**Day 1 - 09.06.25**

**Algorithm Basics:**

* Brute Force Approach
* Heuristic Approach
* Greedy Approach
* Divide and Conquer Approach
* Dynamic Programming

## **1. Brute Force Approach**

## **Explanation:**

* Try **all possible solutions** until the right one is found.
* No shortcuts, no intelligence — just testing everything.

**Real-life examples:**

1. Guessing a PIN code by trying every possible 4-digit combination.
2. Finding a word in an unsorted dictionary by checking page by page.
3. Trying all possible keys on a keychain to unlock a door.
4. Searching for your lost phone by looking in every room one by one.
5. Solving a puzzle blindly by testing all possible piece placements.

## **2. Heuristic Approach**

### **Explanation:**

* Uses **experience-based shortcuts** to find a **good enough solution quickly**, but not guaranteed to be the best.

**Real-life examples:**

1. Google Maps suggesting a "good route" — not always shortest, but practical.
2. Chess-playing AI — uses approximations instead of checking all moves.
3. Emergency vehicle finding quickest way based on traffic signals and road width.
4. Packing your bag fast — guessing which items to fit without checking all ways.
5. Cleaning a messy room quickly — picking up biggest items first to make space.

## **3. Greedy Approach**

### **Explanation:**

* At each step, make the **best possible choice immediately**, without worrying about future consequences.
* Doesn't always guarantee an optimal solution for every problem.

**Real-life examples:**

1. Making change at a shop — giving the highest coin/bill possible first.
2. Filling a backpack with valuable items — picking the most valuable ones first.
3. Selecting activities to maximize free time — choosing tasks that end earliest.
4. Buying items in a supermarket sale — picking the most discounted products first.
5. Laying tiles on the floor — placing the largest tiles first to cover the most area.

## **4. Divide and Conquer Approach**

### **Explanation:**

* **Breaks the problem into smaller parts**, solves each part, then **combines the results**.
* Very efficient for complex problems.

**Real-life examples:**

1. Breaking a chocolate bar into pieces to share easily.
2. Sorting files on a computer by dividing into folders.
3. Dividing an army into smaller squads for attacking from different sides.
4. Solving a crossword puzzle — filling easy words first to help with others.
5. Organizing a large event — splitting work between teams (catering, decoration, seating).

## **5. Dynamic Programming (DP)**

### **Explanation:**

* Break the problem into sub-problems.
* **Solve and store results** of sub-problems to **avoid solving them again**.
* Best for problems with **overlapping sub-problems** and **optimal substructure**.

**Real-life examples:**

1. Planning the cheapest travel route — storing costs of visiting cities to avoid recalculating.
2. Calculating the number of ways to climb stairs — reusing previously calculated steps.
3. Budget planning — using previous months’ expenses to predict future needs.
4. Auto-complete feature in keyboards — remembering past words to suggest the next.
5. Building a wall with bricks — remembering the number of bricks needed for each level.

**Summary**

| **Approach** | **How it works** | **Real Life Example** |
| --- | --- | --- |
| **Brute Force** | Try all possibilities | Guessing passwords, Searching unsorted data |
| **Heuristic** | Use smart guesses or shortcuts | Google Maps route suggestion, Packing without calculation |
| **Greedy** | Best immediate choice at every step | Making change with coins, Scheduling tasks quickly |
| **Divide & Conquer** | Split into smaller problems, solve, combine | Jigsaw Puzzle, Cooking with multiple people |
| **Dynamic Programming** | Solve sub-problems, store results for reuse | Climbing stairs, Calculating Fibonacci numbers |

**Sorting Algorithms:**

* Bubble Sort
* Selection Sort
* Insertion Sort
* Merge Sort
* Quick Sort

1. **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.
2. **Selection Sort:** Selects the smallest (or largest) element from the unsorted part and places it in the correct position.
3. **Insertion Sort:** Builds the final sorted array one item at a time by inserting each element into its correct position.
4. **Merge Sort:** Divides the list into halves, sorts each half, and merges them back together.
5. **Quick Sort:** Picks a pivot, partitions the list into elements smaller and larger than the pivot, and recursively sorts the partitions.

Top 2 Sorting Techniques (Recommended):

### **1. Merge Sort:**

* **Why?** Guaranteed **O(n log n)** time, stable, reliable for large datasets.

### **2. Quick Sort:**

* **Why?** Very fast on average, uses less space (**O(log n)**), ideal for in-memory sorting.

**Conclusion:** > Use **Merge Sort** when stability or large dataset handling is required.  
 > Use **Quick Sort** for speed and lower space usage in general applications.

**Searching Algorithms:**

* Linear Search
* Binary Search

**Linear Search:**

* Checks each element one by one.
* Can work on unsorted or sorted data.
* Very simple to implement.
* Small datasets, unsorted lists.
* Efficiency - Slower for large data.

**Example:**

Find number 5 in the following list:

List: [12, 7, 9, 5, 15, 20]

### Steps:

1. Check 12 — Not 5
2. Check 7 — Not 5
3. Check 9 — Not 5
4. Check 5 — Found it!

Result: 5 found at position 4.

**Binary Search:**

* Divides the list into halves to search.
* Requires sorted data only.
* Slightly more complex due to halving.
* Large, sorted datasets.
* Efficiency - Much faster for large, sorted data.

**Example:**

Find number 5 in the following sorted list:

List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

### Steps:

1. Middle element is 5 (at index 4).
2. Is 5 = 5? Yes!

Result: 5 found at position 5

## **If Searching for 2 in Binary Search:**

1. Middle is 5. Is 2 < 5? Yes — Search left half: [1,2,3,4]
2. New middle is 2. Is 2 = 2? Yes!

Result: 2 found at position 2.

**When to Use What?**

| Condition | Recommended Search |
| --- | --- |
| **Data is unsorted** | Linear Search |
| **Data is sorted** | Binary Search |
| **Small-sized data (any type)** | Linear Search |
| **Large-sized, sorted data (for efficiency)** | Binary Search |

## 

## **Conclusion:**

* Linear Search — Use when the list is unsorted or small.
* Binary Search — Use when the list is sorted for faster search.

**Tree Structure:**

* **Definition:** A hierarchical data structure made of nodes.
* **Each node** contains a value and links (edges) to child nodes.

**Terms:**

* **Root:** Top node.
* **Child:** Node derived from another node.
* **Leaf:** Node with no children.
* **Height:** Length of the longest path to a leaf.

Example Tree:

A (Root)

/ \

B C

/ \ \

D E F

**DFS (Depth-First Search):**

* Search technique that goes as deep as possible in one branch before backtracking.
* Approaches: Pre-order, In-order, Post-order.

Example Binary Tree:

A

/ \

B C

/ \

D E

## Pre-order Traversal **(Root → Left → Right)**:

* Visit root first, then left subtree, then right subtree.

**Order:** A → B → D → E → C

## In-order Traversal **(Left → Root → Right)**:

* Visit left subtree first, then root, then right subtree.

**Order:** D → B → E → A → C

## Post-order Traversal **(Left → Right → Root)**:

* Visit left subtree first, then right subtree, then root.

**Order:** D → E → B → C → A

## BFS (Breadth-First Search):

* Search technique that visits all nodes at the present depth level before going deeper.

A (Root)

/ \

B C

/ \ \

D E F

Example Traversal for the above tree:  
 A → B → C → D → E → F

**BST (Binary Search Tree):**

A tree where:

* **Left child < Parent < Right child**
* All values in the left subtree are smaller; those in the right are larger.

Example:

8

/ \

3 10

/ \ \

1 6 14

**AVL Tree (Balanced BST):**

* AVL Tree is a **self-balancing Binary Search Tree (BST)**.
* After every insertion or deletion, the tree checks and maintains its balance.
* Named after inventors **Adelson-Velsky and Landis**.

**Example:** (Balances itself even if you insert sorted numbers)

### Balance Factor:

* Balance Factor = Height of Left Subtree – Height of Right Subtree
* For every node:

Balance Factor = -1, 0, or +1 (Tree is balanced)

If |Balance Factor| > 1 → Tree needs rebalancing (Rotation)

### Rotations (to maintain balance):

1. Left-Left (LL) Rotation: When a node is inserted in the left subtree of left child.
2. Right-Right (RR) Rotation: When inserted in the right subtree of right child.
3. Left-Right (LR) Rotation: When inserted in the right subtree of left child.
4. Right-Left (RL) Rotation: When inserted in the left subtree of right child.

### Example:

Insert: 10 → 20 → 30

Without AVL:

10

\

20

\

30 Unbalanced (Right-Right case).

After RR Rotation:

20

/ \

10 30 Balanced AVL Tree.

## **Why do we use BST (Binary Search Tree)?**

* BST stores data in a sorted, structured way for quick searching, inserting, and deleting.
* In an ideal balanced BST, operations like search, insert, and delete take O(log n) time.
* Example: Searching for names, phone numbers, IDs.

**Problem:** When many sorted values are inserted (like 1, 2, 3, 4…), BST can become **unbalanced**, behaving like a linked list with **O(n)** time — making operations slow.

## **Why do we need AVL Tree?**

* AVL Tree is a self-balancing BST.
* After every insertion or deletion, the AVL tree performs rotations to stay balanced.
* This ensures:  
  + The tree's height is always minimal.
  + Search, insert, delete remain O(log n) — always fast, even in the worst case.

**Difference between BST and AVL Tree:**

| **Aspect** | **BST** | **AVL Tree** |
| --- | --- | --- |
| Balancing | Not guaranteed (can become unbalanced) | Always balanced (auto-balances) |
| Complexity (Worst) | O(n) (if unbalanced) | O(log n) (balanced always) |
| Insert/Delete | Simple (no balancing needed) | Complex (requires rotations) |
| Speed Stability | May slow down if unbalanced | Always fast and reliable |
| Usage | Small/simple datasets, rarely unbalanced | Databases, large data, real-time systems |

* Use **BST** when data is **small and balancing isn't critical**.
* Use **AVL** Tree when you want consistently **fast performance**, especially with **large or frequently changing data**.

| **Concept** | **Purpose** | **Example Use** |
| --- | --- | --- |
| **Tree** | Hierarchical data storage | XML/HTML structure, file systems |
| **DFS** | Deep path exploration | Maze, puzzle solving |
| **BFS** | Level-by-level exploration | Shortest path in maps |
| **BST** | Fast search, insert, delete | Dictionaries, user records |
| **AVL** | Balanced BST for guaranteed speed | Databases, real-time apps |

**Day 2 - 10.06.25**

**Agile:**

* **Agile** is a modern approach to software development that focuses on flexibility, customer collaboration, and delivering small, working pieces of software frequently (called increments).
* Instead of building the whole product at once, Agile builds it step-by-step with continuous feedback.

**Agile Manifesto:**

The Agile Manifesto is a set of guiding values and principles for Agile development.

### **4 Key Values in Agile Manifesto:**

1. Individuals & Interactions over Processes & Tools  
    ➡️ Team communication is more important than strict tools or processes.
2. Working Software over Comprehensive Documentation  
    ➡️ A working product is more valuable than detailed paperwork.
3. Customer Collaboration over Contract Negotiation  
    ➡️ Ongoing customer involvement is better than sticking strictly to contracts.
4. Responding to Change over Following a Plan  
    ➡️ Being flexible and adjusting to change is better than rigid plans.

## **12 Agile Principles (Summary):**

The Agile Manifesto also outlines 12 principles, like:

* Satisfy customer via early & continuous delivery.
* Welcome changing requirements.
* Deliver working software frequently.
* Build projects around motivated individuals.
* Face-to-face conversation is best.
* Regular reflection for team improvement.

**Benefits of Agile over Traditional Models:**

| **Aspect** | **Agile** | **Traditional (Waterfall)** |
| --- | --- | --- |
| Flexibility | High – changes allowed anytime | Low – changes costly once planned |
| Delivery | Frequent small releases | One final delivery at the end |
| Customer Involvement | Continuous | Only at beginning & end |
| Risk | Less – issues found early | High – issues found late |
| Feedback | Regular from customer | Limited to end stage |
| Documentation | Minimal necessary | Heavy documentation |

**SDLC (Software Development Life Cycle):**

SDLC is a process that defines the steps to develop software, ensuring quality and correctness.

### **Common SDLC Phases:**

1. Requirement Gathering
2. System Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

**Types of SDLC Models:**

| **Model** | **Description** | **Example Use** |
| --- | --- | --- |
| **Waterfall** | Linear, step-by-step, no going back. | Fixed-scope government projects. |
| **V-Model** | Similar to Waterfall, but each development stage has a testing phase. | Safety-critical systems. |
| **Incremental** | Develop system in small parts (increments). | Medium projects where basic features needed fast. |
| **Agile Model** | Flexible, iterative, based on customer feedback. | Mobile Apps, Startups, SaaS Platforms. |
| **Spiral Model** | Focuses on risk analysis; mix of Waterfall & Prototyping. | High-risk, large budget projects. |

**Scrum Overview:**

Scrum is a popular Agile framework used to manage complex projects.

### **Scrum Roles:**

* Product Owner (PO): Sets product vision & priorities.
* Scrum Master: Facilitates the process, solves team issues.
* Development Team: Cross-functional members who build the product.

### **Scrum Activities:**

1. Sprint Planning: Plan what to deliver in the next 2–4 weeks (Sprint).
2. Daily Scrum (Stand-up): 15-minute daily meeting to sync.
3. Sprint Review: Show work done to stakeholders.
4. Sprint Retrospective: Discuss what went well & what can be improved.

**Day 3 - 11.06.2025**

**SQL 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 <<
2. **Which system database stores all login accounts and configuration settings?** a) tempdb  
    b) model  
    c) master <<  
    d) msdb
3. **What is the purpose of the model database in SQL Server?** a) Backup  
    b) Log storage  
    c) Template for new databases <<  
    d) System configuration
4. **What are the two main types of database files in SQL Server?** a) MDF and NDF  
    b) LDF and MDF <<  
    c) NDF and BAK  
    d) BAK and TRN
5. **Which SQL command is used to create a new database?** a) MAKE DATABASE  
    b) NEW DATABASE  
    c) CREATE DATABASE <<  
    d) INIT DATABASE
6. **What happens when you execute DROP DATABASE SalesDB?** a) SalesDB is backed up  
    b) SalesDB is renamed  
    c) SalesDB is deleted permanently <<  
    d) SalesDB is restored
7. **Which command renames a database in SQL Server?** a) RENAME DATABASE old\_name TO new\_name  
    b) ALTER DATABASE old\_name MODIFY NAME = new\_name <<  
    c) UPDATE DATABASE NAME  
    d) SET DATABASE NAME

**Section 2: Managing Tables**

1. **Which data type should be used to store a date of birth?** a) VARCHAR  
    b) DATE <<  
    c) INT  
    d) TEXT
2. **What command is used to create a table?** a) MAKE TABLE  
    b) INSERT TABLE  
    c) CREATE TABLE <<  
    d) DEFINE TABLE
3. **How do you add a new column to an existing table?** a) ALTER TABLE table\_name ADD column\_name datatype<<  
    b) MODIFY TABLE table\_name ADD column\_name  
    c) UPDATE TABLE table\_name ADD column\_name  
    d) APPEND column\_name TO table\_name
4. **Which command is used to rename a table?** a) RENAME TABLE old\_name TO new\_name  
    b) ALTER TABLE old\_name RENAME TO new\_name  
    c) EXEC sp\_rename 'old\_name', 'new\_name'<<  
    d) MODIFY TABLE RENAME
5. **What is the command to delete a table permanently?** a) DELETE TABLE table\_name  
    b) ERASE TABLE table\_name  
    c) DROP TABLE table\_name<<  
    d) REMOVE TABLE table\_name

**Section 3: DML - Manipulating Data**

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

**Day 4 - 12.06.2025**

**Manipulating Data by Using DML Statements - Assignment**

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 database School;

use School;

create table Students

(

Student\_ID int auto\_increment primary key,

Student\_Name varchar(50) not null,

Class int

);

select \* from Students;

insert into Students values

(1, 'Kishore', 10),

(2, 'Abirami', 10),

(3, 'Nahulraj', 10),

(4, 'Siva', 10),

(5, 'Harish', 10);

insert into Students values (6,'Ruban', 10);

alter table Students add Marks\_obtained decimal(10,2);

update Students set Marks\_obtained = 86.72 where Student\_ID=1;

update Students set Marks\_obtained = 89.31 where Student\_ID=2;

update Students set Marks\_obtained = 94.23 where Student\_ID=3;

update Students set Marks\_obtained = 76.70 where Student\_ID=4;

update Students set Marks\_obtained = 80.62 where Student\_ID=5;

update Students set Marks\_obtained = 70.15 where Student\_ID=6;

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

create database Store;

use Store;

create table Customers

(

Customer\_ID int auto\_increment primary key,

FullName varchar(50) not null

);

select \* from Customers;

insert into Customers values (1,'Jennifer Lawrence'),

(2,'Timothy Nathan'),(3,'Robert Chase'),(4, 'Gregory House'),(5,'Lisa Cuddy');

SELECT

FullName,

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

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

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

LENGTH(SUBSTRING\_INDEX(FullName, ' ', -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

SELECT

Sales\_Date,

MONTHNAME(Sales\_Date) AS month\_name,

YEAR(Sales\_Date) AS sale\_year,

DATEDIFF(CURDATE(), Sales\_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

create table employees

(

Employee\_ID int primary key,

Employee\_Name varchar(50) not null,

Employee\_Salary decimal(10,2)

);

select \* from employees;

insert into employees values

(1,'Foreman',5000),

(2,'Wilson',9000),

(3,'Cameron',5700),

(4,'Sammy',7800),

(5,'Valens',8900);

select

Employee\_ID,

Employee\_Name,

Employee\_Salary,

Employee\_Salary \* 1.10 as Hike\_salary,

round(Employee\_Salary,-2) as rounded\_salary

from employees;

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

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'

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

price DECIMAL(10,2)

);

INSERT INTO products (product\_id, product\_name, price) VALUES

(1, 'Laptop', 750.00),

(2, 'Smartphone', NULL),

(3, 'Headphones', 120.00),

(4, 'Keyboard', 45.50),

(5, 'Monitor', NULL),

(6, 'Mouse', 25.00);

select \* from products;

SELECT

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

IFNULL(CAST(price AS CHAR), 'Not Available') AS display\_price

FROM products;

7. 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,

transaction\_date DATE,

sale\_amount DECIMAL(10,2)

);

INSERT INTO transactions (transaction\_id, transaction\_date, sale\_amount) VALUES

(1, '2025-06-10', 250.00),

(2, '2025-06-11', 180.50),

(3, '2025-06-11', 400.00),

(4, '2025-06-12', 95.25),

(5, '2025-06-12', 620.00),

(6, '2025-06-12', 300.00);

select \* from transactions;

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

SELECT

Category,

SUM(Amount) AS TotalSales,

COUNT(\*) AS NumberOfTransactions

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

SELECT

c.Name AS CustomerName,

o.Amount AS OrderAmount

FROM

customers c

INNER JOIN

orders o ON c.CustomerID = o.CustomerID;

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

SELECT

p.ProductID,

p.ProductName,

o.OrderID,

o.Quantity,

o.Amount

FROM

products p

LEFT JOIN

orders o ON p.ProductID = o.ProductID;

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.

SELECT

c.CustomerID,

c.Name AS CustomerName,

ct.Phone,

ct.Email

FROM

contacts ct

RIGHT JOIN

customers c ON ct.CustomerID = c.CustomerID;

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

-- Left join part

SELECT

s.SupplierID,

s.SupplierName,

p.ProductID,

p.ProductName

FROM

suppliers s

LEFT JOIN

products p ON s.SupplierID = p.SupplierID

UNION

-- Right join part

SELECT

s.SupplierID,

s.SupplierName,

p.ProductID,

p.ProductName

FROM

suppliers s

RIGHT JOIN

products p ON s.SupplierID = p.SupplierID;

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

SELECT

p.ProductID,

p.ProductName,

o.OfferID,

o.OfferDescription

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

SELECT

p.Category,

SUM(o.Quantity) AS TotalQuantitySold,

AVG(p.Price) AS AveragePrice

FROM

orders o

JOIN

products p ON o.ProductID = p.ProductID

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

SELECT

s.Name AS StudentName,

AVG(m.Marks) AS AverageMarks

FROM

students s

JOIN

marks m ON s.StudentID = m.StudentID

GROUP BY

s.Name

HAVING

AVG(m.Marks) > 75;

**Day 5 - 13-06-2025**

**Subqueries:**

1. Querying Data by Using Subqueries

Get students whose average marks are above 80.

SELECT StudentID, Name

FROM students

WHERE StudentID IN (

SELECT StudentID

FROM marks

GROUP BY StudentID

HAVING AVG(Marks) > 80

);

### 2. Using the EXISTS

Get customers who have placed at least one order.

SELECT CustomerID, Name

FROM customers c

WHERE EXISTS (

SELECT 1 FROM orders o WHERE o.CustomerID = c.CustomerID

);

### 3. Using ANY

Get products priced higher than any product in the 'Electronics' category.

SELECT ProductName, Price

FROM products

WHERE Price > ANY (

SELECT Price FROM products WHERE Category = 'Electronics'

);

### 4. Using ALL

Get students who scored more than all students in 'Science'.

SELECT StudentID, Subject, Marks

FROM marks

WHERE Subject = 'Science' AND Marks > ALL (

SELECT Marks FROM marks WHERE Subject = 'Science'

);

### 5. Using Nested Subqueries

Get product names that belong to categories with total sales over 10000.

SELECT ProductName

FROM products

WHERE CategoryID IN (

SELECT CategoryID

FROM (

SELECT CategoryID, SUM(Amount) AS TotalSales

FROM orders o

JOIN products p ON o.ProductID = p.ProductID

GROUP BY CategoryID

) AS category\_sales

WHERE TotalSales > 10000

);

### 6. Using Correlated Subqueries

Get students whose marks are above the average for each subject.

SELECT StudentID, Subject, Marks

FROM marks m1

WHERE Marks > (

SELECT AVG(Marks)

FROM marks m2

WHERE m2.Subject = m1.Subject

);

### 7. Using UNION

Combine students from two regional student tables.

SELECT Name, Email FROM students\_north

UNION

SELECT Name, Email FROM students\_south;

### 8. Using INTERSECT

Find customers who are also suppliers (common IDs).

SELECT CustomerID FROM customers

INTERSECT

SELECT SupplierID FROM suppliers;

### 9. Using EXCEPT

Find products in inventory not yet ordered.

SELECT ProductID FROM products

EXCEPT

SELECT DISTINCT ProductID FROM orders;

### 10. Using MERGE (Simulated with INSERT/UPDATE)

If customer exists, update email; else insert new customer.

INSERT INTO customers (CustomerID, Name, Email)

VALUES (101, 'John Doe', 'john@example.com')

ON CONFLICT (CustomerID)

DO UPDATE SET Email = EXCLUDED.Email;

**Day 5 - 13-06-2025**

**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.

| Features | SQL | NoSQL |
| --- | --- | --- |
| Data Model | Relational (tabes- row & column) | Non-relational (graphs, document) |
| Schema | Fixed Schema | No fixed schema |
| Query language | Structured Query Language (SQL) | Varies |
| Scalability | Vertically scalable | Horizontally scalable |
| Example | MySQL, PostgreSQL, Oracle | MongoDB, Cassandra, Redis |

SQL:

* Advantages:

1. Ideal for structured data and complex queries.

Example: Banking systems use SQL for transactional accuracy.

1. ACID compliance ensures data consistency.

Example: E-commerce platforms use SQL databases for payment records.

* Disadvantages:

1. Struggles with massive unstructured data.

Example: Scaling a MySQL database for a global chat app can be difficult.

1. Changing schema in production can be time-consuming.

Example: Altering table structure in a live inventory system can cause downtime.

NoSQL:

* Advantages:

1. Great for unstructured/semi-structured data.

Example: MongoDB used by content platforms like Medium to store article metadata.

1. It has High Scalability.

Example: Cassandra used by Netflix for massive, distributed data handling.

* Disadvantages:

1. Lacks strong consistency.

Example: In a social media feed, posts might not appear instantly due to replication delay.

1. Not ideal for complex queries.

Example: Building a complex report from related documents in MongoDB is harder.

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; ensure atomic values.

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

* 2NF: Remove partial dependencies.

1. Student(StudentID, Name)
2. Enrollment(StudentID, CourseID)
3. Course(CourseID, CourseName, InstructorName, InstructorPhone)

* 3NF: Remove transitive dependencies.

1. Student(StudentID, Name)
2. Enrollment(StudentID, CourseID)
3. Course(CourseID, CourseName, InstructorName)
4. Instructor(InstructorName, InstructorPhone)

Q3. (5 marks)

a) Create a database named StudentDB.

CREATE DATABASE StudentDB;

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

USE StudentDB;

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100)

);

c) Rename the table to Student\_Info.

RENAME TABLE Students TO Student\_Info;

d) Add a column PhoneNumber.

ALTER TABLE Student\_Info

ADD PhoneNumber VARCHAR(15);

e) Drop the table.

DROP TABLE Student\_Info;

**Section B: DML & Filtering Data (15 Marks)**

Q4. (5 marks)

a) Insert 3 student records into Student\_Info.

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

(1, 'Alice Sharma', '2001-05-14', 'alice@gmail.com', '9876543210'),

(2, 'Rahul Mehta', '1999-11-30', 'rahul@yahoo.com', '9123456780'),

(3, 'Sneha Roy', '2002-03-25', 'sneha@outlook.com', '9988776655');

b) Update one student's phone number.

UPDATE Student\_Info

SET PhoneNumber = '9000000000'

WHERE StudentID = 2;

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

DELETE FROM Student\_Info

WHERE Email LIKE '%@gmail.com'

LIMIT 1;

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

SELECT Name, Email

FROM Student\_Info

WHERE YEAR(DOB) > 2000;

e) Retrieve distinct domain names from the email column.

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

FROM Student\_Info;

Q5. (5 marks)

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

SELECT \*

FROM Student\_Info

WHERE Name LIKE 'A%';

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

SELECT \*

FROM Student\_Info

WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';

c) Retrieve students using IN operator on city names.

SELECT \*

FROM Student\_Info

WHERE City IN ('Delhi', 'Mumbai', 'Bangalore');

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

SELECT \*

FROM Student\_Info

WHERE (DOB > '2000-06-13' AND Email LIKE '%@gmail.com')

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

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

SELECT S.Name AS StudentName, S.DOB AS BirthDate

FROM Student\_Info AS S;

Q6. (5 marks)

Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows.

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks INT

);

INSERT INTO Marks (StudentID, Subject, Marks) VALUES

(1, 'Math', 85),

(2, 'Science', 78),

(3, 'English', 55);

a) Display student IDs and their subjects where marks > 70.

SELECT StudentID, Subject

FROM Marks

WHERE Marks > 70;

b) Display subjects with average marks.

SELECT Subject, AVG(Marks) AS AverageMarks

FROM Marks

GROUP BY Subject;

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

SELECT Subject, AVG(Marks) AS AverageMarks

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) BETWEEN 60 AND 90;

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

Q7. (5 marks)

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

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

b) Extract month and year from a DOB column.

SELECT

MONTH(DOB) AS BirthMonth,

YEAR(DOB) AS BirthYear

FROM Student\_Info;

c) Convert a student's name to uppercase.

SELECT UPPER(Name) AS UpperName

FROM Student\_Info;

d) Round off marks to 2 decimal places.

SELECT ROUND(Marks, 2) AS RoundedMarks

FROM Marks;

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

SELECT USER() AS CurrentUser;

SELECT DATABASE() AS CurrentDatabase;

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) AS HighestMark

FROM Marks

GROUP BY Subject;

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

SELECT Subject, AVG(Marks) AS AverageMarks

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) > 75;

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

Q9. (5 marks)

a) Inner Join to retrieve students and their courses.

SELECT S.StudentID, S.Name, C.CourseName

FROM Student\_Info S

INNER JOIN Enrollment E ON S.StudentID = E.StudentID

INNER JOIN Course C ON E.CourseID = C.CourseID;

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

SELECT S.StudentID, S.Name, C.CourseName

FROM Student\_Info S

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

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

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

SELECT S.StudentID, S.Name, C.CourseName

FROM Course C

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

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

d) Full Outer Join equivalent using UNION.

SELECT S.StudentID, S.Name, C.CourseName

FROM Student\_Info S

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

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

UNION

SELECT S.StudentID, S.Name, C.CourseName

FROM Course C

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

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

e) Cross Join to show all combinations.

SELECT S.StudentID, S.Name, C.CourseID, C.CourseName

FROM Student\_Info S

CROSS JOIN Course C;

Q10. (5 marks)

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

SELECT StudentID, Marks

FROM Marks

WHERE Subject = 'Maths'

AND Marks > (

SELECT AVG(Marks)

FROM Marks

WHERE Subject = 'Maths'

);

b) Students not in the Marks table.

SELECT \*

FROM Student\_Info

WHERE StudentID NOT IN (

SELECT DISTINCT StudentID

FROM Marks

);

c) Use EXISTS to get students with at least one subject.

SELECT \*

FROM Student\_Info S

WHERE EXISTS (

SELECT 1

FROM Marks M

WHERE M.StudentID = S.StudentID

);

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

SELECT StudentID, Marks

FROM Marks

WHERE Marks > ALL (

SELECT Marks

FROM Marks

WHERE Subject = 'Science'

)

AND Subject = 'Science';

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

SELECT StudentID, Marks

FROM Marks

WHERE Marks > ANY (

SELECT Marks

FROM Marks

WHERE Subject = 'English'

)

AND Subject = 'English';

Q11. (5 marks)

a) UNION of student names from two tables.

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Alumni;

b) INTERSECT to find common students.

SELECT Name FROM Student\_Info

INTERSECT

SELECT Name FROM Alumni;

SELECT S.Name

FROM Student\_Info S

INNER JOIN Alumni A ON S.Name = A.Name;

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

SELECT StudentID FROM Student\_Info

EXCEPT

SELECT StudentID FROM Marks;

SELECT StudentID

FROM Student\_Info

WHERE StudentID NOT IN (

SELECT StudentID FROM Marks

);

d) MERGE concept or simulate with UPDATE and INSERT.

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

VALUES (4, 'New Student', '2003-07-15', 'new@student.com', '9090909090')

ON DUPLICATE KEY UPDATE

Name = VALUES(Name),

Email = VALUES(Email);

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

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

FROM Marks M

WHERE M.Marks > (

SELECT AVG(M2.Marks)

FROM Marks M2

WHERE M2.Subject = M.Subject

);

**Day 5 - 13-06-2025**

**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.

Scenarios Where NoSQL is Preferred:

| Scenario | Reason |
| --- | --- |
| 1. Handling Large Volumes of Unstructured Data | NoSQL can easily store logs, documents, multimedia, etc., unlike SQL which requires fixed schemas. |
| 2. Rapid Development with Frequent Schema Changes | NoSQL databases like MongoDB allow flexible schemas, ideal for agile development. |
| 3. Horizontal Scaling Needs | NoSQL systems are designed to scale out using distributed clusters, while SQL scaling is usually vertical. |
| 4. High Write/Read Performance in Real-Time Applications | NoSQL is optimized for performance in scenarios like IoT, messaging apps, and caching systems. |

Types of NoSQL Databases and Real-Time Applications:

| Type | Description | Real-Time Example |
| --- | --- | --- |
| 1. Document-based | Stores data as JSON-like documents. | MongoDB – Used in content management systems (e.g., blogging platforms). |
| 2. Key-Value Store | Simple key-value pairs, very fast | Redis – Used in session storage and real-time analytics dashboards. |
| 3. Column-Family Store | Data stored in columns rather than rows. | Apache Cassandra – Used by Netflix for time-series data and scalability. |
| 4. Graph-based | Focuses on relationships between data nodes. | Neo4j – Used in social media apps for friend recommendations. |

NoSQL is ideal for modern applications needing speed, scalability, and flexible data models. While SQL is still strong for transactional systems, NoSQL excels in dynamic, large-scale environments.

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.

* 1 NF:

1. CustomerOrders (CustomerID, Name, OrderID, ProductID, Quantity, ProductName)

* 2 NF:

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

* 3 NF:

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

* BCNF:

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

**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.

-- Create Database

CREATE DATABASE RetailDB;

USE RetailDB;

-- Create Customers Table

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100)

);

-- Create Products Table

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Price DECIMAL(10,2)

);

-- Create Orders Table

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT,

OrderDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

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

ALTER TABLE Orders

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

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

-- Add Discount column

ALTER TABLE Products

ADD Discount DECIMAL(5,2);

-- Update some values

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:

a) Insert 3 sample orders per customer.

-- Sample Inserts for Customers

INSERT INTO Customers (CustomerID, Name, Email)

VALUES

(1, 'Anika Sharma', 'anika@example.com'),

(2, 'Ravi Patel', 'ravi@example.com');

-- Sample Inserts for Products

INSERT INTO Products (ProductID, ProductName, Price, Discount)

VALUES

(1, 'Laptop', 60000.00, 10.00),

(2, 'Headphones', 3000.00, 5.00);

-- Insert 3 Orders for each Customer

INSERT INTO Orders (OrderID, CustomerID, ProductID, Quantity, OrderDate)

VALUES

(101, 1, 1, 2, '2025-06-01'),

(102, 1, 2, 6, '2025-06-02'),

(103, 1, 2, 1, '2025-06-03'),

(104, 2, 1, 3, '2025-06-01'),

(105, 2, 2, 7, '2025-06-02'),

(106, 2, 1, 1, '2025-06-03');

b) Update prices with 10% increase where quantity sold > 5.

UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID

FROM Orders

GROUP BY ProductID

HAVING SUM(Quantity) > 5

);

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

DELETE FROM Orders

WHERE ProductID IN (

SELECT P.ProductID

FROM Products P

LEFT JOIN Orders O ON P.ProductID = O.ProductID

GROUP BY P.ProductID

HAVING SUM(O.Quantity) IS NULL

);

Q5. (5 marks)

Retrieve the following:

a) Customers who ordered more than 3 different products.

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

b) Products not ordered by any customer.

SELECT \*

FROM Products

WHERE ProductID NOT IN (

SELECT DISTINCT ProductID

FROM Orders

);

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

SELECT CustomerID, COUNT(\*) AS OrderCount

FROM Orders

WHERE OrderDate >= CURDATE() - INTERVAL 30 DAY

GROUP BY CustomerID;

**Section C: Advanced Functions and Aggregations (10 Marks)**

Q6. (5 marks)

a) Use string functions to standardize and extract parts from customer email IDs.

-- Convert email to lowercase and extract domain

SELECT

Email,

LOWER(Email) AS LowercaseEmail,

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

FROM Customers;

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

SELECT

OrderID,

DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder

FROM Orders;

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

SELECT

CURRENT\_USER() AS CurrentUser,

@@hostname AS HostName;

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

SELECT

CONCAT('Hello ', UPPER(SUBSTRING(Name, 1, 1)), LOWER(SUBSTRING(Name, 2))) AS Greeting

FROM Customers;

Q7. (5 marks)

a) Aggregate total revenue by product category.

SELECT

P.Category,

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

FROM

Orders O

JOIN

Products P ON O.ProductID = P.ProductID

GROUP BY

P.Category;

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

SELECT

P.Category,

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

FROM

Orders O

JOIN

Products P ON O.ProductID = P.ProductID

GROUP BY

P.Category WITH ROLLUP;

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

SELECT

P.Category,

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

FROM

Orders O

JOIN

Products P ON O.ProductID = P.ProductID

GROUP BY

P.Category

HAVING

SUM(P.Price \* O.Quantity) > 100000;

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

Q8. (5 marks)

a) Self join to list customers referred by other customers.

SELECT

C1.Name AS ReferredCustomer,

C2.Name AS Referrer

FROM

Customers C1

JOIN

Customers C2 ON C1.ReferredBy = C2.CustomerID;

b) Equi join across Orders and Products.

SELECT

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

FROM

Orders O

JOIN

Products P ON O.ProductID = P.ProductID;

c) Join Customers and Orders to display top 3 spenders using window function.

SELECT \*

FROM (

SELECT

C.CustomerID,

C.Name,

SUM(P.Price \* O.Quantity) AS TotalSpent,

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

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

) Ranked

WHERE SpenderRank <= 3;

d) LEFT OUTER JOIN with WHERE NULL to identify inactive customers.

SELECT

C.CustomerID, C.Name

FROM

Customers C

LEFT JOIN

Orders O ON C.CustomerID = O.CustomerID

WHERE

O.OrderID IS NULL;

e) Cross join for all product combinations in a bundle offer.

SELECT

P1.ProductName AS ProductA,

P2.ProductName AS ProductB

FROM

Products P1

CROSS JOIN

Products P2

WHERE

P1.ProductID < P2.ProductID;

Q9. (5 marks)

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

SELECT

O.CustomerID, O.OrderID, (P.Price \* O.Quantity) AS OrderAmount

FROM

Orders O

JOIN

Products P ON O.ProductID = P.ProductID

WHERE

(P.Price \* O.Quantity) > (

SELECT

AVG(P2.Price \* O2.Quantity)

FROM

Orders O2

JOIN

Products P2 ON O2.ProductID = P2.ProductID

WHERE

O2.CustomerID = O.CustomerID

);

b) Subquery using EXISTS to find customers with at least 2 different products.

SELECT

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 ProductID) >= 2

);

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

SELECT

CustomerID

FROM

Orders

GROUP BY

CustomerID

HAVING

COUNT(\*) > ALL (

SELECT

COUNT(\*)

FROM

Orders

GROUP BY

CustomerID

HAVING

CustomerID <> Orders.CustomerID

);

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

SELECT

ProductName, Price

FROM

Products

WHERE

Price > ANY (

SELECT

Price

FROM

Products

WHERE

Category = 'Electronics'

);

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

SELECT

ProductID, TotalSold

FROM (

SELECT

ProductID,

SUM(Quantity) AS TotalSold,

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

FROM

Orders

GROUP BY

ProductID

) Ranked

WHERE rnk <= 3;

Q10. (5 marks)

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

-- Customers who ordered from Category 'Electronics'

SELECT DISTINCT CustomerID

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

WHERE P.Category = 'Electronics'

-- Customers who ordered from Category 'Clothing'

SELECT DISTINCT CustomerID

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

WHERE P.Category = 'Clothing'

-- Simulate INTERSECT using INNER JOIN

SELECT DISTINCT E.CustomerID

FROM (

SELECT CustomerID

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

WHERE P.Category = 'Electronics'

) AS E

INNER JOIN (

SELECT CustomerID

FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

WHERE P.Category = 'Clothing'

) AS C

ON E.CustomerID = C.CustomerID;

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

SELECT P.ProductID, P.ProductName

FROM Products P

LEFT JOIN Orders O ON P.ProductID = O.ProductID

WHERE O.ProductID IS NULL;

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

INSERT INTO Customers (CustomerID, Name, Email)

VALUES (101, 'Alex Doe', 'alex@example.com')

ON DUPLICATE KEY UPDATE

Name = 'Alex Doe',

Email = '[alex@example.com](mailto:alex@example.com)';

d) Use UNION to combine two regional customer tables.

SELECT CustomerID, Name, Region FROM East\_Customers

UNION

SELECT CustomerID, Name, Region FROM West\_Customers;

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

WITH CustomerSpending AS (

SELECT

C.CustomerID,

C.Name,

SUM(P.Price \* O.Quantity) AS TotalSpent,

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

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

)

SELECT \*

FROM CustomerSpending

WHERE SpendingRank <= 5;