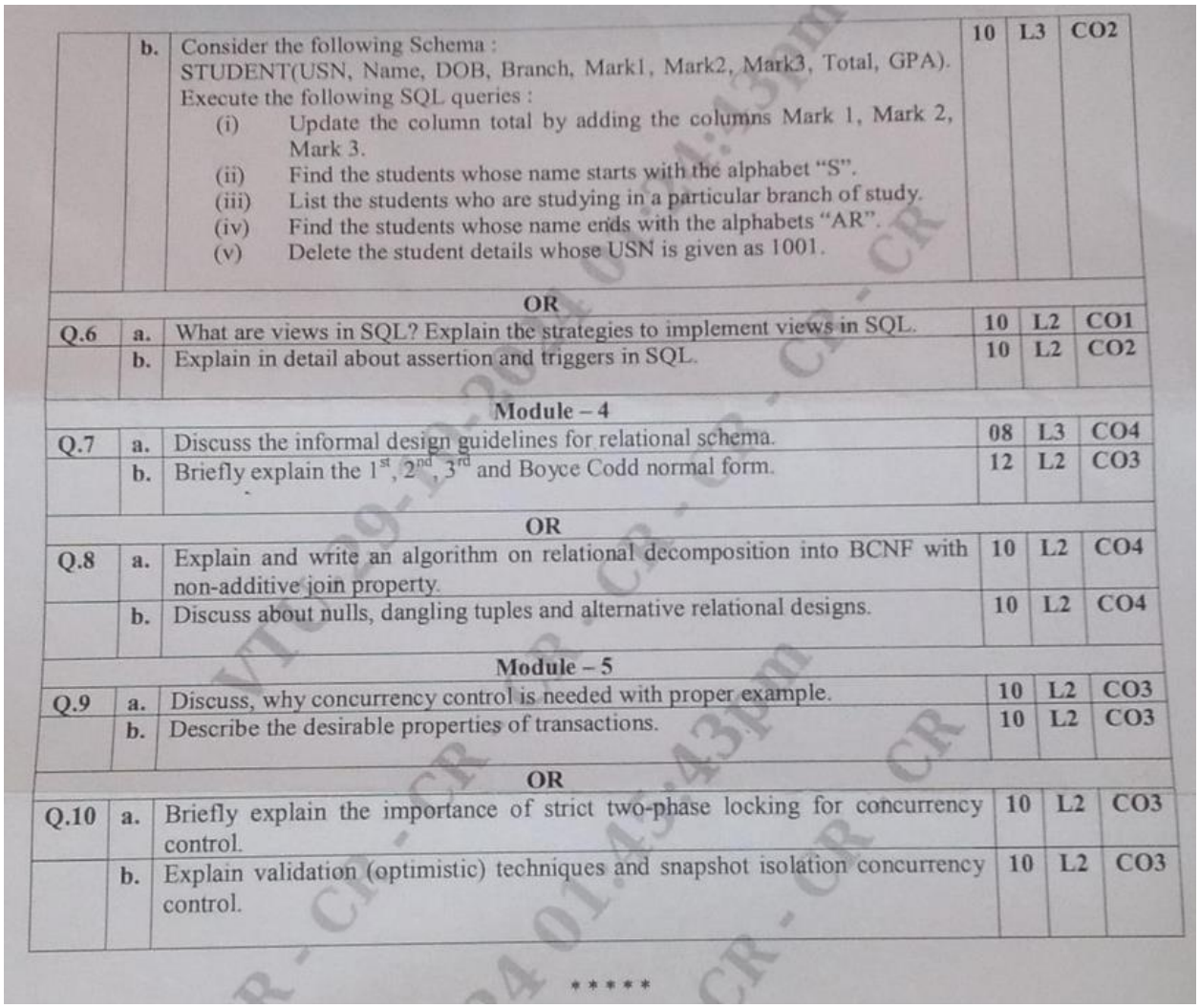
SecondSemester MCA Degree Examination, June/July 2024
Database Management Systems

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	CO
Q.1	a.	Define database management system. List and explain its characteristics.	10	L1	CO1
	b.	Explain 3-schema architecture. What do you mean by data independence? Explain briefly about different types of it.	10	L1	CO1
		OR			
Q.2	a.	With a neat diagram, explain the DBMS component modules.	10	L1	CO1
	b.	Briefly explain types of attributes in E-R model.	04	L1	CO1
	c.	Construct an ER-diagram for company database with proper assumption.	06	L3	CO3
		Module – 2			
Q.3	a.	Discuss about domain constraint and key constraint and constraint on NULL values.	10	L2	CO1
	b.	Explain the following with example : (i) SELECT (ii) PROJECT (iii) RENAME (iv) Division Operation	10	L2	CO4
		OR			
Q.4	a.	Consider the following schema : EMPLOYEE(Fname, Minit, Lname, Ssn, Bdate, Address, Sex, Salary, Dno) DEPARTMENT(Dname, Dnumber, Mgr-ssn, Mgr-start, date) PROJECT(Pname, Pnumber, Plocation, Dnum) WORKS_ON(Essn, Pno, Hours) Write the queries in relational algebra for following : (i) Reterieve tuples for all employees who either work in department 4 and make over $25,000 per year or work in department 5 and make over $30,000. (ii) Retrieve each employee's first and last name and salary. (iii) Retrieve the names of employees who work on all the projects that 'John Smith' works on. (iv) Retrieve the name and address of all employees who work for the 'Research' department.	10	L3	CO4
	b.	Summarize the steps involved in relational database using ER-to-Relational mapping.	10	L3	CO4
		Module – 3			
Q.5	a.	Bring out the different clauses of SELECT-FROM-WHERE-GROUP BY-HAVING with an example for each.	10	L2	CO2



	b.	Consider the following Schema : STUDENT(USN, Name, DOB, Branch, Mark1, Mark2, Mark3, Total, GPA). Execute the following SQL queries : (i) Update the column total by adding the columns Mark 1, Mark 2, Mark 3. (ii) Find the students whose name starts with the alphabet "S". (iii) List the students who are studying in a particular branch of study. (iv) Find the students whose name ends with the alphabets "AR". (v) Delete the student details whose USN is given as 1001.	10	L3	CO2
		OR			
Q.6	a.	What are views in SQL? Explain the strategies to implement views in SQL.	10	L2	CO1
	b.	Explain in detail about assertion and triggers in SQL.	10	L2	CO2
		Module – 4			
Q.7	a.	Discuss the informal design guidelines for relational schema.	08	L3	CO4
	b.	Briefly explain the 1st, 2nd, 3rd and Boyce Codd normal form.	12	L2	CO3
		OR			
Q.8	a.	Explain and write an algorithm on relational decomposition into BCNF with non-additive join property.	10	L2	CO4
	b.	Discuss about nulls, dangling tuples and alternative relational designs.	10	L2	CO4
		Module – 5			
Q.9	a.	Discuss, why concurrency control is needed with proper example.	10	L2	CO3
	b.	Describe the desirable properties of transactions.	10	L2	CO3
		OR			
Q.10	a.	Briefly explain the importance of strict two-phase locking for concurrency control.	10	L2	CO3
	b.	Explain validation (optimistic) techniques and snapshot isolation concurrency control.	10	L2	CO3

* * * * *

1a. Define DBMS, list, and explain its characteristics.

Definition of DBMS:

A **Database Management System (DBMS)** is a software system that enables users to store, manage, and retrieve data in a structured manner. It provides an interface for interacting with databases, supporting operations like **create**, **read**, **update**, and **delete** (CRUD).

2. Characteristics of a DBMS:

1. **Data Independence**:
    - Users and applications are isolated from changes in the database schema.
    - **Logical Data Independence**: Changes to the logical schema do not affect the external schema.

    - **Physical Data Independence**: Changes to physical storage do not affect the logical schema.
2. **Data Redundancy Control**:
    - Minimizes duplicate data by centralizing storage and using normalization techniques.
3. **Data Integrity**:
    - Ensures data accuracy and consistency using integrity constraints:
        - **Entity Integrity**: Unique identification (e.g., primary key).
        - **Referential Integrity**: Maintains relationships between tables (e.g., foreign key).
        - **Domain Integrity**: Ensures valid values in columns (e.g., data type constraints).
4. **Data Security**:
    - Provides user authentication, authorization, and data encryption to protect sensitive data.
5. **Concurrency Control**:
    - Allows multiple users to access the database simultaneously without conflicts, using mechanisms like **locking** and **transaction management**.
6. **Backup and Recovery**:
    - Supports data backup and recovery features to prevent data loss due to system failures or crashes.
7. **Data Abstraction**:
    - Hides the complexity of data storage and representation at three levels:
        - **Physical Level**: How data is stored.
        - **Logical Level**: What data is stored (e.g., tables, schemas).
        - **View Level**: User-specific views of the data.
8. **Query Processing**:
    - Supports high-level query languages (e.g., **SQL**) for efficient data retrieval and manipulation.
9. **Transaction Management**:
    - Ensures that database transactions are **ACID-compliant** (Atomicity, Consistency, Isolation, Durability).
10. **Multi-user Support**:
    - Manages concurrent access by multiple users, ensuring data consistency and preventing conflicts.
11. **Efficiency and Performance**:
    - Optimizes data storage, retrieval, and querying through techniques like indexing and query optimization.

1b. Explain 3 schema architecture. What do you mean by data independence? Explain briefly about its types

The **3-Schema Architecture** is a conceptual framework for database design that separates the database into three levels to promote **data independence** and to ensure that users interact with the database without needing to know its underlying complexities.

1. **Internal Schema (Physical Schema)**:
    - **Definition**: This schema defines **how the data is physically stored** on the storage devices (e.g., hard drives, SSDs), including the file organization, indexing, and storage structures.
    - **Purpose**: It focuses on the efficient storage and retrieval of data, and it is not visible to end-users or application programs.
    - **Example**: It involves details such as disk storage methods, indexing techniques, and how data is organized on disk.
2. **Conceptual Schema (Logical Schema)**:
    - **Definition**: This schema provides a **logical view** of the entire database, describing the structure of the data without considering physical storage. It defines tables, relationships, views, and constraints.
    - **Purpose**: It acts as an abstract representation of the database that is independent of both physical storage and user views.
    - **Example**: It specifies that a "Student" table contains "StudentID", "Name", and "Course" attributes, and that there is a relationship between the "Student" table and the "Course" table.
3. **External Schema (View Schema)**:
    - **Definition**: This schema defines how data is presented to users or applications. It represents **specific user views** and their interactions with the database, tailored to different needs.
    - **Purpose**: It allows for customization of data views based on different user roles, and it hides the complexity of the internal and conceptual schemas.
    - **Example**: A university's administrative staff may have a view that shows only "StudentID" and "Name", while a professor may have a view that includes "StudentID", "Name", and "Grades".

Relationship Between the Three Schemas

- The **Internal Schema** handles the physical storage details.
- The **Conceptual Schema** defines the logical structure of the data, independent of physical storage.
- The **External Schema** provides a user-specific view, which is a tailored representation of the data, independent of the logical and physical structures.

Data Independence

Data Independence is the ability to modify the schema (structure) of a database without affecting the application programs or users interacting with the database. This is one of the key advantages of using a DBMS, as it provides flexibility in managing and evolving the database without disrupting its users.

There are two types of data independence:

1. **Logical Data Independence**:
    - **Definition**: The ability to change the **logical schema** (how the data is structured or organized) without affecting the **external schema** (user views) or application programs.
    - **Example**: Adding new attributes to a table (like adding a "PhoneNumber" field to a "Student" table) or changing relationships between tables (e.g., adding a new foreign key) without requiring changes to how end users interact with the database.
    - **Why it matters**: It allows the logical structure of the database to evolve without impacting the users or applications that rely on it.
2. **Physical Data Independence**:
    - **Definition**: The ability to change the **physical schema** (how the data is stored on disk, indexing, file organization) without affecting the **logical schema** or the **external schema**.
    - **Example**: Moving data from one storage medium to another, changing the indexing method, or reorganizing the files to optimize performance, all without affecting the logical structure of the database or the user views.
    - **Why it matters**: It allows changes to be made to improve performance or storage management without disrupting the data access layer for users or applications.

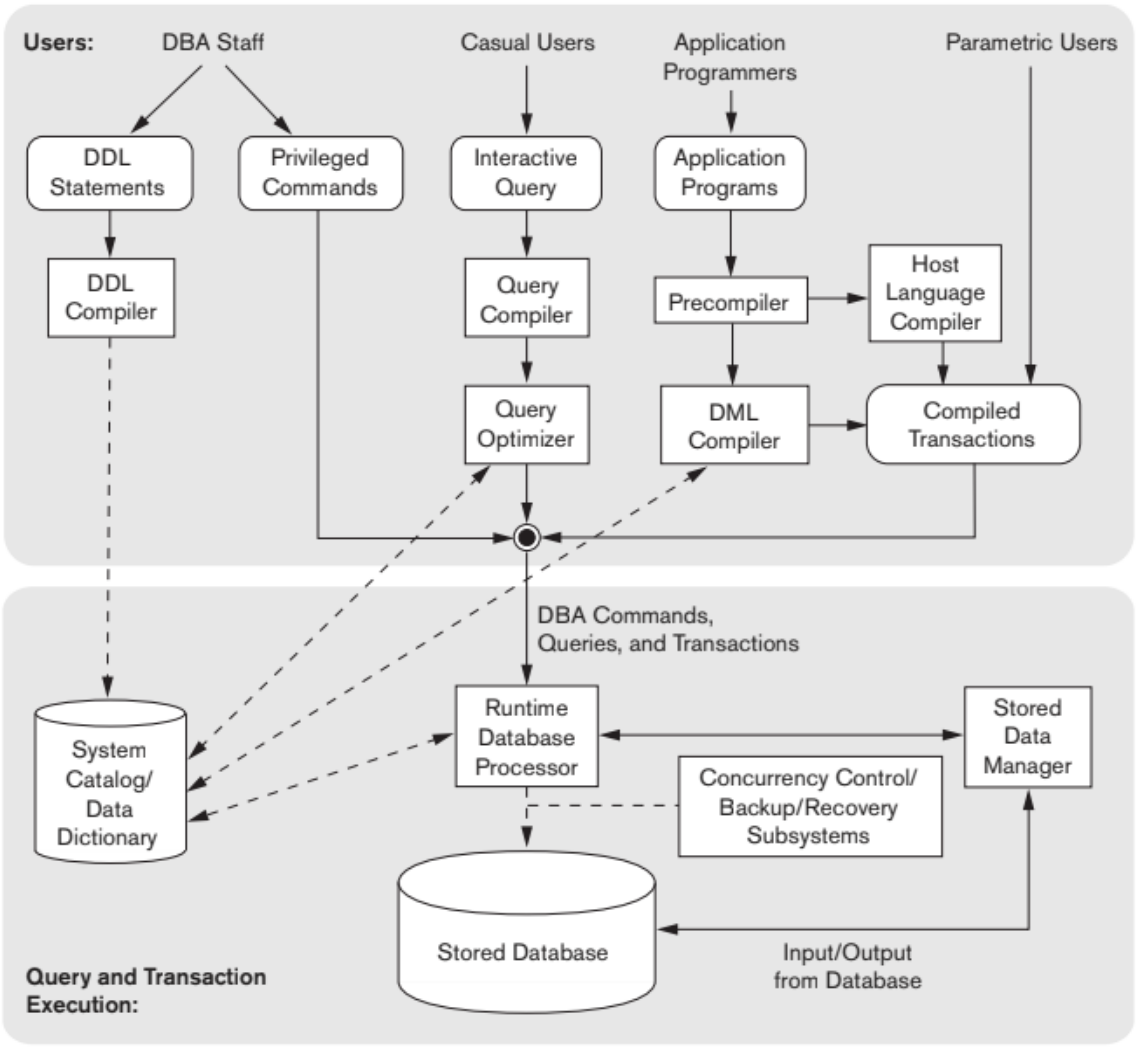
2a. With a neat diagram explain DBMS component modules

A **Database Management System (DBMS)** consists of several interrelated **components or modules** that work together to manage data efficiently, provide access control, ensure data integrity, and optimize performance. Below is an explanation of these core DBMS components, followed by a diagram that illustrates how they are connected.

Core DBMS Component Modules

1. **DBMS Engine**:
    - **Definition**: The DBMS engine is the **core** of the system that handles all the database operations, including reading, writing, and managing data on storage devices.
    - **Responsibilities**:
        - Managing data storage, retrieval, and manipulation.
        - Optimizing queries for efficient data retrieval.
        - Ensuring transactions follow ACID properties (Atomicity, Consistency, Isolation, Durability).
2. **Database Schema**:
    - **Definition**: The schema defines the logical structure of the database, including tables, fields, data types, relationships, and constraints (e.g., primary keys, foreign keys).
    - **Responsibilities**:
        - Organizing the data into a structured format.
        - Enforcing rules about the organization and relationships of data.
3. **Query Processor**:
    - **Definition**: The query processor interprets and executes user queries, typically written in a query language like SQL (Structured Query Language).
    - **Responsibilities**:

        - Parsing queries to check for syntax and semantic errors.
        - Optimizing queries to ensure efficient data retrieval.
        - Translating SQL queries into low-level instructions that the DBMS engine can execute.
4. **Transaction Manager**:
    - **Definition**: The transaction manager ensures that all transactions are processed correctly, adhering to the ACID properties to guarantee the integrity and consistency of the database.
    - **Responsibilities**:
        - Handling multiple transactions simultaneously while maintaining consistency.
        - Ensuring rollback in case of errors or system crashes.
        - Managing concurrency control to prevent conflicts between transactions (e.g., using locks or timestamps).
5. **Storage Manager**:
    - **Definition**: The storage manager is responsible for managing the physical storage of data in the database, including how data is stored on disk or other storage mediums.
    - **Responsibilities**:
        - Managing the physical allocation of space on storage devices.
        - Handling data structures like indexes, tables, and files.
        - Ensuring that data is stored efficiently and can be quickly retrieved.
        - Managing data buffers and cache for fast access.
6. **Data Dictionary**:
    - **Definition**: The data dictionary, also known as the **system catalog**, stores metadata (data about data). This includes details about the database schema, tables, columns, constraints, and relationships.
    - **Responsibilities**:
        - Storing and maintaining information about the structure of the database.
        - Supporting query processing by providing metadata needed for optimizing queries.
        - Keeping track of data types, constraints, indexes, and relationships between tables.
7. **Authorization and Access Control Manager**:
    - **Definition**: This module is responsible for ensuring that only authorized users and applications can access and modify data.
    - **Responsibilities**:
        - Enforcing security policies and user access controls (e.g., granting/revoking permissions).
        - Managing user authentication (e.g., usernames, passwords, roles).
        - Auditing and logging access to the database for security monitoring.
8. **Recovery Manager**:
    - **Definition**: The recovery manager ensures that the database can recover from failures such as system crashes, hardware failures, or transaction errors.
    - **Responsibilities**:
        - Maintaining logs of transactions (write-ahead logs).
        - Performing rollbacks and rollforwards to restore the database to a consistent state after a failure.
        - Ensuring the durability of committed transactions.



2b. Briefly explain types of attributes in ER Models

In an **Entity-Relationship (ER) model**, attributes are used to describe the properties or characteristics of entities (objects) and relationships. Different types of attributes serve various purposes in defining the characteristics of entities and relationships. Below is a brief explanation of the main **types of attributes** in the ER model:

1. Simple (Atomic) Attribute

- **Definition**: An attribute that cannot be subdivided into smaller parts.
- **Example**: A "Phone Number" or "Age" is a simple attribute, as it is indivisible in the context of the ER model.

2. Composite Attribute

- **Definition**: An attribute that can be divided into smaller sub-parts, each of which represents a meaningful piece of information.
- **Example**: An "Address" attribute could be broken down into "Street", "City", "State", and "Zip Code".
- **Representation**: Represented as a group of attributes connected to a single main attribute.

3. Multivalued Attribute

- **Definition**: An attribute that can have multiple values for a single entity or relationship.
- **Example**: A "Phone Numbers" attribute for a "Person" entity, where a person can have multiple phone numbers.
- **Representation**: Represented by a double ellipse in an ER diagram.

4. Derived Attribute

- **Definition**: An attribute whose value is derived or calculated from other attributes in the database.
- **Example**: "Age" can be derived from the "Date of Birth" attribute.
- **Representation**: Represented by a dashed ellipse in the ER diagram.

5. Key Attribute

- **Definition**: An attribute (or a combination of attributes) that uniquely identifies an entity within an entity set.
- **Example**: A "StudentID" attribute in a "Student" entity set uniquely identifies each student.
- **Representation**: Represented as underlined text in an ER diagram.

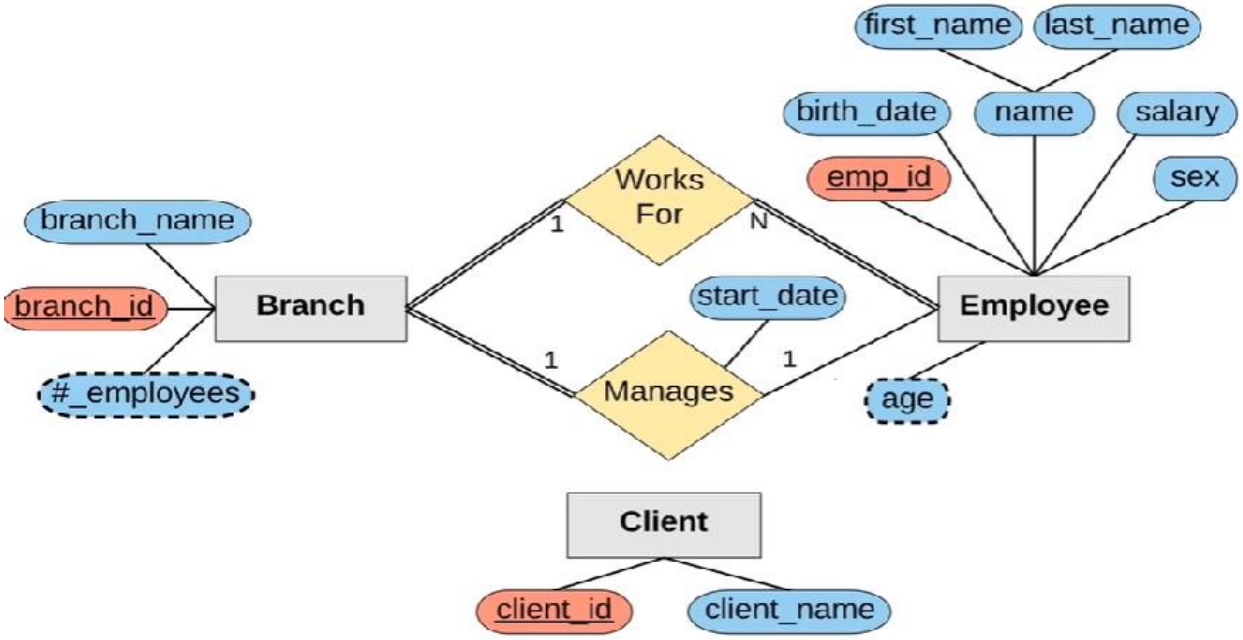
6. Single-valued Attribute

- **Definition**: An attribute that holds only a single value for each entity instance.
- **Example**: A "Date of Birth" attribute for a "Person" entity is typically single-valued.
- **Representation**: Represented as a regular ellipse in an ER diagram.

7. Complex Attribute

- **Definition**: An attribute that is a combination of other attributes (it may include composite and multivalued attributes).
- **Example**: A "Full Name" attribute could be a combination of "First Name" and "Last Name" (both simple attributes).
- **Representation**: Represented by combining simple and composite attributes in the ER diagram.

2c. Briefly explain types of attributes in ER Models



3a. Discuss about domain constraint, key constraint and constraints on NULL values

In a **Database Management System (DBMS)**, **constraints** are rules applied to ensure that the data entered into the database is accurate, consistent, and adheres to the desired properties. Three important types of constraints are:

1. Domain Constraint

Definition: A **domain constraint** ensures that the values of an attribute (or column) are from a specific, predefined set of values. It restricts the type, range, and format of the data that can be stored in a database column. The "domain" of an attribute is essentially its allowable set of values.

- **Purpose**: To enforce data validity by ensuring that only appropriate and valid values are inserted into the database.
- **Example**:
  - If you have an attribute "Age" in a table, the domain constraint could specify that the values for "Age" must be integers between 0 and 150.
  - For a "Gender" attribute, the domain constraint might only allow values like 'Male', 'Female', and 'Other'.
- **Domain Constraint Features**:
  - **Data Type**: Specifies the type of data allowed (e.g., INT, VARCHAR, DATE).
  - **Range**: Specifies a valid range of values (e.g., numbers between 0 and 100).

- **Set of Allowed Values**: Specifies a predefined set of valid values (e.g., only 'Yes' or 'No' for a "Confirmed" field).

```
CREATE TABLE Employees (
    ID INT,
    Age INT CHECK (Age >= 18 AND Age <= 65),
    Gender VARCHAR(10) CHECK (Gender IN ('Male', 'Female'))
);
```

2. Key Constraint

Definition: A **key constraint** ensures that each record in a table can be uniquely identified. The primary key and foreign key are the main types of key constraints.

- **Primary Key Constraint**:
  - **Purpose**: Ensures that each record in a table is uniquely identified. It is a set of one or more attributes (columns) that uniquely identify each row in the table.
  - **Properties**:
    - Each primary key must have unique values.
    - A primary key cannot have NULL values.
  - **Example**: A "StudentID" attribute in a "Student" table is often the primary key because each student has a unique ID.
- **Foreign Key Constraint**:
  - **Purpose**: Enforces referential integrity between two tables. A foreign key in one table points to the primary key of another table, establishing a relationship between the two tables.
  - **Properties**:
    - Foreign keys allow values that either match a value in the referenced primary key or are NULL (depending on the relationship's optionality).
    - Ensures that foreign key values always correspond to existing primary key values or are NULL (if allowed).
  - **Example**: In an "Enrollment" table, the "StudentID" could be a foreign key that points to the "Student" table's primary key.
- **Example in SQL (Primary and Foreign Key)**:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    Age INT
);
```

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    Course VARCHAR(100),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
```

); In a **Database Management System (DBMS)**, **constraints** are rules applied to ensure that the data entered into the database is accurate, consistent, and adheres to the desired properties. Three important types of constraints are:

1. Domain Constraint

Definition: A **domain constraint** ensures that the values of an attribute (or column) are from a specific, predefined set of values. It restricts the type, range, and format of the data that can be stored in a database column. The "domain" of an attribute is essentially its allowable set of values.

- **Purpose**: To enforce data validity by ensuring that only appropriate and valid values are inserted into the database.
- **Example**:
  - If you have an attribute "Age" in a table, the domain constraint could specify that the values for "Age" must be integers between 0 and 150.
  - For a "Gender" attribute, the domain constraint might only allow values like 'Male', 'Female', and 'Other'.
- **Domain Constraint Features**:
  - **Data Type**: Specifies the type of data allowed (e.g., INT, VARCHAR, DATE).
  - **Range**: Specifies a valid range of values (e.g., numbers between 0 and 100).
  - **Set of Allowed Values**: Specifies a predefined set of valid values (e.g., only 'Yes' or 'No' for a "Confirmed" field).
- **Example in SQL**:

```
sql
Copy code
CREATE TABLE Employees (
   ID INT,
   Age INT CHECK (Age >= 18 AND Age <= 65),
   Gender VARCHAR(10) CHECK (Gender IN ('Male', 'Female'))
);
```

2. Key Constraint

Definition: A **key constraint** ensures that each record in a table can be uniquely identified. The primary key and foreign key are the main types of key constraints.

- **Primary Key Constraint**:
  - **Purpose**: Ensures that each record in a table is uniquely identified. It is a set of one or more attributes (columns) that uniquely identify each row in the table.
  - **Properties**:
    - Each primary key must have unique values.
    - A primary key cannot have NULL values.
  - **Example**: A "StudentID" attribute in a "Student" table is often the primary key because each student has a unique ID.
- **Foreign Key Constraint**:
  - **Purpose**: Enforces referential integrity between two tables. A foreign key in one table points to the primary key of another table, establishing a relationship between the two tables.
  - **Properties**:
    - Foreign keys allow values that either match a value in the referenced primary key or are NULL (depending on the relationship's optionality).
    - Ensures that foreign key values always correspond to existing primary key values or are NULL (if allowed).
  - **Example**: In an "Enrollment" table, the "StudentID" could be a foreign key that points to the "Student" table's primary key.
- **Example in SQL (Primary and Foreign Key)**:

```
sql
Copy code
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(100),
   Age INT
);

CREATE TABLE Enrollments (
   EnrollmentID INT PRIMARY KEY,
   StudentID INT,
   Course VARCHAR(100),
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

3. Constraint on NULL Values

Definition: A **NULL value constraint** defines whether or not NULL values (representing "no value" or "unknown") can be stored in a particular attribute (column) of a table. **NULL** is not the same as an empty string (') or zero (0); it means that the value is undefined or missing.

- **NOT NULL Constraint**:
  - **Purpose**: Ensures that an attribute (column) cannot have NULL values. Every record in the table must have a value for that attribute.
  - **Example**: A "Username" attribute in a "User" table would typically have a NOT NULL constraint, ensuring that every user has a valid username.
  - **Example in SQL**:

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) NOT NULL,
    Password VARCHAR(50)
);
```

3b. Explain the following with example

i. **SELECT**
ii. **PROJECT**
iii. **RENAME**
iv. **Division operation**

In the context of **relational databases** and **relational algebra**, the following are the four fundamental operations commonly used to query and manipulate data. Let's look at each one with an explanation and example:

i. SELECT (σ)

Definition: The **SELECT** operation (denoted as **σ**) is used to filter rows from a relation based on a specified condition (predicate). It selects a subset of tuples (rows) that satisfy the condition from the relation.

- **Syntax**:

  σcondition(R)\sigma_{condition}(R)σcondition(R)

  Where:

  - **R** is the relation (table).
  - **condition** is the logical predicate that rows must satisfy.
- **Purpose**: To extract records that meet a specific condition from a table.
- **Example**: Consider a relation **Employees**:

  | EmpID | Name | Department | Salary |

  |-------|-------|------------|--------|

  | 101 | Alice | HR | 50000 |

  | 102 | Bob | IT | 60000 |

| 103 | Carol | IT | 55000 |

| 104 | Dave | HR | 52000 |

Query: Select all employees from the **IT** department.

σDepartment='IT'(Employees)\sigma_{Department = 'IT'}(Employees)σDepartment='IT'(Employees)

ii. PROJECT (π)

Definition: The **PROJECT** operation (denoted as **π**) is used to select specific columns (attributes) from a relation, effectively creating a new relation with only the desired attributes, removing duplicates.

- **Syntax**:

  πattribute1,attribute2,…(R)\pi_{attribute_1, attribute_2, \ldots}(R)πattribute1,attribute2,…(R)

  Where:

  - **R** is the relation (table).
  - **attribute_1, attribute_2, ...** are the columns to be selected.
- **Purpose**: To extract a subset of columns from a table.
- **Example**: Consider the same **Employees** relation:

| EmpID | Name | Department | Salary |

|-------|-------|------------|--------|

| 101 | Alice | HR | 50000 |

| 102 | Bob | IT | 60000 |

| 103 | Carol | IT | 55000 |

| 104 | Dave | HR | 52000 |

Query: Select only the **Name** and **Department** columns.

πName,Department(Employees)\pi_{Name,Department}(Employees)πName,Department(Employees)

iii. RENAME (ρ)

Definition: The **RENAME** operation (denoted as **ρ**) is used to rename the relation or its attributes (columns). It allows the renaming of a table or the attributes for use in other operations or queries.

- **Syntax**:

  ρnew_name(R)\rho_{new_name}(R)ρnew_name(R)

  Or, to rename attributes:

  ρnew_name_of_attributes(R)\rho_{new_name_of_attributes}(R)ρnew_name_of_attributes(R)

- **Purpose**: To rename a relation or its attributes temporarily, often used when combining multiple relations or when working with complex queries.
- **Example**: Consider the **Employees** relation:

  | EmpID | Name | Department | Salary |

  |-------|-------|------------|--------|

  | 101 | Alice | HR | 50000 |

  | 102 | Bob | IT | 60000 |

  | 103 | Carol | IT | 55000 |

  | 104 | Dave | HR | 52000 |

  Query: Rename the **Employees** table to **Staff**, and the "Name" column to "Employee_Name".

  ρStaff(Name→Employee_Name)(Employees)\rho_{Staff(Name \rightarrow Employee_Name)}(Employees)ρStaff(Name→Employee_Name)(Employees)

4b. Summarize the steps involved in relational database using ER to relational mapping

Relational database design using Entity-Relationship (ER) to relational mapping involves converting an ER diagram into a set of relational tables (schemas). Here are the key steps in the process:

1. Map Entity Types to Tables

- Each **entity** in the ER diagram becomes a **table** in the relational schema.
- The attributes of the entity become the **columns** in the table.
- The **primary key** of the entity (often underlined in the ER diagram) becomes the **primary key** of the table.

2. Map Relationship Types to Tables

- **One-to-one (1:1) relationships**: Add the primary key of one entity as a foreign key in the other entity's table.
- **One-to-many (1:M) relationships**: Add the primary key of the "one" side entity as a foreign key in the "many" side entity's table.
- **Many-to-many (M:N) relationships**: Create a new table that includes the primary keys of both related entities as foreign keys. The combination of these foreign keys usually serves as the **composite primary key** for the new table.

3. Map Attribute Types

- For each attribute in the ER diagram, map it to a **column** in the corresponding table.
- If an attribute is multi-valued (can have multiple values), create a new table to represent this, where the primary key of the original entity is included as a foreign key, along with the multi-valued attribute.

4. Map Weak Entities

- For **weak entities** (entities that do not have a primary key on their own), include the **primary key** of the owner entity as a foreign key in the weak entity's table.
- The combination of the foreign key and the weak entity’s own partial key forms the **primary key** for the table.

5. Handle Inheritance (if applicable)

- If the ER diagram includes **generalization or specialization** (inheritance relationships), map the superclasses and subclasses:
    - **Single table** inheritance: Create a single table for the superclass and include all attributes of both the superclass and its subclasses.
    - **Multiple table** inheritance: Create a table for each subclass and include the primary key of the superclass as a foreign key in each subclass table.
    - **Class table** inheritance: Create a table for the superclass and separate tables for each subclass, with foreign keys to the superclass.

6. Normalization (optional but recommended)

- After mapping the ER diagram to relational tables, you may apply **normalization** rules (up to 3NF or BCNF) to eliminate redundancy and ensure data integrity.
- This involves decomposing tables, eliminating transitive dependencies, and ensuring that every non-key attribute is fully functionally dependent on the primary key.

7. Define Constraints

- Specify any **constraints** (like NOT NULL, UNIQUE, etc.) for each column, based on the requirements specified in the ER diagram.
- Ensure that **foreign key constraints** are defined to maintain referential integrity between related tables.

5a. Bring out the different clauses of SELECT-FROM-WHERE-GROUP BY-HAVING with an example for each

1. SELECT Clause

The SELECT clause is used to specify which columns or expressions you want to retrieve from the database.

- **Syntax**: SELECT column1, column2, ...
- **Example**

SELECT name, age

FROM employees;

2. FROM Clause

The FROM clause is used to specify the table or tables from which to retrieve the data.

- **Syntax**: FROM table_name
- **Example**

SELECT name, department

FROM employees;

3. WHERE Clause

The WHERE clause filters rows based on a specified condition, restricting which rows will be included in the result.

- **Syntax**: WHERE condition
- **Example**

SELECT name, age

FROM employees

WHERE age > 30;

4. GROUP BY Clause

The GROUP BY clause groups rows that have the same values in specified columns into summary rows. It is often used with aggregate functions like COUNT(), SUM(), AVG(), etc.

- **Syntax**: GROUP BY column1, column2, ...
- **Example**

SELECT department, COUNT(*)

FROM employees

GROUP BY department;

5. HAVING Clause

The HAVING clause is used to filter the results of a GROUP BY query. It is similar to the WHERE clause, but WHERE filters rows before grouping, while HAVING filters after grouping.

- **Syntax**: HAVING condition
- **Example**

SELECT department, COUNT(*)

FROM employees

GROUP BY department

HAVING COUNT(*) > 5;

5b. Consider the following schema:

STUDENT (USN, name, date_of_birth, branch, mark1, mark2, mark3, total, GPA)

Execute the following queries:

i. Update the column total by adding the columns mark1, mark2, mark3.

create table student(usn int primary key, sname varchar(15),dob varchar(15),

branch varchar(15),mark1 int,mark2 int,mark3 int, total float, gpa float);

Update student set total=mark1+mark2+mark3;

ii. Find the students whose name starts with the alphabet "S".

Select sname from student where sname like 'S%';

iii. List the students who are studying in a particular branch of study.

Select * from student where branch="MCA";

iv. Find the students whose name ends with the alphabets "AR".

Select sname from student where sname like '%AR';

v. Delete the student details whose USN is given as 1001.

Delete from student where usn'1001';

6a. What are views in SQL? Explain the strategies to implement views in SQL

A **view** in SQL is a virtual table that provides a way to represent data from one or more tables. It does not store data itself but contains a SQL query that pulls data from the underlying tables. Views simplify complex queries by encapsulating them into reusable virtual tables, and they provide a way to control access to sensitive data by exposing only specific columns or rows.

Key Characteristics of Views:

- **Virtual Table**: A view does not store data; instead, it dynamically retrieves data when queried.
- **Simplifies Queries**: Views can simplify complex queries by encapsulating joins, aggregations, and filters into a single query.
- **Security**: Views can be used to restrict access to sensitive data, as they can expose only certain columns or rows from the underlying tables.
- **Reusability**: Views can be reused in other queries or applications, making them useful for standardizing data access patterns.

Creating Views:

A view is created using the CREATE VIEW statement in SQL.

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2, ...

FROM table_name

WHERE condition;

Example:

CREATE VIEW employee_details AS

SELECT name, department, salary

FROM employees

WHERE status = 'active';

Types of Views:

1. **Simple View**:
    - A simple view is based on a single table and does not include any complex SQL features like aggregation, joins, or subqueries.
    - Example

CREATE VIEW active_employees AS

SELECT name, department

FROM employees

WHERE status = 'active';

Complex View:

- A complex view is based on multiple tables and often involves joins, aggregations, or subqueries.
- Example

CREATE VIEW department_salaries AS

SELECT e.department, AVG(e.salary) AS avg_salary

FROM employees e

GROUP BY e.department;

Materialized View:

- A materialized view is a type of view that physically stores the result of a query (it persists data).
- Unlike regular views, materialized views are periodically refreshed to reflect changes in the underlying tables.
- **Note**: Not all database systems support materialized views (e.g., PostgreSQL, Oracle).
- Example

```
CREATE MATERIALIZED VIEW department_summary AS

SELECT department, COUNT(*) AS employee_count

FROM employees

GROUP BY department;
```

Strategies to Implement Views in SQL:

When implementing views in SQL, there are several strategies to consider depending on the use case and requirements:

1. Using Views for Data Security and Access Control:

- **Expose Limited Data**: You can create views that only expose certain columns from tables, hiding sensitive or irrelevant information.
- **Row-Level Security**: By applying filters in the view (e.g., `WHERE` clauses), you can restrict access to certain rows of data based on user roles.

What Are Views in SQL?

A **view** in SQL is a virtual table that provides a way to represent data from one or more tables. It does not store data itself but contains a SQL query that pulls data from the underlying tables. Views simplify complex queries by encapsulating them into reusable virtual tables, and they provide a way to control access to sensitive data by exposing only specific columns or rows.

Key Characteristics of Views:

- **Virtual Table**: A view does not store data; instead, it dynamically retrieves data when queried.
- **Simplifies Queries**: Views can simplify complex queries by encapsulating joins, aggregations, and filters into a single query.
- **Security**: Views can be used to restrict access to sensitive data, as they can expose only certain columns or rows from the underlying tables.
- **Reusability**: Views can be reused in other queries or applications, making them useful for standardizing data access patterns.

Creating Views:

A view is created using the `CREATE VIEW` statement in SQL.

Syntax:

sql
Copy code

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
sql
Copy code
CREATE VIEW employee_details AS
SELECT name, department, salary
FROM employees
WHERE status = 'active';
```

In this example, the view `employee_details` will show the `name`, `department`, and `salary` columns from the `employees` table, but only for active employees.

Types of Views:

1. **Simple View**:
   - A simple view is based on a single table and does not include any complex SQL features like aggregation, joins, or subqueries.
   - Example:

     ```
     sql
     Copy code
     CREATE VIEW active_employees AS
     SELECT name, department
     FROM employees
     WHERE status = 'active';
     ```

2. **Complex View**:
   - A complex view is based on multiple tables and often involves joins, aggregations, or subqueries.
   - Example:

     ```
     sql
     Copy code
     CREATE VIEW department_salaries AS
     SELECT e.department, AVG(e.salary) AS avg_salary
     FROM employees e
     GROUP BY e.department;
     ```

3. **Materialized View**:
   - A materialized view is a type of view that physically stores the result of a query (it persists data).
   - Unlike regular views, materialized views are periodically refreshed to reflect changes in the underlying tables.
   - **Note**: Not all database systems support materialized views (e.g., PostgreSQL, Oracle).
   - Example:

```
sql
Copy code
CREATE MATERIALIZED VIEW department_summary AS
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;
```

Strategies to Implement Views in SQL:

When implementing views in SQL, there are several strategies to consider depending on the use case and requirements:

1. **Using Views for Data Security and Access Control:**

- **Expose Limited Data**: You can create views that only expose certain columns from tables, hiding sensitive or irrelevant information.
- **Row-Level Security**: By applying filters in the view (e.g., WHERE clauses), you can restrict access to certain rows of data based on user roles.
- **Example**:

  ```
  sql
  Copy code
  CREATE VIEW public_employee_data AS
  SELECT name, department
  FROM employees;
  ```

  This view hides salary and other sensitive columns from users.

2. **Using Views to Simplify Complex Queries:**

- **Encapsulation of Complex Logic**: Views can simplify complex queries by encapsulating joins, aggregations, and filters. Users can query the view instead of writing complex SQL every time.

3. Using Views to Aggregate Data:

- Views are useful for creating summarized or aggregated data that is frequently needed. For example, creating a view that shows the total sales by each store.

4. Using Views for Data Abstraction:

- Views provide abstraction by allowing users to interact with a simplified, logical representation of the data. For instance, the schema of the underlying tables might be more complex than needed for everyday queries.

5. Performance Optimization with Materialized Views:

- In systems where querying large datasets is frequent, materialized views can store the result of a query physically, improving performance by avoiding repeated execution of complex queries.

6. Maintainability and Reusability:

- **Reuse**: Instead of repeating the same query logic, you can define it in a view and reference the view in multiple places.
- **Maintainability**: Views help maintain a cleaner and more manageable codebase. If the underlying query logic changes, you only need to update the view rather than every query using that logic.

6b. Explain in detail about Assertions and Triggers in SQL

Assertions in SQL

Assertions are a way to ensure that certain conditions hold true within the database. They are used to define a rule that must always be true for the entire database or a particular set of data, regardless of what operations are being performed (e.g., INSERT, UPDATE, DELETE). Assertions are typically used to enforce business logic or integrity constraints that cannot be captured by a single table's constraints.

However, it's important to note that **assertions** are not supported by all relational database management systems (RDBMS). For example, MySQL does not support assertions, while other systems like PostgreSQL and Oracle may have varying levels of support.

Characteristics of Assertions:

1. **Global Rules**: Assertions apply to the entire database or a broader scope of tables and data, rather than being restricted to a single table.
2. **Complex Conditions**: Assertions can enforce more complex conditions or business rules that go beyond standard integrity constraints such as PRIMARY KEY, FOREIGN KEY, or CHECK.
3. **Enforcement**: An assertion is enforced by the DBMS during data modification operations, preventing actions that violate the specified condition.

Example:

Let's assume we want to ensure that the total number of orders for a customer never exceeds 100. An assertion for this might look like:

```
CREATE ASSERTION MaxOrders

CHECK (NOT EXISTS (

  SELECT * FROM Orders

  GROUP BY CustomerID

  HAVING COUNT(*) > 100
```

));

Limitations:

- Support for assertions varies widely, and most modern DBMSs don't support them natively due to performance concerns.
- Some DBMSs, like MySQL, don't have built-in support for assertions, and alternatives such as triggers or application-level logic may need to be used.

Triggers in SQL

Triggers are a type of stored procedure that automatically executes or "fires" when certain events occur on a specific table or view in the database. They are used to enforce business rules, track changes to data, or ensure data consistency and integrity. Triggers can be defined to execute in response to a variety of data modification operations like INSERT, UPDATE, and DELETE.

Triggers can be classified based on:

- **The event** that causes them to fire (INSERT, UPDATE, DELETE).
- **The timing** of their execution relative to the event (BEFORE or AFTER).
- **The scope** of the data changes (FOR EACH ROW or FOR EACH STATEMENT).

Types of Triggers:

1. **BEFORE Triggers**: These triggers are executed before the actual operation (INSERT, UPDATE, DELETE) is performed on the table. They are useful for validating data or modifying the data before it is committed to the table.
    - Example: If you want to ensure that a record cannot be inserted if a certain condition isn't met, you would use a BEFORE trigger.
2. **AFTER Triggers**: These triggers are executed after the operation has been completed. They are useful for actions that depend on the completion of the event, like updating other tables or logging.
    - Example: You might use an AFTER INSERT trigger to update a related table once a new record has been added to a primary table.
3. **INSTEAD OF Triggers**: These triggers replace the event action (INSERT, UPDATE, DELETE). They are commonly used in views when direct modification is not allowed or when complex logic needs to be handled in place of a simple insert, update, or delete.
    - Example: A trigger that replaces an INSERT operation on a view with a more complex set of operations involving multiple underlying tables.

Trigger Syntax:

The basic syntax for creating a trigger is as follows (syntax may vary slightly between database systems):

sql

Copy code

```
CREATE TRIGGER trigger_name

{ BEFORE | AFTER | INSTEAD OF }

{ INSERT | UPDATE | DELETE }

ON table_name

FOR EACH ROW

BEGIN

   -- Trigger logic here

END;
```

Trigger Use Cases:

- **Enforcing business rules**: Automatically reject changes to the data that violate business rules.
- **Data auditing**: Track changes (insert, update, delete) on specific tables and store these changes in audit logs.
- **Referential Integrity**: Prevent updates or deletes that could lead to orphaned records or violate foreign key constraints.
- **Synchronizing tables**: Ensure related tables stay in sync when one table is modified.

Limitations and Considerations:

- **Performance**: Triggers can lead to performance issues if not used carefully, especially when there are many triggers or complex logic involved.
- **Complexity**: Triggers can make the database logic harder to debug and maintain, as operations may be triggered implicitly.
- **Limited visibility**: Debugging or logging the execution of triggers can be challenging because they are often executed automatically in the background without direct user intervention.

7a. Discuss the informal design guidelines for relational schema

1. Use Meaningful Table and Column Names

- **Descriptive Names**: Table and column names should be meaningful and reflect the real-world entities or attributes they represent. This helps with understanding the database design and improves maintainability.
    - For example, use Employee instead of a vague name like Table1 or Data.

    - Use column names like employee_id, first_name, and hire_date rather than generic names like column1, column2.
- **Consistency**: Adopt consistent naming conventions (e.g., snake_case or camelCase) across the schema to ensure readability and standardization.

2. Avoid Redundancy (Data Duplication)

- **Minimize Data Duplication**: Avoid storing the same data multiple times across tables. Redundant data increases the chances of anomalies (inconsistencies, update problems, etc.).
    - Example: Instead of storing an employee's department name in each employee record, store the department_id and reference the department's details in a separate Department table.
- **Use Foreign Keys**: To avoid duplication, use foreign key relationships to link related tables. This ensures that information is stored once and can be referenced efficiently.

3. Choose Appropriate Keys

- **Primary Keys**: Every table should have a primary key that uniquely identifies each record. This key should be simple, minimal, and unique.
    - Example: In an Employee table, employee_id could be the primary key.
- **Foreign Keys**: Foreign keys establish relationships between tables. They help maintain referential integrity by ensuring that the values in one table correspond to valid entries in another table.
    - Example: The Employee table may have a department_id column, which references the Department table's department_id column.
- **Avoid Composite Keys**: Whenever possible, use single-column primary keys, as they simplify indexing, querying, and maintenance. Composite keys (primary keys composed of multiple columns) can be harder to manage and are typically used only when there is no natural single-column key.

4. Normalization (Up to a Reasonable Level)

- **Normalization**: Normalize the schema to reduce redundancy and avoid update anomalies. Typically, the schema should be normalized to at least **Third Normal Form (3NF)**, but sometimes **Boyce-Codd Normal Form (BCNF)** or **Fourth Normal Form (4NF)** might be desirable for more complex data relationships.

  Steps of normalization:

    - **1NF (First Normal Form)**: Ensure each column contains atomic (indivisible) values, and each record is unique.
    - **2NF (Second Normal Form)**: Remove partial dependencies by ensuring that all non-key attributes are fully functionally dependent on the entire primary key (relevant for tables with composite keys).
    - **3NF (Third Normal Form)**: Remove transitive dependencies, ensuring non-key attributes are not dependent on other non-key attributes.

- **Balance with Denormalization**: In some cases, it may be beneficial to denormalize the schema (introduce some redundancy) for performance reasons, especially in read-heavy databases or data warehouses. However, this should be done carefully to avoid problems like update anomalies.

5. Handle Many-to-Many Relationships Effectively

- **Bridge Tables**: Many-to-many relationships cannot be directly modeled with a single table in a relational schema. Instead, create a separate **junction** or **bridge table** that holds references (foreign keys) to both tables involved in the relationship.

  Example: If students can enroll in many courses, and a course can have many students, a Student_Course bridge table can be created with student_id and course_id as foreign keys.

6. Consider the Use of Constraints

- **Use Constraints for Data Integrity**: Constraints such as NOT NULL, UNIQUE, CHECK, and DEFAULT help ensure that the data in the database adheres to business rules and remains consistent.
  - **NOT NULL**: Use for columns where a value is required (e.g., an employee must have a name).
  - **UNIQUE**: Use to enforce that a value (e.g., email address) must be unique in a table.
  - **CHECK**: Use to ensure that values fall within a certain range or match a particular pattern.
  - **DEFAULT**: Use to provide default values when none are specified, which helps in ensuring consistency across rows.

7. Be Mindful of Data Types

- **Use Appropriate Data Types**: Choose data types that reflect the kind of data being stored and the range of values it will hold. This ensures efficient storage and query performance.
  - For example, use DATE for date values, VARCHAR for text strings, and INT for numeric identifiers.
- **Size and Precision**: Be aware of the size of data types. For example, if an employee's salary is unlikely to exceed 10 million, use an appropriate numeric type like DECIMAL(10,2) rather than a large numeric type like BIGINT.

8. Design for Performance

- **Indexing**: Add indexes to columns that are frequently used in search conditions (WHERE clauses), joins, or sorting operations (ORDER BY). However, be cautious about adding too many indexes, as they can slow down insert and update operations.
  - Index primary keys by default, but also consider indexing foreign keys, frequently queried columns, and composite indexes for multi-column queries.

- **Avoid Over-Indexing**: While indexes improve read performance, they come with a cost for write operations (INSERT, UPDATE, DELETE). Strike a balance and only index what's necessary for frequent queries.

9. Document the Schema

- **Provide Documentation**: Keep documentation for the schema, including table relationships, field definitions, and business rules. This helps future developers, DBAs, and analysts understand the design and intent behind the schema.
    - Use comments within SQL code (if supported by your DBMS) to explain complex relationships or decisions.

10. Plan for Scalability

- **Consider Future Growth**: When designing the schema, consider the potential for future changes in data volume, structure, or business needs. Ensure that the design can scale in terms of both performance and flexibility.
    - For example, if you expect an increasing number of users or transactions, consider partitioning large tables or optimizing indexes accordingly.
- **Flexible Design**: Keep the schema flexible to accommodate changes. For example, avoid overly rigid data structures, and use techniques like **soft deletes** (e.g., adding an is_deleted flag) instead of actually removing rows, which allows data recovery or auditing later on.

11. Consider Security and Access Control

- **Restrict Access**: Define appropriate user roles and permissions to restrict who can access or modify the schema. Use **views** and **stored procedures** to control access to sensitive data and maintain data privacy.
- **Sensitive Data**: Protect sensitive information (such as passwords or personal identification) by using encryption, hashing, and other security practices.

7b. Briefly explain the 1st, 2nd, 3rd and Boyce codd normal form

1st Normal Form (1NF)

Definition: A table is in **1st Normal Form (1NF)** if:

1. All columns contain **atomic values** (i.e., indivisible values).
2. Each record (row) is unique.
3. Each column must contain values of a single type (e.g., integers, dates, etc.).

Key Points:

- No repeating groups or arrays within a column.
- Every column must have a unique name.
- There are no multi-valued attributes (e.g., a column that contains a list of values).

Example:

Not in 1NF:

StudentID	Name	Subjects
1	Alice	Math, English
2	Bob	Science, History, Art

To convert this to 1NF:

StudentID	Name	Subject
1	Alice	Math
1	Alice	English
2	Bob	Science
2	Bob	History
2	Bob	Art

2nd Normal Form (2NF)

Definition: A table is in **2nd Normal Form (2NF)** if:

1. It is already in **1st Normal Form**.
2. There is **no partial dependency**; every non-key column is fully functionally dependent on the **entire primary key**, not just part of it (this applies only to tables with a **composite key**).

Key Points:

- 2NF removes partial dependencies where a non-key column depends only on a part of the composite primary key.

Example:

Consider a table with a composite primary key (StudentID, CourseID):

StudentID	CourseID	Instructor	Room

StudentID	CourseID	Instructor	Room
1	101	Mr. Smith	201
1	102	Mrs. Johnson	202
2	101	Mr. Smith	201

Here, Instructor is dependent only on CourseID, not on the whole primary key (StudentID, CourseID).

To convert to 2NF:

- Split the table into two: one for course details, and one for student-course registrations.

Courses:

CourseID	Instructor	Room
101	Mr. Smith	201
102	Mrs. Johnson	202

Enrollments:

StudentID	CourseID
1	101
1	102
2	101

3rd Normal Form (3NF)

Definition: A table is in **3rd Normal Form (3NF)** if:

1. It is already in **2nd Normal Form**.
2. There are **no transitive dependencies**; non-key columns are not dependent on other non-key columns.

Key Points:

- A transitive dependency occurs when a non-key column depends on another non-key column, which in turn depends on the primary key.

Example:

Consider a table with a transitive dependency:

StudentID	StudentName	Department	DepartmentHead
1	Alice	CS	Dr. Smith
2	Bob	CS	Dr. Smith
3	Carol	Math	Dr. Johnson

Here, DepartmentHead is dependent on Department, and Department depends on StudentID, creating a transitive dependency.

To convert to 3NF:

- Split the table into two: one for student details and one for department details.

Students:

StudentID	StudentName	Department
1	Alice	CS
2	Bob	CS
3	Carol	Math

Departments:

Department	DepartmentHead
CS	Dr. Smith
Math	Dr. Johnson

Boyce-Codd Normal Form (BCNF)

Definition: A table is in **Boyce-Codd Normal Form (BCNF)** if:

1. It is already in **3rd Normal Form (3NF)**.
2. For every non-trivial functional dependency, the left side must be a **superkey**.

Key Points:

- BCNF is a stricter version of 3NF. In 3NF, a non-key column can be dependent on another non-key column as long as the dependency doesn't cause any redundancy. In BCNF, every determinant must be a candidate key.

Example:

Consider a table where InstructorID determines both InstructorName and CourseID:

InstructorID	InstructorName	CourseID
1	Dr. Smith	CS101
2	Dr. Johnson	CS102

Here, InstructorID is a candidate key, but InstructorName is not fully dependent on the entire primary key. This violates BCNF.

To convert to BCNF, you must break the table into two:

Instructors:

InstructorID	InstructorName
1	Dr. Smith
2	Dr. Johnson

Courses:

CourseID	InstructorID
CS101	1
CS102	2

8a. Explain and write an algorithm on relational decomposition into BCNF with non addictive join property

Steps to Decompose a Relation into BCNF with Non-Additive Join Property

1. **Check if the Relation is in BCNF:**
   - Identify all functional dependencies (FDs) for the relation.
   - Check if, for every FD X→YX \to YX→Y, the left-hand side (X) is a superkey.
   - If all FDs satisfy this condition, the relation is already in BCNF, and no further decomposition is needed.
2. **Decompose if the Relation is not in BCNF:**
   - If there exists a functional dependency X→YX \to YX→Y such that XXX is **not** a superkey, the relation is **not** in BCNF, and a decomposition is needed.
   - Decompose the relation into two sub-relations:
     - R1(X,Y)R_1(X, Y)R1(X,Y) — this will include the attributes from the FD X→YX \to YX→Y.
     - R2(R−Y)R_2(R - Y)R2(R−Y) — this includes the remaining attributes of the original relation that are not in YYY.
   - Ensure that both sub-relations contain a superkey to satisfy BCNF.
3. **Ensure Non-Additive Join Property:**
   - The decomposition should be **lossless**. For the decomposition R1R_1R1 and R2R_2R2, the non-additive join property is ensured if the intersection of R1R_1R1 and R2R_2R2 contains a key (or superkey) from one of the relations.
   - Specifically, if R1(X,Y)R_1(X, Y)R1(X,Y) and R2(R−Y)R_2(R - Y)R2(R−Y) are the decomposed relations, the decomposition is lossless if the intersection of R1R_1R1 and R2R_2R2 is not empty and contains a key for the original relation.
4. **Repeat Decomposition:**
   - If either R1R_1R1 or R2R_2R2 is not in BCNF, repeat the decomposition process on those relations.
   - Continue until all sub-relations are in BCNF.

Example:

Let's consider a relation R={A,B,C,D}R = \{A, B, C, D\}R={A,B,C,D} with the functional dependencies:

- A→BA \to BA→B
- B→CB \to CB→C
- C→DC \to DC→D
- Start with checking if the relation is in BCNF. The functional dependency A→BA \to BA→B has a left-hand side AAA, which is not a superkey (since it doesn't uniquely identify the whole relation).
- Decompose the relation into R1={A,B}R_1 = \{A, B\}R1={A,B} and R2={B,C,D}R_2 = \{B, C, D\}R2={B,C,D}.
- Repeat the process on R2R_2R2, checking the functional dependencies and further decomposing if needed until all relations are in BCNF.

This algorithm ensures that the decomposition results in relations in BCNF and preserves the non-additive join property by ensuring that each decomposed relation contains sufficient information to avoid loss of data when performing the join.

8b. Discuss about nulls, dangling tuples and alternative relational designs

1. Nulls in Relational Databases

Definition of Nulls:

A **null** in a relational database refers to an absence of a value or an unknown value for an attribute. It is different from a zero or an empty string; rather, it signifies that the value is missing or not applicable.

Use Cases of Nulls:

- **Missing Data:** When information is unavailable, either because it hasn't been entered or because it's not applicable.
- **Unknown Information:** When a value is unknown at the time of data entry (e.g., an unknown age of a person).
- **Not Applicable Values:** When an attribute does not apply to all records in the database (e.g., a column for "end date" in an employee table, where some employees are still employed).

Handling Nulls:

- **Relational Model:** Relational databases often allow null values in any attribute, which must be treated carefully when performing operations such as comparisons and joins.
- **Impacts on Operations:**
    - **Comparisons:** Any comparison involving a null value (e.g., NULL = NULL) is unknown, which complicates querying. Special operators like IS NULL or IS NOT NULL are used to check for null values.
    - **Aggregations:** Most aggregate functions (like SUM, COUNT, etc.) ignore nulls, but this behavior can affect the results if not handled carefully.

Problems with Nulls:

- **Ambiguity:** Null values can create ambiguity because they can represent different situations (unknown, missing, or inapplicable).
- **Integrity Constraints:** Nulls complicate the enforcement of integrity constraints, as constraints like foreign keys, unique keys, and check constraints may require additional rules for null handling.
- **Application Logic:** Handling nulls in queries and business logic requires careful design to avoid errors in reports or decisions based on the data.

2. Dangling Tuples (or Dangling References)

Definition:

A **dangling tuple** (or dangling reference) occurs when a tuple (or record) in one relation is left with a reference to a tuple in another relation that no longer exists. This often arises due to delete operations in relational databases.

Cause of Dangling Tuples:

- **Deletion Anomalies:** If a tuple in a referenced table (i.e., the parent table) is deleted, the dependent tuples (child tuples) in other tables may still exist, but their references (foreign keys) point to a non-existent tuple.

Example:

Consider two tables:

- **Employees (Emp_ID, Name, Dept_ID)**
- **Departments (Dept_ID, Dept_Name)**

If an employee is deleted from the **Employees** table but their Dept_ID is still present in the **Departments** table, then the employee record is "dangling," as it refers to a non-existent department.

Handling Dangling Tuples:

- **Referential Integrity Constraints:** Most modern relational databases enforce referential integrity using foreign key constraints. These constraints can be set to perform automatic actions when a referenced tuple is deleted or updated:
    - **CASCADE:** Automatically delete or update the dependent tuples.
    - **SET NULL:** Set the foreign key in dependent tuples to null when the referenced tuple is deleted or updated.
    - **RESTRICT/NO ACTION:** Prevent the deletion of the referenced tuple if there are dependent tuples.
    - **SET DEFAULT:** Set the foreign key to a default value.

Problems with Dangling Tuples:

- **Data Inconsistency:** Dangling tuples lead to inconsistent data and can cause errors in queries, reports, and updates.
- **Integrity Violation:** If referential integrity is not maintained, the database might return incorrect or incomplete results due to missing relationships.

3. Alternative Relational Designs

Alternative Designs in Relational Databases:

The design of a relational database schema has a significant impact on its efficiency, data integrity, and ease of maintenance. In certain scenarios, designing the schema differently can help avoid issues like null values or dangling tuples.

a) Dealing with Nulls and Missing Data:

- **Normalization vs. Denormalization:**
  - **Normalization:** The process of structuring the database into multiple related tables helps avoid null values by splitting data into smaller, more manageable chunks. For example, having separate tables for contact details (phone number, email, address) allows handling optional information more cleanly. However, normalization can also lead to nulls in many places (e.g., if a customer doesn't have an email, the email attribute will have a null).
  - **Denormalization:** In some cases, denormalizing the schema (combining tables) may reduce the complexity of handling nulls by reducing the number of missing or optional attributes in a given table. However, denormalization introduces redundancy, and maintaining the consistency of data becomes more challenging.
- **Use of Special Design Patterns:**
  - **Entity-Attribute-Value (EAV):** This model is often used when data is sparse or highly variable (e.g., in medical or scientific data). It represents entities and their attributes in a flexible manner, where each attribute of an entity is stored as a separate row.
  - **Sparse Columns:** In some systems, you may design the database using sparse columns that accommodate a large number of potential attributes that can have null values. This design reduces the number of nullable attributes per row.

b) Redundant Data and Dangling Tuples:

- **Avoiding Dangling Tuples Through Referential Integrity:**
  - The use of foreign keys with proper referential integrity constraints ensures that dangling tuples are avoided. Ensuring that no tuple in a child table can reference a non-existent tuple in the parent table can eliminate this issue.
- **Use of Triggers or Stored Procedures:**
  - **Triggers**: Some database systems allow you to define triggers that automatically update or delete tuples in a related table when a change occurs in the parent table. This can help in maintaining referential integrity without relying solely on the foreign key constraints.
  - **Stored Procedures**: In some cases, business logic is implemented using stored procedures to ensure that operations like deletes or updates on tables are handled safely without leaving dangling references.

c) Alternative Data Models (Beyond Relational Model):

- **Document-Based NoSQL Databases (e.g., MongoDB):** For scenarios where null values or schema changes are frequent, NoSQL databases such as MongoDB use a flexible document-based model, allowing for sparse data and no strict schema enforcement. This

can help manage cases with missing data or attributes that are not applicable across all records.

- **Graph Databases:** For scenarios involving complex relationships with potential dangling references, a **graph database** might be a good choice. Graph databases focus on relationships and can dynamically adapt as relationships between entities evolve, potentially avoiding issues with dangling references by modeling relationships explicitly.
- **Wide-Column Stores (e.g., Cassandra, HBase):** These databases use column families instead of rows and allow for more flexibility in dealing with missing or sparse data. Null values are less problematic because column families can grow or shrink dynamically without having to conform to a fixed schema.

9a. Discuss why concurrency control needed with example

1. Lost Update Problem

 Description: Occurs when two transactions update the same data item

simultaneously, causing one update to overwrite the other.

 Example:

o Assume a bank account balance is initially $500.

o Transaction T1T_1T1 reads the balance, adds $100, and sets it to

$600.

o Before T1T_1T1 writes $600 back, Transaction T2T_2T2 reads the

same balance of $500, subtracts $50, and sets it to $450.

o Both transactions complete, but T1T_1T1's update to $600 is

overwritten by T2T_2T2's update to $450, causing the addition of

$100 to be "lost."

2. Dirty Read (Uncommitted Dependency Problem)

 Description: Occurs when a transaction reads data modified by another

transaction that has not yet been committed. If the modifying transaction

rolls back, the reading transaction has read invalid data.

 Example:

o T1T_1T1 updates an account balance from $500 to $600 but does not commit.

o T2T_2T2 reads this uncommitted balance of $600 and uses it for calculations.

o If T1T_1T1 rolls back, the original balance of $500 is restored, but T2T_2T2 has already used the invalid $600 balance.

3. Unrepeatable Read (Inconsistent Retrieval)

 Description: Occurs when a transaction reads the same data multiple times and gets different results due to another transaction's updates.

 Example:

o T1T_1T1 reads the balance of an account as $500.

o T2T_2T2 updates the balance to $600 and commits.

o T1T_1T1 reads the balance again and sees $600, which is inconsistent with the initial read.

4. Phantom Read

 Description: Occurs when a transaction reads a set of rows based on a condition, but another transaction inserts, deletes, or updates rows that affect the result of the original query.

 Example:

o T1T_1T1 reads a list of accounts with a balance greater than $500, finding three accounts.

o Meanwhile, T2T_2T2 inserts a new account with a balance of $700.

o If T1T_1T1 re-reads the list, it will find four accounts, leading to inconsistencies.

Why Concurrency Control and Recovery are Needed in DBMS

1. Concurrency Control:

o Concurrency control mechanisms manage the execution of transactions to ensure data integrity and isolation. They prevent conflicts like lost updates, dirty reads, unrepeatable reads, and phantom reads by coordinating access to shared data.

o Reason: Concurrency control is essential to maintain the ACID properties (especially isolation and consistency) and avoid data inconsistencies. It enables multiple users to access the database concurrently without interference.

2. Recovery:

o Recovery mechanisms restore the database to a consistent state in the event of system crashes, transaction failures, or hardware issues. This includes rolling back incomplete transactions or redoing committed ones.

o Reason: Recovery mechanisms ensure atomicity and durability, allowing transactions to either complete entirely or leave no trace in case of a failure. This maintains database consistency and helps recover from unexpected issues.

9b. Discuss desirable properties of a Transaction

Properties of a Transaction (ACID Properties)

Transactions follow the ACID properties, which are essential to maintaining database integrity:

1. Atomicity:

o Definition: A transaction is an atomic unit of operation, meaning it either fully completes or does not happen at all. If any part of the transaction fails, the entire transaction is rolled back.

o Explanation: If a transaction involves multiple steps (e.g., transferring funds between accounts), either all steps must succeed, or none should take effect.

o Example: In an e-commerce system, if a customer's payment fails, the entire transaction (including inventory update and order creation) should be reversed.

2. Consistency:

o Definition: A transaction must bring the database from one consistent state to another. All integrity constraints must be satisfied both before and after the transaction.

o Explanation: Database rules (like foreign keys or constraints) ensure that data remains valid. If any condition fails, the transaction will not complete.

o Example: A transaction should not result in a negative balance if the system disallows negative account balances.

3. Isolation:

o Definition: Transactions should be executed independently and should not interfere with each other. A transaction's intermediate states should not be visible to other transactions.

o Explanation: This property prevents issues like dirty reads,

unrepeatable reads, and lost updates, maintaining the accuracy of concurrent transactions.

o Example: In a banking application, two simultaneous withdrawals from the same account should occur in isolation to avoid incorrect balance calculations.

4. Durability:

o Definition: Once a transaction is committed, its changes to the database are permanent, even in the event of a system crash or failure.

o Explanation: Durability ensures that the database maintains a record of committed transactions, usually through logging and backup mechanisms.

o Example: If a power failure occurs right after a transaction is committed, the changes should remain when the system restarts.

10a. Briefly explain the importance of strict two phase locking for concurrency control

Strict Two-Phase Locking (2PL) is a concurrency control protocol used in database management systems to ensure the **serializability** of transactions, which means that the outcome of executing transactions concurrently is equivalent to executing them sequentially. Strict 2PL is a variant of the two-phase locking protocol, where transactions are required to hold locks until the end of the transaction, ensuring certain guarantees.

Key Concepts of Strict Two-Phase Locking:

1. **Two Phases**:
    - **Growing Phase**: A transaction can acquire locks but cannot release any locks.
    - **Shrinking Phase**: Once a transaction releases a lock, it cannot acquire any more locks.
2. **Strict 2PL**:

- In **Strict 2PL**, a transaction must hold all its locks until it **commits** or **aborts**, meaning it **cannot release any locks until the transaction completes**.

Why Strict 2PL is Important:

1. **Ensures Serializability**:
   - The primary guarantee provided by Strict 2PL is **serializability**, the highest level of isolation in transaction processing. It ensures that the execution of concurrent transactions produces results that are equivalent to some serial order of transactions.
2. **Prevents Dirty Reads**:
   - A dirty read occurs when a transaction reads uncommitted data from another transaction. Strict 2PL ensures that transactions cannot read data that is not yet committed because locks are held until the end of the transaction.
3. **Prevents Lost Updates and Uncommitted Data**:
   - By ensuring that transactions do not release locks until they commit, Strict 2PL prevents problems like **lost updates** (where one transaction overwrites the changes made by another) and **uncommitted data** (where changes are visible to other transactions before they are finalized).
4. **Avoids Cascading Aborts**:
   - Without strict locking, a failure in one transaction could lead to cascading aborts (where multiple dependent transactions are rolled back). By holding locks until the end, Strict 2PL ensures that the database system doesn't let uncommitted changes affect other transactions, avoiding cascading aborts.
5. **Simplicity in Deadlock Handling**:
   - While Strict 2PL guarantees serializability, it can lead to deadlocks, where two or more transactions are waiting for each other to release locks. However, because the protocol is straightforward (just two phases), deadlock detection and resolution strategies can be implemented effectively.

10b. Explain validation(optimistic) techniques and snapshot isolation concurrency control

1. Validation (Optimistic) Concurrency Control

Concept:

- **Optimistic Concurrency Control** (OCC), also known as **validation**, is based on the assumption that transaction conflicts are rare. Instead of locking resources, OCC allows transactions to execute without restrictions but checks for conflicts only at the end of the transaction (during the **validation phase**).
- The basic idea behind OCC is that transactions are executed optimistically without interference. At the end of the transaction, before committing, the system checks whether

any conflicts have occurred. If there are no conflicts, the transaction is allowed to commit; otherwise, it is aborted and must be retried.

Phases of Optimistic Concurrency Control:

1. **Read Phase:**
   - The transaction reads data from the database and performs the necessary operations without acquiring locks.
2. **Validation Phase:**
   - Before the transaction is committed, the system checks if the data it read during the **Read Phase** has been modified by other transactions. If no conflicts are detected, the transaction is allowed to commit.
3. **Write Phase:**
   - If the transaction passes the validation phase, it is committed to the database. If a conflict is found during validation, the transaction is aborted and must be retried.

Advantages of Optimistic Concurrency Control:

- **No Locking Overhead**: Since transactions don't acquire locks during execution, there is no risk of deadlocks, and there is less overhead compared to locking-based protocols.
- **Increased Parallelism**: Because transactions do not block each other, OCC can result in higher throughput in situations where conflicts are rare.
- **Reduced Contention**: In low-contention environments, OCC can be more efficient than lock-based methods.

Disadvantages of Optimistic Concurrency Control:

- **Higher Abortion Rate**: If there are frequent conflicts, many transactions may be aborted and need to be retried, which can result in performance degradation.
- **Validation Overhead**: The validation phase can introduce overhead, especially in highly concurrent systems where a high number of transactions are running concurrently.

2. Snapshot Isolation (SI) Concurrency Control

Concept:

- **Snapshot Isolation** (SI) is a concurrency control model that provides a **snapshot** of the database to each transaction at the time it starts, ensuring that the transaction sees a consistent view of the database. This approach is based on the idea of providing each transaction with a "snapshot" of the database, effectively freezing the state of the data at the transaction's start time.
- With SI, transactions are not allowed to read uncommitted changes from other transactions (which avoids dirty reads), but they are allowed to read a consistent snapshot of the database, even if other transactions are concurrently modifying the data.

How Snapshot Isolation Works:

- **Read Phase**: Each transaction reads data as of the start time of the transaction, using a snapshot of the database.
- **Write Phase**: The transaction writes its changes to the database at commit time, but only if no other transaction has made conflicting changes to the same data since the transaction started.

Key Points of Snapshot Isolation:

1. **Versioning**: SI uses **multi-version concurrency control (MVCC)** to maintain multiple versions of data items. Each transaction sees its own snapshot of data, and writes are only visible to other transactions once they commit.
2. **No Dirty Reads**: Since transactions see a consistent snapshot, dirty reads (reading uncommitted changes from other transactions) are avoided.
3. **Conflicts**: Conflicts arise if two transactions try to modify the same data item. If two transactions attempt to update the same value, one will be aborted, ensuring consistency.

Advantages of Snapshot Isolation:

- **Prevents Dirty Reads**: Since each transaction works on a snapshot, it only sees committed data at the time the transaction starts.
- **Improved Performance**: SI allows higher concurrency than strict isolation levels (like Serializable), because transactions are not blocked by each other, and conflicts are detected only at commit time.
- **No Locking Conflicts**: SI does not require locks for reading data, which improves performance by avoiding deadlocks and reducing contention.

Disadvantages of Snapshot Isolation:

- **Write Skew**: SI does not prevent all anomalies. One of the potential issues is **write skew**, where two transactions read the same data, perform operations based on that data, and write conflicting results, leading to an inconsistent state.
    - For example, if two transactions simultaneously check the balance of a bank account and decide to update the balance based on their respective snapshots, they could both update the balance based on stale data, violating consistency.
- **Not Serializable**: Although Snapshot Isolation avoids dirty reads and ensures consistency within a transaction, it does not guarantee serializability, which is the highest level of isolation. The database might allow some non-serializable schedules to execute.