Set 1:

Question 1:

A **data model** is a conceptual framework for organizing and defining the structure, relationships, and constraints of data within a database. It serves as a blueprint for designing and managing databases, ensuring data consistency, integrity, and accessibility. Data models help in understanding the data requirements and how data elements interact with each other.

Types of Data Models

1. **Hierarchical Data Model**: The hierarchical data model organizes data in a tree like structure, where each record (node) has a single parent and can have multiple children. This model is suitable for representing data with a clear hierarchical relationship, such as organizational structures or file system

Example: Consider an organizational chart: CEO VP of Sales Sales Manager 1 Sales Manager 2 VP of Marketing Marketing Manager 1 Marketing Manager 2 In this example, the CEO is the root node, and each VP is a child node of the CEO. Each manager is a child node of their respective VP.

2**. Network Data Model**: The netwo data model is an extension of the hierarchical model, allowing more complex relationships by permitting multiple parent records. It uses a graph structure with nodes (records) and edges (relationships), making it suitable for representing many-to-many relationships.

Example: A university database: Students can enroll in multiple Courses. Each Course can have multiple Students. In this example, the network model allows a student to be linked to multiple courses and vice versa, representing the many-to-many relationship.

3. Relational Data Model: The relational data model organizes data into tables (relations) consisting of rows (tuples) and columns (attributes). It uses primary keys to uniquely identify records and foreign keys to establish relationships between tables. This model is widely used in relational database management systems (RDBMS) like MySQL, PostgreSQL, and Oracle.

Example: A customer database: Customers table: CustomerID, Name, Address Orders table: OrderID, CustomerID, OrderDate Products table: ProductID, ProductName, Price In this example, the Orders table references the Customers table using the CustomerID foreign key, establishing a relationship between customers and their orders.

4**. Entity Relationship (ER) Data Model** :The ER data model represents data using entities (objects) and relationships between them. Entities have attributes, and relationships can be one-to-one, one-to-many, or many-to-many. ER diagrams are used to visually represent the data model.

Example: A library database: Entities: Books, Authors, Borrowers Relationships: Books are written by Authors (many-to-many). Books are borrowed by Borrowers (many-to□many). In this example, the ER model helps visualize the relationships between books, authors, and borrowers, making it easier to design the database schema.

5. **Object-Oriented Data Model** :The object-oriented data model integrates object-oriented programming concepts with database design. It represents data as objects, which encapsulate both data (attributes) and behavior (methods). This model is used in object-oriented databases like ObjectDB and db4o.

Example: A multimedia database: Objects: Image, Video, Audio Attributes: Title, Format, Size Methods: Play, Edit, Delete In this example, each media item is an object with attributes and methods, allowing for more complex data structures and behaviors.

6. Document Data Model:The document data model stores data in document formats (e.g., JSON, XML) and is commonly used in NoSQL databases like MongoDB and CouchDB. Each document contains key-value pairs and can have nested structures, making it suitable for semi-structured data.

Example: A content management system: Document: Article Fields: Title, Author, Content, Tags In this example, each article is stored as a document with fields for title, autho content, and tags, allowing for flexible and dynamic data structure

7. Key-Value Data Model: The key-value data model is a simple model where data is stored as key-value pairs. It is highly efficient for lookups based on keys and is us in NoSQL databases like Redis and DynamoDB.

Example: A caching system: Key: UserID Value: UserProfile In this example, each key represents a uniq identifier (UserID), and the value is the cached data (UserProfile), allowing fast and efficient data retrieval

8. Column-Family Data Model The column-family data model organizes data into column families, where each column family contains rows with a variable number of columns. It is used in wide-column stores like Cassandra and HBase.

Example: A time-series database: Row Key: Timestamp Columns: Sensor1, Sensor2, Sensor3 In this example, each row represents a timestamp, and columns store sensor readings, allowing for efficient storage and retriev of time-series data.

9. Graph Data Model The graph data model represents data as nodes (entities) and edges (relationships). It is suitable for applications with complex, interconnected data and is used in graph databases like Neo4j and OrientDB.

Example: A social network database: Nodes: Users Edges: Friendships, Interactions In this example, nodes represent users, and edges represent friendships or interactions, allowing for efficient traversal a analysis of relationships.

Question 2:

Transaction Operations in DBMS A transaction in a Database Management System (DBMS) is a sequence of operations performed as a single logical unit of work. A transaction ensures data integrity and consistency, even in the presence of system failures or concurrent access by multiple users. Transactions follow the ACID properties: Atomicity, Consistency, Isolation, and Durability

ACID Properties

Atomicity: Ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction is rolled back.

Consistency: Ensures that a transaction transforms the database from one consistent state to another consistent state.

Isolation: Ensures that the operations of a transaction are isolated from other transactions, preventing concurrent transactions from interfering with each other.

Durability: Ensures that the results of a completed transaction are permanently recorded in the database, even in the event of a system failure.

Transaction Operations

1. BEGIN TRANSACTION :The BEGIN TRANSACTION operation marks the start of a transaction. It signals the DBMS to treat the subsequent operations as a single unit of work. **Example**:

```
BEGIN TRANSACTION; -- Operations within the transaction UPDATE Accounts SET
Balance = Balance - 500 WHERE AccountID = 1; UPDATE Accounts SET Balance =
Balance + 500 WHERE AccountID = 2; COMMIT;
```

2. READ: The READ operation retrieves data from the database. It is a non-destructive operation that does not modify the data.

Example:

SELECT * FROM Customers WHERE CustomerID = 1;

3. WRITE :The WRITE operation modifies data in the database. It can be an INSERT, UPDAT or DELETE operation.

Example:

UPDATE Customers SET Balance = Balance + 100 WHERE CustomerID = 1;

4. COMMIT :The COMMIT operation marks the successful end of a transaction. It makes all changes made during the transaction permanent in the database.

Example:

COMMIT;

5. ROLLBACK: The ROLLBACK operation undoes all changes made during the transaction. It is used to revert the database to its previous consistent state in case of an error or failure. **Example**:

ROLLBACK;

6. SAVEPOINT: The SAVEPOINT operation sets a point within a transaction to which a transaction can be rolled back. It allows partial rollback of a transaction.

Example:

SAVEPOINT Savepoint1;

7. SET TRANSACTION: The SET TRANSACTION operation sets the properties of a transaction, such as isolation level and access mode.

Example:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Question 3:

Database Anomalies:Database anomalies are issues that arise when data is inserted, updated, or deleted in a database that is not properly normalized.

The three main types of database anomalies are:

- Insertion Anomaly
- Update Anomaly
- Deletion Anomaly

Innnsertion Anomaly: An insertion anomaly occurs when certain attributes cannot be inserted presence of other attributes. This often happens in a database that is not normalized, where redundant data is stored in a single table.

Example: Consider a table StudentCourses that stores information about students and the courses they are enrolled in:

StudentID	**StudentName**	**CourseID**	**CourseName**
1	Alice	101	Math
2	Bob	102	Science

If we want to add a new course to the database without enrolling any students, we cannot do so without providing a StudentID and StudentName. This is an insertion anomaly because the table design forces us to insert redundant or incomplete data.

Update Anomaly: An update anomaly occurs when changes to data in one place require changes to be made in multiple places. This can lead to data inconsistencies if all changes are not made correctly.

Example: Consider the same StudentCourses table:

StudentID	StudentName	CourseID	CourseName
1	Alice	101	Math
2	Bob	102	Science
3	Alice	103	History

If Alice changes her name to Alicia, we need to update her name in multiple rows. If we forget to update one of the rows, we end up with inconsistent data:

StudentID	StudentName	CourseID	CourseName
1	Alicia	101	Math
2	Bob	102	Science
3	Alice	103	History

Deletion Anomaly: A deletion anomaly occurs when deleting data inadvertently causes the loss of additional data that should not be deleted. This often happens when multiple pieces of information are stored in a single table.

Example: Consider the same StudentCourses table:

StudentID	StudentName	CourseID	CourseName
1	Alice	101	Math
2	Bob	102	Science
3	Alice	103	History

If Alice drops the Math course and we delete the corresponding row, we also lose information about Alice

StudentID	StudentName	CourseID	CourseName
2	Bob	102	Science
3	Alice	103	History

If Alice was only enrolled in the Math course, deleting that row would remove all information about Alice from the database.

Preventing Database Anomalies :To prevent these anomalies, databases should be normalized. . The most common normalization forms are:

First Normal Form (1NF): Ensures that each column contains atomic (indivisible) values and each column contains values of a single type.

Second Normal Form (2NF): Ensures that the table is in 1NF and all non-key attributes are fully functionally dependent on the primary key.

Third Normal Form (3NF): Ensures that the table is in 2NF and all attributes are functionally dependent only on the primary key.

Example of Normalization: Let's normalize the StudentCourses table to prevent anomalies: First Normal Form (1NF): Ensure atomic values and single-type columns.

Second Normal Form (2NF): Split the table into Students and Courses tables.

StudentID	**StudentName**
1	Alice
2	Bob

CourseID	**CourseName**
101	Math
102	Science
103	History

Third Normal Form (3NF): Create a StudentCourses table to link students and courses.

StudentID	**CourseID**
1	101
2	102
1	103

By normalizing the database, we separate the data into related tables, reducing redundancy and preventing anomalies.

Set 2:

Question 4:

A. **Temporal Database and Its Types**: A temporal database is designed to manage and store data related to time instances. It allows the storage and querying of data with respect to time, enabling the tracking of historical, current, and future data states. Temporal databases are particularly useful in applications where the history of data changes is crucial, such as financi systems, scientific research, and historical records.

Types of Tempor Databases:

Uni-temporal Database: Manages a single time dimension, either valid time or transaction time.

Bi-temporal Database: Manages two time dimensions, typically valid time and transaction time.

Tri-temporal Database: Manages three time dimensions, including valid time, transaction time, and decision time.

- Vaalid Time: The time period during which a fact is True.
- Transaction Time: The time period during which a fact is stored.
- Decision Time: The time at which a decision about the fact was made.

Example:

Uni-temporal: A database tracking employee records with only valid time.

Bi-temporal: A database tracking employee records with both valid time and transaction time.

Tri-temporal: A database tracking employee records with valid time, transaction time, and decision time.

B. **SQL3 and Its Important Features SQL3**: also known as SQL:1999, is an extension of the SQL standard that introduced several new features to support advanced database applications. It marked a significant evolution from SQL:92 incorporating object-oriented and procedural programming capabilities.

Important Features of SQL3 :

User-Defined Data Types (UDT): Allows users define custom data types, enabling more complex data structures.

Nest Tables: Supports tables within tables, useful for modeling hierarchical data.

Stored Procedures and Functions: Enhances support for procedural programming with SQL/PSM (Persistent Stored Modules).

Triggers: Allows automatic execution of actions in response to specific events on a table.

Recursive Queries: Supports recursive queries using the WITH RECURSIVE clause, simplifying the management of hierarchical data.

Object-Oriented Features: Introduces object-oriented concepts like inheritance, methods, and constructors, enabling more natural data modelling

Question 5:

Locking is a crucial mechanism in Database Management Systems (DBMS) to ensure data consistency and integrity when multiple transactions access the database concurrently. Different lock modes control the level of access granted to transactions, preventing conflicts and ensuring smooth operation.

Types of Lock Modes

1. **Shared Lock (S)**
2. **Exclusive Lock (X)**
3. **Update Lock (U)**
4. **Intent Lock (I)**
5. **Schema Lock (Sch)**
6. **Bulk Update Lock (BU)**

1. Shared Lock (S)

A shared lock allows multiple transactions to read a data item simultaneously but prevents any transaction from modifying it. This lock mode is used when a transaction only needs to read data without making any changes.

Example:

SELECT * FROM Customers WHERE CustomerID = 1;

In this example, multiple transactions can read the same customer record at the same time, but none can modify it while the shared lock is in place.

2. Exclusive Lock (X)

An exclusive lock allows a single transaction to both read and modify a data item. It prevents other transactions from reading or writing to the locked data item until the lock is released.

Example:

UPDATE Customers SET Balance = Balance + 100 WHERE CustomerID = 1;

In this example, the transaction holds an exclusive lock on the customer record, ensuring no other transaction can access it until the update is complete.

3. Update Lock (U)

An update lock is used when a transaction intends to update a data item. It prevents deadlocks by allowing multiple transactions to read a data item but ensures that only one can update it. If a transaction with an update lock decides to modify the data, the lock is upgraded to an exclusive lock.

Example:

-- Transaction 1

SELECT * FROM Customers WHERE CustomerID = 1 FOR UPDATE;

-- Transaction 2

SELECT * FROM Customers WHERE CustomerID = 1 FOR UPDATE;

In this example, both transactions can read the customer record, but only one can proceed to update it, preventing deadlocks.

4. Intent Lock (I)

Intent locks indicate a transaction's intention to acquire a shared or exclusive lock on a data item at a lower level of granularity. They are used to ensure proper locking hierarchy and prevent conflicts between transactions.

Types of Intent Locks:

- **Intent Shared (IS)**: Indicates intention to acquire a shared lock.
- **Intent Exclusive (IX)**: Indicates intention to acquire an exclusive lock.
- **Shared Intent Exclusive (SIX)**: Indicates a shared lock on the current level and an intention to acquire exclusive locks at a lower level.

Example:

-- Transaction 1

LOCK TABLE Customers IN IS MODE;

-- Transaction 2

LOCK TABLE Customers IN IX MODE;

In this example, the intent locks ensure that transactions follow the proper locking hierarchy, preventing conflicts.

5. Schema Lock (Sch)

Schema locks are used to lock the schema of a table, preventing changes to the table structure while the lock is in place. There are two types of schema locks:

- **Schema Modification Lock (Sch-M)**: Acquired when a transaction modifies the table schema (e.g., adding a column).
- **Schema Stability Lock (Sch-S)**: Acquired when a transaction queries the table schema without modifying it.

Example:

-- Transaction 1

ALTER TABLE Customers ADD COLUMN Email VARCHAR(255);

-- Transaction 2

SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'Customers';

In this example, the schema modification lock prevents other transactions from querying or modifying the table schema while the alteration is in progress.

6. Bulk Update Lock (BU)

Bulk update locks are used during bulk operations to improve performance by reducing lock contention. They allow multiple rows to be locked for bulk insert, update, or delete operations.

Example:

BULK INSERT Customers FROM 'customers.csv' WITH (ROWTERMINATOR = '\n');

In this example, the bulk update lock ensures that the bulk insert operation can proceed efficiently without interference from other transactions.

Question 6:

Fragmentation in Databases

Fragmentation is the process of dividing a database into smaller, more manageable pieces called fragments. This technique is commonly used in distributed databases to improve performance, manageability, and availability. By fragmenting the database, data can be stored closer to where it is needed, reducing access time and improving query performance.

Types of Fragmentation

Horizontal Fragmentation

Vertical Fragmentation

Mixed (Hybrid) Fragmentation

1. Horizontal Fragmentation

Horizontal fragmentation divides a table into subsets of rows based on a specified condition. Each fragment contains a subset of the rows from the original table, and these fragments can be distributed across different locations.

Example: Consider a Customer table with the following data:

CustomerID	Name	Region
1	Alice	North
2	Bob	South
3	Carol	North
4	Dave	South

Horizontal fragmentation based on the Region column would result in two fragments:

Fragment 1 (North Region):

SELECT * FROM Customer WHERE Region = 'North';

CustomerID	Name	Region
1	Alice	North
3	Carol	North

Fragment 2 (South Region):

SELECT * FROM Customer WHERE Region = 'South';

CustomerID	Name	Region
2	Bob	South
4	Dave	South

2. Vertical Fragmentation

Vertical fragmentation divides a table into subsets of columns. Each fragment contains a subset of the columns from the original table, and these fragments can be distributed across different locations.

Example: Consider the same Customer table:

CustomerID	Name	Region
1	Alice	North
2	Bob	South
3	Carol	North
4	Dave	South

Vertical fragmentation based on the columns would result in two fragments:

Fragment 1 (Customer Info):

SELECT CustomerID, Name FROM Customer;

CustomerID	Name
1	Alice
2	Bob
3	Carol
4	Dave

Fragment 2 (Region Info):

SELECT CustomerID, Region FROM Customer;

CustomerID	Region
1	North
2	South
3	North
4	South

3. Mixed (Hybrid) Fragmentation

Mixed fragmentation combines both horizontal and vertical fragmentation. This approach allows for more flexible and efficient data distribution by creating fragments that are subsets of both rows and columns.

Example: Consider the same Customer table:

CustomerID	Name	Region
1	Alice	North
2	Bob	South
3	Carol	North
4	Dave	South

Mixed fragmentation could result in the following fragments:

Fragment 1 (North Region, Customer Info):

```
SELECT CustomerID, Name FROM Customer WHERE Region = 'North';
```

CustomerID	Name
1	Alice
3	Carol

Fragment 2 (South Region, Customer Info):

```
SELECT CustomerID, Name FROM Customer WHERE Region = 'South';
```

CustomerID	Name
2	Bob
4	Dave

Fragment 3 (North Region, Region Info):

```
SELECT CustomerID, Region FROM Customer WHERE Region = 'North';
```

CustomerID	Region
1	North
3	North

Fragment 4 (South Region, Region Info):

```
SELECT CustomerID, Region FROM Customer WHERE Region = 'South';
```

CustomerID	Region
2	South
4	South

Benefits of Fragmentation

Improved Performance: By storing data closer to where it is needed, fragmentation reduces access time and improves query performance.

Enhanced Manageability: Smaller fragments are easier to manage and maintain than a large, monolithic database.

Increased Availability: Distributing fragments across multiple locations can improve data availability and fault tolerance.

Security: Sensitive data can be isolated in specific fragments, enhancing data security.

Conclusion