**DAY – 1 ASSIGNMENT (09-06-2025)**

**1) Algorithm Basics –**

* **Brute Force**: Try everything — like testing every possible combination to crack a code. Easy to do, but can be slow
* **Greedy**: Always pick the best option right now — like grabbing the biggest candy each turn. Quick, but might miss the best overall choice .
* **Divide & Conquer**: Split the problem, solve parts, then stitch results back — splitting the puzzle and solving the smaller bits first then combining everything
* **Dynamic Programming**: Remember previous results, don’t re-cook the same dish twice — meal prepping onions.

| **Technique** | **How It Works** | **Real-Life Example** |
| --- | --- | --- |
| **Brute Force** | Try everything | Trying all keys to unlock a door |
| **Greedy** | Pick best now | Choosing the biggest ice-cream scoop |
| **Divide & Conquer** | Split → solve → combine | Pizza slices: cut, top, reassemble |
| **Dynamic Programming** | Split and remember results | Chop onions once and reuse in cooking |

Which on to use when:

| **Approach** | **Use It When...** | **Everyday Example** |
| --- | --- | --- |
| **Brute Force** | Few options, need guaranteed solution | Trying keys on a ring until one unlocks |
| **Greedy** | Quick approximate solution works fine | Picking the biggest snack every time |
| **Divide & Conquer** | Problem naturally breaks into parts | Sorting a big album by splitting and merging |
| **Dynamic Programming** | Many overlapping tasks need reuse | Pre-chopping for multiple recipes |

2)**Sorting Algorithms – comparison**

| **Algorithm** | **Essence** | **When It Works Best** |
| --- | --- | --- |
| **Bubble Sort** | Swap neighbors repeatedly | Lovely for tiny or already sorted lists |
| **Insertion Sort** | Insert element into sorted part — like cards | Great when nearly sorted |
| **Selection Sort** | Select smallest, place at start | Use when memory is tight |
| **Merge Sort** | Split, sort halves, merge — stable & fast | Always reliable for big lists |
| **Quick Sort** | Partition around pivot — smart slicing | Fastest on average for big arrays |

**Best Two Sorting Techniques**

**1. Quick Sort – Like sorting LEGO blocks by size**

pick a “pivot” and split the others into smaller size on the left and larger size on the right.

* Then you do the same for each side, again and again.
* You quickly end up with one big, sorted lineup of legos.
* **Why use it?**
  + It’s fast in real usage
  + Doesn’t need extra space; it sorts while you go

**2.Merge Sort – Like Organizing a Messy Bookshelf**

* You have a huge, jumbled pile of books and want them in ABC order.
* Instead of sorting the whole thing at once you:
  1. Divide the pile into two smaller stacks.
  2. Keep dividing each stack until each has only one book (which will automatically be in order).
  3. Now merge the stacks two at a time—always picking the next book in alphabetical order.
* Step by step, you build up larger sorted stacks, until the whole pile is just one neatly ordered shelf.

**3) Searching Algorithms – Quick Compare**

* **Linear Search**: Look one-by-one — like finding a toy in a messy drawer. Simple, works anywhere, but
* **Binary Search**: Works only on sorted data — pick middle, go left/right like “guess the number”. Can be faster than linear

**when to use :**

* Use linear if data is unsorted or tiny.
* Use binary if the list is sorted and large

**4) BST (Binary Search Tree)**

* A special kind of tree where each node has up to two children.
* All values in the left branch are smaller, and in the right branch, larger

**Real-World Example:**

* Think of arranging your toy cars:
  + You place the first car at the top (root).
  + When a smaller car comes, you put it to the left.
  + A bigger one goes to the right.
  + Repeat this for every new toy, guiding it left or right until it fits in an “empty spot”.

This way, if you need to find a toy of size X, you just start at the top and go left or right based on the size without searching all toys.

**Why We Also Use AVL Trees**

**What’s Wrong with BST Alone?**

* If you keep adding toys in increasing size order (e.g., 1, 2, 3, 4…), the BST becomes one long line—like a linked list.
* This loses the advantage of the tree and feels slow.

**AVL Trees = Balanced BST**

* AVL trees automatically adjust themselves to stay balanced.
* They use small re-arrangements (called rotations) to keep both sides more even
* This balance ensures that every "search route" (from top to bottom) remains shallow and quick.

**Comparison:**

| **Aspect** | **BST (Basic Tree)** | **AVL (Self-Balancing Tree)** |
| --- | --- | --- |
| **Structure** | Left < Parent < Right | Same as BST |
| **Balance** | Can become one-sided | Always stays balanced with small rotations |
| **Ease of Use** | Simple and straightforward | A little more complex (rotations needed) |
| **Order Maintenance** | Maintains sorted order | Also maintains sorted order |

**DAY-2 (10-06-2025)**

**QUIZ**

**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

Answer = d) userdb

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

Answer = c) master

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

Answer = c) Template for new databases

1. **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

Answer = b) LDF and MDF

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

Answer = c) CREATE DATABASE

1. **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

Answer = c) SalesDB is deleted permanently

1. **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

Answer = b) ALTER DATABASE old\_name MODIFY NAME = new\_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

Answer = b) DATE

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

Answer = c) CREATE TABLE

1. **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

Answer = a) ALTER TABLE table\_name ADD column\_name datatype

1. **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

Answer = c) EXEC sp\_rename 'old\_name', 'new\_name'

1. **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

Answer = c) DROP 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

Answer = a) INSERT INTO

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

Answer = b) UPDATE

1. **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

Answer = c) Removes specific rows

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

Answer = c) WHERE

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

Answer = c) DISTINCT

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

Answer = b) Finds pattern-based matches

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

Answer = c) AND / OR

1. **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

Answer = c) Filters values within a range

**DAY-3 (11-06-2025)**

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

**Section A: Managing Databases (10 mins)**

1. List all system databases in SQL Server.

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

2. List physical file paths for all databases.

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

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

CREATE DATABASE TeamDB;

4. Rename the database TeamDB to ProjectDB.

ALTER DATABASE TeamDB MODIFY NAME = ProjectDB;

5. Drop the ProjectDB database.

DROP DATABASE ProjectDB;

**Section B: Managing Tables (10 mins)**

1. Create a table Employees with the following columns:

EmpID INT (Primary Key)

Name VARCHAR(50)

Department VARCHAR(30)

JoiningDate DATE

IsActive BIT

Salary DECIMAL(10,2)

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Department VARCHAR(30),

JoiningDate DATE,

IsActive BIT,

Salary DECIMAL(10,2)

);

2. Add a column Salary (DECIMAL) to the table.

ALTER TABLE Employees ADD Salary DECIMAL(10,2);

3. Rename table Employees to TeamMembers.

EXEC sp\_rename 'Employees', 'TeamMembers';

4. Drop the table TeamMembers.

DROP TABLE TeamMembers;

**Section C: DML Operations (10 mins)**

1. Insert three rows into Employees.

INSERT INTO Employees VALUES

(1, 'Amit', 'HR', '2022-01-01', 1, 50000),

(2, 'Sneha', 'IT', '2021-06-15', 1, 75000),

(3, 'John', 'Finance', '2020-10-10', 0, 65000);

2. Update salary of 'Sneha' to 80000.

UPDATE Employees SET Salary = 80000 WHERE Name = 'Sneha';

3. Delete employee with IsActive = 0.

DELETE FROM Employees WHERE IsActive = 0;

4. Retrieve names and departments of all employees.

SELECT Name, Department FROM Employees;

5. Fetch employees from 'IT' department with salary above 70000.

SELECT \* FROM Employees WHERE Department = 'IT' AND Salary > 70000;

6. Apply filtering using LIKE, BETWEEN, and IN.

SELECT \* FROM Employees WHERE Name LIKE 'S%';

SELECT \* FROM Employees WHERE Salary BETWEEN 60000 AND 80000;

SELECT \* FROM Employees WHERE Department IN ('IT', 'Finance');

**DAY-4 (12-06-2025)**

**Medium-Level Practical SQL Questions**

1. Insert and Update with Integrity: Create a 'students' table with constraints (NOT NULL, UNIQUE). Insert 5 records. Then, update a student's marks ensuring data integrity is maintained

**QUERY**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

marks INT NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL

);

INSERT INTO students VALUES

(1, 'Alice Johnson', 85, 'alice@mail.com'),

(2, 'Bob Smith', 78, 'bob@mail.com'),

(3, 'Charlie Ray', 92, 'charlie@mail.com'),

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

(5, 'Ethan Clark', 74, 'ethan@mail.com');

**-- Update Bob’s marks to 82**

**UPDATE students**

**SET marks = 82**

**WHERE name = 'Bob Smith';**

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

**QUERY**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

full\_name VARCHAR(100)

);

INSERT INTO customers VALUES

(1, 'Alice Johnson'),

(2, 'Bob Smith'),

(3, 'Charlie Ray');

**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 first\_length,**

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

**FROM customers;**

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

**QUERY**

CREATE TABLE sales (

sale\_id INT PRIMARY KEY,

sale\_date DATE,

sale\_amount DECIMAL(10,2),

product\_category VARCHAR(50)

);

INSERT INTO sales VALUES

(1, '2024-06-01', 100.00, 'Beverages'),

(2, '2024-05-15', 50.00, 'Snacks'),

(3, '2024-04-10', 120.00, 'Beverages'),

(4, '2024-06-10', 80.00, 'Dairy');

**SELECT**

**sale\_date,**

**MONTHNAME(sale\_date) AS month,**

**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

**QUERY**

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(100),

salary DECIMAL(10,2)

);

INSERT INTO employees VALUES

(1, 'Alan', 35600),

(2, 'Betty', 42250),

(3, 'Clark', 39870);

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

**QUERY**

SELECT

NOW() AS current\_datetime,

DATABASE() AS current\_db,

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'

**QUERY**

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10,2),

supplier\_id INT

);

INSERT INTO products VALUES

(1, 'Coffee', 'Beverages', 60.00, 1),

(2, 'Chips', 'Snacks', NULL, 2),

(3, 'Milk', 'Dairy', 40.00, NULL);

SELECT

UPPER(product\_name) AS product\_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

**QUERY**

CREATE TABLE transactions (

transaction\_id INT PRIMARY KEY,

sale\_amount DECIMAL(10,2)

);

INSERT INTO transactions VALUES

(1, 100.00),

(2, 200.00),

(3, 50.00);

SELECT

SUM(sale\_amount) AS total\_sales,

AVG(sale\_amount) AS avg\_sale,

MAX(sale\_amount) AS max\_sale,

MIN(sale\_amount) AS min\_sale

FROM transactions;

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

**QUERY**

SELECT

product\_category,

SUM(sale\_amount) AS total\_sales,

COUNT(\*) AS num\_transactions

FROM sales

GROUP BY product\_category;

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

**QUERY**

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

product\_id INT,

quantity INT,

order\_amount DECIMAL(10,2)

);

INSERT INTO orders VALUES

(1, 1, 1, 2, 120.00),

(2, 2, 2, 1, 50.00);

SELECT

c.full\_name,

o.order\_amount

FROM customers c

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

**QUERY**

SELECT

p.product\_name,

o.order\_id,

o.order\_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

**QUERY**

CREATE TABLE contacts (

contact\_id INT PRIMARY KEY,

customer\_id INT,

phone VARCHAR(20)

);

INSERT INTO contacts VALUES

(1, 1, '1234567890'),

(2, 2, '9876543210');

SELECT

c.full\_name,

ct.phone

FROM contacts ct

RIGHT JOIN customers c ON ct.customer\_id = c.customer\_id;

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

**QUERY**

CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

supplier\_name VARCHAR(100)

);

INSERT INTO suppliers VALUES

(1, 'Fresh Co'),

(2, 'Snack Corp');

-- Full Outer Join using UNION

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;

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

**QUERY**

CREATE TABLE offers (

offer\_id INT PRIMARY KEY,

offer\_name VARCHAR(50)

);

INSERT INTO offers VALUES

(1, 'Summer Sale'),

(2, 'Buy 1 Get 1');

SELECT

p.product\_name,

o.offer\_name

FROM products p

CROSS JOIN offers o;

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

**QUERY**

SELECT

p.category,

SUM(o.quantity) AS total\_quantity,

AVG(p.price) AS avg\_price

FROM orders o

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

**QUERY**

CREATE TABLE marks (

student\_id INT,

subject VARCHAR(50),

score INT

);

INSERT INTO marks VALUES

(1, 'Math', 90),

(1, 'Science', 80),

(2, 'Math', 70),

(2, 'Science', 75),

(3, 'Math', 95),

(3, 'Science', 89);

SELECT

s.name AS student\_name,

AVG(m.score) AS avg\_marks

FROM students s

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

GROUP BY s.name

HAVING AVG(m.score) > 75;

**DAY -5 (13-06-2025)**

**Querying Data by Using Subqueries - Examples**

QUESTIONS

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"

ANSWER QUERIES:

-- Create and use the database

CREATE DATABASE IF NOT EXISTS CompanyDB;

USE CompanyDB;

-- Drop old tables

DROP TABLE IF EXISTS employees, departments, job\_history, new\_employees, locations, countries, suppliers, products, categories;

-- Create schema

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50),

location\_id INT

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

salary INT,

department\_id INT,

job\_id VARCHAR(10),

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

CREATE TABLE job\_history (

employee\_id INT,

department\_id INT,

job\_id VARCHAR(10)

);

CREATE TABLE locations (

location\_id INT PRIMARY KEY,

city VARCHAR(50)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(50),

category\_id INT,

supplier\_id INT

);

CREATE TABLE categories (

category\_id INT PRIMARY KEY,

category\_name VARCHAR(50)

);

CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

country\_id INT

);

CREATE TABLE countries (

country\_id INT PRIMARY KEY,

country\_name VARCHAR(50)

);

CREATE TABLE new\_employees (

employee\_id INT,

name VARCHAR(50),

salary INT,

department\_id INT

);

-- Insert sample data

INSERT INTO departments VALUES

(1, 'HR', 100),

(2, 'Engineering', 101),

(3, 'Sales', 102);

INSERT INTO locations VALUES

(100, 'New York'),

(101, 'Tokyo'),

(102, 'London');

INSERT INTO employees VALUES

(101, 'Alice', 80000, 2, 'ENG'),

(102, 'Bob', 50000, 2, 'ENG'),

(103, 'Charlie', 90000, 1, 'HR'),

(104, 'David', 60000, 3, 'SAL');

INSERT INTO job\_history VALUES

(101, 2, 'ENG'),

(103, 1, 'HR');

INSERT INTO categories VALUES

(1, 'Electronics'),

(2, 'Furniture');

INSERT INTO countries VALUES

(1, 'Japan'),

(2, 'Germany');

INSERT INTO suppliers VALUES

(1, 1),

(2, 2);

INSERT INTO products VALUES

(1, 'Laptop', 1, 1),

(2, 'TV', 1, 2);

INSERT INTO new\_employees VALUES

(101, 'Alice Johnson', 85000, 2),

(105, 'Eva', 70000, 3);

-- --------------------------------------------------

-- 1. Basic Subquery: Get employees in HR department

-- --------------------------------------------------

SELECT name

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE department\_name = 'HR'

);

-- --------------------------------------------------

-- 2. EXISTS: Get employees who have job history

-- --------------------------------------------------

SELECT name

FROM employees e

WHERE EXISTS (

SELECT 1

FROM job\_history j

WHERE j.employee\_id = e.employee\_id

);

-- --------------------------------------------------

-- 3. ANY: Get employees earning more than someone in HR

-- --------------------------------------------------

SELECT name

FROM employees

WHERE salary > ANY (

SELECT salary

FROM employees

WHERE department\_id = 1

);

-- --------------------------------------------------

-- 4. ALL: Get employees earning more than everyone in Sales

-- --------------------------------------------------

SELECT name

FROM employees

WHERE salary > ALL (

SELECT salary

FROM employees

WHERE department\_id = 3

);

-- --------------------------------------------------

-- 5. Nested Subquery: Employees in NY-based department

-- --------------------------------------------------

SELECT name

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE location\_id = (

SELECT location\_id

FROM locations

WHERE city = 'New York'

)

);

-- --------------------------------------------------

-- 6. Correlated Subquery: Earn more than dept avg

-- --------------------------------------------------

SELECT name

FROM employees e

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id = e.department\_id

);

-- --------------------------------------------------

-- 7. UNION: Combine employees from dept 1 and 2

-- --------------------------------------------------

(SELECT name FROM employees WHERE department\_id = 1)

UNION

(SELECT name FROM employees WHERE department\_id = 2);

-- --------------------------------------------------

-- 8. INTERSECT (Conceptual only; not in MySQL)

-- Equivalent: use INNER JOIN

-- --------------------------------------------------

SELECT e.employee\_id

FROM employees e

INNER JOIN job\_history j ON e.employee\_id = j.employee\_id

WHERE e.department\_id = 1;

-- --------------------------------------------------

-- 9. EXCEPT (MySQL Alternative using NOT IN)

-- Employees not in job history

-- --------------------------------------------------

SELECT employee\_id

FROM employees

WHERE employee\_id NOT IN (

SELECT employee\_id

FROM job\_history

);

-- --------------------------------------------------

-- 10. MERGE (Simulated using INSERT...ON DUPLICATE KEY)

-- Update existing or insert new employees

-- --------------------------------------------------

-- Simulate MERGE behavior

INSERT INTO employees (employee\_id, name, salary, department\_id, job\_id)

VALUES

(101, 'Alice Johnson', 85000, 2, 'ENG'),

(105, 'Eva', 70000, 3, 'SAL') AS new\_vals

ON DUPLICATE KEY UPDATE

name = new\_vals.name,

salary = new\_vals.salary,

department\_id = new\_vals.department\_id;

**Question Paper – 1**

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

Q1. (4 marks)

Explain with examples the scenarios where NoSQL is preferred over SQL. Discuss types of NoSQL databases and

suggest a real-time application for each.

ANSWER:

**When to Prefer NoSQL Over SQL:**

| **Scenario** | **Why NoSQL is Better** | **Example** |
| --- | --- | --- |
| Schema-less data | NoSQL doesn't require a predefined schema. | Social apps storing posts with different structures. |
| Massive write/read scalability | Horizontal scaling on commodity servers. | Logging system like ELK stack (Elasticsearch). |
| Rapid development | Fields can be added without altering schema. | Startup changing product details frequently. |
| Complex relationships | Graph-based queries work better in NoSQL. | Fraud detection using relationship networks. |

**Types of NoSQL and Real-Time Use Cases:**

| **Type** | **Description** | **Database** | **Example Use Case** |
| --- | --- | --- | --- |
| Document Store | Stores JSON-like flexible documents | MongoDB | E-commerce product catalogs |
| Key-Value Store | Simple key-value pairs | Redis | Real-time leaderboard or session cache |
| Column Store | Column-oriented, fast for analytics | Apache Cassandra | IoT time-series data |
| Graph Store | Nodes & relationships | Neo4j | Social media friend recommendations |

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.

ANSWER:

Given Unnormalized Table:

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

**1NF – Remove repeating groups (flatten nested structure)**

Separate Orders into its own table.

-- Step 1: Create Customer Table (Atomic fields only)

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100)

);

-- Step 2: Create Products Table

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100)

);

-- Step 3: Create Orders Table

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT CHECK (Quantity > 0),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

**2NF – Eliminate Partial Dependencies**

All partial dependencies are already removed because:

* CustomerID → Name is in Customers
* ProductID → ProductName is in Products
* OrderID → CustomerID, ProductID, Quantity is in Orders

So this schema is already in **2NF**.

**3NF – Eliminate Transitive Dependencies**

Already achieved:

* No non-prime attribute depends transitively on another non-key
* All non-key attributes depend **only** on the primary key

**BCNF – Ensure every determinant is a candidate key**

All functional dependencies are satisfied:

| **Table** | **Functional Dependencies** | **BCNF Check** |
| --- | --- | --- |
| Customers | CustomerID → Name | Candidate key |
| Products | ProductID → ProductName | Candidate key |
| Orders | OrderID → CustomerID, ProductID, Quantity | Candidate key |

**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 'Discount' column and update some values.

ANSWER:

-- a) Create Database and Schema

CREATE DATABASE RetailDB;

USE RetailDB;

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100),

ReferredBy INT,

FOREIGN KEY (ReferredBy) REFERENCES Customers(CustomerID)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

Name VARCHAR(100),

Category VARCHAR(100),

Price DECIMAL(10, 2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

OrderDate DATE,

Quantity INT CHECK (Quantity > 0),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

-- b) Already included CHECK constraint above in Orders table

-- c) Alter Products to add Discount 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 = 5.50 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.

ANSWER:

-- a) Insert 3 sample orders per customer

INSERT INTO Orders VALUES

(1, 1, 1, '2025-06-01', 3),

(2, 1, 2, '2025-06-02', 6),

(3, 1, 3, '2025-06-03', 1),

(4, 2, 1, '2025-06-01', 2),

(5, 2, 2, '2025-06-02', 7),

(6, 2, 3, '2025-06-03', 4);

-- b) Increase price by 10% for products sold in quantity > 5

UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID FROM Orders

WHERE Quantity > 5

);

-- c) Delete orders for products never sold (if any exist)

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.

ANSWER:

-- a) Customers who ordered > 3 different products

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

-- b) Products never ordered

SELECT \* FROM Products

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

-- c) Count of orders in last 30 days (from 2025-05-14 to 2025-06-13)

SELECT CustomerID, COUNT(\*) AS OrderCount

FROM Orders

WHERE OrderDate BETWEEN '2025-05-14' AND '2025-06-13'

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 current user and host.

d) Use nested functions to format a customer greeting string.

ANSWER:

-- a) Email standardization

SELECT Email,

LOWER(Email) AS LowerEmail,

SUBSTRING\_INDEX(Email, '@', 1) AS Username,

SUBSTRING\_INDEX(Email, '@', -1) AS Domain

FROM Customers;

-- b) Days between order date and today

SELECT OrderID, DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

-- c) System user & host

SELECT CURRENT\_USER(), @@hostname;

-- d) Greeting message

SELECT CONCAT('Hello ', UPPER(Name), '! Your email is ', Email) 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 clause to filter categories with revenue > 100000.

ANSWER:

-- a) Revenue by product category

SELECT Category, SUM(Price \* Quantity) AS Revenue

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY Category;

-- b) ROLLUP

SELECT Category, SUM(Price \* Quantity) AS Revenue

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY Category WITH ROLLUP;

-- c) HAVING revenue > 100000

SELECT Category, SUM(Price \* Quantity) AS Revenue

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY Category

HAVING SUM(Price \* Quantity) > 100000;

**Section D: Complex Joins, Subqueries, and Set Ops (25 Marks)**

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.

ANSWER:

-- a) Self-join: customers referred by others

SELECT c1.Name AS Customer, c2.Name AS ReferredBy

FROM Customers c1

JOIN Customers c2 ON c1.ReferredBy = c2.CustomerID;

-- b) Equi-join: Orders and Products

SELECT o.OrderID, p.Name, p.Price

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID;

-- c) Top 3 spenders (using window function)

SELECT CustomerID, TotalSpend

FROM (

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

) ranked

WHERE rnk <= 3;

-- d) Inactive customers (never ordered)

SELECT \* FROM Customers c

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

WHERE o.OrderID IS NULL;

-- e) Product bundles (cross join)

SELECT a.Name AS ProductA, b.Name AS ProductB

FROM Products a

CROSS JOIN Products b

WHERE a.ProductID < b.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.

ANSWER:

-- a) Customers with order > their own average

SELECT DISTINCT o1.CustomerID

FROM Orders o1

WHERE (o1.Quantity \* (SELECT Price FROM Products WHERE ProductID = o1.ProductID)) > (

SELECT AVG(o2.Quantity \* (SELECT Price FROM Products WHERE ProductID = o2.ProductID))

FROM Orders o2

WHERE o1.CustomerID = o2.CustomerID

);

-- b) Customers who bought at least 2 different products

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) >= 2;

-- c) Customers who ordered more than every other customer

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(OrderID) > ALL (

SELECT COUNT(OrderID)

FROM Orders

GROUP BY CustomerID

);

-- d) Products costlier than any in Electronics

SELECT \* FROM Products

WHERE Price > ANY (

SELECT Price FROM Products WHERE Category = 'Electronics'

);

-- e) Top 3 best-selling products (by total quantity)

SELECT ProductID, TotalQty

FROM (

SELECT ProductID, SUM(Quantity) AS TotalQty,

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.

SQL Practical Question Paper

Duration: 2 Hours | Total Marks: 60

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

ANSWER:

-- a) Simulate INTERSECT using INNER JOIN

SELECT c1.CustomerID, c1.Name

FROM SegmentA c1

JOIN SegmentB c2 ON c1.CustomerID = c2.CustomerID;

-- b) EXCEPT: products not yet ordered

SELECT ProductID FROM Products

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID FROM Orders

);

-- c) MERGE: if exists, update; else insert

INSERT INTO Customers (CustomerID, Name, Email)

VALUES (10, 'Sam', 'sam@mail.com')

ON DUPLICATE KEY UPDATE Name = 'Sam', Email = 'sam@mail.com';

-- d) UNION of two customer regions

SELECT \* FROM EastCustomers

UNION

SELECT \* FROM WestCustomers;

-- e) CTE: rank and top 5 spenders

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;

**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

**ANSWER:**

| **Criteria** | **SQL** | **NoSQL** |
| --- | --- | --- |
| **Definition** | Relational DB (Structured) | Non-relational DB (Flexible) |
| **Advantages** | 1. ACID compliance2. Easy joins and structured queries | 1. Scalable horizontally2. Supports unstructured data |
| **Disadvantages** | 1. Less scalable for large unstructured data2. Schema rigidity | 1. No standard query language2. Eventual consistency |
| **Examples** | MySQL, PostgreSQL | MongoDB, Cassandra |

Q2. (2 marks)

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

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

**ANSWER:**

Unnormalized:

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

**1NF**: Remove repeating groups

Student(StudentID, Name)

Course(CourseID, CourseName, InstructorName, InstructorPhone)

Enrollment(StudentID, CourseID)

**2NF**: Remove partial dependencies  
Separate Instructor details:

Instructor(InstructorID, Name, Phone)

Course(CourseID, CourseName, InstructorID)

**3NF**: Remove transitive dependencies  
Final tables:

* Student(StudentID, Name)
* Instructor(InstructorID, Name, Phone)
* Course(CourseID, CourseName, InstructorID)
* Enrollment(StudentID, CourseID)

-- Step 1: Create Student Table

CREATE TABLE Student (

StudentID INT PRIMARY KEY,

Name VARCHAR(100)

);

-- Step 2: Create Instructor Table (to store instructor details separately)

CREATE TABLE Instructor (

InstructorID INT PRIMARY KEY,

InstructorName VARCHAR(100),

InstructorPhone VARCHAR(15)

);

-- Step 3: Create Course Table (with reference to Instructor)

CREATE TABLE Course (

CourseID INT PRIMARY KEY,

CourseName VARCHAR(100),

InstructorID INT,

FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID)

);

-- Step 4: Create Enrollment Table (relationship between students and courses)

CREATE TABLE Enrollment (

StudentID INT,

CourseID INT,

PRIMARY KEY (StudentID, CourseID),

FOREIGN KEY (StudentID) REFERENCES Student(StudentID),

FOREIGN KEY (CourseID) REFERENCES Course(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.

**ANSWER:**

-- a

CREATE DATABASE StudentDB;

-- b

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100)

);

-- c

ALTER TABLE Students RENAME TO Student\_Info;

-- d

ALTER TABLE Student\_Info ADD PhoneNumber BIGINT;

-- e

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.

**ANSWER:**

-- a

INSERT INTO Student\_Info VALUES (1, 'Alice', '2002-05-20', 'alice@yahoo.com', 9123456789);

INSERT INTO Student\_Info VALUES (2, 'Bob', '1999-12-10', 'bob@gmail.com', 9876543210);

INSERT INTO Student\_Info VALUES (3, 'Carol', '2001-03-15', 'carol@outlook.com', 9988776655);

-- b

UPDATE Student\_Info SET PhoneNumber = 9000000000 WHERE StudentID = 2;

-- c

DELETE FROM Student\_Info WHERE Email LIKE '%@gmail.com';

-- d

SELECT Name, Email FROM Student\_Info WHERE YEAR(DOB) > 2000;

-- e

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

Q5. (5 marks)

a) Retrieve students with names starting with 'A'.

b) Retrieve students with phone number between 9000000000 and 9999999999.

c) Retrieve students using 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.

**ANSWER:**

-- a

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

-- b

SELECT \* FROM Student\_Info WHERE PhoneNumber BETWEEN 9000000000 AND 9999999999;

-- c

SELECT \* FROM Student\_Info WHERE City IN ('Chennai', 'Mumbai', 'Delhi');

-- d

SELECT \* FROM Student\_Info

WHERE (YEAR(CURDATE()) - YEAR(DOB)) < 25

AND (Email LIKE '%@gmail.com' OR Email LIKE '%@yahoo.com');

-- e

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

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.

**ANSWER:**

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks INT

);

INSERT INTO Marks VALUES (1, 'Maths', 80), (2, 'Science', 60), (3, 'English', 75);

-- a

SELECT StudentID, Subject FROM Marks WHERE Marks > 70;

-- b

SELECT Subject, AVG(Marks) AS AvgMarks FROM Marks GROUP BY Subject;

-- c

SELECT Subject, AVG(Marks) AS AvgMarks FROM Marks

GROUP BY Subject HAVING AVG(Marks) BETWEEN 60 AND 90;

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

Q7. (5 marks)

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

**ANSWER:**

-- a

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

-- b

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

-- c

SELECT UPPER(Name) FROM Student\_Info;

-- d

SELECT ROUND(Marks, 2) FROM Marks;

-- e

SELECT DATABASE() AS CurrentDB, USER() AS CurrentUser;

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.

**ANSWER:**

-- a

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

-- b

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

-- c

SELECT Subject, AVG(Marks) AS AvgMark 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.

**ANSWER:**

Lets Assume:

* Students(StudentID, Name)
* Enrollments(StudentID, CourseID)
* Courses(CourseID, CourseName)

-- a

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

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

SELECT s.Name, c.CourseName

FROM Courses c

RIGHT JOIN Enrollments e ON c.CourseID = e.CourseID

RIGHT JOIN Student\_Info s ON s.StudentID = e.StudentID;

-- d

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 Courses c

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

LEFT JOIN Student\_Info s ON e.StudentID = s.StudentID;

-- e

SELECT \* FROM Student\_Info CROSS JOIN Courses;

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'

**ANSWER:**

-- a

SELECT \* FROM Marks

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

-- b

SELECT \* FROM Student\_Info

WHERE StudentID NOT IN (SELECT DISTINCT StudentID FROM Marks);

-- c

SELECT \* FROM Student\_Info s

WHERE EXISTS (SELECT 1 FROM Marks m WHERE m.StudentID = s.StudentID);

-- d

SELECT \* FROM Marks

WHERE Subject = 'Science' AND Marks > ALL (

SELECT Marks FROM Marks WHERE Subject = 'Science'

);

-- e

SELECT \* FROM Marks

WHERE Subject = 'English' AND Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English'

);

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

**ANSWER:**

-- a

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Alumni;

-- b

SELECT Name FROM Student\_Info

INTERSECT

SELECT Name FROM Alumni;

-- c

SELECT Name FROM Student\_Info

EXCEPT

SELECT Name FROM Marks;

-- d (simulate MERGE)

UPDATE Student\_Info SET Email = 'updated@mail.com' WHERE StudentID = 1;

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

SELECT 4, 'NewStudent', '2004-06-12', 'new@mail.com', 9012345678

WHERE NOT EXISTS (SELECT 1 FROM Student\_Info WHERE StudentID = 4);

-- e

SELECT \* FROM Marks m1

WHERE Marks > (

SELECT AVG(Marks) FROM Marks m2

WHERE m1.Subject = m2.Subject

);