1, 'Amit', '2001-01-01', 'newemail@example.com'

WHERE NOT EXISTS (

SELECT 1 FROM Student\_Info WHERE StudentID = 1

);

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

SELECT M.StudentID, M.Subject, M.Marks

FROM Marks M

WHERE M.Marks > (

SELECT AVG(M2.Marks)

FROM Marks M2

WHERE M2.Subject = M.Subject

);