CBCS SCHEME

USN

21CS53

Fifth Semester B.E. Degree Examination, Dec.2023/Jan.2024

Database Management Systems

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

Module-1

1 a. Define DBMS. Explain all the basic operations that can be performed by DBMS on a database. **(05 Marks)**

b. Explain the different users of a database system. **(10 Marks)**

c. Describe the 3-Schema Architecture. **(05 Marks)**

OR

2 a. Define the following terms:
i) Data model ii) Schema iii) Insurance iv) Canned transaction **(04 Marks)**

b. Describe the structural constraints of a database system with suitable example. **(10 Marks)**

c. Explain all the E-R diagram notations. **(06 Marks)**

Module-2

3 a. Explain the four relational model constraints. **(06 Marks)**

b. Explain all the steps of Relational database design using E-R to relational schema with a suitable example. **(06 Marks)**

c. Discuss the DIVISION operation of relational algebra. Find the Quotient for the following :

A =

SNO	DNO
S_1	P_1
S_1	P_2
S_1	P_3
S_1	P_4
S_2	P_1
S_2	P_2
S_3	P_2
S_4	P_2
S_4	P_4

B_1 =

PNO
P_2

B_2 =

PNO
P_2
P_4

B_3 =

PNO
P_1
P_2
P_4

Find i) A/B_1 ii) A/B_2 iii) A/B_3 **(08 Marks)**

OR

4 a. Explain the characteristics of a relational model. **(06 Marks)**

b. Explain all types of outer join operations in relational algebra. Demonstrate the advantage of outer join operation over the inner join operation. **(06 Marks)**

c. Considering the following schema
Sailors (sid, sname, rating, age)
Boats (bid, bname, color)
Reserves (sid, bid, day)
Write a relational algebra queries for the following :
i) Find the names of sailors who have reserved boat#103.
ii) Find the names of sailors who have reserved a red boat.
iii) Find the names of sailors who have reserved a red or green boat.
iv) Find the names of sailors who have reserved all boats. **(08 Marks)**

Module-3

5 a. Explain the basic data types available for attributes in SQL. (05 Marks)

b. Demonstrate the following constraints in SQL with suitable example:
i) NOT NULL ii) Primary key iii) Foreign key iv) Default v) Check. (10 Marks)

c. What are triggers? Explain with syntax and suitable example. (05 Marks)

OR

6 a. Explain the basic definition of a cursor and its usage with the help of a suitable example. (05 Marks)

b. What are Assertions? Assuming suitable company schema write an Assertion for the condition.
"The salary of an Employee must not be greater than the salary of the manager of the department that the employee works for". (05 Marks)

c. Referring to the below mentioned company schema. Write the SQL queries for the following:

Employee

Fname	Lname	Minit	Ssn	Bdate	Address	Sex	Salary	SuperSsn	Dno

Department

Dname	Dnumber	Mgr_Ssn	Mgr_start_date

Department_location

Dnumber	Dlocation

Project

Pname	Pnumber	Plocation	Dnum

Work_on

Essn	DNo	HRS

Defendant

Essn	Dependentname	Sex	Bdate

i) For each department retrieve the department number, the number of employees in the department and their average salary.

ii) For each project on which more than 2 employees work, retrieve the project number, the project name and the number of employees who work on the project.

iii) For each project, retrieve the project number, the project name and the number of employees from department no. 5 who work on that project.

iv) For each department that has more than 5 employees, retrieve the department number and the number of its employees who are making more than $40,000 salary.

v) Retrieve the names of an employees who have two or more dependents. (10 Marks)

Module-4

7 a. Explain the types of update anomalies with examples. (05 Marks)

b. Explain Armstrong's rules of inference. (05 Marks)

c. What is the need for normalization? Explain 1NF, 2NF and 3NF with examples. (10 Marks)

OR

8 a. Explain the informal design guidelines of a database. (06 Marks)

b. What is equivalence of sets of functional dependencies? Check whether the following sets of F.D's are equivalent or not.

$FD_1 = \{A \rightarrow B, B \rightarrow C, AB \rightarrow D\}$

$FD_2 = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow D\}$ (08 Marks)

c. Write an algorithm to find the closure of functional dependency 'F'. (06 Marks)

Module-5

9 a. Explain the desirable properties of a transaction. (06 Marks)

b. Explain with a neat diagram, the state transition diagram of a transaction. (06 Marks)

c. Explain two phase locking mechanism with suitable example. (08 Marks)

OR

10 a. Discuss on the database inconsistency problem. (10 Marks)

b. Explain Binary locks and shared locks with algorithms. (10 Marks)

1.1 Define DBMS. Explain all the basic operations that can be performed in DBMS database.

Solution-
Database Management System (DBMS) is a software system that provides the necessary tools and functionalities to create, manage, and manipulate databases. It acts as an interface between the users and the database, enabling users to perform various operations on the data while ensuring security, integrity, and efficiency.

A DBMS allows for the definition, creation, querying, update, and administration of databases. Some common examples of DBMS software include MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and MongoDB.

Basic Operations in a DBMS

The basic operations that can be performed on a DBMS database are often summarized by the acronym CRUD:

1. **Create**
2. **Read**
3. **Update**
4. **Delete**

These operations correspond to the fundamental ways in which data can be manipulated within a database.

1. Create

- **Definition**: The CREATE operation is used to add new data into the database. This can involve creating new database structures like tables, indexes, views, or inserting new records into existing tables.

Read

- **Definition**: The READ operation (also known as Retrieve or Query) is used to fetch or retrieve data from the database. This operation is often carried out using the SELECT statement in SQL.

Update

- **Definition**: The UPDATE operation is used to modify existing data in the database. It allows you to change one or more records based on certain conditions.

4. Delete

- **Definition**: The DELETE operation is used to remove data from the database. This can involve deleting specific records or even entire tables or databases.

1.2 Explain different users of a database system.

Solution-
A database system serves various types of users, each with different roles, responsibilities, and

access levels. These users can be broadly categorized based on how they interact with the database and the tasks they perform. Below are the different types of users in a database system:

1. Database Administrators (DBAs)

- **Role**: Database Administrators are responsible for the overall management, maintenance, and security of the database system. They ensure that the database runs efficiently, remains secure, and is accessible to authorized users.
- **Responsibilities**:
  - Installing and upgrading the database server and application tools.
  - Allocating system storage and planning for future storage requirements.
  - Configuring and tuning the database for performance optimization.
  - Managing user access and ensuring data security.
  - Backing up and recovering data to prevent data loss.
  - Monitoring and troubleshooting the database to prevent or resolve issues.
- **Example**: A DBA might set up and configure an Oracle database for a large organization and ensure that it runs efficiently by regularly monitoring performance and applying necessary updates.

2. Database Designers (Data Architects)

- **Role**: Database Designers are responsible for designing the logical and physical structure of the database. They create the schema that defines how data is stored, accessed, and related.
- **Responsibilities**:
  - Designing the database schema, including tables, relationships, indexes, and constraints.
  - Normalizing the database to eliminate redundancy and ensure data integrity.
  - Defining the data types and structures based on the organization's requirements.
  - Ensuring that the database design supports the required business processes.
- **Example**: A database designer might create an ER diagram for a university's database, defining tables for Students, Courses, and Instructors and establishing relationships between them.

3. Application Programmers (Developers)

- **Role**: Application Programmers write the code that interacts with the database. They develop the software applications that use the database to store, retrieve, and manipulate data.
- **Responsibilities**:
  - Writing SQL queries to interact with the database.
  - Developing application code that performs CRUD operations.

    - Integrating the database with other software systems or user interfaces.
    - Ensuring that the application follows the database schema and constraints.
- **Example**: An application programmer might develop a web application for an e-commerce site that retrieves product information from a database and displays it to users.

4. End Users

- **Role**: End users are the individuals who interact with the database system through applications or direct query interfaces. They do not need to have in-depth knowledge of the database but use it to perform their daily tasks.
- **Types of End Users**:
    - **Casual Users**: Use the database occasionally, typically through user-friendly interfaces like forms or reports. They might not know the underlying database structure.
        - **Example**: A sales manager who generates sales reports from a database using a pre-built reporting tool.
    - **Naive Users (Parametric Users)**: Regular users who interact with the database through pre-defined queries or simple application interfaces. They follow a routine and predefined set of operations.
        - **Example**: A bank teller entering transactions into a database using a specific application.
    - **Sophisticated Users**: Experienced users who write their own queries to extract information from the database. They might have a good understanding of the database schema.
        - **Example**: A business analyst who writes SQL queries to generate custom reports.

5. System Analysts

- **Role**: System Analysts bridge the gap between business needs and IT. They analyze business requirements and ensure that the database system meets those needs.
- **Responsibilities**:
    - Gathering and analyzing business requirements.
    - Defining data models and database requirements based on business processes.
    - Collaborating with DBAs and Developers to ensure the database system aligns with business goals.
    - Testing and validating that the database meets user requirements.
- **Example**: A system analyst might work with a retail company to define the data requirements for tracking inventory and ensure that the database supports this functionality.

1.3 Describe a 3 schema architecture.

Solution-
The three-schema architecture is a framework for understanding the various views of a database system. It was developed to separate the user applications and the physical database. Here's a description of each of the three levels:

1. External Schema (View Level):

- Description: This is the topmost level, where users interact with the database. It involves multiple user views, each designed for a specific application or user group.
- Purpose: To provide a customized view of the data for different users, ensuring that they can only access and manipulate data relevant to them. It helps in enhancing security and privacy by restricting access to certain parts of the database.
- Example: In a university database, students might have a view of their grades and courses, while professors might have a different view to manage class rosters and assignments.

2. Conceptual Schema (Logical Level):

- Description: This is the middle level, providing a unified view of the entire database, independent of how the data is stored physically. It defines what data is stored, the relationships among the data, and the constraints on the data.
- Purpose: To describe the structure of the whole database for a community of users. It focuses on the logical structure of the data and defines the entities, data types, relationships, user operations, and constraints.
- Example: The university database might include entities like Students, Courses, and Instructors, and relationships such as Students enrolled in Courses.

3. Internal Schema (Physical Level):

- Description: This is the lowest level, dealing with the physical storage of data on the storage medium. It describes how the data is stored in the database, including indexes, file organization, and access methods.
- Purpose: To manage the physical storage of data, optimize performance, and ensure efficient use of storage resources. It abstracts the physical storage details from the other levels.
- Example: The university database might store student records in a B-tree index to allow quick retrieval by student ID, while course data might be stored in sequential files.

2.1 Define the following terms-
1. Data model
2. Schema
3. Insurance
4. Canned Transaction

Solution-
Data Model

- **Definition**: A data model is an abstract representation of the data structures, relationships, and constraints within a database. It serves as a blueprint for how data is stored, organized, and manipulated in a database management system (DBMS).
- **Purpose**: Data models provide a framework for defining the database structure, guiding the design, and ensuring consistency and integrity. They are used to create the logical and physical structure of the data and include elements such as entities, attributes, relationships, and rules.
- **Examples**:
  - **Hierarchical Model**: Data is organized in a tree-like structure.
  - **Relational Model**: Data is organized in tables (relations) with rows and columns.
  - **Object-Oriented Model**: Data is represented as objects, similar to object-oriented programming.

2. Schema

- **Definition**: A schema is a structured framework or blueprint that defines the organization and structure of a database. It includes definitions of tables, columns, data types, constraints, relationships, and other elements.
- **Types**:
  - **Physical Schema**: Describes how data is physically stored on disk.
  - **Logical Schema**: Describes the logical structure of the database, such as tables and relationships.
  - **External Schema**: Describes user-specific views of the data.
- **Purpose**: Schemas help in managing and organizing data within a database, ensuring that the structure is well-defined and can be easily understood and maintained.
- **Example**: In a university database, the schema might define tables for Students, Courses, and Enrollment, with relationships between them.

3. Insurance

- **Definition**: Insurance is a contract between an individual or entity (the insured) and an insurance company (the insurer) in which the insurer agrees to provide financial protection or reimbursement against losses in exchange for the payment of premiums.
- **Purpose**: Insurance helps individuals and businesses mitigate financial risk by transferring the burden of potential losses to the insurer. It covers various risks, such as life, health, property, and liability.
- **Example**: Health insurance covers medical expenses for illnesses or injuries, while auto insurance covers damage to a vehicle in case of an accident.

4. Canned Transaction

- **Definition**: A canned transaction is a pre-defined, standardized, and commonly used database transaction that can be executed repeatedly without modification. These transactions are typically optimized and stored in the database management system for efficiency.
- **Purpose**: Canned transactions are used to perform routine and repetitive tasks, such as querying customer information or processing orders, ensuring consistency, reliability, and speed.
- **Example**: In a banking system, a canned transaction might be used to check an account balance, transfer funds between accounts, or generate a monthly statement.

2.2 **Define the structural constraint of a database system with ecamples.**

Solution-
Structural constraints in a database system refer to the rules that define the relationships between different entities and the limitations on those relationships. These constraints ensure that the database maintains data integrity and consistency. Structural constraints are typically applied in the context of entity-relationship (ER) models or relational database models.

There are two primary types of structural constraints:

1. Cardinality Constraints

- **Definition**: Cardinality constraints specify the number of instances of one entity that can or must be associated with each instance of another entity. They define the nature of relationships in terms of how many entities in one set can relate to entities in another set.
- **Types**:
    - **One-to-One (1:1)**: An entity in one set is associated with at most one entity in another set, and vice versa.
        - **Example**: In a database for a university, each *Student* might have one unique *Student ID*, and each *Student ID* corresponds to only one *Student*.
    - **One-to-Many (1**

): An entity in one set can be associated with multiple entities in another set, but an entity in the second set is associated with only one entity in the first set.

- **Example**: In a university database, a *Professor* can teach multiple *Courses*, but each *Course* is taught by only one *Professor*.

- **Many-to-Many (M**

): Entities in both sets can be associated with multiple entities in the other set.

- **Example**: In a university database, *Students* can enroll in multiple *Courses*, and each *Course* can have multiple *Students* enrolled.

2. Participation Constraints

- **Definition**: Participation constraints specify whether all or only some entity instances participate in a relationship. This constraint ensures that certain entities are required to participate in a relationship, affecting how data integrity is maintained.
- **Types**:
    - **Total Participation (Mandatory)**: Every instance of an entity must participate in a relationship.
        - **Example**: In a database for a company, every *Employee* must be assigned to a *Department*. This means that there cannot be an employee who is not associated with any department.
    - **Partial Participation (Optional)**: Only some instances of an entity participate in a relationship, meaning that it is optional for an entity to be involved in the relationship.
        - **Example**: In a university database, not every *Professor* might be assigned as a *Department Head*. Some professors might not hold any department head position.

Combining Cardinality and Participation Constraints

These constraints often work together to define the structure of relationships between entities in a database.

Example:

- **Employee-Department Relationship**:
    - **Cardinality Constraint**: One department can have many employees (1

relationship).

-
    - **Participation Constraint**: All employees must be associated with a department (Total Participation), but a department might not have any employees yet (Partial Participation).

Structural constraints are fundamental in designing databases, as they directly impact how data is related and stored, ensuring that the relationships between different entities are meaningful and correctly enforced according to the rules of the system.

2.3 **Explain all the ER diagram notations.**

Solution-
Entity-Relationship (ER) diagrams are used to visually represent the structure of a database, including entities, attributes, and relationships. Various notations are used in ER diagrams to convey these concepts clearly. Here's an explanation of the most common ER diagram notations:

1. Entity

- **Notation**: A rectangle.

- **Description**: Represents an object or concept within the database that can have data stored about it. Entities are typically nouns like "Student," "Course," "Employee," or "Product."
- **Example**:

(Imagine a rectangle labeled "Student")

2. Weak Entity

- **Notation**: A rectangle with a double border.
- **Description**: A weak entity cannot be uniquely identified by its own attributes alone and relies on a "strong" entity for its identification. It typically has a partial key and is dependent on a related strong entity.
- **Example**:

(Imagine a double-bordered rectangle labeled "Dependent")

3. Attribute

- **Notation**: An ellipse (oval).
- **Description**: Represents a property or characteristic of an entity. Attributes are typically adjectives or properties like "Name," "Date of Birth," "Salary," or "Color."
- **Example**:

(Imagine an ellipse labeled "Student Name")

4. Key Attribute

- **Notation**: An ellipse with the attribute name underlined.
- **Description**: Represents an attribute that uniquely identifies each instance of an entity (i.e., a primary key).
- **Example**:

(Imagine an underlined ellipse labeled "Student ID")

5. Composite Attribute

- **Notation**: An ellipse with smaller ellipses branching from it.
- **Description**: Represents an attribute that can be divided into smaller sub-attributes. For example, an "Address" attribute might be broken down into "Street," "City," "State," and "ZIP Code."
- **Example**:

(Imagine an ellipse labeled "Address" with branches to smaller ellipses labeled "Street," "City," "State," and "ZIP Code")

6. Multivalued Attribute

- **Notation**: A double ellipse.

- **Description**: Represents an attribute that can have multiple values for a single entity. For instance, a "Phone Number" attribute for a "Person" entity could have several values.
- **Example**:

(Imagine a double ellipse labeled "Phone Number")

7. Derived Attribute

- **Notation**: A dashed ellipse.
- **Description**: Represents an attribute that can be derived or calculated from other attributes. For example, "Age" might be derived from the "Date of Birth."
- **Example**:

(Imagine a dashed ellipse labeled "Age")

8. Relationship

- **Notation**: A diamond.
- **Description**: Represents a relationship between two or more entities. Relationships can include verbs like "Enrolls," "Teaches," "Manages," or "Buys."
- **Example**:

(Imagine a diamond labeled "Enrolls" connecting "Student" and "Course" entities)

9. Identifying Relationship

- **Notation**: A diamond with a double border.
- **Description**: Represents a relationship where a weak entity is uniquely identified by being associated with another strong entity.
- **Example**:

(Imagine a double-bordered diamond labeled "DependentOf" connecting "Dependent" and "Employee")

10. Cardinality and Participation Constraints

- **Notation**: Numbers or symbols (1, N, M, etc.) placed near the entities involved in the relationship.
- **Description**: Specifies the number of instances of one entity that can or must be associated with each instance of another entity.
  - **1:1 Relationship**: A line with "1" at both ends.
  - **1**

Relationship: A line with "1" at one end and "M" or "N" at the other.

  - **M**

Relationship: A line with "M" or "N" at both ends.

- **Participation**:
  - **Total Participation**: Represented by a double line.
  - **Partial Participation**: Represented by a single line.
- **Example**:

(Imagine a relationship between "Employee" and "Department" where "1

" indicates one department can have many employees, and the line between them is single for partial participation)

11. Associative Entity (or Intersection Entity)

- **Notation**: A rectangle inside a diamond.
- **Description**: Represents a relationship that is treated as an entity because it has its own attributes. This is often used in many-to-many relationships.
- **Example**:

(Imagine a rectangle inside a diamond labeled "Enrollment" connecting "Student" and "Course")

12. Generalization/Specialization

- **Notation**: A triangle.
- **Description**: Represents an inheritance hierarchy, where a higher-level entity (superclass) is divided into lower-level entities (subclasses) based on some distinguishing characteristics.
- **Example**:

(Imagine a triangle connecting "Vehicle" to subclasses "Car" and "Truck")

13. Aggregation

- **Notation**: A dashed rectangle surrounding the involved entities and relationships.
- **Description**: Represents a complex relationship involving a group of entities and relationships that are treated as a single higher-level entity.
- **Example**:

(Imagine a dashed rectangle surrounding entities "Project" and "Worker" and their relationship, connecting to "Department")

3.1 Explain the 4 relational model constraints

Solution-
In the relational model, constraints are rules that ensure the accuracy and consistency of the data within a relational database. There are four primary types of relational model constraints:

1. Domain Constraints

- Definition: Domain constraints specify that all values in a column must be of a specific data type and fall within a particular range or set of allowable values.
- Example: If a column is defined to hold integer values between 1 and 100, any value outside this range would violate the domain constraint.
- Purpose: To ensure that data entered into the database adheres to the expected format and limits, thus maintaining data integrity.

2. Key Constraints

- Definition: Key constraints ensure that every tuple (row) in a relation (table) is uniquely identifiable.
- Types:
    - Primary Key: A primary key is a column or a set of columns that uniquely identifies each row in a table. No two rows can have the same value for the primary key, and it cannot contain NULL values.
    - Candidate Key: Any column or combination of columns that can uniquely identify a row. The primary key is selected from the candidate keys.
    - Unique Key: Similar to the primary key, but it can contain NULL values (though each non-null value must be unique).
- Example: In a table of employees, the EmployeeID might be the primary key, ensuring no two employees share the same ID.
- Purpose: To prevent duplicate records and maintain entity integrity.

3. Entity Integrity Constraints

- Definition: Entity integrity ensures that the primary key of a table is unique and not NULL.
- Example: In a student database, each student must have a unique and non-null StudentID. This ensures that every student can be uniquely identified.
- Purpose: To guarantee that each entity (row) within the table can be uniquely identified by its primary key, which is crucial for relational operations like joining tables.

4. Referential Integrity Constraints

- Definition: Referential integrity constraints ensure that a foreign key in one table matches a primary key in another table. This constraint prevents orphaned records.
- Example: In a database with a Students table and an Enrollments table, the Enrollments table might have a foreign key StudentID that references the StudentID in the Students table. If a student is deleted from the Students table, any corresponding enrollment records must also be deleted, or the deletion should be prevented.
- Purpose: To maintain the logical connections between tables, ensuring that relationships between tables remain consistent.

3.2 Explain all the steps of relational database design using ER to relational schema with a suitable example.

Solution-
Designing a relational database from an Entity-Relationship (ER) model involves several steps. The process translates the conceptual design represented by the ER diagram into a logical schema that can be implemented in a relational database management system (RDBMS). Below are the steps, illustrated with a suitable example.

Steps in Relational Database Design

1. Identify Entities and Relationships

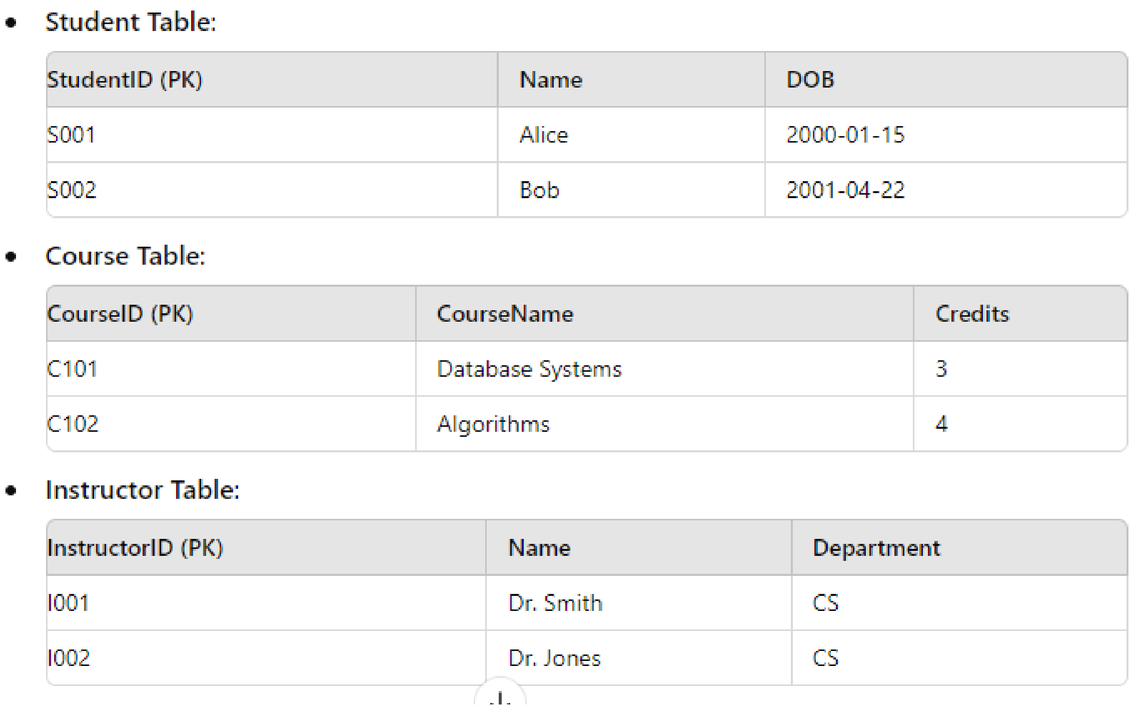
- Step: Start by identifying the entities and relationships from the ER diagram.
- Example: Suppose we are designing a database for a university. The entities could be Student, Course, and Instructor. The relationships could be:
    - Enrollment: between Student and Course
    - Teaches: between Instructor and Course

2. Define Primary Keys for Entities

- Step: Choose a unique identifier (primary key) for each entity.
- Example:
    - Student: StudentID
    - Course: CourseID
    - Instructor: InstructorID

3. Create Tables for Each Entity

- Step: Create a table for each entity identified in the ER diagram. Each attribute of the entity becomes a column in the table.
- Example:
    - Student Table:



- Student Table:

StudentID (PK)	Name	DOB
S001	Alice	2000-01-15
S002	Bob	2001-04-22

- Course Table:

CourseID (PK)	CourseName	Credits
C101	Database Systems	3
C102	Algorithms	4

- Instructor Table:

InstructorID (PK)	Name	Department
I001	Dr. Smith	CS
I002	Dr. Jones	CS

4. Map Relationships

- Step: Convert relationships into foreign keys and possibly additional tables.
  - One-to-One Relationships: Place the primary key of one entity as a foreign key in the other entity’s table.
  - One-to-Many Relationships: Place the primary key of the "one" side as a foreign key in the "many" side's table.
  - Many-to-Many Relationships: Create a new table to handle the many-to-many relationship, with foreign keys pointing to both related entities.

5. Normalize the Schema

- **Step**: Normalize the database schema to eliminate redundancy and ensure data integrity. Ensure the schema is in at least Third Normal Form (3NF).
- **Example**:
  - Ensure that each table is in 1NF by removing duplicate columns.
  - Ensure 2NF by removing partial dependencies; i.e., no non-key attribute should be dependent on a part of a composite primary key.
  - Ensure 3NF by ensuring no transitive dependencies; i.e., non-key attributes should depend only on the primary key.

6. Define Constraints

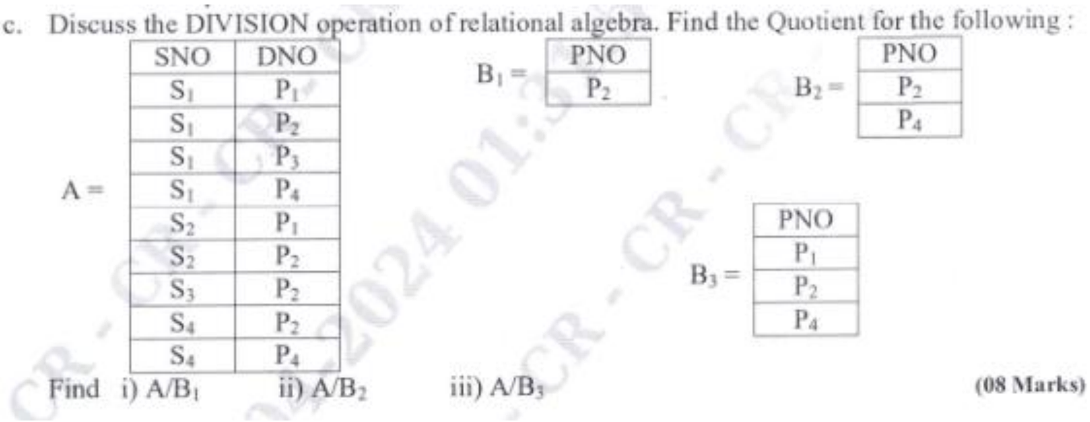
- **Step**: Define primary keys, foreign keys, unique constraints, and other constraints such as NOT NULL, CHECK, etc., to enforce data integrity.

- **Example**:
    - Primary Key: StudentID in Student, CourseID in Course.
    - Foreign Key: StudentID in Enrollment, InstructorID in Course.
    - Unique Constraint: Each CourseName might need to be unique.

7. Refine and Optimize the Design

- **Step**: Review the schema to optimize performance, add indexes, and refine the structure for clarity and efficiency.
- **Example**:
    - Create indexes on StudentID and CourseID in the Enrollment table to speed up queries.
    - If CourseName is frequently searched, consider indexing it.

3.3



c. Discuss the DIVISION operation of relational algebra. Find the Quotient for the following :

A =

SNO	DNO
S_1	P_1
S_1	P_2
S_1	P_3
S_1	P_4
S_2	P_1
S_2	P_2
S_3	P_2
S_4	P_2
S_4	P_4

B_1 =

PNO
P_2

B_2 =

PNO
P_2
P_4

B_3 =

PNO
P_1
P_2
P_4

Find i) A/B_1 ii) A/B_2 iii) A/B_3 **(08 Marks)**

Solution

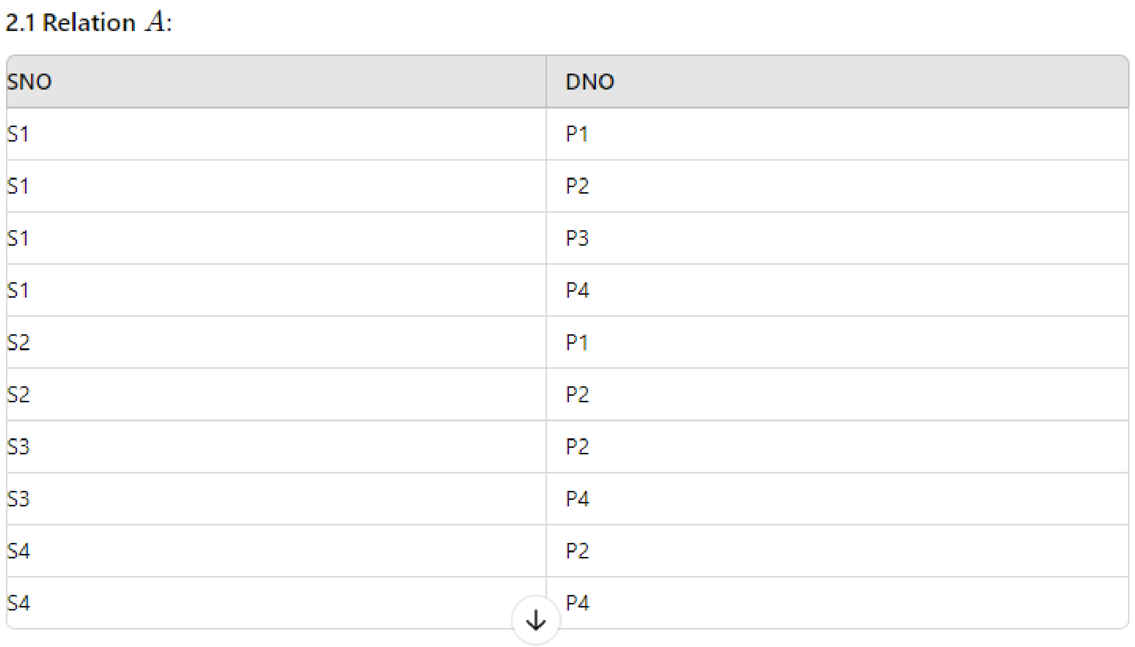
1. Division Operation in Relational Algebra

The DIVISION operation in relational algebra is used to find tuples in one relation (let's call it AAA) that match with all tuples in another relation (let's call it BBB). The operation A÷BA \div BA÷B returns the set of all tuples ttt in relation AAA such that for every tuple uuu in BBB, the combination of ttt and uuu appears in AAA.

2. Solve the Quotients

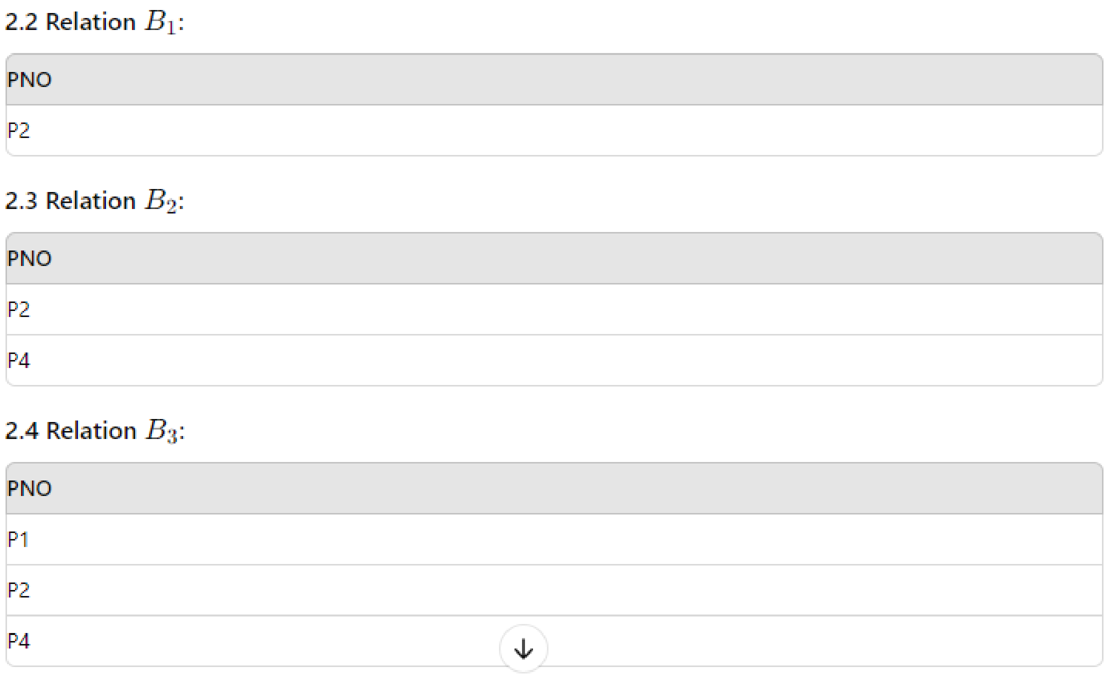
Given the relation AAA and relations B1B_1B1, B2B_2B2, and B3B_3B3, we need to compute A/B1A/B_1A/B1, A/B2A/B_2A/B2, and A/B3A/B_3A/B3.

2.1 Relation AAA:



2.1 Relation A:

SNO	DNO
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S3	P4
S4	P2
S4	P4



2.2 Relation B_1:

PNO
P2

2.3 Relation B_2:

PNO
P2
P4

2.4 Relation B_3:

PNO
P1
P2
P4

Solution:

i) A/B1A/B_1A/B1

We are looking for all SNO in AAA such that every tuple with DNO = P2 is associated with them.

From AAA, we can see that:

- S1S1S1 is associated with P2.
- S2S2S2 is associated with P2.
- S3S3S3 is associated with P2.
- S4S4S4 is associated with P2.

Thus, A/B1={S1,S2,S3,S4}A/B_1 = \{ S1, S2, S3, S4 \}A/B1={S1,S2,S3,S4}.

ii) A/B2A/B_2A/B2

We are looking for all SNO in AAA such that every tuple with DNO values P2 and P4 is associated with them.

From AAA, we can see that:

- S1S1S1 is associated with both P2 and P4.
- S4S4S4 is associated with both P2 and P4.
- S3S3S3 is associated with both P2 and P4.

Thus, A/B2={S1,S3,S4}A/B_2 = \{ S1, S3, S4 \}A/B2={S1,S3,S4}.

iii) A/B3A/B_3A/B3

We are looking for all SNO in AAA such that every tuple with DNO values P1, P2, and P4 is associated with them.

From AAA, we can see that:

- S1S1S1 is associated with P1, P2, and P4.

Thus, A/B3={S1}A/B_3 = \{ S1 \}A/B3={S1}.

Final Answers:

- A/B1={S1,S2,S3,S4}A/B_1 = \{ S1, S2, S3, S4 \}A/B1={S1,S2,S3,S4}
- A/B2={S1,S3,S4}A/B_2 = \{ S1, S3, S4 \}A/B2={S1,S3,S4}
- A/B3={S1}A/B_3 = \{ S1 \}A/B3={S1}

4.1 **Explain the characterstics of a relational model.**

Solution

The **relational model** is one of the most widely used database models, introduced by Edgar F. Codd in 1970. It organizes data into structured tables (relations) and provides a powerful framework for data management, ensuring data integrity, flexibility, and ease of use. Below are the key characteristics of the relational model:

1. Data Structure: Tables (Relations)

- **Definition**: Data is organized into tables, also known as relations.
- **Components**:
  - **Rows (Tuples)**: Each row represents a single record or instance of an entity.
  - **Columns (Attributes)**: Each column represents a specific attribute or property of the entity.
- **Example**: A Students table with columns StudentID, Name, Major, and GPA.

2. Tuples and Attributes

- **Tuples**: Represent individual records within a table. Each tuple contains a unique set of values for the table's attributes.
    - **Example**: (1001, "Alice Smith", "Computer Science", 3.8) is a tuple in the Students table.
- **Attributes**: Define the properties of the entities. Each attribute has a specific data type and domain.
    - **Example**: StudentID might be an integer, Name a string, GPA a decimal.

3. Primary Keys

- **Definition**: A primary key is an attribute (or a combination of attributes) that uniquely identifies each tuple in a table.
- **Purpose**: Ensures that each record can be uniquely retrieved, preventing duplicate entries.
- **Example**: StudentID serves as a primary key in the Students table.

4. Foreign Keys

- **Definition**: A foreign key is an attribute that creates a link between two tables, referencing the primary key of another table.
- **Purpose**: Establishes and enforces relationships between tables, maintaining referential integrity.
- **Example**: DepartmentID in the Courses table might be a foreign key referencing DepartmentID in the Departments table.

5. Data Integrity Constraints

The relational model enforces several types of integrity constraints to maintain the accuracy and consistency of data:

- **Entity Integrity**:
    - **Definition**: Ensures that each table has a primary key and that the primary key attributes are unique and not null.
    - **Example**: No two students can have the same StudentID, and StudentID cannot be null.
- **Referential Integrity**:
    - **Definition**: Ensures that foreign keys correctly and consistently reference existing primary keys in related tables.
    - **Example**: Every DepartmentID in the Courses table must exist in the Departments table.
- **Domain Integrity**:
    - **Definition**: Ensures that all attributes in a table adhere to their defined data types, formats, and permissible values.

    - **Example**: The GPA attribute must be a decimal between 0.0 and 4.0.

- **User-Defined Integrity**:
    - **Definition**: Enforces specific business rules that do not fall under entity, referential, or domain integrity.
    - **Example**: A rule that a student's GPA must be recalculated every semester.

6. Normalization

- **Definition**: The process of organizing data to reduce redundancy and improve data integrity.
- **Purpose**: Divides tables into smaller, related tables and defines relationships between them.
- **Normal Forms**:
    - **First Normal Form (1NF)**: Ensures that each table cell contains only atomic (indivisible) values.
    - **Second Normal Form (2NF)**: Removes partial dependencies, ensuring that non-key attributes are fully functional dependent on the primary key.
    - **Third Normal Form (3NF)**: Removes transitive dependencies, ensuring that non-key attributes are not dependent on other non-key attributes.
- **Example**: Splitting a Students table into Students and Enrollments tables to handle many-to-many relationships with Courses.

7. Relational Algebra and Query Language

- **Relational Algebra**: A set of operations (such as select, project, join, union, intersection, difference) used to manipulate and retrieve data from tables.
- **SQL (Structured Query Language)**: A declarative language based on relational algebra that allows users to perform queries, updates, insertions, and deletions.
- **Example**:
    - **Relational Algebra**: SELECT Name FROM Students WHERE GPA > 3.5
    - **SQL**: SELECT Name FROM Students WHERE GPA > 3.5;

8. Data Independence

- **Logical Data Independence**:
    - **Definition**: Changes to the logical schema (e.g., adding a new column) do not affect the external schemas or application programs.
- **Physical Data Independence**:
    - **Definition**: Changes to the physical storage (e.g., indexing methods) do not affect the logical schema.
- **Purpose**: Enhances the flexibility and maintainability of the database by isolating different levels of abstraction.

9. Mathematical Foundation

- **Set Theory and Predicate Logic**: The relational model is grounded in mathematical concepts, providing a rigorous and precise framework for data manipulation and query formulation.
- **Benefit**: Ensures consistency, predictability, and reliability in how data is handled and queried.

10. Schema and Subschema

- **Schema**: The overall logical structure of the database, defining tables, columns, data types, relationships, and constraints.
  - **Example**: A university database schema including Students, Courses, Professors, and Departments tables with their respective relationships.
- **Subschema (External Schema)**: Specific views or subsets of the schema tailored to different users or applications.
  - **Example**: A student view showing only their own grades and enrolled courses, while a professor view shows the courses they teach and the students enrolled.

11. Atomicity of Data

- **Definition**: Ensures that each attribute in a table holds atomic, indivisible values.
- **Purpose**: Simplifies data manipulation and querying by avoiding complex data structures within individual fields.
- **Example**: Storing FullName as separate FirstName and LastName attributes instead of a single FullName attribute.

12. Flexibility and Scalability

- **Flexibility**: The tabular structure allows for easy addition, deletion, and modification of data without significant restructuring.
- **Scalability**: Efficiently handles large volumes of data and complex queries through indexing, normalization, and optimized query processing.

13. Data Manipulation and Integrity

- **Transactional Support**: Ensures that database transactions are processed reliably, adhering to ACID (Atomicity, Consistency, Isolation, Durability) properties.
- **Concurrency Control**: Manages simultaneous data access by multiple users, preventing conflicts and ensuring data consistency.

14. Security and Access Control

- **Role-Based Access**: Defines user roles and permissions at various levels (table, column, row) to secure data.
- **Example**: Only authorized personnel can access sensitive information like student grades or financial records.

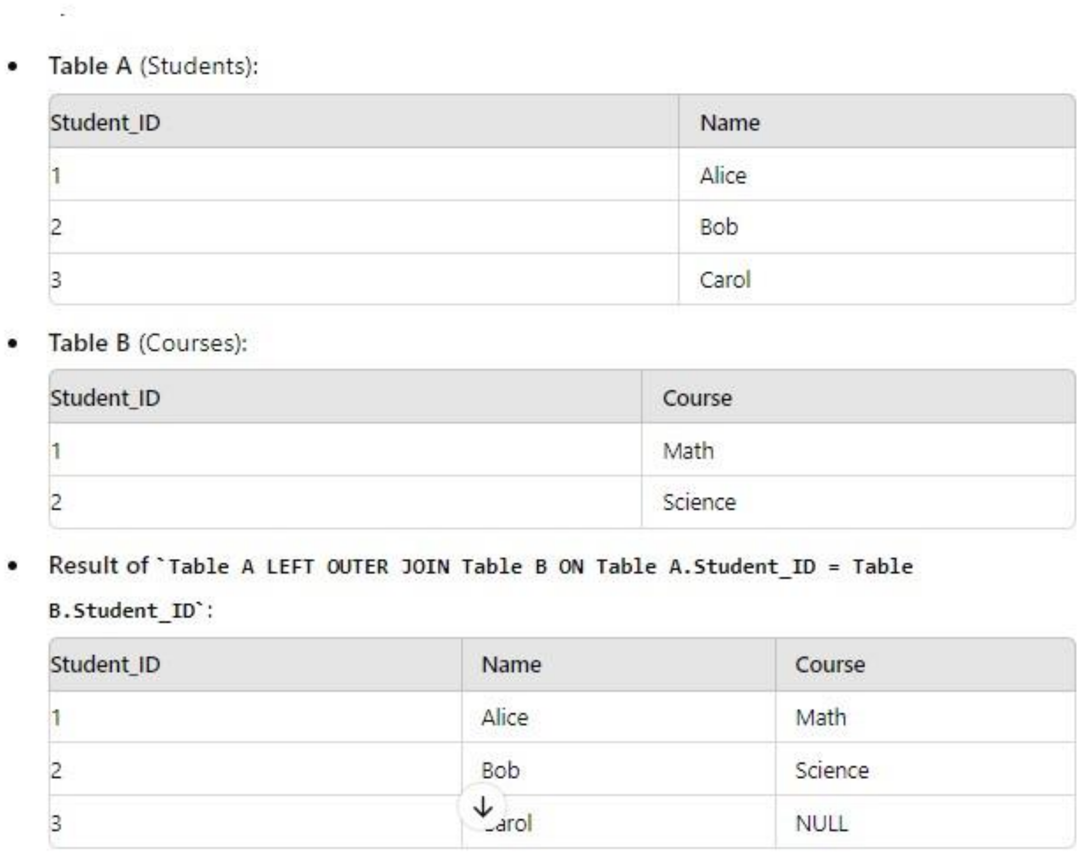
15. Declarative Nature

- **Definition**: Users specify *what* data they need without needing to define *how* to retrieve it.
- **Advantage**: Simplifies query writing and allows the DBMS to optimize the execution plan for efficient data retrieval.
- **Example**: Writing SELECT Name FROM Students WHERE GPA > 3.5; without specifying the steps to access the data.

4.2 **Explain all type of outer join operations in relational algebra. Demonstrate the advantage of outer join over inner join.**

Solution
In relational algebra, **joins** are operations that allow you to combine rows from two or more tables based on a related column between them. The **outer join** operations are a type of join that, unlike inner joins, include rows that do not have matching values in both tables. There are three main types of outer joins:

1. Left Outer Join (LEFT JOIN)

- **Description**: This operation returns all the rows from the **left** table and the matching rows from the right table. If there is no match, the result will include the rows from the left table with NULL values in the columns from the right table.
- **Relational Algebra Notation**: R ⟕ S
- 

  - Table A (Students):

Student_ID	Name
1	Alice
2	Bob
3	Carol

  - Table B (Courses):

Student_ID	Course
1	Math
2	Science

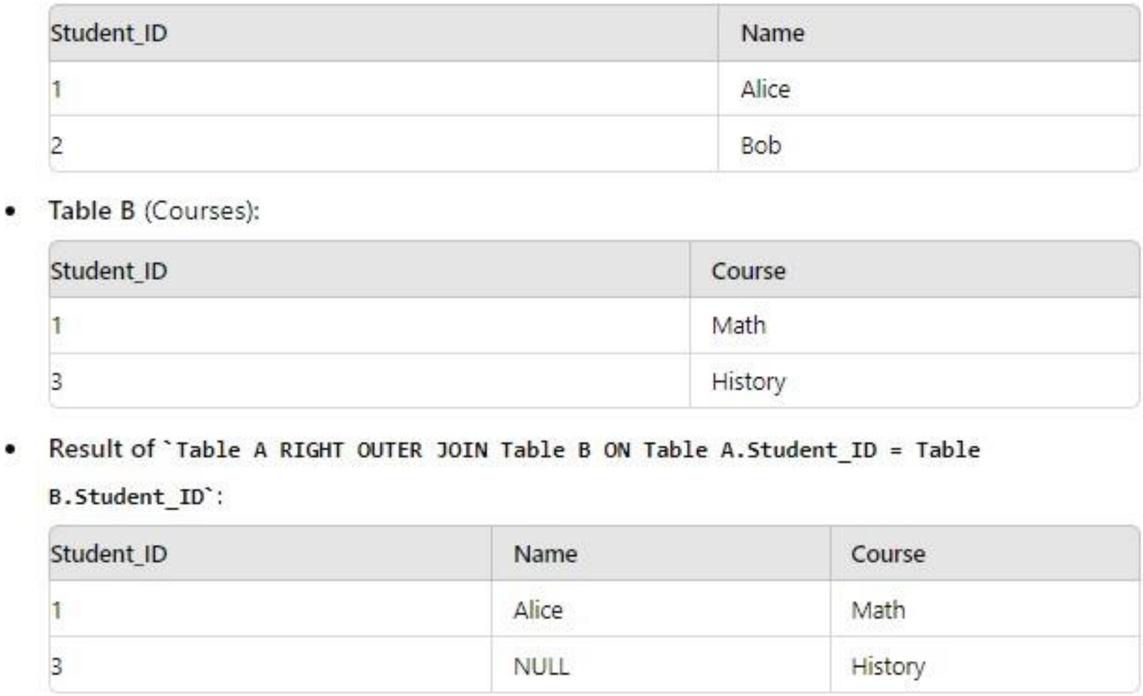
  - Result of `Table A LEFT OUTER JOIN Table B ON Table A.Student_ID = Table B.Student_ID`:

Student_ID	Name	Course
1	Alice	Math
2	Bob	Science
3	Carol	NULL

-

Right Outer Join (RIGHT JOIN)

- **Description**: This operation returns all the rows from the **right** table and the matching rows from the left table. If there is no match, the result will include the rows from the right table with NULL values in the columns from the left table.
- **Relational Algebra Notation**: R ⟖ S
- **Example**



Student_ID	Name
1	Alice
2	Bob

- Table B (Courses):

Student_ID	Course
1	Math
3	History

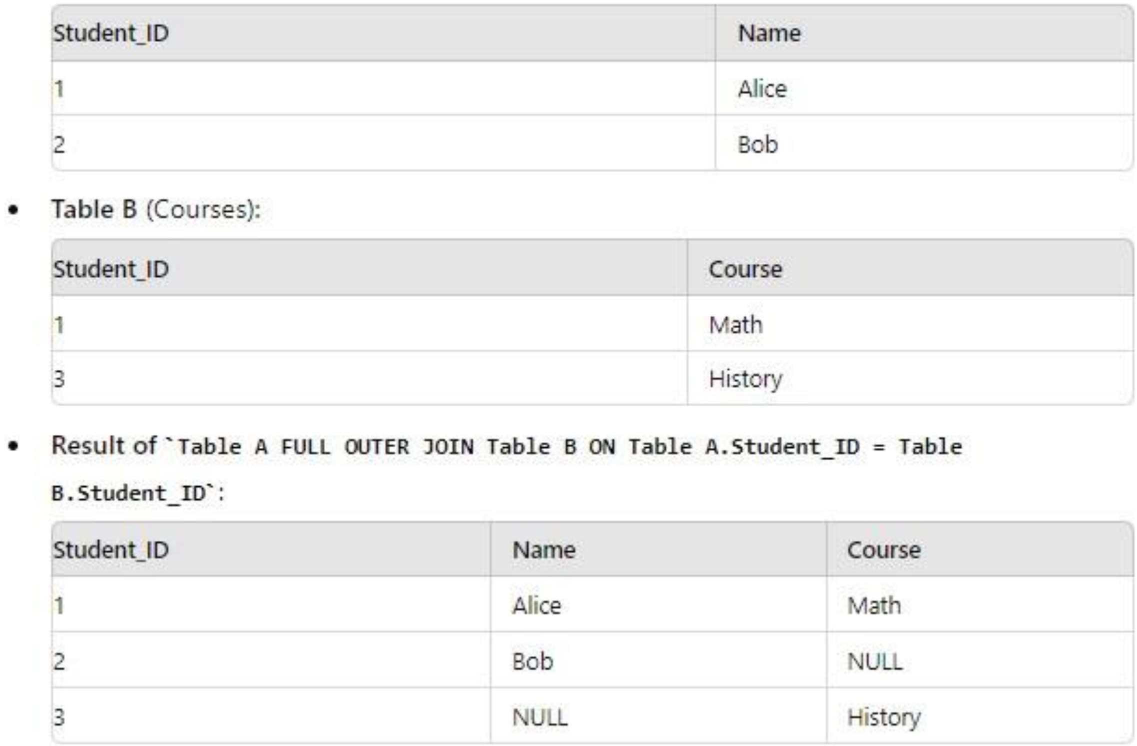
- Result of `Table A RIGHT OUTER JOIN Table B ON Table A.Student_ID = Table B.Student_ID`:

Student_ID	Name	Course
1	Alice	Math
3	NULL	History

-

Full Outer Join (FULL JOIN)

- **Description**: This operation returns all the rows when there is a match in either the left or right table. If there is no match, the result will include the rows with NULL values in the columns from the table that doesn't have the match.
- **Relational Algebra Notation**: R ⟗ S
- **Example**:



Student_ID	Name
1	Alice
2	Bob

- Table B (Courses):

Student_ID	Course
1	Math
3	History

- Result of `` `Table A FULL OUTER JOIN Table B ON Table A.Student_ID = Table B.Student_ID` ``:

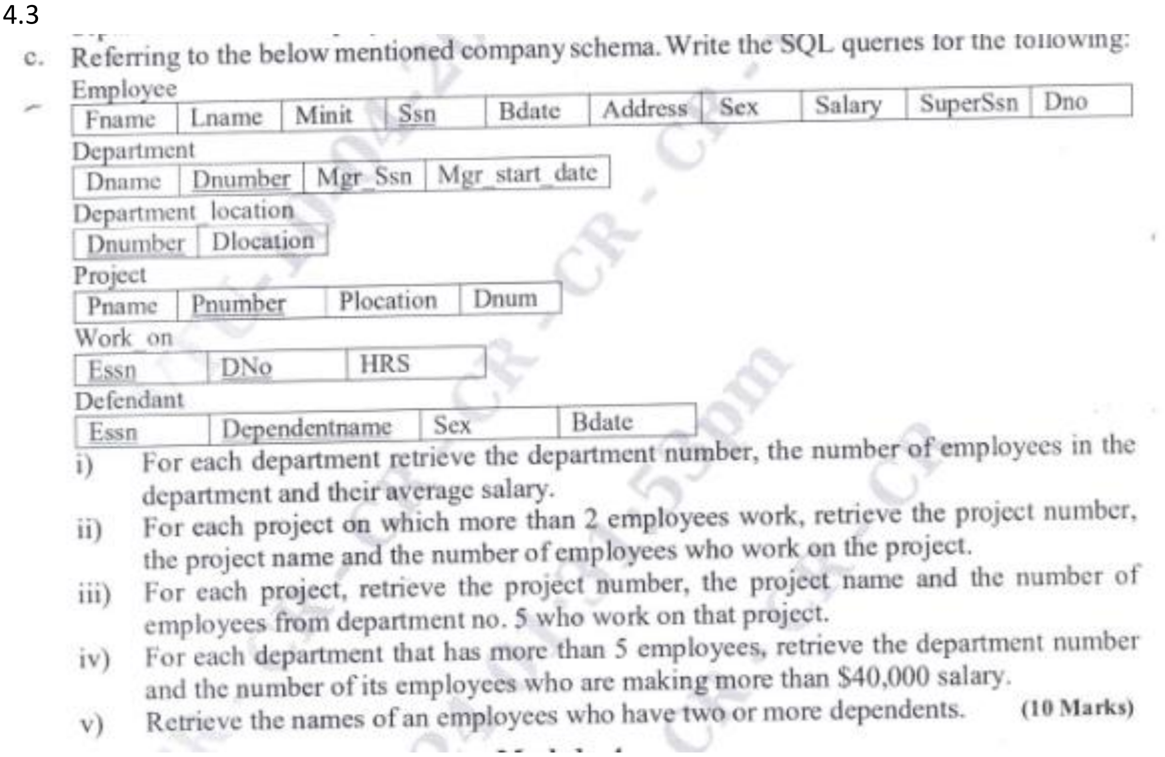
Student_ID	Name	Course
1	Alice	Math
2	Bob	NULL
3	NULL	History

-

Advantage of Outer Join Over Inner Join

- **Inner Join** returns only the rows that have matching values in both tables. This can be limiting when you need to include rows that don't have a match, especially if you're working with incomplete or sparse data.
- **Outer Join** operations (Left, Right, Full) address this limitation by including non-matching rows and filling in the gaps with NULL values. This is particularly useful when you need to:
    1. **Identify unmatched data**: Outer joins help to identify rows that do not have corresponding matches in the other table. For instance, you might want to find all students who haven't enrolled in any courses or all courses that no students have signed up for.
    2. **Preserve all data**: When you need to include all records from one or both tables, regardless of whether they have a match, outer joins are essential.
    3.

4.3



c. Referring to the below mentioned company schema. Write the SQL queries for the following:

Employee

Fname	Lname	Minit	Ssn	Bdate	Address	Sex	Salary	SuperSsn	Dno

Department

Dname	Dnumber	Mgr_Ssn	Mgr_start_date

Department_location

Dnumber	Dlocation

Project

Pname	Pnumber	Plocation	Dnum

Work_on

Essn	DNo	HRS

Defendant

Essn	Dependentname	Sex	Bdate

i) For each department retrieve the department number, the number of employees in the department and their average salary.

ii) For each project on which more than 2 employees work, retrieve the project number, the project name and the number of employees who work on the project.

iii) For each project, retrieve the project number, the project name and the number of employees from department no. 5 who work on that project.

iv) For each department that has more than 5 employees, retrieve the department number and the number of its employees who are making more than $40,000 salary.

v) Retrieve the names of an employees who have two or more dependents. **(10 Marks)**

Solution

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT
   Dnumber,
   COUNT(E.Ssn) AS Number_of_Employees,
   AVG(Salary) AS Average_Salary
FROM
   Department D
JOIN
   Employee E ON D.Dnumber = E.Dno
GROUP BY
   Dnumber;
```

For each project on which more than 2 employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```
SELECT
   P.Pnumber,
   P.Pname,
   COUNT(W.Essn) AS Number_of_Employees
```

```
FROM
   Project P
JOIN
   Work_on W ON P.Pnumber = W.Pno
GROUP BY
   P.Pnumber, P.Pname
HAVING
   COUNT(W.Essn) > 2;
```

For each project, retrieve the project number, the project name, and the number of employees from department no. 5 who work on that project.

```
SELECT
   P.Pnumber,
   P.Pname,
   COUNT(W.Essn) AS Number_of_Employees_from_Dept_5
FROM
   Project P
JOIN
   Work_on W ON P.Pnumber = W.Pno
JOIN
   Employee E ON W.Essn = E.Ssn
WHERE
   E.Dno = 5
GROUP BY
   P.Pnumber, P.Pname;
```

For each department that has more than 5 employees, retrieve the department number and the number of its employees who are making more than $40,000 salary.

```
SELECT
   E.Dno,
   COUNT(E.Ssn) AS Number_of_Employees
FROM
   Employee E
JOIN
```

```
  Department D ON E.Dno = D.Dnumber
WHERE
  E.Salary > 40000
GROUP BY
  E.Dno
HAVING
  COUNT(E.Ssn) > 5;
```

Retrieve the names of all employees who have two or more dependents.

```
SELECT
  E.Fname,
  E.Lname
FROM
  Employee E
JOIN
  Dependent D ON E.Ssn = D.Essn
GROUP BY
  E.Ssn, E.Fname, E.Lname
HAVING
  COUNT(D.Dependentname) >= 2;
```

5.1 Explain basic data types available for attributes in SQL.

Solution
In SQL, attributes (or columns) in a database table can be of various data types. These data types define the kind of data that can be stored in that attribute, ensuring data integrity and optimizing storage. Here's an overview of the basic data types available in SQL:

1. Numeric Data Types

These are used to store numbers, which can be either integers or real numbers.

- **INT/INTEGER**:
  - Stores whole numbers (both positive and negative).
  - Example: INT, INTEGER.
  - Range: Typically -2,147,483,648 to 2,147,483,647 (4 bytes).
- **SMALLINT**:

  - Stores smaller range of whole numbers.
  - Range: Typically -32,768 to 32,767 (2 bytes).
- **BIGINT**:
  - Stores larger whole numbers.
  - Range: Typically -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (8 bytes).
- **DECIMAL(p, s)** or **NUMERIC(p, s)**:
  - Stores fixed-point numbers where p is the precision (total number of digits) and s is the scale (number of digits to the right of the decimal point).
  - Example: DECIMAL(10, 2) can store numbers up to 10 digits with 2 decimal places (e.g., 12345678.90).
- **FLOAT** and **REAL**:
  - Stores floating-point numbers, which are numbers with decimal points that can accommodate a wide range of values.
  - Example: FLOAT (can store very large or very small numbers but with precision trade-offs).

2. Character/String Data Types

These are used to store text.

- **CHAR(n)** or **CHARACTER(n)**:
  - Stores fixed-length strings. If the length of the string is less than n, it will be padded with spaces.
  - Example: CHAR(10) will always store 10 characters, even if you enter fewer.
- **VARCHAR(n)** or **CHARACTER VARYING(n)**:
  - Stores variable-length strings up to n characters. It only uses as much space as needed.
  - Example: VARCHAR(50) can store strings of varying lengths up to 50 characters.
- **TEXT**:
  - Stores large strings of text. The maximum length varies depending on the database system, but it typically allows storage of much larger text than VARCHAR.
  - Example: TEXT can be used for storing paragraphs or entire documents.

3. Date and Time Data Types

These are used to store dates, times, and timestamps.

- **DATE**:
  - Stores date values (year, month, day).

  - Example: DATE stores 'YYYY-MM-DD' format.

- **TIME**:
  - Stores time values (hours, minutes, seconds).
  - Example: TIME stores 'HH:MI

' format.

- **TIMESTAMP**:
  - Stores both date and time (year, month, day, hours, minutes, seconds).
  - Example: TIMESTAMP stores 'YYYY-MM-DD HH:MI

' format.

- **DATETIME**:
  - Stores date and time with a wider range compared to TIMESTAMP.
  - Example: DATETIME is similar to TIMESTAMP but with broader application.
- **INTERVAL**:
  - Stores a time interval. It is used for representing a span of time (e.g., hours, days, months).
  - Example: INTERVAL '1' DAY adds 1 day to a date.

4. Boolean Data Type

- **BOOLEAN**:
  - Stores Boolean values, which can be either TRUE or FALSE.
  - Example: BOOLEAN values can be used in conditional statements.

5. Binary Data Types

These are used to store binary data (such as images or files).

- **BINARY(n)**:
  - Stores fixed-length binary data.
  - Example: BINARY(16) will store 16 bytes of binary data.
- **VARBINARY(n)**:
  - Stores variable-length binary data up to n bytes.
  - Example: VARBINARY(256) can store up to 256 bytes of binary data.
- **BLOB** (Binary Large Object):
  - Stores large binary data. It is used for storing files like images, audio, or video.
  - Example: BLOB can store data in binary format, commonly used for multimedia.

6. Other Data Types

- **ENUM**:
  - Stores one of the predefined values. It is commonly used when you want to restrict the value of a column to a specific list of values.
  - Example: ENUM('small', 'medium', 'large') can store any of the three values.
- **SET**:
  - Stores a set of values. Similar to ENUM but allows multiple values to be selected.
  - Example: SET('A', 'B', 'C') can store any combination of the specified values.

5.2 Demonstrate the following constraints in SQL
NOT NULL, primary, check, foreign key, default

Solution
In SQL, constraints are used to enforce rules on data columns in a database table. Below are examples demonstrating how to implement the **NOT NULL**, **PRIMARY KEY**, **CHECK**, **FOREIGN KEY**, and **DEFAULT** constraints in SQL.

1. NOT NULL Constraint

The NOT NULL constraint ensures that a column cannot have a NULL value.

sql

Copy code

```
CREATE TABLE Employee (
    EmpID INT NOT NULL,         -- EmpID cannot be NULL
    FName VARCHAR(50) NOT NULL, -- First Name cannot be NULL
    LName VARCHAR(50)           -- Last Name can be NULL
);
```

2. PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table. It also implies NOT NULL.

sql

Copy code

```
CREATE TABLE Department (
    DeptID INT PRIMARY KEY,     -- DeptID is the primary key and cannot be NULL
    DeptName VARCHAR(100) NOT NULL
);
```

Note: You can define the primary key at the column level as shown above, or at the table level:

sql

Copy code

```
CREATE TABLE Department (
    DeptID INT,
    DeptName VARCHAR(100) NOT NULL,
    PRIMARY KEY (DeptID)        -- Defining the primary key at the table level
);
```

3. CHECK Constraint

The CHECK constraint ensures that all values in a column satisfy a specific condition.

sql

Copy code

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    FName VARCHAR(50) NOT NULL,
    Age INT CHECK (Age >= 18),  -- Age must be 18 or older
    Salary DECIMAL(10, 2) CHECK (Salary > 0)  -- Salary must be greater than 0
);
```

4. FOREIGN KEY Constraint

The FOREIGN KEY constraint is used to link two tables. It ensures that the value in a column (or a group of columns) must match values in another table's primary key or unique key.

sql

Copy code

```
CREATE TABLE Project (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100) NOT NULL,
    DeptID INT,               -- Foreign key references the Department table
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)  -- Enforces referential integrity
);
```

5. DEFAULT Constraint

The DEFAULT constraint provides a default value for a column when no value is specified.

sql

Copy code

```
CREATE TABLE Employee (
   EmpID INT PRIMARY KEY,
   FName VARCHAR(50) NOT NULL,
   LName VARCHAR(50),
   StartDate DATE DEFAULT CURRENT_DATE,  -- Sets the default start date to the current date
   Salary DECIMAL(10, 2) DEFAULT 50000   -- Sets the default salary to 50000
);
```

5.3 What are triggers and explain the syntax.

Solution
A **trigger** in SQL is a database object that is automatically executed or fired when certain events occur in a table or view. Triggers are used to enforce business rules, validate data, or update data in a database. They are typically associated with INSERT, UPDATE, and DELETE operations.

Triggers can be used to:

- Automatically update a timestamp when a record is updated.
- Log changes to a table.
- Enforce referential integrity.
- Maintain derived or aggregate values.
- Validate input data.

Types of Triggers

1. **BEFORE Trigger**: Executes before the operation (INSERT, UPDATE, DELETE) is performed.
2. **AFTER Trigger**: Executes after the operation is completed.
3. **INSTEAD OF Trigger**: Used with views to execute custom actions instead of the operation (common with views).

Basic Syntax of a Trigger

The syntax for creating a trigger may vary slightly between different database management systems (DBMSs). Here is a general structure:

sql

Copy code

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
ON table_name
```

```
[FOR EACH ROW]

[WHEN (condition)]

BEGIN

   -- SQL statements to be executed

END;
```

Components of the Syntax

- **CREATE TRIGGER trigger_name**: This statement is used to create a new trigger, where trigger_name is the name of the trigger.
- **{BEFORE | AFTER | INSTEAD OF}**: Specifies when the trigger should be executed in relation to the triggering event:
    - BEFORE: The trigger activates before the triggering event.
    - AFTER: The trigger activates after the triggering event.
    - INSTEAD OF: The trigger replaces the triggering event, commonly used with views.
- **{INSERT | UPDATE | DELETE}**: Specifies the triggering event that will activate the trigger.
    - INSERT: Trigger fires when a new record is inserted.
    - UPDATE: Trigger fires when a record is updated.
    - DELETE: Trigger fires when a record is deleted.
- **ON table_name**: Specifies the table or view on which the trigger is set.
- **[FOR EACH ROW]**: This clause indicates that the trigger will execute once for each row affected by the triggering event.
- **[WHEN (condition)]**: An optional clause that specifies a condition that must be met for the trigger to execute.
- **BEGIN ... END**: Defines the block of SQL code that will execute when the trigger fires. Inside this block, you can write one or more SQL statements.

Example 1: BEFORE INSERT Trigger

This example creates a trigger that automatically sets the created_at field to the current date and time before a new record is inserted into the Employees table.

sql

Copy code

```
CREATE TRIGGER set_created_at

BEFORE INSERT

ON Employees

FOR EACH ROW
```

```
BEGIN
  SET NEW.created_at = CURRENT_TIMESTAMP;
END;
```

- **NEW.created_at** refers to the created_at field in the new record being inserted.
- The trigger sets the created_at field to the current timestamp before the insertion is completed.

Example 2: AFTER UPDATE Trigger

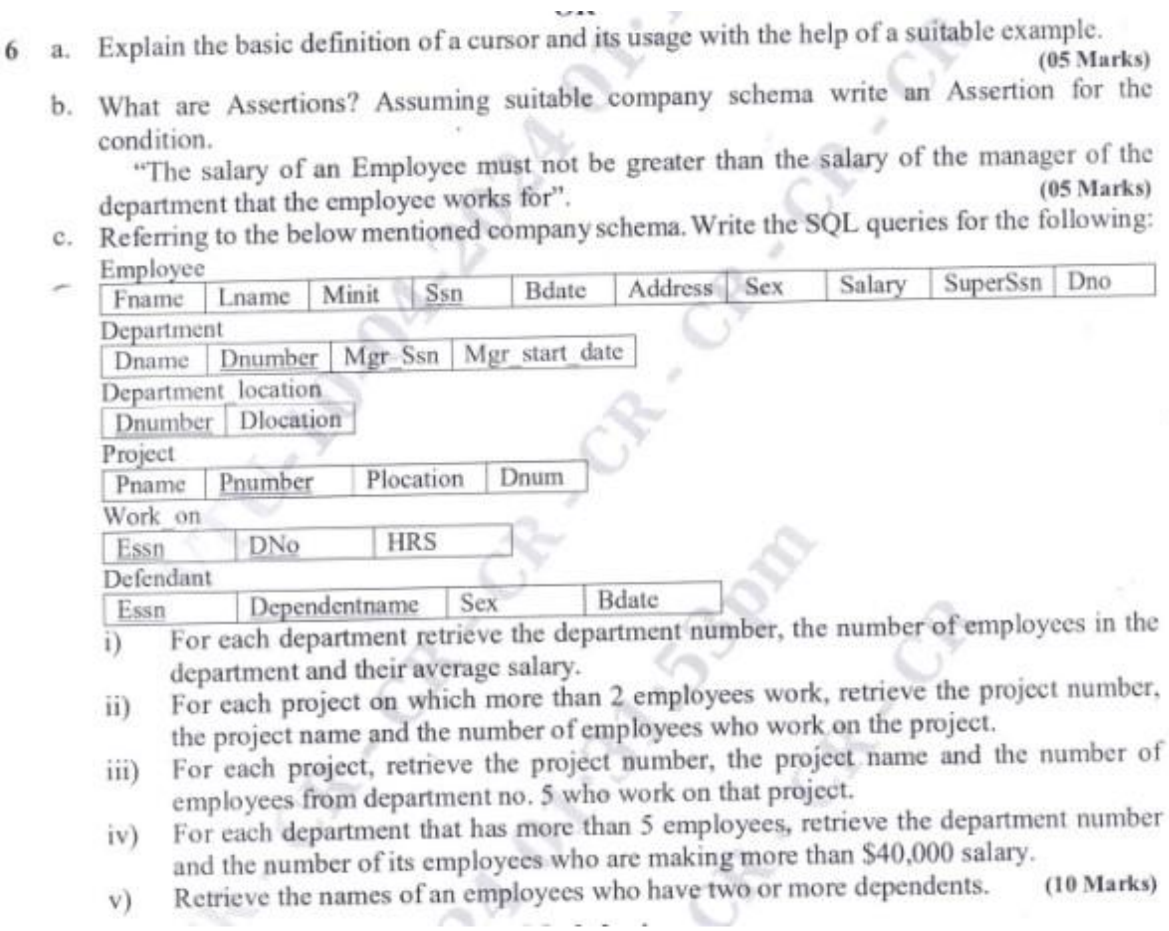
This example creates a trigger that logs changes to the Salary field in an Employee table after an update.

sql

Copy code

```
CREATE TRIGGER log_salary_update
AFTER UPDATE OF Salary
ON Employee
FOR EACH ROW
BEGIN
  INSERT INTO Salary_Log (EmpID, OldSalary, NewSalary, ChangeDate)
  VALUES (OLD.EmpID, OLD.Salary, NEW.Salary, CURRENT_TIMESTAMP);
END;
```

- **OLD.Salary** refers to the value of the Salary field before the update.
- **NEW.Salary** refers to the value of the Salary field after the update.
- The trigger inserts a log entry into the Salary_Log table every time the Salary field is updated.



6 a. Explain the basic definition of a cursor and its usage with the help of a suitable example. (05 Marks)

b. What are Assertions? Assuming suitable company schema write an Assertion for the condition.

"The salary of an Employee must not be greater than the salary of the manager of the department that the employee works for". (05 Marks)

c. Referring to the below mentioned company schema. Write the SQL queries for the following:

Employee

Fname	Lname	Minit	Ssn	Bdate	Address	Sex	Salary	SuperSsn	Dno

Department

Dname	Dnumber	Mgr_Ssn	Mgr_start_date

Department_location

Dnumber	Dlocation

Project

Pname	Pnumber	Plocation	Dnum

Work_on

Essn	DNo	HRS

Defendant

Essn	Dependentname	Sex	Bdate

i) For each department retrieve the department number, the number of employees in the department and their average salary.

ii) For each project on which more than 2 employees work, retrieve the project number, the project name and the number of employees who work on the project.

iii) For each project, retrieve the project number, the project name and the number of employees from department no. 5 who work on that project.

iv) For each department that has more than 5 employees, retrieve the department number and the number of its employees who are making more than $40,000 salary.

v) Retrieve the names of an employees who have two or more dependents. (10 Marks)

Solution-

6 a. Explain the basic definition of a cursor and its usage with the help of a suitable example.

Cursor: A cursor in SQL is a database object used to retrieve, manipulate, and iterate through the result set of a query one row at a time. Cursors are particularly useful when you need to process individual rows returned by a query, especially when performing operations that cannot be executed with set-based SQL operations.

Usage Example:

sql

Copy code

```
DECLARE @EmpID INT, @EmpName VARCHAR(50)


DECLARE employee_cursor CURSOR FOR

SELECT EmpID, FName FROM Employee


OPEN employee_cursor


FETCH NEXT FROM employee_cursor INTO @EmpID, @EmpName
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
  -- Example operation: Print each employee's name
  PRINT 'Employee ID: ' + CAST(@EmpID AS VARCHAR) + ', Name: ' + @EmpName

  FETCH NEXT FROM employee_cursor INTO @EmpID, @EmpName
END

CLOSE employee_cursor
DEALLOCATE employee_cursor
```

In this example, the cursor employee_cursor iterates over all rows returned by the SELECT query and prints the employee ID and name for each row.

6 b. What are Assertions? Assuming a suitable company schema, write an Assertion for the condition: "The salary of an Employee must not be greater than the salary of the manager of the department that the employee works for."

Assertions: An assertion in SQL is a condition that must always hold true for the database. Assertions are used to enforce complex integrity constraints that involve multiple rows or tables. However, note that not all SQL databases support assertions.

Example Assertion:

```
CREATE ASSERTION Salary_Assertion
CHECK (
  NOT EXISTS (
    SELECT *
    FROM Employee E JOIN Department D ON E.Dno = D.Dnumber
    WHERE E.Salary > (
      SELECT Mgr.Salary
      FROM Employee Mgr
      WHERE Mgr.Ssn = D.Mgr_Ssn
    )
  )
);
```

This assertion ensures that no employee's salary exceeds the salary of the manager of the department they work in.

6 c. SQL Queries based on the provided company schema:

i) For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT D.Dnumber, COUNT(E.Ssn) AS NumEmployees, AVG(E.Salary) AS AvgSalary

FROM Department D

JOIN Employee E ON D.Dnumber = E.Dno

GROUP BY D.Dnumber;
```

ii) For each project on which more than 2 employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```
SELECT P.Pnumber, P.Pname, COUNT(W.Essn) AS NumEmployees

FROM Project P

JOIN Work_on W ON P.Pnumber = W.Pno

GROUP BY P.Pnumber, P.Pname

HAVING COUNT(W.Essn) > 2;
```

iii) For each project, retrieve the project number, the project name, and the number of employees from department no. 5 who work on that project.

```
SELECT P.Pnumber, P.Pname, COUNT(W.Essn) AS NumEmployees

FROM Project P

JOIN Work_on W ON P.Pnumber = W.Pno

JOIN Employee E ON W.Essn = E.Ssn

WHERE E.Dno = 5

GROUP BY P.Pnumber, P.Pname;
```

iv) For each department that has more than 5 employees, retrieve the department number and the number of its employees who are making more than $40,000 salary.

```
SELECT D.Dnumber, COUNT(E.Ssn) AS NumHighEarners

FROM Department D

JOIN Employee E ON D.Dnumber = E.Dno

WHERE E.Salary > 40000

GROUP BY D.Dnumber

HAVING COUNT(E.Ssn) > 5;
```

v) Retrieve the names of employees who have two or more dependents.

```
SELECT E.Fname, E.Lname

FROM Employee E

JOIN Dependent D ON E.Ssn = D.Essn

GROUP BY E.Fname, E.Lname

HAVING COUNT(D.Dependentname) >= 2;
```

7.1 Explain the types of update anomalies with example.

Solution-

Update anomalies occur in relational databases when data redundancy and poor table design lead to inconsistencies and unintended consequences during data modification operations. These anomalies typically arise in tables that are not properly normalized. There are three main types of update anomalies: **Insertion Anomaly**, **Update Anomaly**, and **Deletion Anomaly**. Below is an explanation of each type, along with examples.

1. Insertion Anomaly

Definition: An insertion anomaly occurs when certain attributes cannot be inserted into the database without the presence of other attributes.

Example: Suppose we have a table Student_Course that stores information about students and the courses they are enrolled in:

StudentID	StudentName	CourseID	CourseName	Instructor
1	Alice	101	Data Science	Dr. Smith
2	Bob	102	AI	Dr. White

- **Anomaly**: If a new course is introduced (e.g., CourseID = 103, CourseName = "Database Systems", Instructor = "Dr. Green"), but no students are yet enrolled in it, the insertion cannot happen without also inserting a StudentID and StudentName. This forces the insertion of incomplete or misleading data (e.g., dummy student data).

2. Update Anomaly

Definition: An update anomaly occurs when data is duplicated in multiple rows, and a change in one row requires changes in multiple rows to maintain consistency.

Example: Using the same Student_Course table:

StudentID	StudentName	CourseID	CourseName	Instructor
1	Alice	101	Data Science	Dr. Smith
2	Bob	101	Data Science	Dr. Smith

StudentID	StudentName	CourseID	CourseName	Instructor
3	Carol	101	Data Science	Dr. Smith

- **Anomaly**: If the instructor for the "Data Science" course changes to "Dr. Brown," every occurrence of "Dr. Smith" for CourseID = 101 must be updated. If one instance is missed, the data will become inconsistent.

3. Deletion Anomaly

Definition: A deletion anomaly occurs when the deletion of data representing certain facts also results in the unintended loss of additional data.

Example: Continuing with the Student_Course table:

StudentID	StudentName	CourseID	CourseName	Instructor
1	Alice	101	Data Science	Dr. Smith
2	Bob	102	AI	Dr. White

- **Anomaly**: If Alice is the only student enrolled in the "Data Science" course and we delete her record, we also lose all information about the course "Data Science" and its instructor, "Dr. Smith," even though this information could still be relevant.

Preventing Update Anomalies with Normalization

To avoid update anomalies, database tables should be normalized, which involves organizing the tables in such a way that:

- **Redundancy** is minimized, reducing the chances of inconsistencies.
- **Data dependencies** are logical, meaning related data is stored together, and unrelated data is separated.

For example, in a normalized design, instead of having a single Student_Course table, the data might be divided into two tables:

1. **Student** table:

StudentID	StudentName
1	Alice
2	Bob

2. **Course** table:

CourseID	CourseName	Instructor
101	Data Science	Dr. Smith
102	AI	Dr. White

3. **Enrollment** table:

StudentID	CourseID
1	101
2	102

This design ensures that changes to a course's instructor only need to be made in one place (the Course table), preventing update anomalies. Similarly, deleting a student does not result in the loss of information about the course or its instructor.

7.2 Explain Armstrong rule for inference.

Solution-
Armstrong's Axioms are a set of inference rules used in relational database theory to derive all the functional dependencies on a relational database. These rules are fundamental in the process of normalization, which involves decomposing database tables to eliminate redundancy and prevent update anomalies.

Armstrong's Axioms consist of three basic rules: **Reflexivity**, **Augmentation**, and **Transitivity**. These can be combined to infer additional functional dependencies from a given set.

Armstrong's Axioms

1. **Reflexivity (Trivial Functional Dependency)**
   - **Rule**: If Y⊆XY \subseteq XY⊆X, then X→YX \rightarrow YX→Y.
   - **Explanation**: Any set of attributes always functionally determines itself and any subset of its attributes.
   - **Example**: If you have a set of attributes X={A,B,C}X = \{A, B, C\}X={A,B,C}, then {A,B,C}→{A}\{A, B, C\} \rightarrow \{A\}{A,B,C}→{A} and {A,B,C}→{A,C}\{A, B, C\} \rightarrow \{A, C\}{A,B,C}→{A,C} are valid dependencies.
2. **Augmentation (Expansion)**
   - **Rule**: If X→YX \rightarrow YX→Y, then XZ→YZXZ \rightarrow YZXZ→YZ for any ZZZ.
   - **Explanation**: If XXX determines YYY, then adding extra attributes to both XXX and YYY does not change the validity of the functional dependency.
   - **Example**: If {A}→{B}\{A\} \rightarrow \{B\}{A}→{B}, then {A,C}→{B,C}\{A, C\} \rightarrow \{B, C\}{A,C}→{B,C} is also valid.
3. **Transitivity (Chain Rule)**
   - **Rule**: If X→YX \rightarrow YX→Y and Y→ZY \rightarrow ZY→Z, then X→ZX \rightarrow ZX→Z.
   - **Explanation**: If XXX determines YYY, and YYY determines ZZZ, then XXX must determine ZZZ.
   - **Example**: If {A}→{B}\{A\} \rightarrow \{B\}{A}→{B} and {B}→{C}\{B\} \rightarrow \{C\}{B}→{C}, then {A}→{C}\{A\} \rightarrow \{C\}{A}→{C}.

Derived Rules

In addition to the basic axioms, there are some additional rules that can be derived using Armstrong's Axioms:

1. **Union**:
   - **Rule**: If X→YX \rightarrow YX→Y and X→ZX \rightarrow ZX→Z, then X→YZX \rightarrow YZX→YZ.
   - **Explanation**: If XXX determines both YYY and ZZZ, then XXX determines the combination of YYY and ZZZ.
   - **Example**: If {A}→{B}\{A\} \rightarrow \{B\}{A}→{B} and {A}→{C}\{A\} \rightarrow \{C\}{A}→{C}, then {A}→{B,C}\{A\} \rightarrow \{B, C\}{A}→{B,C}.
2. **Decomposition (Projectivity)**:
   - **Rule**: If X→YZX \rightarrow YZX→YZ, then X→YX \rightarrow YX→Y and X→ZX \rightarrow ZX→Z.
   - **Explanation**: If XXX determines the combination of YYY and ZZZ, then XXX must individually determine both YYY and ZZZ.
   - **Example**: If {A}→{B,C}\{A\} \rightarrow \{B, C\}{A}→{B,C}, then {A}→{B}\{A\} \rightarrow \{B\}{A}→{B} and {A}→{C}\{A\} \rightarrow \{C\}{A}→{C}.
3. **Pseudotransitivity**:
   - **Rule**: If X→YX \rightarrow YX→Y and WY→ZW Y \rightarrow ZWY→Z, then XW→ZXW \rightarrow ZXW→Z.
   - **Explanation**: If XXX determines YYY, and the combination of WWW and YYY determines ZZZ, then the combination of XXX and WWW must determine ZZZ.
   - **Example**: If {A}→{B}\{A\} \rightarrow \{B\}{A}→{B} and {C,B}→{D}\{C, B\} \rightarrow \{D\}{C,B}→{D}, then {A,C}→{D}\{A, C\} \rightarrow \{D\}{A,C}→{D}.

Importance of Armstrong's Axioms

Armstrong's Axioms are crucial in the field of database design for the following reasons:

- **Inference**: They allow the derivation of all possible functional dependencies that can logically exist in a database schema, given a set of known dependencies.
- **Normalization**: They aid in the normalization process, where database tables are decomposed into smaller tables to remove redundancy and ensure that the database is in the highest possible normal form without losing information.
- **Dependency Preservation**: They help ensure that during normalization, all functional dependencies are preserved in the resulting tables.

7.3 Explain the need of normalization and explain 1NF, 2 NF & 3 NF.

Solution-

Normalization is a database design technique that organizes the data in a relational database to reduce redundancy and improve data integrity. The process of normalization involves dividing large tables into smaller, related tables and defining relationships between them. The primary goal of normalization is to eliminate redundancy, ensure data consistency, and make the database more efficient.

Why Normalization is Needed:

1. **Eliminate Redundancy**: Redundancy occurs when the same piece of data is stored in multiple places within a database. This can lead to data anomalies (insertion, update, and deletion anomalies) and increases the storage requirement. Normalization reduces this redundancy by ensuring that data is stored only once.
2. **Improve Data Integrity**: By minimizing redundancy, normalization helps maintain the accuracy and consistency of data within the database. If data exists in only one place, it is easier to enforce data integrity rules.
3. **Avoid Data Anomalies**: Without normalization, databases are prone to insertion, update, and deletion anomalies. Normalization helps prevent these anomalies by organizing the data into well-structured tables.
4. **Efficient Data Management**: Normalized databases are easier to maintain, update, and query because the data is organized logically. This leads to better performance and scalability.

Normal Forms

Normalization is typically carried out in stages, known as **normal forms** (NFs). Each normal form addresses specific issues, and achieving each level of normalization results in a more refined database design. Below are the explanations of the first three normal forms: 1NF, 2NF, and 3NF.

1. First Normal Form (1NF)

Definition: A table is in **1NF** if:

- All the values in a column are atomic (indivisible).
- Each column contains only one value (no repeating groups or arrays).
- Each row is unique and identified by a primary key.

Example: Consider the following unnormalized table:

StudentID	Name	Courses
1	Alice	Math, Science
2	Bob	Science, English

This table is not in 1NF because the Courses column contains multiple values. To convert it to 1NF, we split the data so that each value is atomic:

StudentID	Name	Course
1	Alice	Math
1	Alice	Science
2	Bob	Science
2	Bob	English

2. Second Normal Form (2NF)

Definition: A table is in **2NF** if:

- It is already in 1NF.
- All non-key attributes (attributes that are not part of the primary key) are fully functionally dependent on the entire primary key.

Explanation: 2NF eliminates partial dependencies, where an attribute depends on only part of a composite primary key rather than on the whole key.

Example: Consider the following table in 1NF:

StudentID	Course	Instructor
1	Math	Dr. Smith
1	Science	Dr. Brown
2	Science	Dr. Brown
2	English	Dr. White

In this example, StudentID and Course together form the composite primary key. However, Instructor depends only on Course and not on StudentID. To normalize to 2NF, we split the table:

Student_Course Table:

StudentID	Course
1	Math
1	Science
2	Science
2	English

Course_Instructor Table:

Course	Instructor
Math	Dr. Smith

Course	**Instructor**
Science	Dr. Brown
English	Dr. White

Now, each non-key attribute is fully dependent on the entire primary key.

3. Third Normal Form (3NF)

Definition: A table is in **3NF** if:

- It is already in 2NF.
- There are no transitive dependencies, meaning no non-key attribute depends on another non-key attribute.

Explanation: 3NF eliminates transitive dependencies, where one non-key attribute depends on another non-key attribute.

Example: Consider the following table in 2NF:

StudentID	**Course**	**Instructor**	**InstructorOffice**
1	Math	Dr. Smith	Room 101
1	Science	Dr. Brown	Room 102
2	Science	Dr. Brown	Room 102
2	English	Dr. White	Room 103

In this table, InstructorOffice is dependent on Instructor, which is a transitive dependency (since Instructor is a non-key attribute). To convert to 3NF, we split the table further:

Student_Course Table:

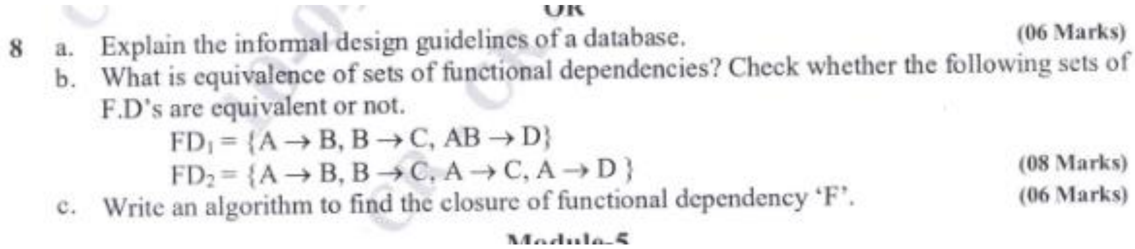
StudentID	**Course**	**Instructor**
1	Math	Dr. Smith
1	Science	Dr. Brown
2	Science	Dr. Brown
2	English	Dr. White

Instructor_Office Table:

Instructor	**InstructorOffice**
Dr. Smith	Room 101
Dr. Brown	Room 102

Instructor	InstructorOffice
Dr. White	Room 103

In this 3NF structure, InstructorOffice depends only on Instructor, not on any other non-key attribute.



OR

8 a. Explain the informal design guidelines of a database. (06 Marks)

b. What is equivalence of sets of functional dependencies? Check whether the following sets of F.D's are equivalent or not.

$FD_1 = \{A \rightarrow B, B \rightarrow C, AB \rightarrow D\}$

$FD_2 = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow D\}$ (08 Marks)

c. Write an algorithm to find the closure of functional dependency 'F'. (06 Marks)

Module-5

Solution-

Question 8: Database Design and Functional Dependencies

a. Explain the informal design guidelines of a database.

Informal design guidelines provide general principles and best practices for designing databases. They aim to create well-structured, efficient, and maintainable databases. Here are some key informal design guidelines:

1. **Normalization:**
   - **Eliminate redundant data:** Avoid storing the same data in multiple places.
   - **Maintain data integrity:** Ensure data consistency and accuracy.
   - **Improve query performance:** Optimized queries can be executed more efficiently.
   - **Common normalization forms:** First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF).
2. **Atomicity:**
   - **Divide attributes into atomic values:** Break down attributes into their smallest indivisible units.
   - **Prevent inconsistencies:** Ensure data is stored in a consistent and meaningful way.
3. **Consistency:**
   - **Maintain data integrity:** Ensure that data follows predefined rules and constraints.
   - **Prevent inconsistencies:** Avoid conflicting or contradictory information.
4. **Isolation:**
   - **Protect concurrent transactions:** Ensure that transactions are executed independently and do not interfere with each other.
   - **Maintain data integrity:** Prevent data inconsistencies during concurrent updates.
5. **Durability:**
   - **Ensure data persistence:** Once a transaction is committed, its changes should be permanent and recoverable even in case of system failures.

b. What is equivalence of sets of functional dependencies? Check whether the following sets of F.D's are equivalent or not.

Equivalence of sets of functional dependencies means that two sets of functional dependencies have the same closure. In other words, they imply the same set of dependencies.

To check equivalence, we can compare their closures.

FD1: {A → B, B → C, AB → D} **FD2:** {A → B, B → C, CA → A, A → C, A → D}

Closure of FD1:

- AB → D (given)
- A → B (given)
- B → C (given)
- A → C (transitivity from A → B and B → C)
- AC → A (reflexivity)
- CA → A (commutativity)

Closure of FD2:

- A → B (given)
- B → C (given)
- CA → A (given)
- A → C (given)
- A → D (given)
- AC → A (reflexivity)

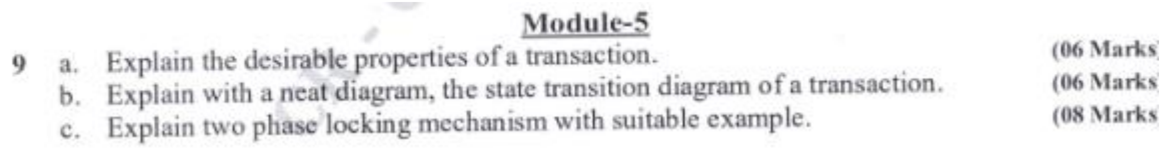
Comparing the closures, we can see that both sets of functional dependencies imply the same set of dependencies. Therefore, they are equivalent.

c. Write an algorithm to find the closure of functional dependency 'F'.

Algorithm to find the closure of functional dependency 'F':

1. **Initialize closure:** Set closure = F.
2. **Iterate until no new dependencies can be inferred:** a. For each pair of attributes (X, Y) in the closure: i. If X is a subset of the closure, add Y to the closure.
3. **Return the closure.**

9.1



Module-5

9 a. Explain the desirable properties of a transaction. (06 Marks

b. Explain with a neat diagram, the state transition diagram of a transaction. (06 Marks

c. Explain two phase locking mechanism with suitable example. (08 Marks

Solution-

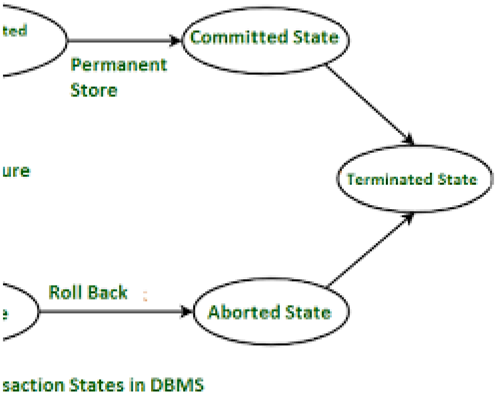
a. Explain the desirable properties of a transaction.

Transactions in a database system should possess the following desirable properties, often referred to as ACID properties:

1. **Atomicity:** A transaction is an indivisible unit of work. It must either be executed completely or not at all. If any part of the transaction fails, the entire transaction is rolled back to its original state.
2. **Consistency:** A transaction must maintain the database's consistency. It should leave the database in a valid state, ensuring that all data integrity constraints are satisfied.
3. **Isolation:** Transactions should operate independently of each other. The actions of one transaction should not be visible to other concurrent transactions until it is committed. This prevents conflicts and ensures data integrity.
4. **Durability:** Once a transaction is committed, its changes should be permanent and recoverable even in case of system failures. The effects of a committed transaction should survive system crashes and restarts.

b. Explain with a neat diagram, the state transition diagram of a transaction.

A transaction can be in one of several states during its execution. The state transition diagram illustrates the possible transitions between these states:



Opens in a new window www.geeksforgeeks.org

state transition diagram for a transaction

The diagram shows the following states and transitions:

- **Active:** The transaction is currently executing.
- **Partially Committed:** The transaction has completed its operations but is not yet fully committed.
- **Committed:** The transaction has successfully completed and its changes are made permanent.
- **Aborted:** The transaction has failed and been rolled back. Its changes are undone.

c. Explain two phase locking mechanism with suitable example.

Two-phase locking (2PL) is a concurrency control mechanism used to ensure transaction isolation and prevent conflicts. It involves two phases:

1. **Growing phase:** Transactions acquire locks on data items they need to access. Locks can be shared (read-only) or exclusive (write-only).
2. **Shrinking phase:** Transactions release locks on data items they no longer need.

Example:

Consider two transactions, T1 and T2.

T1:

- Reads data item A.
- Updates data item A.
- Writes data item A.

T2:

- Reads data item A.
- Updates data item A.
- Writes data item A.

Using 2PL:

1. **T1:** Acquires a shared lock on A.
2. **T2:** Acquires a shared lock on A.
3. **T1:** Updates A and acquires an exclusive lock on A.
4. **T2:** Tries to acquire an exclusive lock on A but is blocked because T1 already holds an exclusive lock.
5. **T1:** Commits and releases the exclusive lock on A.
6. **T2:** Acquires an exclusive lock on A and updates it.
7. **T2:** Commits and releases the exclusive lock on A.

10.1 Discus on database inconsistence problem.

Solution-
Database Inconsistency Problems

Database inconsistency occurs when data within a database is contradictory or inaccurate. This can lead to various issues, including incorrect results, data loss, and system failures. Here are some common types of database inconsistency problems:

1. Data Redundancy

- **Definition:** Storing the same data item in multiple places within the database.

- **Problems:**
    - **Update anomalies:** When data is updated in one location but not in others, leading to inconsistencies.
    - **Deletion anomalies:** Deleting a data item in one location might unintentionally delete related data in another.
    - **Insertion anomalies:** New data cannot be inserted without also inserting related data in other locations.

2. Data Anomalies

- **Update anomalies:** Difficulties in updating data due to inconsistencies.
- **Deletion anomalies:** Accidental deletion of data that is related to other data.
- **Insertion anomalies:** Inability to insert new data without violating data integrity constraints.

3. Lost Updates

- **Definition:** When two transactions update the same data item concurrently, and one transaction's changes are overwritten by the other.
- **Problem:** Data integrity is compromised, and the database may become inconsistent.

4. Dirty Read

- **Definition:** When a transaction reads data that has been modified by another transaction but not yet committed.
- **Problem:** The transaction may see an inconsistent state of the database.

5. Uncommitted Read

- **Definition:** When a transaction reads data that has been modified by another transaction but not yet committed.
- **Problem:** The transaction may see an inconsistent state of the database.

6. Phantom Read

- **Definition:** When a transaction reads a set of data items, then another transaction inserts new data items into the same set, and the first transaction reads the set again, seeing the new data items.
- **Problem:** This can lead to inconsistent results and make it difficult to reason about the data.

7. Lost Write

- **Definition:** When a transaction's updates are lost due to a system failure or other reasons.
- **Problem:** Data integrity is compromised, and the database may become inconsistent.

To address these problems, database systems employ various techniques, including:

- **Normalization:** Decomposing data into smaller, normalized tables to reduce redundancy and improve data integrity.

- **Concurrency control mechanisms:** Techniques like locking and timestamping to prevent conflicts between concurrent transactions.
- **Recovery mechanisms:** Techniques like logging and checkpoints to recover from system failures and ensure data durability.

10.2 Explain binary locks and shared locks with example.

Solution-

Binary Locks and Shared Locks

Binary locks and **shared locks** are concurrency control mechanisms used in database systems to manage access to data items and prevent conflicts between concurrent transactions.

Binary Locks

- **Definition:** A binary lock is a simple type of lock that can be in one of two states: locked or unlocked.
- **Usage:** When a transaction wants to access a data item, it must acquire a binary lock on that item. If the lock is already held by another transaction, the requesting transaction must wait until the lock is released.
- **Example:**
    - Transaction T1 wants to update a data item A.
    - T1 acquires a binary lock on A.
    - T2 also wants to update A.
    - T2 is blocked because T1 holds the lock on A.
    - T1 completes its update and releases the lock.
    - T2 can now acquire the lock on A and update it.

Shared Locks

- **Definition:** A shared lock allows multiple transactions to read a data item simultaneously.
- **Usage:** Shared locks are used for read-only operations.
- **Example:**
    - Transaction T1 wants to read data item A.
    - T1 acquires a shared lock on A.
    - Transaction T2 also wants to read data item A.
    - T2 can acquire a shared lock on A, allowing both transactions to read the data concurrently.
    - If T1 or T2 wants to update A, they must acquire an exclusive lock, which will block other transactions from accessing A.

In summary:

- **Binary locks** provide exclusive access to a data item, preventing other transactions from accessing it until the lock is released.
- **Shared locks** allow multiple transactions to read a data item concurrently but prevent modifications.

The choice between binary locks and shared locks depends on the specific requirements of the database system and the types of operations being performed. In some cases, a combination of both types of locks may be used to optimize concurrency and data integrity.