DAY 1 – 09.06.2025

ASSIGNMENT 1

1) Explain all the algorithm basics in brief and compare

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

3) compare searching algorithm

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

1) algorithm basics in brief and compare

a) Brute Force

- Tries every possible solution until the correct one is found.
- Simple to implement, but inefficient for large problems.
- Example: Trying every combination on a 3-digit lock (000 to 999).
- Use when :Problem size is small.
- Drawback: Slow and resource-heavy for large datasets.

a) Heuristic

- Makes educated guesses to find a good enough solution quickly.
- Doesn’t guarantee the optimal or correct answer.
- Example: Searching for a book in the “Science” section without using a catalog.
- Use when: Approximate answers are acceptable; exact solution is too complex.
- Drawback: May miss the correct or best solution.

b) Greedy Approach

- Chooses the best immediate option at each step.
- Doesn’t look ahead or consider the overall situation.
- Example: Making ₹43 using the largest coins first (₹20, ₹10, ₹10, ₹2, ₹1).
- Use when: Local choices lead to the global optimal solution.
- Drawback: May fail if local optimum isn't part of global optimum.

c) Divide and Conquer

- Divides the problem into smaller parts, solves them individually, and combines the results.
- Efficient for recursive solutions and large datasets.
- Example: Sorting papers by splitting them into halves, sorting each half, and merging.
- Used in: Merge Sort, Quick Sort, Binary Search.
- Benefit: Fast and efficient for large problems.

d) Dynamic Programming (DP)

- Breaks a problem into overlapping subproblems.
- Stores results of subproblems (memoization/tabulation) to avoid recomputation.
- Example: Calculating ways to climb stairs using results from previous steps.
- Use when: Problems have overlapping subproblems and optimal substructure.
- Benefit: More efficient than brute force.
- Drawback: Can be complex to design and implement.

Approach	pros	cons
Brute Force	Simple to implement Fast	Inefficient for large inputs
Heuristic	saves time in complex times	May not find the correct or best solution
Greedy Approach	Fast and easy to implement	May fail if local choices don't lead to optimal
Divide and Conquer	Efficient for large datasets	Can be complex to manage recursion
Dynamic Programming (DP)	Avoids repeated work, very efficient	Requires extra memory, harder to implement

2) Compare all sorting algorithm

1.Bubble Sort

Repeatedly compares and swaps adjacent items until everything is in order.
Real-Life Example: Arranging books by height by comparing pairs and swapping until all are correctly placed.

2.Insertion Sort

Builds the sorted list one element at a time by placing each item in its correct position.
Real-Life Example: Sorting playing cards in your hand by inserting each card into the right spot.

3.Selection Sort

Selects the smallest (or largest) item and places it in its final position, one by one.
Real-Life Example: Picking the smallest item from a shelf and placing it first, then repeating for the rest.

4. Merge Sort

Splits the dataset into halves, sorts each half, then merges them together.
Real-Life Example: Two people sort different halves of a deck of cards and then combine them in order.

5. Quick Sort

Chooses a pivot, places smaller elements on one side and larger on the other, and recursively sorts them.
Real-Life Example: Choosing a reference book height, separating shorter and taller books, and sorting the groups.

Algorithm	Space Complexity	Stability
Insertion Sort	O(1)	Yes
Selection Sort	O(1)	No
Merge Sort	O(n)	Yes
Quick Sort	O(log n)	No

3) compare searching algorithm

Algorithm	**Idea**	**Space**
Linear Search	Checks each element one by one	O(1)
Binary Search	Repeatedly divide sorted list into halves	O(1)

4) Use of BST and difference between BST and AVL tree

Binary Search Tree (BST) is used to store data in a structured and sorted way, enabling fast.

- Search
- Insertion
- Deletion

Feature	BST	AVL Tree
Structure	Simple binary tree with left < root < right rule	Self-balancing BST that adjusts after every insertion/deletion
Balance Factor	Not maintained	Maintains balance factor (−1, 0, or +1) for every node
Speed	Slightly faster insertion/deletion (no balancing)	Slightly slower (due to balancing overhead)

DAY 2 – 10.06.2025

ASSIGNMENT – 2

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

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

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

10. **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

11. **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

12. **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

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

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

15. 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

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

17. Which keyword ensures no duplicate records are returned?
a) UNIQUE
b) NO_REPEAT
c) DISTINCT
d) ONLY

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

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

20. 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

DAY 3 – 11.06.2025

ASSIGNMENT – 3

SECTION A

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, 'Ajith', 'HR', '2022-10-05', 1, 65000),
(2, 'Raji', 'IT', '2021-08-11', 1, 78000),
(3, 'Nithya', 'Finance', '2020-01-11', 0, 69000);
```

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');

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.

```
CREATE TABLE students (

   StudentID INT PRIMARY KEY,

   Name VARCHAR(100) NOT NULL,

   Email VARCHAR(100) UNIQUE NOT NULL,

   Marks INT CHECK (Marks BETWEEN 0 AND 100)

);

INSERT INTO students (StudentID, Name, Email, Marks) VALUES
(1, 'Aarav', 'aarav@example.com', 85),
(2, 'Divya', 'divya@example.com', 78),
(3, 'Karthik', 'karthik@example.com', 92),
(4, 'Meera', 'meera@example.com', 69),
(5, 'Rohan', 'rohan@example.com', 88);

UPDATE students
SET Marks = 82
WHERE Name = 'Divya';

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

```
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_name_length,
  LENGTH(SUBSTRING_INDEX(full_name, ' ', -1)) AS last_name_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

```
CREATE TABLE sales (
    sale_id INT AUTO_INCREMENT PRIMARY KEY,
    sale_date DATE NOT NULL
);
INSERT INTO sales (sale_date) VALUES
('2025-06-10'),
('2025-05-20'),
('2025-04-01');
SELECT
    sale_id,
    sale_date,
    MONTHNAME(sale_date) AS month_name,
    YEAR(sale_date) AS year,
```

```sql
    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

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    salary DECIMAL(10,2) NOT NULL
);
INSERT INTO employees (employee_id, name, salary) VALUES
(101, 'Priya', 4600.00),
(102, 'Arjun', 5250.00),
(103, 'Sneha', 4842.00),
(104, 'Ravi', 3999.99),
(105, 'Kiran', 6585.50);
SELECT
  employee_id,
  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

```sql
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'

```sql
CREATE TABLE products (
   product_id INT AUTO_INCREMENT PRIMARY KEY,
   product_name VARCHAR(100) NOT NULL,
   price DECIMAL(10, 2) DEFAULT NULL
);

INSERT INTO products (product_name, price) VALUES
('Laptop', 80000.00),
('Tablet', NULL),
('Smartphone', 50000.00),
('Headphones', NULL);

SELECT
   UPPER(product_name) AS product_name_upper,
   IFNULL(CAST(price AS CHAR), '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

```
CREATE TABLE transactions (
  transaction_id INT PRIMARY KEY,
  customer_name VARCHAR(100),
  sale_amount DECIMAL(10,2),
  transaction_date DATE
);

INSERT INTO transactions (transaction_id, customer_name, sale_amount, transaction_date) VALUES
(1, 'Priya', 1200.50, '2025-06-01'),
(2, 'Arjun', 2300.00, '2025-06-03'),
(3, 'Sneha', 850.75, '2025-06-05'),
(4, 'Ravi', 1799.99, '2025-06-07'),
(5, 'Kiran', 2540.20, '2025-06-10');

SELECT
  SUM(sale_amount) AS total_sales,
  AVG(sale_amount) AS average_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

```
CREATE TABLE sales (
  sale_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  category VARCHAR(50),
  sale_amount DECIMAL(10,2),
  sale_date DATE
);
INSERT INTO sales (sale_id, product_name, category, sale_amount, sale_date)
VALUES
(1, 'Laptop', 'Electronics', 55000.00, '2025-06-01'),
(2, 'Mobile', 'Electronics', 20000.00, '2025-06-03'),
(3, 'Table', 'Furniture', 8000.00, '2025-06-05'),
(4, 'Chair', 'Furniture', 3500.00, '2025-06-06'),
(5, 'Shoes', 'Fashion', 2500.00, '2025-06-10'),
(6, 'T-shirt', 'Fashion', 1200.00, '2025-06-11');
SELECT
 category,
 COUNT(*) AS number_of_transactions,
 SUM(sale_amount) AS total_sales
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

```
CREATE TABLE customers (
  customer_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100)
);

CREATE TABLE orders (
  order_id INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT,
  order_amount DECIMAL(10, 2),
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO customers (name) VALUES
('Alice'), ('Bob'), ('Charlie');

INSERT INTO orders (customer_id, order_amount) VALUES
(1, 500.00),
(2, 1200.75),
(1, 300.00);  -- Charlie didn't order
```

10. Left Join for Products with or without Orders:

Show all products with:

- Their order details (if available)
- Use LEFT JOIN

```
CREATE TABLE products (
   product_id INT PRIMARY KEY,
   product_name VARCHAR(100)
);

CREATE TABLE orders (
   order_id INT PRIMARY KEY,
   product_id INT,
   quantity INT,
   FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO products VALUES
(1, 'Laptop'), (2, 'Phone'), (3, 'Tablet');

INSERT INTO orders VALUES
(101, 1, 2), (102, 2, 1);  -- No order for Tablet

SELECT
   p.product_name,
   o.order_id,
   o.quantity
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

```sql
CREATE TABLE customers (
   customer_id INT PRIMARY KEY,
   name VARCHAR(100)
);

CREATE TABLE contacts (
   contact_id INT PRIMARY KEY,
   customer_id INT,
   email VARCHAR(100),
   FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO customers VALUES
(1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

INSERT INTO contacts VALUES
(201, 1, 'alice@mail.com'), (202, 2, 'bob@mail.com'); -- Charlie has no contact

SELECT
   c.customer_id,
   c.name,
   ct.email
FROM contacts ct
RIGHT JOIN customers c ON c.customer_id = ct.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

```
CREATE TABLE suppliers (
    supplier_id INT PRIMARY KEY,
    supplier_name VARCHAR(100)
);
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    supplier_id INT
);
INSERT INTO suppliers (supplier_id, supplier_name) VALUES
(1, 'Alpha Supplies'),
(2, 'Beta Traders'),
(3, 'Gamma Goods');
INSERT INTO products (product_id, product_name, supplier_id) VALUES
(101, 'Pen', 1),
(102, 'Notebook', 1),
(103, 'Marker', 2),
(104, 'Eraser', NULL),
(105, 'Stapler', 4);
SELECT
    s.supplier_id,
    s.supplier_name,
    p.product_id,
    p.product_name
```

```
FROM
    suppliers s
FULL OUTER JOIN
    products p ON s.supplier_id = p.supplier_id;
SELECT
    s.supplier_id,
    s.supplier_name,
    p.product_id,
    p.product_name
FROM
    suppliers s
LEFT JOIN
    products p ON s.supplier_id = p.supplier_id
UNION
SELECT
    s.supplier_id,
    s.supplier_name,
    p.product_id,
    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

```
CREATE TABLE offers (
   offer_id INT PRIMARY KEY,
   offer_name VARCHAR(50)
);
INSERT INTO offers VALUES
(1, '10% Off'), (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

```
CREATE TABLE products (
   product_id INT PRIMARY KEY,
   product_name VARCHAR(100),
   category VARCHAR(50),
   price DECIMAL(10,2)
);

CREATE TABLE orders (
   order_id INT PRIMARY KEY,
   product_id INT,
   quantity INT,
   FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```sql
INSERT INTO products (product_id, product_name, category, price) VALUES
(1, 'Pen', 'Stationery', 10.00),
(2, 'Notebook', 'Stationery', 30.00),
(3, 'Stapler', 'Office Supplies', 50.00),
(4, 'Mouse', 'Electronics', 500.00);

INSERT INTO orders (order_id, product_id, quantity) VALUES
(101, 1, 5),
(102, 1, 3),
(103, 2, 2),
(104, 3, 4),
(105, 4, 1);
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;
```

Querying Data by Using Subqueries – Examples

Querying Data by Using Subqueries

SELECT Name FROM Employees WHERE Sal < (SELECT AVG(Sal) FROM Employees);

Querying Data by Using Subqueries Using the EXISTS

SELECT Name FROM Employees e WHERE EXISTS (SELECT 1 FROM Employees WHERE Department = 'IT' AND e.Department = Department);

Querying Data by Using Subqueries using ANY

SELECT Name FROM Employees WHERE Sal> ANY (SELECT Sal FROM Employees WHERE Department = 'HR');

Querying Data by Using Subqueries using ALL Keywords

SELECT Name FROM Employees WHERE Sal > ALL (SELECT Salary FROM Employees WHERE Department = 'HR');

Querying Data by Using Subqueries using Nested Subqueries

SELECT Name FROM Employees WHERE Sal = (SELECT MAX(Sal) FROM Employees WHERE Department = (SELECT Department FROM Employees WHERE Name = 'Charlie'));

Querying Data by Using Subqueries Using Correlated Subqueries

SELECT Name FROM Employees e1 WHERE Sal > (SELECT AVG(Salary) FROM Employees e2 WHERE e1.Department = e2.Department);

Querying Data by Using Subqueries Using UNION

SELECT Name FROM Employees WHERE Department = 'HR' UNION SELECT Name FROM Employees WHERE Sal> 60000;

Querying Data by Using Subqueries using INTERSECT

SELECT Name FROM Employees WHERE Department = 'IT' INTERSECT SELECT Name FROM Employees WHERE Sal > 50000;

Querying Data by Using Subqueries using EXCEPT

SELECT Name FROM Employees WHERE Department = 'IT' EXCEPT SELECT Name FROM Employees WHERE Sal> 70000;

SQL Practical Question Paper 1

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.

SQL	NOSQL
Tables contains rows and columns.	Document-based, key-value, column-family, or graph-based.
It is Vertically scalable (upgrading hardware)	It is Horizontally scalable (adding more servers)
Efficient for complex queries and transactions	Better for large-scale data and fast read/write operations

SQL- ADV

- Ideal for structured data and enforcing relationships using foreign keys.
- Ensures reliable transactions (like a transfer of money).

DIS.ADV:

- Vertical scaling (adding more CPU/RAM) is expensive and limited.
- Requires predefined schemas. Changing structure later is hard.

NOSQL-ADV:

- Easily handles huge data volumes across multiple servers.
- No fixed schema—ideal for unstructured or semi-structured data.

DIS.ADV:

- No standard query language across NoSQL databases.
- Sacrifices strong consistency for performance.

Q2. (2 marks)

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

1NF: Remove repeating groups

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

2NF: 1. Student Table**:**

Student(StudentID, Name)

2. Course Table:

Course(CourseID, CourseName, InstructorName, InstructorPhone)

3. Enrollment Table:

Enrollment(StudentID, CourseID)

3NF: Remove transitive dependencies.

1. Instructor Table:

Instructor(InstructorName, InstructorPhone)

2. Updated Course Table:

Course(CourseID, CourseName, InstructorName)

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.

Code:

```
create database StudentDB;
use StudentDB;

create table Students (
   StudentID int primary key,
   Name varchar(100),
   DOB date,
   Email varchar(100)
);
rename table Students to Student_Info;
alter table Student_Info add PhoneNum varchar(15);
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.

Code:

```
a) INSERT INTO Student_Info (StudentID, Name, DOB, Email, PhoneNum)
VALUES
(1, 'Durga', '2001-06-20', 'Dur123@gmail.com', '9876543210'),
(2, 'Pooja', '1999-09-15', 'pooja123@yahoo.com', '9867543210'),
(3, 'Hema', '2002-10-03', 'hema123@outlook.com', '9856543210');

b) UPDATE Student_Info
   SET PhoneNum = '9123456789'
   WHERE StudentID = 1;

c) DELETE FROM Student_Info
   WHERE Email LIKE '%@gmail.com'
   LIMIT 1;

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.

Code:

```
a)SELECT * FROM Student_Info
  WHERE Name LIKE 'A%';

b) SELECT * FROM Student_Info
  WHERE PhoneNum BETWEEN '9000000000' AND '9999999999';

c) SELECT * FROM Student_Info
  WHERE City IN ('Chennai', 'Bangalore', 'Hyderabad');
```

d) SELECT * FROM Student_Info
WHERE (YEAR(CURDATE()) - YEAR(DOB)) > 21
AND (Email LIKE '%@gmail.com' OR Email LIKE '%@yahoo.com');

e) 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.

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.

Code:

```
CREATE TABLE Marks (
   StudentID INT,
   Subject VARCHAR(50),
   Marks INT
);

INSERT INTO Marks (StudentID, Subject, Marks)
VALUES
(1, 'Math', 85),
(2, 'Science', 65),
(1, 'English', 78);
```

a) SELECT StudentID, Subject FROM Marks
WHERE Marks > 70;

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

c) SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks
GROUP BY Subject
HAVING AVG(Marks) BETWEEN 60 AND 90;

Section C: Functions & Grouping (10 Marks)

Q7. (5 marks)

a) Get the current date and format it as "YYYY-MM-DD".
b) Extract month and year from a DOB column.
c) Convert a student's name to uppercase.
d) Round off marks to 2 decimal places.
e) Use system function to return user name or current database.

Code:

```
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) AS UpperCaseName FROM Student_Info;

d) SELECT ROUND(Marks, 2) AS RoundedMarks FROM Marks;

e) SELECT USER() AS CurrentUser;
   SELECT DATABASE() AS CurrentDatabase;
```

Q8. (5 marks)
a) Display total marks of each student.
b) Display subject-wise highest mark.
c) Use GROUP BY and HAVING to display subjects with average marks > 75.

Code:

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

Section D: Joins and Subqueries (25 Marks)

Q9. (5 marks)
a) Inner Join to retrieve students and their courses.
b) Left Join to get all students even if not enrolled.
c) Right Join to get all courses even if no students.
d) Full Outer Join equivalent using UNION.
e) Cross Join to show all combinations.

Code:

```
a) SELECT s.StudentID, s.Name, c.CourseName
   FROM Student_Info s
   INNER JOIN Courses c ON s.StudentID = c.StudentID;

b) SELECT s.StudentID, s.Name, c.CourseName
   FROM Student_Info s
   LEFT JOIN Courses c ON s.StudentID = c.StudentID;

c) SELECT s.StudentID, s.Name, c.CourseName
   FROM Student_Info s
   RIGHT JOIN Courses c ON s.StudentID = c.StudentID;

d)SELECT s.StudentID, s.Name, c.CourseName
   FROM Student_Info s
   LEFT JOIN Courses c ON s.StudentID = c.StudentID

   UNION

   SELECT s.StudentID, s.Name, c.CourseName
   FROM Student_Info s
   RIGHT JOIN Courses c ON s.StudentID = c.StudentID;

e) SELECT s.Name AS StudentName, c.CourseName
   FROM Student_Info s
   CROSS JOIN Courses c;
```

Q10. (5 marks)
a) Students who scored more than average in 'Maths'.
b) Students not in the Marks table.
c) Use EXISTS to get students with at least one subject.
d) Use ALL to find those scoring more than all in 'Science'.

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

Code:

```
a) SELECT s.StudentID, s.Name, m.Marks 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) SELECT s.StudentID, s.Name
      FROM Student_Info s
      WHERE s.StudentID NOT IN (
       SELECT DISTINCT StudentID FROM Marks
    );

  c) SELECT s.StudentID, s.Name
      FROM Student_Info s
      WHERE EXISTS (
       SELECT 1
       FROM Marks m
       WHERE m.StudentID = s.StudentID
    );

    d) SELECT StudentID, Subject, Marks
        FROM Marks
        WHERE Subject = 'Science'
          AND Marks > ALL (
         SELECT Marks
         FROM Marks
         WHERE Subject = 'Science'
      );

    e) SELECT StudentID, Subject, Marks
        FROM Marks
        WHERE Subject = 'English'
          AND Marks > ANY (
            SELECT Marks
```

```
    FROM Marks
    WHERE Subject = 'English'
);
```

Q11. (5 marks)

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

Code:

a)
```
SELECT Name FROM Student_Info UNION
SELECT Name FROM Other_Students;
```

b)
```
SELECT Name FROM Student_Info
WHERE Name IN ( SELECT Name FROM Other_Students
);
```

c)
```
SELECT s.StudentID, s.Name
FROM Student_Info s
LEFT JOIN Marks m ON s.StudentID = m.StudentID
WHERE m.StudentID IS NULL;
```

d)
```
INSERT INTO Marks (StudentID, Subject, Marks)
VALUES (1, 'Maths', 88)
ON DUPLICATE KEY UPDATE Marks = 88;
```

e)
```
SELECT StudentID, Subject, Marks FROM Marks m1
WHERE Marks > ( SELECT AVG(Marks)FROM Marks m2
    WHERE m2.Subject = m1.Subject
);
```

SQL Practical Question Paper 2
Duration: 2 Hours | Total Marks: 60

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.

Code:

- High volume of unstructured or semi-structured data
- Scalability
- Flexible schema
- Real-time big data analytics

Types:

1. Document-Based (e.g., MongoDB, CouchDB)
2. Key-Value Stores (e.g., Redis, DynamoDB)
3. Column-Family Stores (e.g., Apache Cassandra, HBase)
4. Graph-Based (e.g., Neo4j, Amazon Neptune)

1. Document-Based (e.g., MongoDB, CouchDB)

- **Structure**: Stores data as JSON or BSON documents.
- **Use Case**: Content Management Systems, e-commerce product catalogs.

Real-time Application:
Amazon-like Product Catalog

- Each product has different fields (e.g., electronics vs. clothing)
- Easily stored as documents

2. Key-Value Stores (e.g., Redis, DynamoDB)

- **Structure**: Key mapped to a single value (string, JSON, etc.)
- **Use Case**: Caching, session management, real-time leaderboards

Real-time Application:
Online Gaming Session Storage

- Player ID → session state
- High-speed reads and writes

3. Column-Family Stores (e.g., Apache Cassandra, HBase)

- **Structure**: Stores data in rows and columns like SQL, but columns can vary by row

- **Use Case**: Large-scale time-series data, analytics

Real-time Application:
IoT Sensor Data Platform

- Sensors push timestamped values every second
- Efficiently stores millions of readings

4. Graph-Based (e.g., Neo4j, Amazon Neptune)

- **Structure**: Nodes (entities) and Edges (relationships)
- **Use Case**: Social networks, fraud detection, recommendation engines

Real-time Application:
LinkedIn's Social Network Graph

- People connected to others
- Fast relationship queries like: "friends of friends"

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.

Code:

1NF Rule: Eliminate repeating groups.
Customer(CustomerID, Name)
Orders(OrderID, CustomerID, ProductID, Quantity, ProductName)

2NF Tables:

1. **Customer**
   Customer(CustomerID, Name)
2. **Orders**
   Orders(OrderID, CustomerID)
3. **OrderDetails**
   OrderDetails(OrderID, ProductID, Quantity)
4. **Product**
   Product(ProductID, ProductName)

3NF Rule:

- Be in 2NF
- No transitive dependencies

Step 4: Convert to BCNF

- Every determinant must be a candidate key

In our schema:

- All functional dependencies are on keys (e.g., ProductID → ProductName)
- No violations of BCNF.

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.

Code:

a. Create Database RetailDB and Design Schema

```
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 CHECK (Quantity > 0),
   OrderDate DATE,
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
   FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

b. Implement a CHECK Constraint on Quantity > 0

```
Quantity INT CHECK (Quantity > 0)
```

c. Alter Products Table to Add a 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 = 5.50
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.

Code:

a.Insert 3 sample orders per customer

```
INSERT INTO Customers VALUES
(1, 'Alice', 'alice@example.com'),
(2, 'Bob', 'bob@example.com');

INSERT INTO Products(ProductID, ProductName, Price, Discount)
VALUES
(101, 'Laptop', 60000.00, 5.00),
(102, 'Phone', 20000.00, 10.00),
(103, 'Tablet', 30000.00, 7.50);

INSERT INTO Orders VALUES
(1, 1, 101, 2, '2024-06-01'),
(2, 1, 102, 1, '2024-06-03'),
(3, 1, 103, 3, '2024-06-05'),
(4, 2, 101, 6, '2024-06-02'),
(5, 2, 102, 7, '2024-06-04'),
(6, 2, 103, 2, '2024-06-06');
```

b. Update product 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. (5 marks) Retrieve the following:

a) Customers who ordered more than 3 different products.

b) Products not ordered by any customer. C

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

Code:

```
a) SELECT CustomerID FROM Orders
   GROUP BY CustomerID
   HAVING COUNT(DISTINCT ProductID) > 3;

b) SELECT ProductID, ProductName FROM Products
   WHERE ProductID NOT IN (SELECT DISTINCT ProductID FROM Orders
   );

c) SELECT CustomerID, COUNT(*) AS OrderCount FROM Orders
  WHERE OrderDate >= CURDATE() - INTERVAL 30 DAY
  GROUP BY CustomerID;
```

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

Code:

a. SELECT CustomerID,
    Email,
    LOWER(SUBSTRING_INDEX(Email, '@', 1)) AS Username,
    LOWER(SUBSTRING_INDEX(Email, '@', -1)) AS Domain
    FROM Customers;

b. SELECT OrderID, CustomerID, OrderDate,
    DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder
    FROM Orders;

c. SELECT CURRENT_USER() AS CurrentUser,
    USER() AS LoggedInUser,
    VERSION() AS MySQLVersion;

d. SELECT CustomerID,
    Name, CONCAT('Hello ', UPPER(LEFT(Name, 1)), LOWER(SUBSTRING(Name, 2)), '!') AS
    Greeting
    FROM Customers;

Q7. (5 marks)

a) Aggregate total revenue by product category.

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

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

Code:

a) SELECT p.Category,
    SUM(p.Price * o.Quantity) AS TotalRevenue FROM Orders o
    JOIN Products p ON o.ProductID = p.ProductID
    GROUP BY p.Category;

b) SELECT p.Category,
    SUM(p.Price * o.Quantity) AS Revenue FROM Orders o
    JOIN Products p ON o.ProductID = p.ProductID
    GROUP BY p.Category WITH ROLLUP;

c) SELECT p.Category,
    SUM(p.Price * o.Quantity) AS Revenue FROM Orders o
    JOIN Products p ON o.ProductID = p.ProductID

```
GROUP BY p.Category
HAVING Revenue > 100000;
```

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.

Code:

```
a) SELECT  c1.Name AS CustomerName,
   c2.Name AS ReferredByName FROM Customers c1
   JOIN Customers c2 ON c1.ReferredBy = c2.CustomerID;

b) SELECT o.OrderID, o.CustomerID, p.ProductName, o.Quantity, p.Price,
   o.Quantity * p.Price AS Total
   FROM Orders o
   JOIN Products p ON o.ProductID = p.ProductID;

c) SELECT * FROM ( SELECT c.CustomerID, c.Name,
     SUM(p.Price * o.Quantity) AS TotalSpent,
     RANK() OVER (ORDER BY SUM(p.Price * o.Quantity) DESC) AS RankPos
     FROM Customers c
     JOIN Orders o ON c.CustomerID = o.CustomerID
     JOIN Products p ON o.ProductID = p.ProductID
     GROUP BY c.CustomerID
   ) ranked WHERE RankPos <= 3;

d) SELECT c.CustomerID, c.Name
   FROM Customers c
   LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
   WHERE o.CustomerID IS NULL;

e) SELECT  p1.ProductName AS Product1, p2.ProductName AS Product2
   FROM Products p1
   CROSS JOIN Products p2
   WHERE p1.ProductID < p2.ProductID;
```

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

Code:

```
a)SELECT DISTINCT o.CustomerID FROM Orders on
  JOIN Products p ON o.ProductID = p.ProductID
  WHERE (o.Quantity * p.Price) > (
  SELECT AVG(o2.Quantity * p2.Price)
  FROM Orders o2
  JOIN Products p2 ON o2.ProductID = p2.ProductID
  WHERE o2.CustomerID = o.CustomerID
  );

b) SELECT c.CustomerID, c.Name FROM Customers c
    WHERE EXISTS (
     SELECT 1
     FROM Orders o
     WHERE o.CustomerID = c.CustomerID
     GROUP BY o.CustomerID
     HAVING COUNT(DISTINCT o.ProductID) >= 2
  );

c) SELECT CustomerID FROM (
     SELECT CustomerID, SUM(Quantity) AS TotalQty
     FROM Orders
     GROUP BY CustomerID
    ) AS totals
    WHERE TotalQty > ALL (
      SELECT SUM(Quantity)
      FROM Orders
      GROUP BY CustomerID
      HAVING CustomerID != totals.CustomerID
    );
```

d) SELECT ProductID, ProductName, Price
FROM Products
WHERE Price > ANY (
SELECT Price
FROM Products
WHERE Category = 'Electronics'
);

e) SELECT ProductID, ProductName, TotalQty FROM (
SELECT
p.ProductID,
p.ProductName,
SUM(o.Quantity) AS TotalQty,
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
) ranked
WHERE rnk <= 3;

Q10. (5 marks)

a) Simulate INTERSECT using INNER JOIN on two customer segments.

b) Use EXCEPT to find products in inventory not yet ordered.

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

d) Use UNION to combine two regional customer tables.

e) Write a WITH CTE that ranks customers by total spend and filters top 5.

Code:

a) SELECT c1.CustomerID, c1.Name
FROM East_Customers c1
INNER JOIN West_Customers c2 ON c1.CustomerID = c2.CustomerID;

b) SELECT i.ProductID FROM Inventory i
LEFT JOIN Orders o ON i.ProductID = o.ProductID
WHERE o.ProductID IS NULL;

c) INSERT INTO Customers (CustomerID, Name, Region)
VALUES (101, 'Amit', 'North') ON DUPLICATE KEY UPDATE
Name = 'Amit',

Region = 'North';

d) SELECT * FROM East_Customers
UNION
SELECT * FROM West_Customers;

e) 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 SpendRank
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
GROUP BY c.CustomerID
)
SELECT * FROM CustomerSpending
WHERE SpendRank <= 5;