1. Comprehensive NoSQL Exercise.

a) Design a NoSQL schema for a social media app using a document-based system (MongoDB): collections for users and posts, with embedded/nested documents for comments.

b) Write insertion queries (MongoDB Syntax) for sample data, demonstrating schema-less flexibility (e.g., some users have "bio", other don't).

c) Explain how to implement sharding on the posts collection by a key like user_id, and create a materialized view for popular posts (if supported).

A) Social media applications need to handle large volumes of user-generated data such as posts, comments, likes and profile information. A NoSQL document-oriented database like MongoDB is ideal because:

- It provides a flexible schema
- It supports nested/embedded documents
- It is highly scalable with horizontal sharding.

In this schema design, we will use two main collections.

① Users Collection

-> The users collection holds data about registered users.
-> Each document corresponds to a single user
-> Fields are flexible
-> Structure:



```
{
  "_id": objectId,
  "user_id": "U001",
  "name": "Rahul",
  "email": "xyz@gmail.com",
  "bio": "Dreamer",
  "profile_pic": https://cdn.app.com/u001.jpg",
  "joined_data": ISODate("2025-01-01"),
  "followers": ["U002", "U003"],
  "following": ["U004"]
}
```

- _id: MongoDB's unique identifier
- user_id: Application-level unique ID for the user.
- bio and profile_pic are optional fields, showing MongoDB's schema-less nature.
- follower and following arrays help in implementing social connections

② Posts Collection.

-> The posts collection stores all the posts made by users.
-> Each post references the user who created it through user_id
-> Posts may contain text, media or both.
-> Comments are stored as embedded/nested documents inside the post.

Structure :-

```
{ "_id": objectId,
  "post_id": "P101",
  "user_id": "U001",
```

```
"content" : "My first post on this app!",
"media" : "https://cdn.app.com/p101.jpg",
"created_at" : ISODate("2025-01-10"),
"likes" : 150,
"comments : [
  {
   "comment_id" : "C001",
   "user_id" : "U002",
   "text" : "Nice post Rahul!",
   "created_at" : ISODate("2025-01-11")
  },
  {
   "comment_id" : "C002",
   "user_id" : "U003",
   "text" : "Nice",
   "created_at" : ISODate("2025-01-11")
  }
 ]
}
```

- Each post has post_id, content, media, created_at and likes.
- Comments is an array of nested documents.
- Each comment includes its own comment_id, the user_id & the commenter text, and timestamp.
- Embedding comments inside posts - improves performance because the application can fetch a post with all its comments in a single query.

Advantages of this schema:-

1. flexibility
2. Performance
3. scalability
4. Realistic Social Media features:-



b) Insertion Queries (MONGODB)

MongoDB Insertion Syntax

Insertion Syntax for Many documents

```
db.collection_name.insert([
   { "field1" : "value",
   },
   { "field1" : "value1",
   }
]);
```

Insertion Synatax for one document

```
db.collection_name.insertOne({
     "field1" : "value1",
     "field2" : "value2",
     "field3" : "value3"
});
```

Example

Insert users (with and without optional fields like bio, profile_pic)

```
db.users.insertOne({
  "user_id" : "U001",
   "name" : "Rahul",
   "email" : "xyz@gmail.com",
   "bio" : "explorer",
   "profile_pic" : "https://Rahul.jpg",
   "Joined_date" : newDate("2025-01-01")
});
```

```
db.users.insertOne({
  "user_id" : "U002",
  "name" : "mahesh",
  "email" : "Abc@gmail.come",
  "joined_date" : new Date("2025-01-05")
});
```

- User U001 has bio and profile_pic
- User U002 does not have these fields (MongoDB allows missing fields).

## c) Sharding and Materialized View in MongoDB.

### 1. Implementing Sharding on posts collection:-

Sharding is used to distribute data across multiple servers to improve performance and scalability in large databases.

Steps to shard posts collection by user_id:

1. enable sharding on the database

   sh.enableSharding("social_app")

   -> this prepares the database for sharding
2. create an index on the shard key

   db.posts.createIndex({user_id:1})

   -> The shard key (user_id) must be indexed.
3. shard the Collection

   sh.shardCollection("social_app.posts", {user_id:1})

   -> MongoDB splits the Posts collection across shards based on user_id.

   -> All posts of a particular user will usually go to the same shard, balancing the data.

Benefits

- Supports horizontal scaling
- Queries filtering by user_id are faster
- Large Collections do not overload a single server

### 2. Creating a Materialized view for popular posts.

-> A Materialized view stores the precomputed results of a query for faster access.

-> In MongoDB, Materialized views can be simulated using aggregation with $out or $merge.

Ex: create a view of Posts with more than 100 likes.

```
db.posts.aggregate([
    {$match: {likes: {$gt:100}}},
    {$project: {post_id:1, user_id:1, content:1, likes:1}},
    {$out: "popular_posts"}
]);
```

-> The Result is stored in new collection called popular_posts.

-> whenever this query is run, it updates the materialized view.

Benefits:

-> Quick access to frequently queried data.

-> Reduces Computation during runtime.

2. SQL functions: Date, Time, Numeric, and String
a. Consider a table orders with columns (OrderID, OrderDate, CustomerName, TotalAmount).
b. Write an SQL query to select orders placed in the last 30 days using DATE functions (e.g., CURDATE(), DATE_ADD()).
c. Use numeric functions (e.g., ROUND(), CEIL()) to round TotalAmount to the nearest integer and calculate a 10% discount (arithmetic: TotalAmount * 0.9).
d. Use String functions (e.g., CONCAT(), UPPER(), SUBSTRING()) to project a formatted column "CUSTOMERSUMMARY" as "UPPER (customerName) - OrderID".
Insert Sample data for 4 orders and write a query combining all these.

A) SQL functions: Date, Time, Numeric, and String

a) CREATE ORDERS Table

```
CREATE TABLE order(
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerName VARCHAR(50),
    TotalAmount DECIMAL(10,2));
```

- orderID → unique identifier for each order.
- orderDate → Date when the order was placed.
- customerName → Name of the customer.



- TotalAmount → Total price of the order.

b) INSERT Sample Data (4 orders)

```
INSERT INTO orders(orderID, OrderDate, CustomerName,
TotalAmount) VALUES
(101, CURDATE(), 'RahulReddy', 1234.56),
(102, DATE_SUB(CURDATE(), INTERVAL 10 DAY), '
Bharath', 789.45),
(103, DATE_SUB(CURDATE(), INTERVAL 35 DAY),
'Devil', 456.78),
(104, DATE_SUB(CURDATE(), INTERVAL 20 DAY),
'ARJUN', 999.99);
```

- Order 103 is older than 30 days → will be excluded in last 30 days query.
- CURDATE() and DATE_SUB() are Date functions.

c) Query Using Date & Numeric functions

```
SELECT OrderID, OrderDate, TotalAmount, ROUND(TotalAmount)
As RoundedAmount, CEIL(TotalAmount * 0.9) As
DiscountedAmount FROM orders WHERE
OrderDate >= DATE_SUB(CURDATE(), INTERVAL 30
DAY);
```

- ROUND(TotalAmount) → rounds to nearest integer.
- CEIL(TotalAmount * 0.9) → applies 10% discount & roundsup.

· WHERE OrderDate >= DATE_SUB(CURDATE(), INTERVAL 30 DAY) -> Selects last 30 days orders

d) Query Using String functions

```
SELECT OrderID, CustomerName, CONCAT(UPPER(
CustomerName), '-', OrderID) AS CustomerSummary
From Orders;
```

· UPPER(CustomerName) -> Converts name to Upper Case.
· CONCAT(UPPER(customerName), '-', OrderID) -> Creates a formatted column CustomerSummary.

Output for C

| OrderID | OrderDate | TotalAmount | Rounded Amount | Discounted Amount |
|---|---|---|---|---|
| 101 | 2025-09-29 | 1234.56 | 1235 | 1112 |
| 102 | 2025-09-19 | 789.45 | 789 | 711 |
| 104 | 2025-09-09 | 999.99 | 1000 | 900 |

**Note:** Order 103 is excluded because it is older than 30days.



Output for D

| OrderID | CustomerName | CustomerSummary |
|---|---|---|
| 101 | Rahul Reddy | RAHULREDDY-101 |
| 102 | Bharath | BHARATH-102 |
| 103 | Devil | DEVIL-103 |
| 104 | Arjun | ARJUN-104 |

3. Implement Relationships and Referential Integrity

a. Consider tables for an e-commerce system: Customers (CustID, Name) and Orders (OrderID, OrderDate, CustID).

b. Write SQL to create both tables, with CustID as PRIMARY KEY in orders with ON DELETE CASCADE for referential integrity.

c. Add Constraints: OrderDate NOT NULL, and a DEFAULT Value for OrderDate as CURRENT_DATE.

d. Insert data for 3 customers and 5 orders, then delete a customer and explain how CASCADE affects orders.

A) Relationships define how tables are connected in a database.
Referential Integrity ensures that relationships between tables remain Consistent.

a) Consider Tables

Tables for an e-commerce system

1. Customers -> (custID, Name)
2. Orders -> (OrderID, OrderDate, CustID)

- CustID in Orders is a foreign key referencing customers.

b) SQL to create Tables with Referntial Integrity

```
CREATE TABLE Customers(
    CustID INT PRIMARY KEY,
    Name VARCHAR(50)
);
CREATE TABLE Orders(
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    Cust ID INT,
    FOREIGN KEY (custID) REFERENCES Customers
    (custID) ON DELETE CASCADE
);
```

- CustID in Customers -> Primary key
- CustID in Orders -> FOREIGN key
- ONDELETE CASCADE -> deleting a Customer deletes their orders automatically.

c) Add Constraints



```
CREATE TABLE Orders(
    OrderID INT PRIMARY KEY,
    OrderDate DATE NOT NULL DEFAULT CURRENT_
    DATE,
    CustID INT,
    FOREIGN KEY (custID) REFERENCES Customers
    (custID) ON DELETE CASCADE
);
```

- OrderDate NOT NULL -> ensures every order has a date.
- DEFAULT CURRENT_DATE -> if no date is given, today's date is used.

d) Insert Sample Data and Demonstrate CASCADE:-

Customers

```
INSERT INTO Customers VALUES(1, 'Ram');
INSERT INTO Customers VALUES(2, 'Mahi');
INSERT INTO Customers VALUES(3, 'Sita');
```

Orders

```
INSERT INTO Orders VALUES(101, '2025-09-29', 1);
INSERT INTO orders VALUES(102, '2025-09-28', 2);
INSERT INTO Orders VALUES(103, '2025-09-27', 1)
INSERT INTO orders VALUES(104, '2025-09-26', 3)
INSERT INTO Orders VALUES(105, '2025-09-25',
```

d) Delete a Customer:-

DELETE FROM Customers WHERE CustID=2;

Effect of CASCADE:

- Customer with CustID=2 is deleted
- All orders linked to this customer (OrderID 102 and 105) are automatically deleted from the Orders table.
- Ensures referential integrity -> no orders exist for a deleted Customer.

4. Different Types of Joins using tables Departments (DeptID, DeptName) and Employees (EmpID, Name, DeptID, Salary).

a. Write SQL for an INNER JOIN to list ~~&~~ employees with their dept names.

b. User LEFT OUTER JOIN to include departments with no employees ~~with their~~ and RIGHT OUTER JOIN for the reverse.

c. implement a FULL OUTER JOIN (or emulate if not supported) and a CROSS JOIN to show all possible combinations.

d. Insert Sample data (4 departments, 6 employees) and explain difference in results.



A) JOINS:

Joins is an operation that Combines rows from two or more tables based on a related column ~~before~~ between them.

Tables Considered:-

Departments -> (DeptID, DeptName)

Employees -> (EmpID, Name, DeptID, Salary)

a) INNER JOIN - Employees with Department Names

Tables Creation

i) Departments

```
CREATE TABLE Departments(
    DeptID INT PRIMARY KEY,
    DeptName VARCHAR (50) NOT NULL
);
```

ii) Employees:-

```
CREATE TABLE Employees(
    EmpID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    DeptID INT,
    Salary DECIMAL(10,20),
    FOREIGN KEY(DeptID) REFERENCES Departments(I
    ON DELETE SET NULL );
```

d) Values Insertion

```
INSERT INTO Departments (1, 'HR');
INSERT INTO Departments(2, 'IT');
INSERT INTO Departments(3, 'Sales');
INSERT INTO Departments(4, 'finance');
```

Employees

```
INSERT INTO Employees(101, 'Rahul', 1, 50000);
INSERT INTO Employees(102, 'Bharath', 2, 60000);
INSERT INTO Employees(103, 'Mahesh', 2, 55000);
INSERT INTO Employees(104, 'Arjun', 3, 45000);
INSERT INTO Employees(105, 'Ravi', NULL, 40000);
INSERT INTO Employees(106, 'Nehanth', 2, 58000);
```

-> Employee 105 has no department
-> Departments 1,2,3,4 all exist, but finance initially has no employee.

```
SELECT e.EmpID, e.Name, d.DeptName, e.Salary
FROM Employees e
INNER JOIN Departments d ON e.DeptID=d.DeptID;
```

| EmpID | Name | DeptName | Salary |
|---|---|---|---|
| 101 | Rahul | HR | 50000 |
| 102 | Bharath | IT | 60000 |
| 103 | Mahesh | IT | 55000 |
| 104 | Arjun | Sales | 45000 |
| 106 | Nehanth | IT | 58000 |



Employee 105 is excluded becuase DeptID is NULL.

D) LEFT OUTER JOIN

```
SELECT d.DeptID, d.DeptName, e.EmpID, e.Name
FROM Departments d
LEFT JOIN Employees e ON d.DeptID = e.DeptID;
```

| DeptID | DeptName | EmpID | Name |
|---|---|---|---|
| 1 | HR | 101 | Rahul |
| 2 | IT | 102 | Bharath |
| 2 | IT | 103 | Mahesh |
| 2 | IT | 106 | Nehanth |
| 3 | Sales | 104 | Arjun |
| 4 | finance | NULL | NULL |

finance has no employees, so it is NULL

RIGHT OUTER JOIN

```
SELECT e.EmpID, e.Name, d.DeptID, d.DeptName
FROM Employees e
RIGHT JOIN Departments d ON e.DeptID=d.DeptID
```

| EmpID | Name | DeptID | DeptName |
|---|---|---|---|
| 101 | Rahul | 1 | HR |
| 102 | Bharath | 2 | IT |
| 103 | Mahesh | 2 | IT |
| 106 | Nehanth | 2 | IT |
| 104 | Arjun | 3 | Sales |
| NULL | NULL | 4 | finance |

## 5) FULL OUTER JOIN

```
SELECT e.EmpID, e.Name, d.DeptID, d.DeptName
FROM Employees e
FULL OUTER JOIN Departments d ON e.DeptID = d.DeptID;
```

| EmpID | Name | DeptID | DeptName |
|---|---|---|---|
| 101 | Rahul | 1 | HR |
| 102 | Bharath | 2 | IT |
| 103 | Mahesh | 2 | IT |
| 104 | Arjun | 3 | Sales |
| 105 | Ravi | NULL | NULL |
| 106 | Nehanth | 2 | IT |
| NULL | NULL | 4 | finance |

## CROSS JOIN

```
SELECT e.Name As Employee, d.DeptName As
Department FROM Employees e CROSS JOIN Departments d;
```



## Expected Output

- Total rows = 6 employees × 4 departments = 24 rows.
- Every employee appears with every department, ignoring actual ~~actual~~ assignments.

## Difference IN Results

| Join Type | Result Includes |
|---|---|
| INNER JOIN | only matching rows from both tables. |
| LEFT OUTER JOIN | All rows from Departments + matching employees. |
| RIGHT OUTER JOIN | All rows from Employees + Matching departments. |
| FULL OUTER JOIN | All rows from both Tables, with NULLs for missing matches |
| CROSS JOIN | All possible Combinations. |