**DATE : 11.06.2025**

**SQL QUIZ-1**

**Section 1: Managing Databases**

1. **Which of the following is NOT a system database in SQL Server?**a) master  
   b) model  
   c) tempdb  
   **d) userdb**
2. **Which system database stores all login accounts and configuration settings?**a) tempdb  
   b) model  
   **c) master**  
   d) msdb
3. **What is the purpose of the model database in SQL Server?**  
   a) Backup  
   b) Log storage  
   **c) Template for new databases**  
   d) System configuration
4. **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
5. **Which SQL command is used to create a new database?**  
   a) MAKE DATABASE  
   b) NEW DATABASE  
   c**) CREATE DATABASE**  
   d) INIT DATABASE
6. **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
7. **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**

1. **Which data type should be used to store a date of birth?**  
   a) VARCHAR  
   **b) DATE**  
   c) INT  
   d) TEXT
2. **What command is used to create a table?**  
   a) MAKE TABLE  
   b) INSERT TABLE  
   **c) CREATE TABLE**  
   d) DEFINE TABLE
3. **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
4. **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
5. **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**

1. **Which command adds data into a table?**  
   **a) INSERT INTO**  
   b) ADD ROW  
   c) CREATE DATA  
   d) APPEND TO
2. **Which clause is used to update data in a table?**  
   a) MODIFY  
   **b) UPDATE**  
   c) CHANGE  
   d) SET TABLE
3. **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
4. **Which clause is used to filter rows in a SELECT statement?**  
   a) HAVING  
   b) SELECT  
   **c) WHERE**  
   d) ORDER BY
5. **Which keyword ensures no duplicate records are returned?**  
   a) UNIQUE  
   b) NO\_REPEAT  
   **c) DISTINCT**  
   d) ONLY
6. **What does the LIKE keyword do in SQL?**  
   a) Finds exact matches  
   **b) Finds pattern-based matches**  
   c) Sorts records  
   d) Deletes matches
7. **Which operator is used to combine multiple conditions in a WHERE clause?**  
   a) TO  
   b) WITH  
   **c) AND / OR**d) IF / ELSE
8. **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

**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**

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');

**DATE: 12.06.2025**

**Medium-Level Practical SQL Questions**

**1. Insert and Update with Integrity**

Create the 'students' table with constraints:

create table students (  
 student\_id int primary key,  
 name varchar(50) not null,  
 email varchar(100) unique not null,  
 marks int check (marks >= 0 and marks <= 100)  
);

Insert 5 records:

Insert into students values  
(1, 'thaara', 'thaara@example.com', 98),  
(2, 'renu', 'renu@example.com', 85),  
(3, 'mohana', 'mohana@example.com', 65),  
(4, 'abi', 'abi@example.com', 76),  
(5, 'gowtham', 'gowtham@example.com', 88);

Update a student's marks:

Update students  
set marks = 90  
where student\_id = 4;

**2. String Function Challenge**

Split full name and display name lengths:

Select   
 full\_name,  
 substring\_index(full\_name, ' ', 1) as first\_name,  
 substring\_index(full\_name, ' ', -1) as last\_name,  
 length(substring\_index(full\_name, ' ', 1)) as firstname\_length,  
 length(substring\_index(full\_name, ' ', -1)) as lastname\_length  
from customers;

**3. Date Function Usage**

Extract month, year and days ago from sale\_date:

Select   
 sale\_date,  
 monthname(sale\_date) as month\_name,  
 year(sale\_date) as sale\_year,  
 datediff(curdate(), sale\_date) as days\_ago  
from sales;

**4. Mathematical Functions on Salary**

Calculate salary hike and round salary:

Select   
 salary,  
 salary \* 1.10 as salary\_after\_hike,  
 round(salary, -2) as rounded\_salary  
from employees;

**5. System Function Check**

Retrieve system information:

Select   
 now() as current\_datetime,  
 database() as current\_database,  
 user() as login\_user;

**6. Demo: Custom Result Set**

Display product name in uppercase and handle NULL prices:

Select   
 upper(product\_name) as product\_name\_upper,  
from products;

Update products

Set price = 'not available'

Where price is null;

SQL Joins and Aggregates Practice

**7. Aggregate Functions Practice**

From a 'transactions' table:

Select   
 sum(amount) as total\_sales,  
 avg(amount) as average\_sale,  
 max(amount) as max\_sale,  
 min(amount) as min\_sale  
from transactions;

**8. Grouping with Aggregation**

Group by product category:

Select   
 category,  
 sum(sale\_amount) as total\_sales,  
 count(\*) as transaction\_count  
from sales  
group by category;

**9. Inner Join for Orders and Customers**

Join 'orders' and 'customers' to show only customers with orders:

Select   
 c.name as customer\_name,  
 o.amount as order\_amount  
from orders o  
inner join customers c on o.customer\_id = c.id;

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

Show all products and their order details (if available):

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**

Show all customers, even if they don't have contact info:

Select   
 c.customer\_id,  
 c.name,  
 ct.phone\_number  
from contacts ct  
right join customers c on ct.customer\_id = c.customer\_id;

**12. Full Outer Join for Suppliers and Products**

List all suppliers and products with matching or NULL values:

Select   
 s.supplier\_name,  
 p.product\_name  
from suppliers s  
full outer join products p on s.supplier\_id = p.supplier\_id;

**13. Cross Join for Offers**

Show all possible combinations of products and offers.

Select

p.product\_name,

o.offer\_name

From products p

Cross join offers o;

**14. Join with Aggregation**

Join orders and products, group by product category, and show total quantity sold and average price.

Select

p.category,

sum(o.quantity) 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.category;

**15. Join with Grouping and Filter**

Join students and marks, display student name and average marks, and filter to show only students with average marks > 75

Select

s.name as student\_name,

avg(m.marks) as average\_marks

From students s

Join marks m on s.student\_id = m.student\_id

Group by s.name

Having avg(m.marks) > 75;

**DATE: 13.06.2025**

**SQL Practical Question Paper**

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

**Q1. Difference between SQL and NoSQL (3 marks)**

| **Feature** | **SQL (Relational)** | **NoSQL (Non-relational)** |
| --- | --- | --- |
| Structure | Structured tables with fixed schemas | Flexible schemas: document, key-value, graph |
| Query Language | Uses SQL | Varies by type (MongoDB uses JSON-like queries) |
| Scalability | Vertical scaling | Horizontal scaling |
| ACID Compliance | Fully supports ACID transactions | BASE properties (eventual consistency) |

Examples:

* SQL: MySQL, PostgreSQL (e.g., Banking System)
* NoSQL: MongoDB, Cassandra (e.g., Social Media Posts)

Advantages:

* SQL: Structured data, powerful queries.
* NoSQL: High scalability, flexible with semi-structured/unstructured data.

Disadvantages:

* SQL: Not ideal for huge-scale distributed systems.
* NoSQL: Lacks standardization, weaker transaction support.

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

* 1NF: Remove multivalued attributes.

StudentID | Name | CourseID | CourseName | InstructorName | InstructorPhone

* 2NF: Remove partial dependencies (assumes composite key is CourseID + StudentID)

Students(StudentID, Name)

Courses(CourseID, CourseName, InstructorName, InstructorPhone)

Enrollments(StudentID, CourseID)

* 3NF: Remove transitive dependencies

Instructors(InstructorID, InstructorName, InstructorPhone)

Courses(CourseID, CourseName, InstructorID)

**Q3. Database Operations (5 marks)**

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.

EXEC sp\_rename 'Students', 'Student\_Info'; -- SQL Server

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.

INSERT INTO Student\_Info VALUES

(1, 'John Doe', '2000-05-15', 'john@example.com', '9876543210'),

(2, 'Alice Smith', '2001-08-20', 'alice@gmail.com', '9123456789'),

(3, 'Bob Johnson', '1999-11-30', 'bob@yahoo.com', '9988776655');

b) Update one student's phone number.

UPDATE Student\_Info

SET PhoneNumber = '9999999999'

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 YEAR(DOB) > 2000;

e) Retrieve distinct domain names from the email column.

SELECT DISTINCT SUBSTRING(Email, CHARINDEX('@', Email) + 1, LEN(Email)) AS Domain

FROM Student\_Info;

**Q5. (5 marks)**

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 ('Coimbatore', 'Chennai', 'Banglore');

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

SELECT \* FROM Student\_Info

WHERE (YEAR(GETDATE()) - YEAR(DOB) > 20)

AND (Email LIKE '%@yahoo.com' OR Email LIKE '%@hotmail.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 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),

PRIMARY KEY (StudentID, Subject)

);

INSERT INTO Marks VALUES

(1, 'Maths', 85.5),

(1, 'Science', 78.0),

(2, 'Maths', 92.3),

(3, 'English', 68.5);

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".

SELECT DATE\_FORMAT(NOW(), '%Y-%m-%d') AS FormattedDate;

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 UCASE(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 CURRENT\_USER() AS CurrentUser, DATABASE() AS CurrentDatabase;

**Q8. (5 marks)**

a) Display total marks of each student.

Select Subject, SUM(marks) as Total\_Marks

From Marks

Group By StudentID;

b) Display subject-wise highest mark.

Select Subject, MAX(marks) as Highest\_Mark

From Marks

Group By Subject;

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

Select Subject, AVG(marks) as Average\_Marks

From Marks

Group By Subject

Having AVG(marks)>75;

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

**Q9. (5 marks)**

a) Inner Join to retrieve students and their courses.

Select S.StudentID, S.Name, C.CourseName

From Student\_Info As S

Inner join Courses As C ON S.StudentID = C.StudentID;

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

Select S.StudentID, S.Name, C.CourseName

From Student\_Info As S

Left join Courses As C ON S.StudentID = C.StudentID;

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

Select S.StudentID, S.Name, C.CourseName

From Student\_Info As S

Right join Courses As C ON S.StudentID = C.StudentID;

d) Full Outer Join equivalent using UNION.

Select S.StudentID, S.Name, C.CourseName

From Student\_Info AS S

Left join Courses AS C ON S.StudentID = C.StudentID

UNION

Select S.StudentID, S.Name, C.CourseName

From Student\_Info AS S

Right join Courses AS C ON S.StudentID = C.StudentID;

e) Cross Join to show all combinations.

Select S.StudentID, S.Name, C.CourseName

From Student\_Info AS S

Cross join Courses AS C;

**Q10. (5 marks)**

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

SELECT s.\*

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject = 'Maths' AND m.Marks > (

SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths'

);

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 m.StudentID = s.StudentID

);

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

SELECT s.\*

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Marks > ALL (

SELECT Marks FROM Marks WHERE Subject = 'Science'

);

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

SELECT s.\*

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English'

);

**Q11. (5 marks)**

a) UNION of student names from two tables

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Another\_Student\_Table;

b) INTERSECT to find common students

SELECT DISTINCT s.Name

FROM Student\_Info s

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

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

SELECT s.Name

FROM Student\_Info s

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

WHERE m.StudentID IS NULL;

d) MERGE concept or simulate with UPDATE and INSERT

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

VALUES

(4, 'New Student', 'new@example.com', '1234567890'),

(5, 'Existing Student', 'update@example.com', '9876543210')

ON DUPLICATE KEY UPDATE

Name = VALUES(Name),

Email = VALUES(Email),

PhoneNumber = VALUES(PhoneNumber);

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

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

FROM Student\_Info s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Marks > (

SELECT AVG(Marks)

FROM Marks

WHERE Subject = m.Subject

)

ORDER BY m.Subject, m.Marks DESC;

**SQL Practical Question Paper**

**Section A: Advanced Concepts & Schema Design (10 Marks)**

**Q1. Explain with examples the scenarios where NoSQL is preferred over SQL. Discuss types of NoSQL databases and suggest a real-time application for each.(4 marks)**

NoSQL vs SQL Scenarios:

* Use NoSQL when:
  + Data is unstructured/semi-structured: e.g., social media posts, IoT sensor data.
  + Need high scalability: e.g., real-time big data analytics or web-scale applications.
  + Frequent schema changes: NoSQL adapts easily.
  + High write loads and distributed systems.

Types of NoSQL & Real-Time Applications:

| Type | Description | Real-Time Example |
| --- | --- | --- |
| Document-based | Stores JSON-like documents | MongoDB for product catalogs |
| Key-Value | Simple key-value pairs | Redis for caching session data |
| Column-family | Stores data in columns | Cassandra for time-series logs |
| Graph-based | Stores data as nodes and edges | Neo4j for social networks |
|  |  |  |

**Q2. 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.( 6 marks)**

1NF (First Normal Form):

* Remove repeating groups (Orders)
* Create separate Order table

Customer (CustomerID, Name)

Order (OrderID, CustomerID, ProductID, Quantity, ProductName)

2NF (Second Normal Form):

* Remove partial dependencies (ProductName depends only on ProductID)
* Create Product table

Customer (CustomerID, Name)

Order (OrderID, CustomerID, ProductID, Quantity)

Product (ProductID, ProductName)

3NF (Third Normal Form):

* No transitive dependencies (already satisfied)

BCNF (Boyce-Codd Normal Form):

* Every determinant is a candidate key (already satisfied)

Final schema:

Customer (CustomerID PK, Name)

Product (ProductID PK, ProductName)

Order (OrderID PK, CustomerID FK, ProductID FK, Quantity)

**Q3. (5 marks)**

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) NOT NULL

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100) NOT NULL,

Price DECIMAL(10,2) NOT NULL

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID),

ProductID INT FOREIGN KEY REFERENCES Products(ProductID),

Quantity INT NOT NULL,

OrderDate DATE DEFAULT GETDATE(),

CONSTRAINT CHK\_Quantity CHECK (Quantity > 0)

);

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

CONSTRAINT CHK\_Quantity CHECK (Quantity > 0)

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

ALTER TABLE Products ADD Discount DECIMAL(5,2) DEFAULT 0;

UPDATE Products SET Discount = 10.00 WHERE ProductID IN (1, 3, 5);

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

a) Insert 3 sample orders per customer.

INSERT INTO Orders VALUES

(1, 101, 1, 2, '2023-06-01'),

(2, 101, 2, 1, '2023-06-05'),

(3, 101, 3, 3, '2023-06-10'),

(4, 102, 1, 5, '2023-06-02'),

(5, 102, 4, 2, '2023-06-07'),

(6, 102, 5, 1, '2023-06-12'),

(7, 103, 2, 6, '2023-06-03'),

(8, 103, 3, 4, '2023-06-08'),

(9, 103, 4, 3, '2023-06-11');

b) 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

);

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

DELETE FROM Orders

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID FROM Orders

);

**Q5. (marks) Retrieve the following:**

a) Customers who ordered more than 3 different products.

SELECT c.CustomerID, c.Name

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerID, c.Name

HAVING COUNT(DISTINCT o.ProductID) > 3;

b) Products not ordered by any customer.

SELECT p.ProductID, p.ProductName

FROM Products p

LEFT JOIN Orders o ON p.ProductID = o.ProductID

WHERE o.OrderID IS NULL;

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

SELECT c.CustomerID, c.Name, COUNT(o.OrderID) AS OrderCount

FROM Customers c

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

WHERE o.OrderDate >= DATEADD(day, -30, GETDATE())

GROUP BY c.CustomerID, c.Name;

Section C: Advanced Functions and Aggregations

**Q6. (5 marks)**

a) 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;

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

SELECT

OrderID,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

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

SELECT CURRENT\_USER() AS User,

HOST\_NAME() AS Host;

SELECT SYSTEM\_USER();

d)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)**

a) 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;

b) 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;

c) 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)**

a) 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;

B) Equi join across Orders and Products.

SELECT o.OrderID, p.ProductName, o.Quantity

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 \*

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;

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 ProductA,

p2.ProductName AS ProductB

FROM Products p1

CROSS JOIN Products p2

WHERE p1.ProductID < p2.ProductID;

**Q9. (5 marks)**

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

SELECT \*

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);

b) 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

);

c) 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

);

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

SELECT \*

FROM Products

WHERE Price > ANY (

SELECT Price

FROM Products

WHERE Category = 'Electronics'

);

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

SELECT \* 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)**

a) 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;

b) 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;

c) 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);

d) Use UNION to combine two regional customer tables.

SELECT CustomerID, Name, Email FROM EastRegionCustomers

UNION

SELECT CustomerID, Name, Email FROM WestRegionCustomers;

e) 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;

**Queries Practice**

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) --Find employees who earn more than the average salary

SELECT employee\_id, first\_name, last\_name, salary

FROM employees

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

2) -- Find departments that have at least one employee

SELECT department\_id, department\_name

FROM departments d

WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department\_id = d.department\_id);

3) -- Find employees who earn more than any employee in department 50

SELECT employee\_id, first\_name, last\_name, salary

FROM employees

WHERE salary > ANY (SELECT salary FROM employees WHERE department\_id = 50);

4) -- Find employees who earn more than all employees in department 60

SELECT employee\_id, first\_name, last\_name, salary

FROM employees

WHERE salary > ALL (SELECT salary FROM employees WHERE department\_id = 60);

5) -- Find employees who work in departments located in the US

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location\_id IN (

SELECT location\_id

FROM locations

WHERE country\_id = 'US'

)

);

6) -- Find employees who earn more than the average salary in their department

SELECT e.employee\_id, e.first\_name, e.last\_name, e.salary, e.department\_id

FROM employees e

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id = e.department\_id

);

7) -- Combine customers and employees who live in California

SELECT customer\_id AS id, customer\_name AS name, 'Customer' AS type

FROM customers

WHERE state = 'CA'

UNION

SELECT employee\_id, first\_name || ' ' || last\_name, 'Employee'

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location\_id IN (

SELECT location\_id

FROM locations

WHERE state\_province = 'California'

)

);

8) -- Find products that were ordered in both January and February

SELECT product\_id

FROM order\_details

WHERE order\_id IN (SELECT order\_id FROM orders WHERE order\_date BETWEEN '2023-01-01' AND '2023-01-31')

INTERSECT

SELECT product\_id

FROM order\_details

WHERE order\_id IN (SELECT order\_id FROM orders WHERE order\_date BETWEEN '2023-02-01' AND '2023-02-28');

9) -- Find customers who haven't placed any orders

SELECT customer\_id, customer\_name

FROM customers

EXCEPT

SELECT c.customer\_id, c.customer\_name

FROM customers c

JOIN orders o ON c.customer\_id = o.customer\_id;

10) -- Merge data from source table into target table

MERGE INTO target\_table t

USING (

SELECT product\_id, product\_name, category\_id

FROM source\_products

WHERE updated\_date > SYSDATE-30

) s

ON (t.product\_id = s.product\_id)

WHEN MATCHED THEN

UPDATE SET t.product\_name = s.product\_name, t.category\_id = s.category\_id

WHEN NOT MATCHED THEN

INSERT (product\_id, product\_name, category\_id)

VALUES (s.product\_id, s.product\_name, s.category\_id);