DAY - 1 9-6-25

Home work:-

1) Explain all the algorithm basics in brief and compare

ANS:**1. Heuristic / Brute Force Technique**

* **Brute Force**: Tries all possible solutions to find the correct one. It's simple but inefficient for large inputs (e.g., trying all combinations to guess a password).
* **Heuristic**: An approximate method used when exact solutions are impractical; often used in AI and optimization problems (e.g., A\* search).

**2. Greedy Approach**

* Solves problems by choosing the best option at each step with the hope that this leads to the global optimum.
* Works well for problems like **activity selection**, **Huffman coding**, and **prim's algorithm**.

**3. Divide and Conquer**

* Splits problems into smaller subproblems, solves them independently, and then combines solutions.
* Used in algorithms like **Merge Sort**, **Quick Sort**, and **Binary Search**.

**4. Dynamic Programming**

* Solves problems by breaking them into subproblems, storing the results (memoization) to avoid redundant computations.
* Efficient for **Fibonacci numbers**, **knapsack problem**, **Longest Common Subsequence (LCS)**.

2) Compare all sorting algorithm and choose any two best according to you and why

ANS:

* **Bubble, Selection, Insertion**: Easy, O(n²), slow for big data.
* **Merge Sort**: Stable, always O(n log n), uses extra space.
* **Quick Sort**: Very fast in practice, O(n log n) average, in-place.
* Best: **Quick Sort** (practical) & **Merge Sort** (stable and reliable).

3) compare searching algorithm

ANS: **Linear Search**: Simple, O(n), works on unsorted data.

**Binary Search**: O(log n), needs sorted data, much faster.  
 Use **Binary Search** for speed, if data is sorted.

4) why we use BST and what is the need of AVL and difference between BST and AVL tree

ANS:

**Use of BST (Binary Search Tree)**

* To **store elements in a sorted structure**
* Allows **fast searching, insertion, and deletion**
* Time complexity:
  + **Best/Average case**: O(log n)
  + **Worst case (unbalanced)**: O(n)
* Supports **inorder traversal** to get sorted data

**AVL Tree**

* **Problem with normal BST**: It can become **unbalanced** (like a linked list) if data is inserted in sorted order.
* AVL Tree ensures:
  + **Balance factor is maintained** (–1, 0, +1)
  + **Always height-balanced**

**Difference Between BST and AVL Tree**

| **Feature** | **BST (Binary Search Tree)** | **AVL Tree (Adelson-Velsky and Landis Tree)** |
| --- | --- | --- |
| **Balance** | Not self-balancing | Self-balancing |
| **Worst Case Time** | O(n) (if unbalanced) | O(log n) (always balanced) |
| **Performance** | Fast only if balanced | Consistently fast |
| **Rotations** | No rotations used | Uses rotations to balance |
| **Implementation** | Simple to implement | More complex due to rotations |
| **Use Case** | Small or mostly balanced datasets | Large and frequently changing datasets |

DAY - 3 11-6-25

1) create database HexaAirlines

ANS: create database HexaAirlines;

use HexaAirlines;

2) create tables flight, piolets ,airhostess , foodTeam, customers

3) fill 6 to 7 rows in these and have primary and foreign key

ANS: CREATE TABLE Flight (

FlightID INT AUTO\_INCREMENT PRIMARY KEY,

AirlineName VARCHAR(50),

Destination VARCHAR(50)

);

INSERT INTO Flight (AirlineName, Destination) VALUES

('IndiGo', 'Bangalore'),

('JetAirways', 'Mumbai'),

('AirIndia', 'Goa'),

('IndiGo', 'Chennai'),

('AirIndia', 'Delhi'),

('JetAirways', 'Goa');

CREATE TABLE Pilots (

PilotID INT AUTO\_INCREMENT PRIMARY KEY,

PilotName VARCHAR(50),

FlightID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID)

);

INSERT INTO Pilots (PilotName, FlightID) VALUES

('Captain Raj', 1),

('Captain Ali', 2),

('Captain Veer', 3),

('Captain John', 4),

('Captain Arya', 5),

('Captain Neil', 6);

CREATE TABLE AirHostess (

HostessID INT AUTO\_INCREMENT PRIMARY KEY,

HostessName VARCHAR(50),

FlightID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID)

);

INSERT INTO AirHostess (HostessName, FlightID) VALUES

('Neha', 1),

('Sara', 2),

('Pooja', 3),

('Anjali', 4),

('Tina', 5),

('Zara', 6);

CREATE TABLE FoodTeam (

FoodID INT AUTO\_INCREMENT PRIMARY KEY,

FoodType VARCHAR(30)

);

INSERT INTO FoodTeam (FoodType) VALUES

('Veg'),

('Non-Veg');

CREATE TABLE Customers (

CustomerID INT AUTO\_INCREMENT PRIMARY KEY,

CustomerName VARCHAR(50),

FlightID INT,

FoodID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID),

FOREIGN KEY (FoodID) REFERENCES FoodTeam(FoodID)

);

INSERT INTO Customers (CustomerName, FlightID, FoodID) VALUES

('Ravi', 1, 1),

('Priya', 2, 2),

('Amit', 3, 1),

('Sneha', 1, 2),

('Karan', 3, 1),

('Divya', 2, 1),

('Asha', 6, 2);

4) select customers going to bangalore in indigo

ANS: SELECT CustomerName

FROM Customers

JOIN Flight ON Customers.FlightID = Flight.FlightID

WHERE Destination = 'Bangalore' AND AirlineName = 'IndiGo';

5) select customers going to mumbai in jetairways

ANS: SELECT CustomerName

FROM Customers

JOIN Flight ON Customers.FlightID = Flight.FlightID

WHERE Destination = 'Mumbai' AND AirlineName = 'JetAirways';

6) select customers going to goa in airindia and having veg food

ANS: SELECT CustomerName

FROM Customers

JOIN Flight ON Customers.FlightID = Flight.FlightID

JOIN FoodTeam ON Customers.FoodID = FoodTeam.FoodID

WHERE Destination = 'Goa' AND AirlineName = 'AirIndia' AND FoodType = 'Veg';

7) add new column in piolets table, covid vaccination date and fill values in column

ANS: ALTER TABLE Pilots

ADD CovidVaccinationDate DATE;

UPDATE Pilots SET CovidVaccinationDate = '2021-03-15' WHERE PilotID = 1;

UPDATE Pilots SET CovidVaccinationDate = '2021-04-10' WHERE PilotID = 2;

UPDATE Pilots SET CovidVaccinationDate = '2021-05-01' WHERE PilotID = 3;

UPDATE Pilots SET CovidVaccinationDate = '2021-03-28' WHERE PilotID = 4;

UPDATE Pilots SET CovidVaccinationDate = '2021-04-22' WHERE PilotID = 5;

UPDATE Pilots SET CovidVaccinationDate = '2021-05-15' WHERE PilotID = 6;

8) change the age of airhostess who are elder then 35 to 34

ANS:

9) update the price for the customer food or flight charge and make changes

10) delete piolets whose age is greater then 60

11) delete airhostess who are working from last 6 years or elder then 50

12) list out the name of piolets

13) list out the name of airhostess

14) use where clause to retrieve data from above tables

15) use in keyword who filter fooditems uses milk or almonds

DAY - 4 12-6-25

1. Insert and Update with Integrity:

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

student's marks ensuring data integrity is maintained.

ANS: CREATE TABLE students (

student\_id INT PRIMARY KEY,

full\_name VARCHAR(100) NOT NULL,

marks INT NOT NULL,

email VARCHAR(100) UNIQUE

);

INSERT INTO students VALUES

(1, 'John', 85, 'john@1.com'),

(2, 'Alice', 92, 'alice@2.com'),

(3, 'Bob', 76, 'bob@3.com'),

(4, 'Diana', 88, 'diana@4.com'),

(5, 'Clark', 95, 'clark@5.com');

UPDATE students SET marks = 90 WHERE student\_id = 3;

select \* from students;

2. String Function Challenge:

Given a 'customers' table with a 'full\_name' column, write a query to display:

- First name

- Last name

- Length of each name

ANS: CREATE TABLE customer (

customer\_id INT PRIMARY KEY,

full\_name VARCHAR(100) NOT NULL

);

INSERT INTO customer VALUES

(1, 'Janalyn maroula'),

(2, 'Anand suresh'),

(3, 'Benjamin fracklin'),

(4, 'Priya mani'),

(5, 'Tommy smith');

SELECT

SUBSTRING\_INDEX(full\_name, ' ', 1) AS first\_name,

SUBSTRING\_INDEX(full\_name, ' ', -1) AS last\_name,

LENGTH(SUBSTRING\_INDEX(full\_name, ' ', 1)) AS first\_name\_length,

LENGTH(SUBSTRING\_INDEX(full\_name, ' ', -1)) AS last\_name\_length

FROM customer;

3. Date Function Usage:

From a 'sales' table with a 'sale\_date' column, write a query to:

- Extract the month name and year

- Display how many days ago the sale happened

ANS: CREATE TABLE sales (

sale\_id INT PRIMARY KEY,

sale\_date DATE,

amount DECIMAL(10, 2),

category VARCHAR(50)

);

INSERT INTO sales VALUES

(1, '2025-06-01', 500.00, 'Electronics'),

(2, '2025-05-28', 250.00, 'Grocery'),

(3, '2025-06-10', 1000.00, 'Furniture'),

(4, '2025-04-15', 300.00, 'Grocery'),

(5, '2025-05-30', 750.00, 'Clothing');

SELECT

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

ANS: CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(100),

salary DECIMAL(10, 2)

);

INSERT INTO employees VALUES

(1, 'Ravi', 38500),

(2, 'Sneha', 41250),

(3, 'Kiran', 27800),

(4, 'Amit', 49200),

(5, 'Geetha', 36900);

SELECT

name,salary,salary \* 1.10 AS salary\_after\_hike,ROUND(salary, -2) AS rounded\_salary

FROM employees;

5. System Function Check:

Retrieve:

- Current date and time

- Database name and logged-in user

ANS: SELECT

NOW() AS current\_datetime,

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'

ANS: CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

price DECIMAL(10, 2)

);

INSERT INTO products VALUES

(1, 'Laptop', 50000),

(2, 'Smartphone', NULL),

(3, 'Headphones', 1500),

(4, 'Keyboard', NULL),

(5, 'Monitor', 7500);

SELECT

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

IFNULL(price, '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

ANS: CREATE TABLE transaction (

trans\_id INT PRIMARY KEY,

amount DECIMAL(10, 2)

);

INSERT INTO transaction VALUES

(1, 10),

(2, 150),

(3, 140),

(4, 134),

(5, 120);

SELECT

SUM(amount) AS total\_sales,

AVG(amount) AS average\_sale,

MAX(amount) AS max\_sale,

MIN(amount) AS min\_sale

FROM transaction;

8. Grouping with Aggregation:

From a 'sales' table:

- Group by product category

- Show total sales and number of transactions in each category

ANS: SELECT

category,

SUM(amount) AS total\_sales,

COUNT(\*) AS num\_transactions

FROM sales

GROUP BY category;

9. Inner Join for Orders and Customers:

Join 'orders' and 'customers' to show:

- Customer name

- Order amount

- Only for customers who made orders

ANS: CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

amount DECIMAL(10,2),

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

);

INSERT INTO orders VALUES

(101, 1, 300),

(102, 3, 500),

(103, 2, 250),

(104, 1, 450);

SELECT

c.full\_name AS customer\_name,

o.amount AS order\_amount

FROM customer c

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

10. Left Join for Products with or without Orders:

Show all products with:

- Their order details (if available)

- Use LEFT JOIN

ANS: ALTER TABLE orders ADD product\_id INT;

UPDATE orders SET product\_id = 1 WHERE order\_id = 101;

UPDATE orders SET product\_id = 2 WHERE order\_id = 102;

UPDATE orders SET product\_id = 5 WHERE order\_id = 103;

UPDATE orders SET product\_id = 3 WHERE order\_id = 104;

SELECT

p.product\_name,

o.order\_id,

o.amount

FROM products p

LEFT JOIN orders o ON p.product\_id = o.product\_id;

11. Right Join for Customer Contacts:

Use a RIGHT JOIN between 'contacts' and 'customers' to display:

- All customers, even if they don't have contact info

ANS: CREATE TABLE contacts (

contact\_id INT PRIMARY KEY,

customer\_id INT,

phone\_number VARCHAR(15)

);

INSERT INTO contacts VALUES

(1, 1, '9999999999'),

(2, 3, '8888888888');

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

ANS:

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

ANS: CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

supplier\_name VARCHAR(100)

);

ALTER TABLE products ADD supplier\_id INT;

INSERT INTO suppliers VALUES

(1, 'Tech World'),

(2, 'Gadget Hub');

UPDATE products SET supplier\_id = 1 WHERE product\_id IN (1, 2);

UPDATE products SET supplier\_id = 2 WHERE product\_id = 3;

SELECT

s.supplier\_name,

p.product\_name

FROM suppliers s

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

UNION

SELECT

s.supplier\_name,

p.product\_name

FROM suppliers s

RIGHT JOIN products p ON s.supplier\_id = p.supplier\_id;

14. Join with Aggregation:

Join 'orders' and 'products', then group by product category and:

- Show total quantity sold and average price per category

ANS: ALTER TABLE products ADD category VARCHAR(50);

UPDATE products SET category = 'Electronics' WHERE product\_id IN (1, 2, 5);

UPDATE products SET category = 'Accessories' WHERE product\_id IN (3, 4);

SELECT

p.category,

SUM(o.amount) AS total\_quantity\_sold,

AVG(p.price) AS average\_price

FROM products p

JOIN orders o ON p.product\_id = o.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

ANS: CREATE TABLE marks (

mark\_id INT PRIMARY KEY,

student\_id INT,

marks INT,

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

);

INSERT INTO marks VALUES

(1, 1, 80),

(2, 1, 90),

(3, 2, 95),

(4, 3, 70),

(5, 3, 85),

(6, 4, 88),

(7, 5, 92);

SELECT

s.full\_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.full\_name

HAVING AVG(m.marks) > 75;

DAY-5 13-6-25

1) Querying Data by Using Subqueries

ANS: create database joins;

use joins;

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

name VARCHAR(50),

department\_id INT,

salary INT

);

INSERT INTO employees VALUES

(1, 'Alice', 10, 50000),

(2, 'Bob', 10, 60000),

(3, 'Carol', 20, 55000),

(4, 'David', 30, 45000),

(5, 'Eva', 20, 62000);

select \* from employees;

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50),

location VARCHAR(50)

);

INSERT INTO departments VALUES

(10, 'HR', 'Chennai'),

(20, 'IT', 'Mumbai'),

(30, 'Sales', 'Chennai');

select \* from departments;

-- TYPE 1: Single-row Subquery

SELECT name, salary

FROM employees

WHERE salary > (

SELECT AVG(salary) FROM employees

);

-- TYPE 2: Multiple-row Subquery

SELECT name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location = 'Chennai'

);

-- TYPE 3: Multiple-column Subquery

SELECT name, department\_id, salary

FROM employees

WHERE (department\_id, salary) IN (

SELECT department\_id, salary

FROM employees

WHERE salary > 50000

);

-- TYPE 4: Correlated Subquery

SELECT e1.name, e1.salary, e1.department\_id

FROM employees e1

WHERE salary > (

SELECT AVG(e2.salary)

FROM employees e2

WHERE e1.department\_id = e2.department\_id

);

-- TYPE 5: Nested Subquery (Multi-level)

SELECT name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location = (

SELECT location FROM departments WHERE department\_id = 10

)

);

2) Querying Data by Using Subqueries Using the EXISTS,

ANS: SELECT department\_name

FROM departments d

WHERE EXISTS (

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

);

3) Querying Data by Using Subqueries using ANY,

ANS: SELECT name, salary

FROM employees

WHERE salary > ANY (

SELECT salary FROM employees WHERE department\_id = 10

);

4) Querying Data by Using Subqueries using ALL Keywords

ANS: SELECT name, salary

FROM employees

WHERE salary > ALL (

SELECT salary FROM employees WHERE department\_id = 10

);

5) Querying Data by Using Subqueries using Using Nested Subqueries

ANS: SELECT name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location = (

SELECT location

FROM departments

WHERE department\_id = 10

)

);

6) Querying Data by Using Subqueries Using Correlated Subqueries

ANS: SELECT e1.name, e1.salary

FROM employees e1

WHERE salary > (

SELECT AVG(salary)

FROM employees e2

WHERE e1.department\_id = e2.department\_id

);

7) Querying Data by Using Subqueries Using UNION,

ANS: SELECT name FROM employees

UNION

SELECT department\_name FROM departments;

8) Querying Data by Using Subqueries using INTERSECT,

9) Querying Data by Using Subqueries using EXCEPT,

10)Querying Data by Using Subqueries using MERGE

SQL Practical Question Paper

Duration: 2 Hours | Total Marks: 60

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.

ANS:

Advantages of SQL:

\*)Strong consistency – Ideal for banking systems (e.g., MySQL in finance).

\*)Supports complex joins and queries.

Disadvantages of SQL:

\*)Scaling is difficult for huge data.

\*)Schema modifications are rigid.

Advantages of NoSQL:

\*)Scalable – used in large apps like Instagram (e.g., MongoDB).

\*)Schema-less – easy for rapid development.

Disadvantages of NoSQL:

\*)Joins not supported well.

\*)Weaker transactional support.

Q2. (2 marks)

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

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

1NF:

Remove multivalued attributes.

StudentID | Name | CourseID | CourseName | InstructorName | InstructorPhone

1 | John | 101 | Maths | Alice | 1237896

1 | John | 102 | CS | Bob | 4568535

2NF:

Remove partial dependencies:

Students(StudentID, Name)

Courses(CourseID, CourseName, InstructorName, InstructorPhone)

Enrollment(StudentID, CourseID)

3NF:

Remove transitive dependencies:

Instructors(InstructorName, InstructorPhone)

Courses(CourseID, CourseName, InstructorName)

Q3. (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.

ALTER TABLE Students RENAME TO Student\_Info;

d) Add a column PhoneNumber.

ALTER TABLE Student\_Info ADD PhoneNumber INT;

e) Drop the table.

DROP TABLE Student\_Info;

Section B: DML & Filtering Data (15 Marks)

Q4. (5 marks)

CREATE TABLE Student\_Info (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100),

PhoneNumber BIGINT,

City VARCHAR(100)

);

a) Insert 3 student records into Student\_Info.

INSERT INTO Student\_Info VALUES

(1, 'ABI', '2002-01-10', 'abi@gmail.com', 9876543210, 'Chennai'),

(2, 'BABU', '1999-06-15', 'babu@gmail.com', 9123456789, 'Delhi'),

(3, 'CHRISTY', '2003-08-25', 'christy@gmail.com', 9988776655, 'Mumbai');

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 'y%@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 Name, Email FROM Student\_Info WHERE YEAR(DOB) > 2000;

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 ('Chennai', 'Delhi');

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%' OR City = 'Mumbai';

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

SELECT SI.Name AS StudentName, SI.DOB AS DateOfBirth FROM Student\_Info SI;

Q6. (5 marks)

Create a new table Marks(StudentID, Subject, Marks).

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks INT

);

Insert at least 3 rows.

INSERT INTO Marks VALUES

(1, 'Maths', 85),

(2, 'Maths', 60),

(3, 'Science', 90);

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 AvgMark FROM Marks GROUP BY Subject;

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

SELECT Subject 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 CURDATE();

b) Extract month and year from a DOB column.

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

c) Convert a student's name to uppercase.

SELECT UPPER(Name) FROM Student\_Info;

d) Round off marks to 2 decimal places.

SELECT ROUND(Marks, 2) FROM Marks;

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

SELECT USER(), DATABASE();

Q8. (5 marks)

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) FROM Marks GROUP BY Subject;

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

SELECT Subject FROM Marks GROUP BY Subject HAVING AVG(Marks) > 75;

Section D: Joins and Subqueries (25 Marks)

Q9. (5 marks)

CREATE TABLE Courses (CourseID INT, CourseName VARCHAR(50));

CREATE TABLE Enrollment (StudentID INT, CourseID INT);

INSERT INTO Courses VALUES (101, 'Maths'), (102, 'Science');

INSERT INTO Enrollment VALUES (1, 101), (3, 102);

a) Inner Join to retrieve students and their courses.

SELECT SI.Name, C.CourseName

FROM Student\_Info SI

JOIN Enrollment E ON SI.StudentID = E.StudentID

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

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

SELECT SI.Name, C.CourseName

FROM Student\_Info SI

LEFT JOIN Enrollment E ON SI.StudentID = E.StudentID

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

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

SELECT SI.Name, C.CourseName

FROM Courses C

RIGHT JOIN Enrollment E ON C.CourseID = E.CourseID

RIGHT JOIN Student\_Info SI ON SI.StudentID = E.StudentID;

d) Full Outer Join equivalent using UNION.

SELECT SI.Name, C.CourseName

FROM Student\_Info SI

LEFT JOIN Enrollment E ON SI.StudentID = E.StudentID

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

UNION

SELECT SI.Name, C.CourseName

FROM Courses C

LEFT JOIN Enrollment E ON C.CourseID = E.CourseID

LEFT JOIN Student\_Info SI ON SI.StudentID = E.StudentID;

e) Cross Join to show all combinations.

SELECT SI.Name, C.CourseName

FROM Student\_Info SI

CROSS JOIN Courses C;

Q10. (5 marks)

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

SELECT StudentID FROM Marks

WHERE Subject = 'Maths' AND Marks > (SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths');

b) Students not in the Marks table.

SELECT StudentID 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 SI

WHERE EXISTS (SELECT 1 FROM Marks M WHERE M.StudentID = SI.StudentID);

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

SELECT \* FROM Marks M1

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 \* FROM Marks M2

WHERE Subject = 'English' AND Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English'

);

Q11. (5 marks)

CREATE TABLE Students2 (

StudentID INT,

Name VARCHAR(100)

);

INSERT INTO Students2 VALUES (4, 'Diana'), (3, 'Ann');

a) UNION of student names from two tables.

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Students2;

b) INTERSECT to find common students.

SELECT Name FROM Student\_Info

INTERSECT

SELECT Name FROM Students2;

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

SELECT Name FROM Student\_Info

INTERSECT

SELECT Name FROM Students2;

d) MERGE concept or simulate with UPDATE and INSERT.

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

VALUES (5, 'Emmi', '2001-11-05', 'emmi@mail.com', 9090909090, 'Bangalore')

ON DUPLICATE KEY UPDATE Name = 'emmi';

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

SELECT \* FROM Marks M1

WHERE Marks > (SELECT AVG(Marks) FROM Marks M2 WHERE M1.Subject = M2.Subject);

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.

NOSQL always stores unstructured data while sql has fixed schema to store the data . NOSQL can handle large amount of data than sql. NOSQL need not to be predefined it can be changed at any time.

Types of Nosql : Document-oriented,keyvalue stores,column-oriented, graph databases

Realtime application:-sql:- ERP(oracle) NOSQL:-social media plateforms

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.

1nf:-

Customer(CustomerID, Name) OrderDetails(OrderID, CustomerID, ProductID, Quantity, ProductName)

2nf:-

Customer(CustomerID, Name) Orders(OrderID, CustomerID) OrderItems(OrderID, ProductID, Quantity) Product(ProductID, ProductName)

BCNF:-

Customer(CustomerID, Name) Orders(OrderID, CustomerID) OrderItems(OrderID, ProductID, Quantity) Product(ProductID, ProductName)

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;

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));

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

alter table orders add constraint chk\_quantity check(quantity > 0);

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

alter table products add discount decimal(5,2); update products set discount=10.00 where productid=1; update products set discount=5.00 where productid=2;

Q4. (5 marks)

Using the above schema:

1. Insert 3 sample orders per customer.

insert into orders(orderid,customerid,productid,quantity,orderdate) values(1,1,1,2,'2025-06-10'),(2,1,2,4,'2025-06-11'),(3,1,3,1,'2025-06-12'),(4,2,1,6,'2025-06-10'),(5,2,2,3,'2025-06-11'),(6,2,3,8,'2025-06-12');

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 orders 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.

select customerid,count(\*) as order\_count 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 customerid,lower(email) as standardized\_email,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 days\_since\_order from orders;

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

select user(),current\_user(),version();

1. Use nested functions to format a customer greeting string.

select concat('hello ',upper(left(name,1)),lower(substring(name,2))) as greeting from customers;

Q7. (5 marks)

alter table products add category varchar(50);

1. Aggregate total revenue by product category.

select p.category,sum(p.price\*o.quantity) as total\_revenue from products p join orders o on p.productid=o.productid group by p.category;

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

select p.category,sum(p.price\*o.quantity) as total\_revenue from products p join orders o on p.productid=o.productid group by p.category with rollup;

1. Use HAVING clause to filter categories with revenue > 100000.

select p.category,sum(p.price\*o.quantity) as total\_revenue from products p join orders o on p.productid=o.productid group by p.category having total\_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.customerid as referred\_customer,c1.name as customer\_name,c2.name as referrer\_name from customers c1 join customers c2 on c1.referred\_by=c2.customerid;

1. Equi join across Orders and Products.

select o.orderid,o.customerid,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 customerid,name,total\_spent from (select c.customerid,c.name,sum(p.price\*o.quantity) as total\_spent,row\_number() over(order by sum(p.price\*o.quantity) 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,c.name) 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.customerid is null;

1. 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)

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

select o.customerid,o.orderid,sum(p.price\*o.quantity) as order\_total from orders o join products p on o.productid=p.productid group by o.orderid,o.customerid having order\_total > (select avg(p2.price\*o2.quantity) from orders o2 join products p2 on o2.productid=p2.productid where o2.customerid=o.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.customerid having count(distinct o.productid)>=2);

1. Use ALL to find customers who ordered more than every other customer.

select customerid from (select customerid,count(\*) as order\_count from orders group by customerid) as t where order\_count > all(select count(\*) from orders group by 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 productid,productname,total\_sold from (select p.productid,p.productname,sum(o.quantity) as total\_sold,row\_number() over(order by sum(o.quantity) desc) as rk from products p join orders o on p.productid=o.productid group by p.productid,p.productname) as ranked where rk<=3;

Q10. (5 marks)

1. Simulate INTERSECT using INNER JOIN on two customer segments.

select cn.customerid,cn.name from customer\_north cn inner join customer\_south cs on cn.customerid=cs.customerid;

1. Use EXCEPT to find products in inventory not yet ordered.

select \* from products where productid not in(select distinct productid from orders);

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

update customers set email='newemail@gmail.com' where customerid=4; insert into customers(customerid,name,email) select 4,'newcustomer','new@gmail.com' where not exists(select 1 from customers where customerid=4);

1. Use UNION to combine two regional customer tables.

Write a WITH CTE that ranks customers by total spend and filters

select customerid,name,email from customer\_north union select customerid,name,email from customer\_south;