CBCS SCHEME

USN

BCS403

Fourth Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
Database Management System

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	C
Q.1	a.	Define the following terms: (i) Database (ii) Schema (iii) Entity (iv) DDL (v) Degree of a relationship	05	L1	CO1
	b.	Briefly explain characteristics of database approach.	05	L2	CO1
	c.	List and explain advantages of using DBMS approach.	10	L2	CO1
		OR			
Q.2	a.	Define the following terms: (i) Cardinality (ii) Weak entity (iii) Program data independence (iv) DML (v) Value sets	05	L1	CO1
	b.	Describe three-schema architecture. Why do we need mappings between schema levels?	05	L2	CO1
	c.	Explain different types of attributes in ER model with suitable example for each.	10	L2	CO1
		Module – 2			
Q.3	a.	With suitable example, explain the entity integrity and referential integrity constraints. Why each is considered important?	05	L2	CO2
	b.	Discuss equijoin and natural join with suitable example using relational algebra notation.	05	L2	CO2
	c.	Given the relational tables:	10	L3	CO2

Employee:

EID	Name	DepID	Salary
1	Alice	10	5000
2	Bob	20	6000
3	Eve	20	6500

Department:

DeptID	DeptName
10	HR
20	IT
30	Sales

Project

PID	Project Name	DeptID
101	Project Alpha	10
102	Project Beta	20
103	Project Gamma	30

Write relational algebra expression for the following:
(i) Find the names and salaries of all employees in the 'IT' department.
(ii) Find the ID's and names of employees who are in the 'IT' department and have a salary greater than 6000.
(iii) Find the ID's and names of employees who are either in the 'HR' department or have a salary greater than 6000.
(iv) Find the names of employees who are not in the 'IT' department
(v) Find the names of employees along with their department names.

OR

Q.4	**a.**	Explain any two operations that change the state of relation in a database. Provide suitable examples.	**05**	**L2**	**CO2**
	b.	Discuss the aggregation functions and grouping in relational algebra with suitable examples.	**05**	**L2**	**CO2**
	c.	Given the relational tables: Student: SID | Name a | Alice b | Bob c | Carol Project: PID | Project Name p | Alpha q | Beta r | Gamma Language: LID | Language Name x | Python y | Java z | C++ Enrollment: SID | PID a | p a | q b | q c | r Write relational algebra expression for the following: (i) Rename the student table to Learner and display it. (ii) Find the students (learners) who are not enrolled in any project. (iii) Find the students who are enrolled in all projects. (iv) Find the students who are not enrolled in any project. (v) Find the students who are enrolled in both the 'Alpha' and 'Beta' projects.	**10**	**L3**	**CO2**

Module – 3

Q.5	**a.**	Explain Armstrong inference rules.	**05**	**L2**	**CO4**
	b.	What is the need for normalization? Explain 1NF, 2NF and 3NF with examples.	**05**	**L2**	**CO4**
	c.	What is functional dependency? Write an algorithm to find minimal cover for set of functional dependencies. Construct minimal cover M for set of functional dependencies which are: E = {B → A, D → A, AB → D}	**10**	**L3**	**CO4**

OR

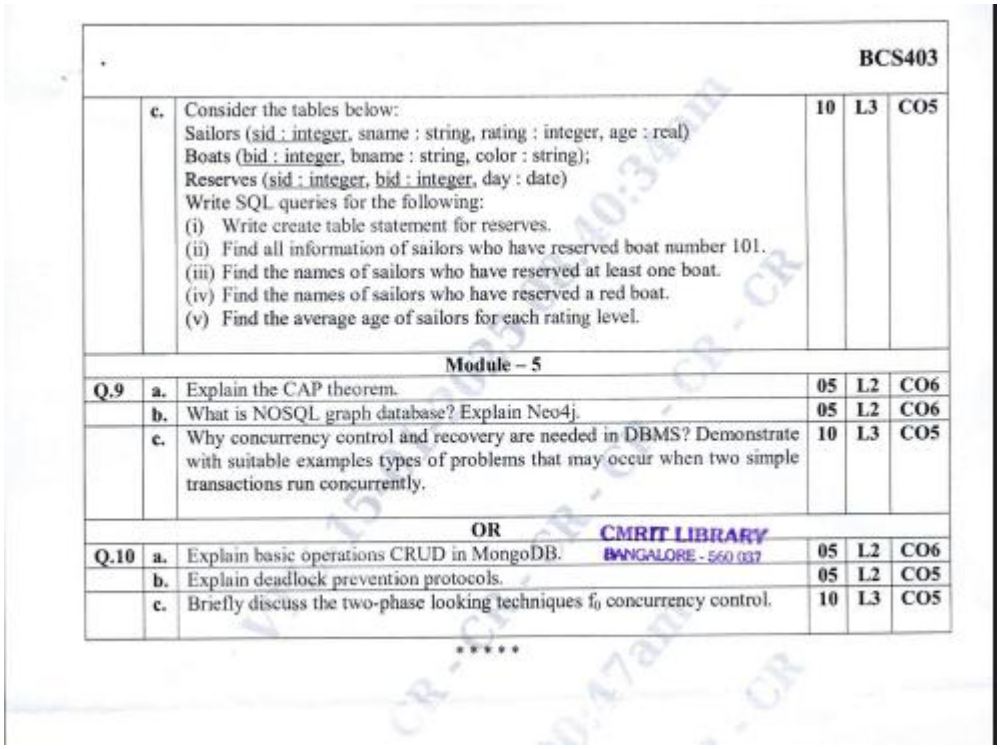
Q.6	**a.**	Explain the types of update anomalies in SQL with an example.	**05**	**L2**	**CO4**
	b.	Explain types of JBBC drivers.	**05**	**L2**	**CO5**
	c.	Consider the schema R = ABCD, subjected to FDs F = {A →B, B →C}, and the non-binary partition D1 = {ACD, AB, BC}. State whether D1 is a lossless decomposition? [give all steps in detail].	**10**	**L3**	**CO4**

Module – 4

Q.7	**a.**	Define transaction. Discuss ACID properties.	**05**	**L2**	**CO5**
	b.	With a neat diagram, explain transition diagram of a transaction.	**05**	**L2**	**CO5**
	c.	Demonstrate working of assertion and triggers in SQL with example.	**10**	**L3**	**CO5**

OR

Q.8	**a.**	Explain cursor and its properties in embedded SQL with suitable example.	**05**	**L2**	**CO5**
	b.	Determine if the following schedule is serializable and explain your reasoning: i) T1 : R(X)W(X) T2 : R(X)W(X) T1 : COMMIT T2 : COMMIT ii) T1 : W(X)R(Y) T2 : R(X)W(Y) T1 : COMMIT T2 : COMMIT	**05**	**L2**	**CO5**



		BCS403			
	c.	Consider the tables below: Sailors (sid : integer, sname : string, rating : integer, age : real) Boats (bid : integer, bname : string, color : string); Reserves (sid : integer, bid : integer, day : date) Write SQL queries for the following: (i) Write create table statement for reserves. (ii) Find all information of sailors who have reserved boat number 101. (iii) Find the names of sailors who have reserved at least one boat. (iv) Find the names of sailors who have reserved a red boat. (v) Find the average age of sailors for each rating level.	10	L3	CO5
		Module – 5			
Q.9	a.	Explain the CAP theorem.	05	L2	CO6
	b.	What is NOSQL graph database? Explain Neo4j.	05	L2	CO6
	c.	Why concurrency control and recovery are needed in DBMS? Demonstrate with suitable examples types of problems that may occur when two simple transactions run concurrently.	10	L3	CO5
		OR			
Q.10	a.	Explain basic operations CRUD in MongoDB.	05	L2	CO6
	b.	Explain deadlock prevention protocols.	05	L2	CO5
	c.	Briefly discuss the two-phase looking techniques for concurrency control.	10	L3	CO5

*****

Q.1	a.	Define the following terms: (i) Database (ii) Schema (iii) Entity (iv) DDL (v) Degree of a relationship	05	L1	CO1
	b.	Briefly explain characteristics of database approach.	05	L2	CO1
	c.	List and explain advantages of using DBMS approach.	10	L2	CO1

Q.1 a. Define the following terms: (i) Database (ii) Schema (iv) DDL (v) Degee of a relationship

(Each Definition 1 Mark)

1. Database

A **database** is an organized collection of related data that is stored electronically and managed so that it can be easily accessed, updated, and retrieved. It provides a convenient and efficient way of storing and managing information.

Example: Student database containing student details, course details, and marks.

2. Schema

A **schema** is the overall logical design or blueprint of a database that specifies how data is organized, including tables, attributes, relationships, and constraints. It describes the structure of the database but does not contain the actual data.

Example: STUDENT(RollNo, Name, Branch) and COURSE(CourseID, CourseName).

3. Entity

An **entity** is a distinguishable real-world object, person, place, or concept about which data is stored in a database. Each entity is represented by a set of attributes.

Example: Student, Employee, Department, and Book are entities.

4. DDL (Data Definition Language)

Data Definition Language (DDL) is a category of SQL commands used to define, create, modify, and remove database objects such as tables, views, and indexes.

Common DDL commands:

- **CREATE** – Creates database objects.
- **ALTER** – Modifies the structure of existing objects.
- **DROP** – Deletes database objects.
- **TRUNCATE** – Removes all records from a table.

5. Degree of Relationship

The **degree of a relationship** refers to the number of entity sets participating in a relationship type.

- **Unary Relationship:** Involves one entity set.
    - Example: Employee supervises Employee.
- **Binary Relationship:** Involves two entity sets.
    - Example: Student enrolls in Course.
- **Ternary Relationship:** Involves three entity sets.
    - Example: Supplier supplies Part to Project.

- **N-ary Relationship:** Involves more than three entity sets.

Q1. b. Briefly explain characteristics of database approach (5M)

The **database approach** is a method of storing and managing data using a Database Management System (DBMS). Unlike the traditional file processing system, the database approach provides centralized control over data and offers several important characteristics that improve efficiency, consistency, and security.

1. Self-Describing Nature of a Database System

A database system contains not only the actual data but also information about the structure and constraints of the data. This information, called **metadata**, is stored in the **system catalog or data dictionary**.

The metadata describes:

- Names of relations and attributes.
- Data types of attributes.
- Constraints and relationships among data.

Thus, the database is self-describing in nature.

2. Program–Data Independence

In the database approach, application programs are independent of the physical storage details of data. Changes made in the database structure or storage organization do not require modifications to the application programs.

This characteristic provides:

- **Logical data independence**
- **Physical data independence**

Hence, changes in one level do not affect the other levels.

3. Support for Multiple Views of Data

Different users have different requirements. A DBMS allows multiple users to view the same database in different ways according to their needs.

For example:

- A student may view marks and attendance details.
- A faculty member may view student and course information.
- The administrator may access the entire database.

Thus, multiple views provide simplicity and security.

4. Data Sharing and Multiuser Transaction Processing

A database can be shared simultaneously by many users and applications. The DBMS provides mechanisms for controlling concurrent access to maintain consistency.

It supports:

- Concurrent transactions.
- Data sharing among multiple users.
- Recovery from failures.
- Maintenance of database consistency.

Hence, several users can access and update the database without causing conflicts.

5. Controlled Redundancy

The database approach minimizes unnecessary duplication of data. Since the same data is shared by multiple users, redundancy is reduced considerably.

Advantages include:

- Reduced storage space.
- Elimination of inconsistencies.
- Improved data integrity.

6. Data Integrity and Consistency

The DBMS enforces integrity constraints to ensure that the stored data remains accurate and consistent.

Examples include:

- Entity integrity.
- Referential integrity.
- Domain constraints.

Thus, the correctness of data is maintained.

7. Data Security

A database system provides mechanisms to protect data from unauthorized access.

Security features include:

- User authentication.
- Authorization and access privileges.
- Password protection.
- Encryption techniques.

Therefore, only authorized users can access sensitive information.

8. Backup and Recovery

DBMS provides facilities for backing up the database and recovering it in case of hardware failures, software failures, or power outages.

Recovery mechanisms help restore the database to a consistent state and prevent loss of data.

9. Transaction Processing and Concurrency Control

A DBMS supports transaction management based on the ACID properties. It allows multiple transactions to execute concurrently while maintaining consistency and isolation.

This feature ensures:

- Reliable execution of transactions.
- Prevention of anomalies.
- Correctness of concurrent operations.

10. Enforcement of Standards

The database approach allows standards for naming conventions, storage formats, and data representation to be enforced uniformly throughout the organization.

This promotes:

- Better data administration.

- Easier maintenance.
- Improved interoperability.

Q1. C List and explain advantages of using DBMS approach (10M)

A database management system (DBMS) is a collection of programs enabling users to create and maintain a database. More specifically, a DBMS is a general purpose software system facilitating each of the following (with respect to a database):

● definition: specifying data types (and other constraints to which the data must conform) and data organization

● construction: the process of storing the data on some medium (e.g., magnetic disk) that is controlled by the DBMS

● manipulation: querying, updating, report generation

● sharing: allowing multiple users and programs to access the database "simultaneously"

● system protection: preventing database from becoming corrupted when hardware or software failures occur

● security protection: preventing unauthorized or malicious access to database.

ADVANTAGES OF USING DBMS:

i. Controlling Redundancy

Redundancy refers to the unnecessary duplication of the same data at multiple locations in a database. The database approach aims to **control and minimize redundancy** by integrating related data into a single database and allowing different users and applications to share the same data.

In traditional file processing systems, the same information may be stored repeatedly in different files. Such duplication leads to several problems and inconsistencies. Therefore, controlling redundancy is one of the important characteristics of the database approach.

ii. Restricting Unauthorized Access

• A DBMS should provide a security and authorization subsystem • most users will not be authorized to access all information in the database

iii. Providing Persistent Storage for Program Objects

• Databases can be used to provide persistent storage for program objects and

data structures

iv. Providing Storage Structures and Search Techniques for Efficient Query Processing

• the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records – index

• The DBMS often has a buffering or caching module that maintains parts of the database in main memory buffers

• The query processing and optimization module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures.

v. Providing Backup and Recovery

responsible for recovery the database is restored to the state it was in

before the transaction started executing

vi. Providing Multiple User Interfaces

• a DBMS should provide a variety of user interfaces o query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for standalone users

vii. Representing Complex Relationships among Data

• A DBMS must have the capability to represent a variety of complex relationships among the data, to define new relationships as they arise, and to retrieve and update related data easily and efficiently

viii. Enforcing Integrity Constraints

• The simplest type of integrity constraint involves specifying a data type for each data item

• referential integrity constraint: specifying that a record in one file

must be related to records in other files

• Primary key constraint: uniqueness on data item values.

ix. Permitting inferencing and Actions Using Rules: A trigger is a

form of a rule activated by updates to the table, which results in

performing some additional operations to some other tables,

sending messages, and so on.

x. Additional Implications of Using the Database Approach

Potential for Enforcing Standards: The DBA can enforce standards in a centralized database environment o Reduced Application Development Time: Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system o Flexibility: Modern DBMSs allow certain types of evolutionary changes to the structure of the database without affecting the stored data and the existing application programs o Availability of Up-to-Date Information: As soon as one user's update is applied to the database, all other users can immediately see this update o Economies of Scale: reduce the wasteful overlap activities thereby reducing the overall costs of operation and management.

Q.2	a.	Define the following terms: (i) Cardinality (ii) Weak entity (iii) Program data independence (iv) DML (v) Value sets
	b.	Describe three-schema architecture. Why do we need mappings between schema levels?
	c.	Explain different types of attributes in ER model with suitable example for each.

Q2 a. Define the following terms:

(i) Cardinality (ii) Weak entity (iii) Program data independence (iv) DML (v) Value sets

(5M each definition)

1. Cardinality

Cardinality specifies the number of entities in one entity set that can be associated with entities in another entity set through a relationship.

Types of Cardinality:

- **One-to-One (1:1):** One entity is related to only one entity of another set.
- **One-to-Many (1:N):** One entity is related to many entities of another set.
- **Many-to-One (N:1):** Many entities are related to one entity of another set.
- **Many-to-Many (M:N):** Many entities are related to many entities of another set.

2. Weak Entity

A **weak entity** is an entity type that does not have sufficient attributes to form a primary key and whose existence depends on a strong entity. It is identified by combining its partial key with the primary key of the owner entity.

Example: Dependent entity related to Employee entity.

3. Program Data Independence

Program data independence is the ability to modify the database schema at one level without affecting the application programs or the schema at the next higher level. It separates data from application programs and improves flexibility.

Types:

- **Physical Data Independence**
- **Logical Data Independence**

4. DML (Data Manipulation Language)

Data Manipulation Language (DML) is a set of SQL commands used to retrieve, insert, update, and delete data stored in database tables.

Common DML Commands:

- **SELECT** – Retrieves data.
- **INSERT** – Adds new records.
- **UPDATE** – Modifies existing records.
- **DELETE** – Removes records.

5. Value Sets

A **value set** (or domain) is the set of all possible values that an attribute can take. It specifies the data type and permissible values for an attribute.

Example:

- Age: Integer values from 0 to 120.
- Gender: {Male, Female, Other}.

Q. 2.b. Describe three-schema architecture. Why do we need mappings between schema levels

5M: Definition (1M), Each schema description (3 marks - 1 mark each), mappings between schema level(1M)

Three-schema architecture is a framework that helps achieve database system properties of data independence and data abstraction using three levels of data description.

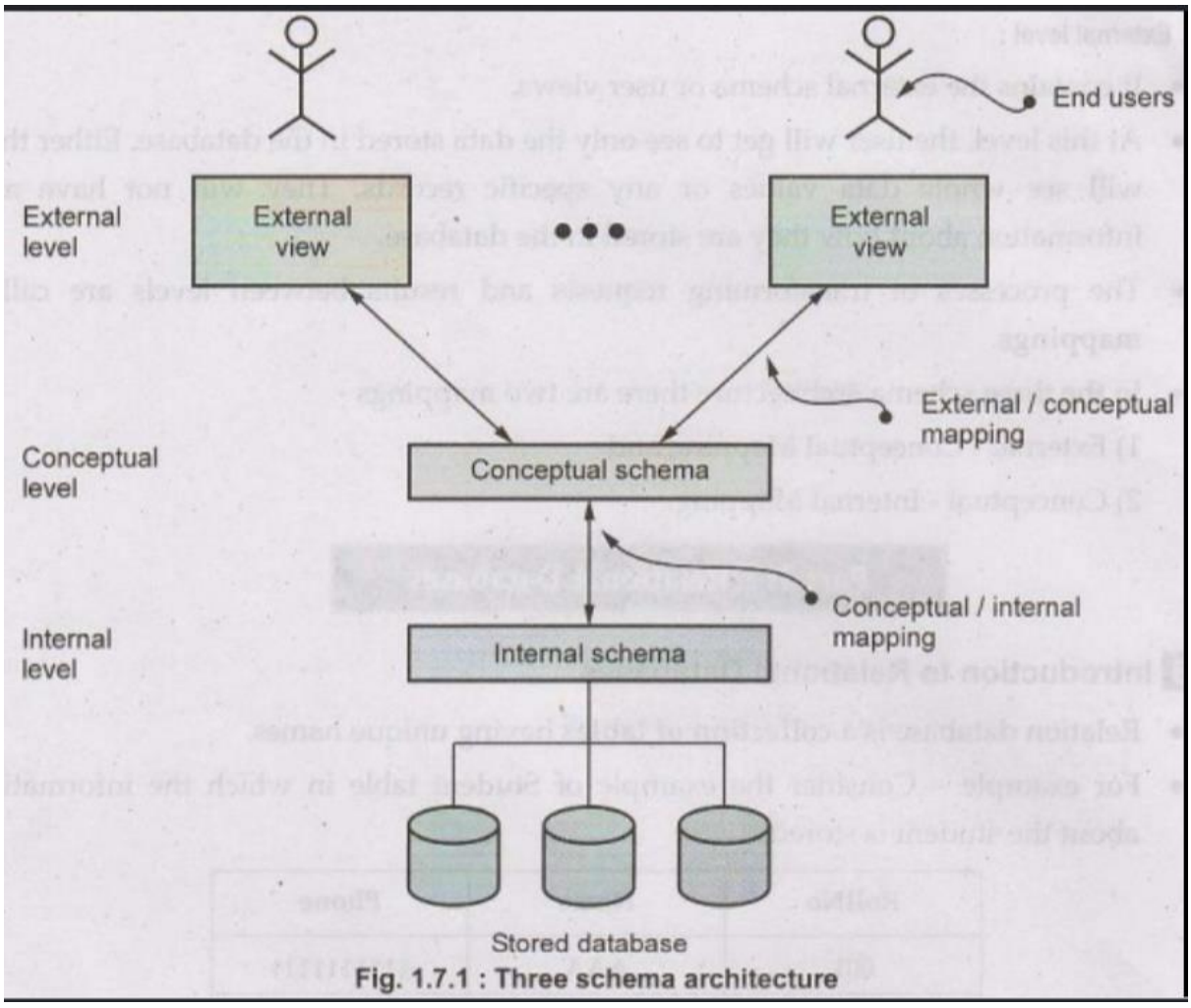


Fig. 1.7.1 : Three schema architecture

1. The internal level has an internal schema, -describes the physical storage structure of the database. -uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The conceptual level has a conceptual schema, -describes the structure of the whole database for a community of users -hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints -representational data model is used to describe the conceptual schema when a database system is implemented -This implementation conceptual schema is often based on a conceptual schema design in a high-level data model.

3. The external or view level includes a number of external schemas or user views, -describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. -As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level data model.

The DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are calledmappings. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

Mappings between schema levels are required to establish correspondence between the **external, conceptual, and internal schemas** of a database. They enable **data independence**, allowing changes made at one schema level to be isolated from the other levels. Thus, application programs and user views remain unaffected when the logical structure or physical storage of the database is modified.

Q2. C Explain different types of attributes in ER model with suitable example for each

(10M)

An **attribute** is a property or characteristic used to describe an entity or a relationship in an Entity-Relationship (ER) model. Attributes provide information about entities and help in representing real-world objects accurately. Based on their characteristics and behavior, attributes are classified into the following types.

1. Simple (Atomic) Attribute

A **simple attribute** is an attribute that cannot be divided further into smaller meaningful components. It represents a single indivisible piece of information about an entity. Since it contains only one value and has no internal structure, it is considered the basic unit of data in the ER model. Simple attributes simplify database design and are directly stored in the database.

Example:

Consider an entity **STUDENT** having attributes:

- Student_ID
- Age
- Gender

Here, **Age** and **Gender** are simple attributes because they cannot be subdivided further.

2. Composite Attribute

A **composite attribute** is an attribute that can be decomposed into smaller sub-attributes, each having its own independent meaning. The individual components collectively represent the complete attribute. They improve data organization and facilitate efficient data retrieval and manipulation.

Example:

In an EMPLOYEE entity, the attribute Name can be divided into:

First_Name

Middle_Name

Last_Name

Similarly, Address can be divided into:

House_No

Street

City

State

PIN_Code

Example Entity:

EMPLOYEE

Emp_ID

Name = {First_Name, Middle_Name, Last_Name}

Address = {House_No, Street, City, State, PIN_Code}

Composite attributes help represent complex information in a structured manner.

3. Single-Valued Attribute

A **single-valued attribute** is an attribute that contains only one value for each occurrence of an entity. Every entity instance can possess exactly one value for such an attribute. These attributes are commonly used when the property being represented is unique and unambiguous for every entity. They contribute to maintaining consistency and eliminating redundancy in the database.

Example:
Each student has only one:

- Date_of_Birth
- Blood_Group
- Gender

Therefore, these are single-valued attributes.

For example, a student can have only one date of birth.

4. Multi-Valued Attribute

A **multi-valued attribute** is an attribute that can have more than one value associated with a single entity occurrence. Such attributes are required when an entity needs to store multiple values for the same property. Multi-valued attributes increase the complexity of database design and are generally represented separately during the normalization process to avoid redundancy and maintain data integrity.

Example:
An employee may have multiple:

- Phone numbers
- Email addresses
- Skills

For instance, an employee may possess the skills:

- Java
- Python
- SQL

In ER diagrams, multi-valued attributes are represented using **double ellipses**.

5. Stored Attribute

A **stored attribute** is an attribute whose value is physically stored in the database. These attributes contain original data entered by users or obtained from external sources. Stored attributes serve as the basis for deriving other attributes and occupy physical storage space in the database. Their values are maintained explicitly and are readily available whenever required.

Example:

Date_of_Birth

The value of **Date_of_Birth** is stored directly in the database.

Stored attributes require memory space and are maintained permanently.

6. Derived Attribute

A **derived attribute** is an attribute whose value can be computed or obtained from the values of one or more existing attributes. These attributes are not necessarily stored physically in the database, thereby reducing redundancy and saving storage space. Derived attributes provide flexibility and ensure that the computed values remain consistent with the underlying data.

Example:

Age can be calculated from Date_of_Birth.

Age = Current Date – Date_of_Birth

- Date_of_Birth → Stored attribute
- Age → Derived attribute

Derived attributes reduce redundancy because they need not be stored separately. In ER diagrams, they are represented using **dashed ellipses**.

7. Key Attribute

A **key attribute** is an attribute whose values uniquely identify each entity in an entity set. No two entities can possess the same value for a key attribute. Key attributes play a vital role in maintaining entity integrity and establishing relationships among different entities. They are essential for efficient searching, indexing, and retrieval of records within the database.

Example:

For the entity **STUDENT**:
Student_ID

Name
Branch
Semester

Here, **Student_ID** uniquely identifies each student.

Similarly:

- Employee_ID identifies an employee.
- Account_Number identifies a bank account.

8. Null Attribute

A **null attribute** is an attribute whose value may be unknown, unavailable, missing, or not applicable for certain entity occurrences. A null value does not indicate zero or an empty string; rather, it signifies the absence of a valid value. Null attributes provide flexibility in representing incomplete information and accommodate situations where certain data may not exist at a particular time.

A **null attribute** is an attribute whose value may be unknown, unavailable, or not applicable for certain entities.

Example:

Emp_ID

Name

Spouse_Name

- Middle_Name may be absent.
- Passport_Number may not be available for all persons.

A null value does not mean zero; it indicates the absence of a value.

Module – 2

Q.3	a.	With suitable example, explain the entity integrity and referential integrity constraints. Why each is considered important?	05	L2	CO2
	b.	Discuss equijoin and natural join with suitable example using relational algebra notation.	05	L2	CO2
	c.	Given the relational tables: Employee: EID | Name | DepID | Salary 1 | Alice | 10 | 5000 2 | Bob | 20 | 6000 3 | Eve | 20 | 6500 Department: DeptID | DeptName 10 | HR 20 | IT 30 | Sales Project PID | Project Name | DeptID 101 | Project Alpha | 10 102 | Project Beta | 20 103 | Project Gamma | 30 Write relational algebra expression for the following: (i) Find the names and salaries of all employees in the 'IT' department. (ii) Find the ID's and names of employees who are in the 'IT' department and have a salary greater than 6000. (iii) Find the ID's and names of employees who are either in the 'HR' department or have a salary greater than 6000. (iv) Find the names of employees who are not in the 'IT' department (v) Find the names of employees along with their department names.	10	L3	CO2

Q3. a. With suitable example, explain the entity integrity and referential integrity constraints. Why each is considered important?

(5M)

Definition

The **Entity Integrity Constraint** states that **no primary key attribute of a relation can have a NULL value**. Since the primary key is used to uniquely identify each tuple in a relation, every tuple must possess a valid and unique primary key value. Therefore, attributes forming the primary key are not permitted to contain null values.

Entity integrity ensures that each entity represented in a relation is distinguishable from every other entity. Without a valid primary key value, it would be impossible to uniquely identify a particular record, leading to ambiguity and inconsistency in the database.

In a relation, the primary key serves as the identity of each tuple. If the value of the primary key is unknown or missing, the corresponding tuple cannot be uniquely identified, thereby violating the basic principle of the relational model.

Example

Consider the relation **STUDENT (Student_ID, Name, Branch)** where **Student_ID** is the primary key.

Student_ID	**Name**	**Branch**
101	Ravi	CSE
102	Priya	ISE
103	Arjun	ECE

All tuples have valid primary key values and therefore satisfy the entity integrity constraint.Suppose the following tuple is inserted:

Student_ID	**Name**	**Branch**
NULL	Kiran	AIML

Since the primary key value is NULL, the tuple cannot be uniquely identified. Hence, the entity integrity constraint is violated.

Importance of Entity Integrity Constraint

Entity integrity constraint is considered important because:

- It ensures that every tuple in a relation is uniquely identifiable.
- It eliminates ambiguity in identifying records.
- It prevents the occurrence of duplicate and inconsistent data.
- It preserves the correctness and reliability of information stored in the database.
- It forms the basis for establishing relationships among relations.
- It facilitates efficient searching, updating, and deletion of records.
- It maintains the integrity and uniqueness of entities represented in the database.

Thus, entity integrity constraint guarantees that every entity in a relation has a unique identity and can always be distinguished from other entities.

2. Referential Integrity Constraint

The **Referential Integrity Constraint** specifies that a value appearing in a foreign key attribute of one relation must either match the value of a primary key in another relation or be NULL. This constraint ensures that relationships established among different relations remain valid and meaningful.

A foreign key is used to establish associations between tuples belonging to different relations. Referential integrity guarantees that a tuple in one relation always refers to an existing tuple in another relation. Consequently, it prevents the existence of invalid references and maintains consistency among related tables.

Without referential integrity, a relation could contain references to tuples that do not exist, resulting in inconsistent and meaningless data. Therefore, referential integrity is essential for preserving the logical relationships among entities.

Example

Consider the following relations:

Dept_ID	Dept_Name
D1	CSE
D2	ISE
D3	ECE

where **Dept_ID** is the primary key.

STUDENT

Student_ID	Name	Dept_ID
101	Ravi	D1
102	Priya	D2
103	Arjun	D3

In the STUDENT relation, **Dept_ID** is a foreign key that refers to the primary key **Dept_ID** of the DEPARTMENT relation.

Suppose the following tuple is inserted:

Student_ID	Name	Dept_ID
104	Kiran	D5

Since there is no department with **Dept_ID = D5** in the DEPARTMENT relation, the foreign key value does not correspond to any existing primary key value. Therefore, the referential integrity constraint is violated.

Importance of Referential Integrity Constraint

Referential integrity constraint is considered important because:

- It maintains consistency among related relations.
- It ensures that every foreign key value corresponds to an existing primary key value.
- It prevents the occurrence of invalid references.
- It preserves the correctness of relationships among entities.
- It avoids anomalies that may arise during insertion, deletion, and update operations.
- It guarantees that the database represents real-world relationships accurately.
- It enhances the reliability and integrity of the overall database system.
- It ensures that related information remains synchronized across multiple relations.

Thus, referential integrity constraint preserves the logical association among relations and ensures that no tuple refers to a non-existent entity.

Q3 b. Discuss equijoin and natural join with suitable example using relational algebra notation (5M)

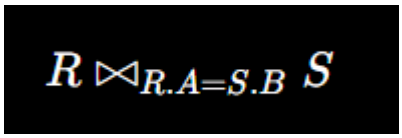
Join operations are used to combine tuples from two or more relations based on a specified condition. They are fundamental operations in relational algebra and are extensively used to retrieve related information from multiple relations. Among the various types of joins, **Equijoin** and **Natural Join** are commonly used.

1. Equijoin

An **Equijoin** is a join operation in which the join condition involves only the equality (=) operator. It combines tuples from two relations whenever the values of specified attributes are equal. In an equijoin, both join attributes are retained in the resulting relation, which may lead to duplicate columns.

Thus, an equijoin is a special case of the theta join in which the comparison operator used is equality.

Mathematically, Equijoin is represented as:



$$R \bowtie_{R.A=S.B} S$$

where R.A and S.B are the attributes on which the equality condition is applied.

Example

Consider the following relations:

EMPLOYEE

Emp_ID	Name	Dept_No
E1	Ravi	D1
E2	Priya	D2
E3	Arjun	D1

DEPARTMENT

Dept_No	Dept_Name
D1	CSE
D2	ISE
D3	ECE

To obtain employee details along with department information, an equijoin is performed on the attribute **Dept_No**.

Relational Algebra Notation

$$EMPLOYEE \bowtie_{EMPLOYEE.Dept\_No=DEPARTMENT.Dept\_No} DEPARTMENT$$

Result

Emp_ID	Name	Dept_No	Dept_No	Dept_Name
E1	Ravi	D1	D1	CSE
E2	Priya	D2	D2	ISE
E3	Arjun	D1	D1	CSE

It can be observed that both copies of **Dept_No** are present in the resultant relation.

Characteristics of Equijoin

- Uses only the equality operator.
- It is a special case of theta join.
- Join attributes from both relations are retained.
- Duplicate columns may appear in the result.
- It is useful when both join attributes are required in the output relation.

2. Natural Join

A **Natural Join** is a special type of equijoin in which the join is performed automatically on all attributes having the same name and compatible domains in the participating relations.

Unlike equijoin, duplicate attributes are eliminated from the resultant relation.

Therefore, a natural join combines related tuples and removes redundant columns, producing a more meaningful relation.

Mathematically, Natural Join is represented as:

No explicit join condition is required because the common attributes are identified automatically.

Example

Consider the same relations:

EMPLOYEE

Emp_ID	Name	Dept_No
E1	Ravi	D1
E2	Priya	D2
E3	Arjun	D1

DEPARTMENT

Dept_No	Dept_Name
D1	CSE
D2	ISE
D3	ECE

Relational Algebra Notation

$$EMPLOYEE \bowtie DEPARTMENT$$

Result

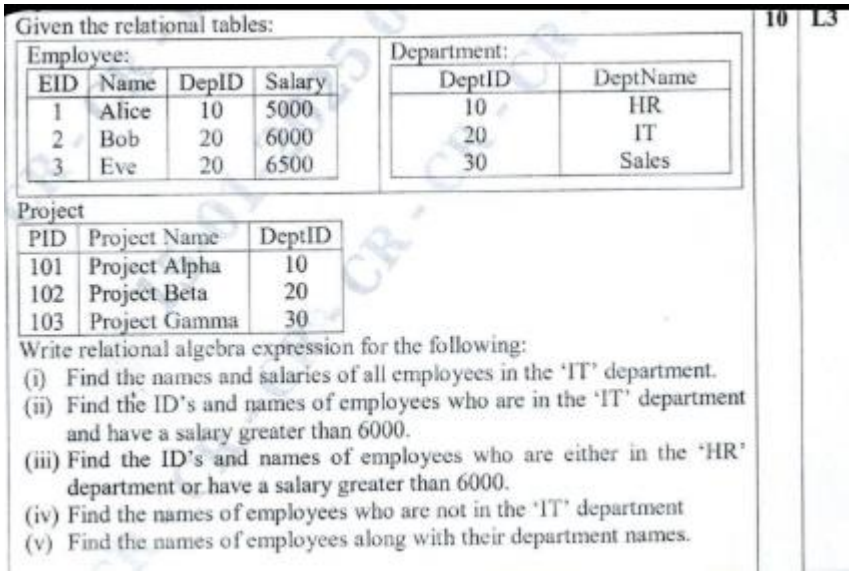
Emp_ID	Name	Dept_No	Dept_Name
E1	Ravi	D1	CSE
E2	Priya	D2	ISE
E3	Arjun	D1	CSE

Here, only one copy of **Dept_No** is retained, and duplicate columns are removed automatically.

Characteristics of Natural Join

- It is a special case of equijoin.
- Common attributes having the same name are used automatically.
- Duplicate columns are eliminated from the result.
- No explicit join condition is required.
- Produces a simpler and more meaningful relation.

C.



Given the relational tables: 10 L3

Employee:

EID	Name	DepID	Salary
1	Alice	10	5000
2	Bob	20	6000
3	Eve	20	6500

Department:

DeptID	DeptName
10	HR
20	IT
30	Sales

Project

PID	Project Name	DeptID
101	Project Alpha	10
102	Project Beta	20
103	Project Gamma	30

Write relational algebra expression for the following:

(i) Find the names and salaries of all employees in the 'IT' department.

(ii) Find the ID's and names of employees who are in the 'IT' department and have a salary greater than 6000.

(iii) Find the ID's and names of employees who are either in the 'HR' department or have a salary greater than 6000.

(iv) Find the names of employees who are not in the 'IT' department

(v) Find the names of employees along with their department names.

(i) Find the names and salaries of all employees in the 'IT' department.

First, join **Employee** and **Department** on DepID = DeptID, select tuples with DeptName = 'IT', and then project the required attributes.

Relational Algebra Expression

$$\pi_{Name,\ Salary}\left(\sigma_{DeptName='IT'}\left(Employee \bowtie_{Employee.DepID=Department.DeptID} Department\right)\right)$$

Result

Name	Salary
Bob	6000
Eve	6500

(ii) Find the ID's and names of employees who are in the 'IT' department and have a salary greater than 6000.

Join Employee and Department, select tuples satisfying both conditions, and project EID and Name.

Relational Algebra Expression

$$\pi_{EID,\,Name}\left(\sigma_{DeptName='IT' \,\wedge\, Salary>6000}(Employee \bowtie_{Employee.DepID=Department.DeptID} Department)\right)$$

Result

EID	Name
3	Eve

(iii) Find the ID's and names of employees who are either in the 'HR' department or have a salary greater than 6000.

Relational Algebra Expression

$$\pi_{EID,\,Name}\left(\sigma_{DeptName='HR' \,\vee\, Salary>6000}(Employee \bowtie_{Employee.DepID=Department.DeptID} Department)\right)$$

Result

EID	Name
1	Alice
3	Eve

(iv) Find the names of employees who are not in the 'IT' department.

Relational Algebra Expression

$$\pi_{Name}\left(\sigma_{DeptName \neq 'IT'}(Employee \bowtie_{Employee.DepID=Department.DeptID} Department)\right)$$

Result

Name
Alice

(v) Find the names of employees along with their department names.

Join Employee and Department and project the required attributes.

Relational Algebra Expression

$$\pi_{Name,\ DeptName}(Employee \bowtie_{Employee.DepID=Department.DeptID} Department)$$

Result

Name	DeptName
Alice	HR
Bob	IT
Eve	IT

Q.4	a.	Explain any two operations that change the state of relation in a database. Provide suitable examples.	05	L2	CO2
	b.	Discuss the aggregation functions and grouping in relational algebra with suitable examples.	05	L2	CO2
	c.	Given the relational tables: Student: SID | Name a | Alice b | Bob c | Carol Project: PID | Project Name p | Alpha q | Beta r | Gamma Language: LID | Language Name x | Python y | Java z | C++ Enrollment: SID | PID a | p a | q b | q c | r Write relational algebra expression for the following: (i) Rename the student table to Learner and display it. (ii) Find the students (learners) who are not enrolled in any project. (iii) Find the students who are enrolled in all projects. (iv) Find the students who are not enrolled in any project. (v) Find the students who are enrolled in both the 'Alpha' and 'Beta' projects.	10	L3	CO2

Q4. a. Explain any two operations that change the state of relation in a database. provide suitable examples (5M)

The operations that modify the contents of a relation are called **update operations**. These operations change the state of the database by adding, deleting, or modifying tuples. The three basic update operations are **Insertion**, **Deletion**, and **Modification (Updation)**. Any two are explained below.

1. Insertion operation

is used to add a new tuple (record) into a relation. When a new entity is introduced into the database, a corresponding tuple is inserted into the appropriate relation. The insertion operation changes the state of the relation by increasing the number of tuples.

While inserting a tuple, it is necessary to ensure that all integrity constraints such as domain constraints, entity integrity constraints, and referential integrity constraints are satisfied. If any constraint is violated, the insertion operation is rejected.

Example

Consider the relation **STUDENT(Student_ID, Name, Branch)**

Student_ID	Name	Branch
101	Ravi	CSE
102	Priya	ISE

Suppose a new student is admitted. The following tuple is inserted:

(103, Arjun, ECE)

After insertion, the relation becomes:

Student_ID	Name	Branch
101	Ravi	CSE
102	Priya	ISE
103	Arjun	ECE

Thus, the insertion operation changes the state of the relation by adding a new tuple.

Importance

- Adds new information to the database.
- Increases the number of records in a relation.
- Helps maintain up-to-date data.
- Must satisfy all integrity constraints.

2. Deletion Operation

The **Deletion operation** is used to remove one or more tuples from a relation. It changes the state of the database by reducing the number of tuples present in the relation. Deletion may be performed on a specific tuple or on a set of tuples satisfying a particular condition.

Before deleting tuples, care must be taken to ensure that referential integrity constraints are not violated. Deleting a tuple that is referenced by another relation may lead to dangling references and inconsistency in the database.

Example

Consider the relation **STUDENT(Student_ID, Name, Branch)**

Student_ID	Name	Branch
101	Ravi	CSE
102	Priya	ISE
103	Arjun	ECE

Suppose the record corresponding to student 102 is deleted.

After deletion, the relation becomes:

Student_ID	Name	Branch
101	Ravi	CSE
103	Arjun	ECE

Hence, the deletion operation changes the state of the relation by removing a tuple.

- Removes obsolete or unwanted data.
- Reduces the number of tuples in a relation.
- Maintains the relevance of information stored in the database.
- Must preserve referential integrity.

Q4.b. Discuss the aggregation functions and grouping in relational algebra with suitable examples (5M)

Aggregation and grouping are extended operations in relational algebra that are used to perform various computations on groups of tuples. They are particularly useful for summarizing information and generating statistical results from relations. Common aggregation functions include **COUNT, SUM, AVERAGE (AVG), MAX, and MIN**. Grouping allows tuples having the same value for one or more attributes to be treated as a single group before applying aggregate functions.

Aggregation Functions

Aggregation functions are operations that take a collection of values as input and produce a single value as output. They are used to summarize data and derive meaningful information from relations. Aggregate functions are generally applied to a set of tuples rather than individual tuples.

In extended relational algebra, aggregation is represented using the **γ (gamma)** operator.

The general form is

$$\gamma_{F_1(A_1),F_2(A_2),\ldots}(R)$$

where:

- $F_1, F_2, \ldots$ are aggregate functions.
- $A_1, A_2, \ldots$ are attributes.
- R is the relation.

Types of Aggregation Functions

1. COUNT

The **COUNT** function returns the total number of tuples in a relation or group.

Example

Consider relation EMPLOYEE(EID, Name, DeptID, Salary)

EID	Name	DeptID	Salary
1	Alice	10	5000
2	Bob	20	6000
3	Eve	20	6500

To find the number of employees:

$$\gamma_{COUNT(EID)}(EMPLOYEE)$$

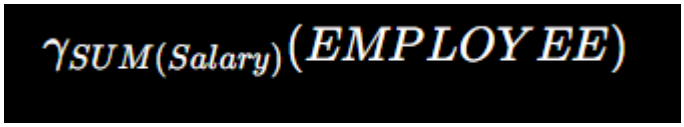
Result:

COUNT(EID) = 3

2. SUM

The **SUM** function computes the total of all values of an attribute.

To find the total salary of all employees:



Result:

SUM(Salary) = 17500

3. AVERAGE (AVG)

The **AVG** function calculates the arithmetic mean of attribute values.

To determine the average salary:

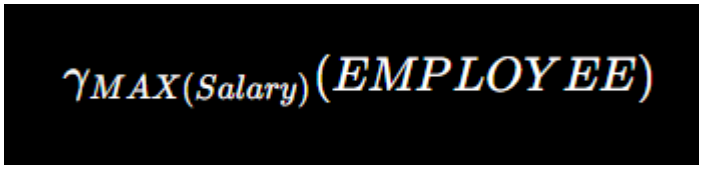
Result:

AVG(Salary) = 5833.33

4. MAX

The **MAX** function returns the highest value of an attribute.

To find the maximum salary:



Result:

MAX(Salary) = 6500

5. MIN

The **MIN** function returns the smallest value of an attribute.

To find the minimum salary:

Result:

MIN(Salary) = 5000

Grouping in Relational Algebra

Grouping is the process of partitioning tuples having the same values of certain attributes into separate groups. Aggregate functions are then applied independently to each group. Grouping enables the analysis of data on a category-wise basis rather than on the entire relation.

In extended relational algebra, grouping is represented using the gamma (γ) operator with grouping attributes.

General notation:

$$\gamma_{Grouping\ Attribute,\ Aggregate\ Function(Attribute)}(R)$$

Example

Consider the relation EMPLOYEE:

EID	Name	DeptID	Salary
1	Alice	10	5000
2	Bob	20	6000
3	Eve	20	6500
4	John	10	5500

Suppose we want to find the average salary department-wise.

Relational Algebra Expression

$$\gamma_{DeptID,\ AVG(Salary)}(EMPLOYEE)$$

Result

DeptID	AVG(Salary)
10	5250
20	6250

Here, tuples are first grouped according to **DeptID**, and then the average salary is calculated for each group separately.

Advantages of Aggregation and Grouping

- They provide summarized information from large relations.
- They facilitate statistical analysis of data.
- They enable department-wise, category-wise, or group-wise computations.
- They reduce the complexity of query processing.
- They help in decision-making and report generation.

c.	Given the relational tables: Student: SID | Name a | Alice b | Bob c | Carol Project: PID | Project Name p | Alpha q | Beta r | Gamma Language: LID | Language Name x | Python y | Java z | C++ Enrollment: SID | PID a | p a | q b | q c | r Write relational algebra expression for the following: (i) Rename the student table to Learner and display it. (ii) Find the students (learners) who are not enrolled in any project. (iii) Find the students who are enrolled in all projects. (iv) Find the students who are not enrolled in any project. (v) Find the students who are enrolled in both the 'Alpha' and 'Beta' projects.	10	L3	CO2

(i)

$$\rho_{Learner(SID,\ Name)}(Student)$$

(ii)

$$\pi_{SID,\ Name}(Student) - \pi_{SID,\ Name}(Student \bowtie Enrollment)$$

(iii)

$$\pi_{SID,PID}(Enrollment) \div \pi_{PID}(Project)$$

(iv)

$$\pi_{SID,\ Name}(Student) - \pi_{SID,\ Name}(Student \bowtie Enrollment)$$

(v)

$$\pi_{Name}\left(Student \bowtie (\pi_{SID}(\sigma_{PID='p'}(Enrollment)) \cap \pi_{SID}(\sigma_{PID='q'}(Enrollment)))\right)$$

Q.5	**a.**	Explain Armstrong inference rules.	**05**	**L2**	**CO4**
	b.	What is the need for normalization? Explain 1NF, 2NF and 3NF with examples.	**05**	**L2**	**CO4**
	c.	What is functional dependency? Write an algorithm to find minimal cover for set of functional dependencies. Construct minimal cover M for set of functional dependencies which are: E = {B → A, D → A, AB → D}	**10**	**L3**	**CO4**

Q. 5 a. Explain Armstrong inference rules (5M)

Armstrong's axioms provide a systematic method for reasoning about functional dependencies and determining the closure of attribute sets.

The three basic Armstrong's inference rules are **Reflexivity, Augmentation, and Transitivity**. Several other rules such as Union, Decomposition, and Pseudo transitivity are derived from these basic rules.

1. Reflexivity Rule

The **Reflexivity Rule** states that if an attribute set Y is a subset of another attribute set X, then X functionally determines Y.

$$\text{If } Y \subseteq X, \text{ then } X \rightarrow Y$$

This rule is based on the intuitive idea that a set of attributes always determines itself and any subset of itself. Such dependencies are called **trivial functional dependencies**.

Reflexivity is considered the simplest and most fundamental rule among Armstrong's axioms. It guarantees that every attribute set inherently possesses enough information to determine its own components.

Example

If

$$X = \{A, B, C\}$$

and

$$Y = \{A, B\}$$

then

$$ABC \rightarrow AB$$

since AB is a subset of ABC.

Importance

- Produces trivial dependencies.
- Serves as the basis for deriving more complex dependencies.
- Helps identify attributes that are naturally dependent on a given attribute set.
- Plays a significant role in determining attribute closures.

2. Augmentation Rule

Definition

The **Augmentation Rule** states that if an attribute set X functionally determines another attribute set Y, then adding a common set of attributes Z to both sides preserves the dependency.

$$\text{If } X \rightarrow Y, \text{ then } XZ \rightarrow YZ$$

This rule implies that adding additional information to both sides of a functional dependency does not invalidate the dependency. The relationship between the original attributes remains intact even after augmentation.

Augmentation is particularly useful when combining several dependencies during normalization and decomposition of relations.

Example

If

$$A \rightarrow B$$

then by adding C to both sides,

$$AC \rightarrow BC$$

Similarly, if

$$EmployeeID \rightarrow DepartmentID$$

then

$$EmployeeID, ProjectID \rightarrow DepartmentID, ProjectID$$

- Helps derive larger functional dependencies.

- Preserves dependencies when additional attributes are introduced.
- Used extensively in proving dependency preservation.
- Facilitates decomposition and schema refinement.

3. Transitivity Rule

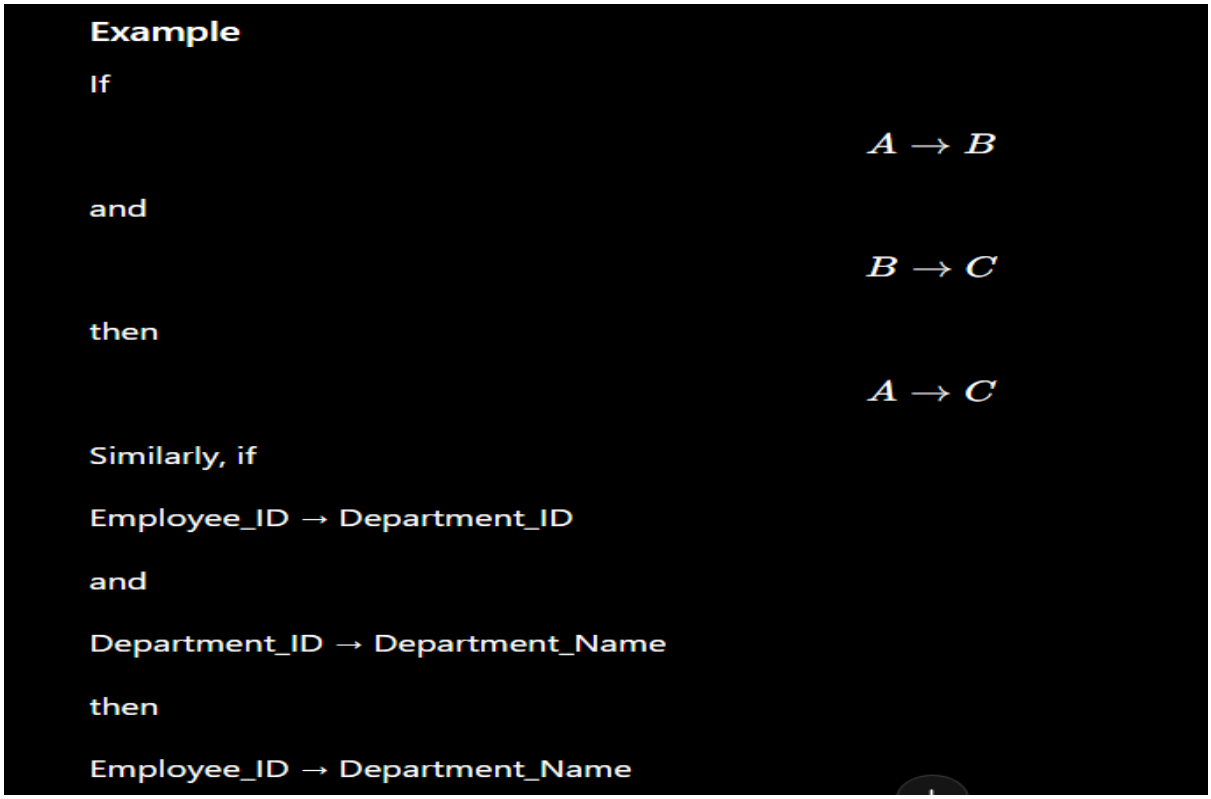
Definition

The **Transitivity Rule** states that if X functionally determines Y and Y functionally determines Z, then X functionally determines Z.

$$\text{If } X \rightarrow Y \text{ and } Y \rightarrow Z, \text{ then } X \rightarrow Z$$

This rule is similar to the transitive property in mathematics. It establishes indirect relationships among attributes and is very important in identifying transitive dependencies in relations.

Transitive dependencies are undesirable in higher normal forms, and therefore this rule is widely used in normalization.



- Helps identify indirect dependencies.
- Used in determining transitive dependencies.
- Plays an important role in achieving Third Normal Form (3NF).
- Assists in finding attribute closures and candidate keys.

Derived Rules of Armstrong's Axioms

Apart from the three basic axioms, several other inference rules can be derived. These are called **secondary or derived rules**.

Union Rule

The Union Rule states that if X functionally determines Y and X functionally determines Z, then X functionally determines the combination of Y and Z.

- Combines multiple dependencies into a single dependency.
- Reduces the number of functional dependencies to be considered.

Decomposition Rule

The Decomposition Rule states that if X functionally determines YZ, then X separately determines Y and Z.

This rule is also called the **Projectivity Rule**.

Importance

- Breaks a complex dependency into simpler dependencies.
- Useful in normalization and dependency analysis.

Pseudotransitivity Rule

The Pseudotransitivity Rule states that if X determines Y and WY determines Z, then WX determines Z.

This rule is an extension of the transitivity rule and is useful in deriving more complex dependencies.

- Helps derive indirect dependencies involving composite attributes.
- Used in advanced normalization procedures.

Q.5 b What is the need for normalization? Explain 1NF, 2NF, and 3NF with examples (5M)

Need for Normalization

Normalization is a systematic process of organizing data in a database to minimize redundancy and eliminate undesirable characteristics such as insertion, deletion, and update

anomalies. It involves decomposing a relation into smaller and well-structured relations while preserving data integrity and consistency.

In poorly designed relations, the same information may be stored repeatedly, leading to wastage of storage space and inconsistency in data. Normalization provides a set of guidelines, known as normal forms, that help in designing efficient relational schemas.

Normalization is required for the following reasons:

- To eliminate data redundancy and duplication.
- To avoid insertion anomalies.
- To prevent deletion anomalies.
- To eliminate update anomalies.
- To ensure data consistency and integrity.
- To simplify the database structure.
- To improve efficiency in storage and maintenance.
- To facilitate easy modification and retrieval of data.
- To achieve a flexible and reliable database design.

The various stages of normalization are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF), and higher normal forms.

First Normal Form (1NF)

A relation is said to be in **First Normal Form (1NF)** if all its attributes contain only atomic (indivisible) values and each attribute contains a single value. In other words, repeating groups and multivalued attributes are not permitted in a relation satisfying 1NF.

The primary objective of 1NF is to eliminate repeating groups and ensure that every field contains only one value.

Example

Consider the relation:

STUDENT

SID	Name	Subjects
1	Ravi	DBMS, OS
2	Priya	CN
3	Arjun	DBMS, CN

The attribute **Subjects** contains multiple values and hence the relation is not in 1NF.

To convert it into 1NF, each subject is stored separately.

STUDENT

SID	Name	Subject
1	Ravi	DBMS
1	Ravi	OS
2	Priya	CN

3	Arjun	DBMS
3	Arjun	CN

Now each attribute contains only atomic values. Therefore, the relation is in First Normal Form.

Characteristics of 1NF

- Eliminates repeating groups.
- Ensures atomicity of attribute values.
- Removes multivalued attributes.
- Forms the basis for higher normal forms.

Second Normal Form (2NF)

A relation is said to be in **Second Normal Form (2NF)** if it is already in 1NF and every non-prime attribute is fully functionally dependent on the entire primary key. That is, no non-key attribute should depend on only a part of a composite key.

The main objective of 2NF is to eliminate partial dependencies.

Example

Consider the relation:

ENROLLMENT

SID	**CourseID**	**StudentName**	**CourseName**
101	C1	Ravi	DBMS

101	C2	Ravi	OS
102	C1	Priya	DBMS

The composite primary key is:

$$(SID, CourseID)$$

Functional dependencies are:

$$SID \rightarrow StudentName$$

$$CourseID \rightarrow CourseName$$

Since StudentName depends only on SID and CourseName depends only on CourseID, partial dependencies exist. Therefore, the relation is not in 2NF.

Decomposition

STUDENT

SID	StudentName
101	Ravi
102	Priya

COURSE

CourseID	CourseName
C1	DBMS
C2	OS

ENROLLMENT

SID	CourseID
101	C1
101	C2
102	C1

Now all non-key attributes depend on the whole primary key, and hence the relations are in Second Normal Form.

Characteristics of 2NF

- Relation must already be in 1NF.
- Eliminates partial functional dependencies.
- Removes redundant storage of data.
- Reduces update anomalies.

Third Normal Form (3NF)

A relation is said to be in **Third Normal Form (3NF)** if it is already in 2NF and there are no transitive dependencies among non-key attributes. In other words, no non-prime attribute should depend on another non-prime attribute.

The main objective of 3NF is to eliminate transitive dependencies.

Example

Consider the relation:

EMPLOYEE

EmpID	EmpName	DeptID	DeptName
E1	Ravi	D1	CSE
E2	Priya	D2	ISE
E3	Arjun	D1	CSE

Primary Key:

$$EmpID$$

Functional dependencies are:

$$EmpID \rightarrow DeptID$$

$$DeptID \rightarrow DeptName$$

Therefore,

$$EmpID \rightarrow DeptName$$

Here, DeptName depends on DeptID, which in turn depends on EmpID. Hence, a transitive dependency exists, and the relation is not in 3NF.

Decomposition

EMPLOYEE

EmpID	EmpName	DeptID
E1	Ravi	D1
E2	Priya	D2
E3	Arjun	D1

DEPARTMENT

DeptID	DeptName
D1	CSE
D2	ISE

Now,

- EmpName depends directly on EmpID.
- DeptName depends directly on DeptID.

There are no transitive dependencies, and hence the relations are in Third Normal Form.

Characteristics of 3NF

- Relation must already be in 2NF.
- Eliminates transitive dependencies.
- Reduces redundancy further.
- Prevents insertion, deletion, and update anomalies.
- Improves data consistency and integrity.

Q 5. C.

c.	What is functional dependency? Write an algorithm to find minimal cover for set of functional dependencies. Construct minimal cover M for set of functional dependencies which are: E = {B → A, D → A, AB → D}	**10**	**L3**	**CO4**

A **functional dependency (FD)** is a constraint between two sets of attributes in a relation. An attribute or set of attributes **X** is said to functionally determine another attribute or set of attributes **Y**, denoted by

$$X \rightarrow Y$$

if each value of **X** is associated with exactly one value of **Y**.

In other words, if two tuples have the same value for attribute set **X**, then they must also have the same value for attribute set **Y**.

Example

Consider the relation:

USN	Name	Branch
101	Ravi	CSE
102	Priya	ISE

Here,

$$USN \rightarrow Name$$

$$USN \rightarrow Branch$$

because a particular USN uniquely determines the corresponding Name and Branch.

Algorithm to Find Minimal Cover

Let FFF be a set of functional dependencies.

Step 1: Convert RHS to Single Attribute

Replace every dependency having multiple attributes on the right-hand side with equivalent dependencies having a single attribute.

If

$$X \to Y_1Y_2$$

then replace it by

$$X \to Y_1$$

and

$$X \to Y_2$$

Step 2: Remove Extraneous Attributes from LHS

For each functional dependency $X \to A$, examine whether any attribute in X is redundant.

If

$$(X - \{B\})^+ \supseteq A$$

then attribute B is extraneous and can be removed.

Step 3: Remove Redundant Functional Dependencies

For each dependency $X \to A$, temporarily remove it and compute the closure of X with respect to the remaining dependencies.

If

$$A \in X^+$$

then the dependency is redundant and can be eliminated.

Step 4: The Remaining Set of Dependencies Forms the Minimal Cover

The resulting set of dependencies is called the **minimal cover** or **canonical cover**.

Construction of Minimal Cover

Given

$$E = \{B \to A,\ D \to A,\ AB \to D\}$$

Step 1: Ensure Single Attribute on RHS

All dependencies already have a single attribute on the RHS.

Hence,

$$E = \{B \to A,\ D \to A,\ AB \to D\}$$

Step 2: Remove Extraneous Attributes from Left-Hand Side

Consider

$$AB \rightarrow D$$

Check whether A is extraneous

Compute closure of B:

$$B^+ = \{B\}$$

Using

$$B \rightarrow A$$

we get

$$B^+ = \{A, B\}$$

Using

$$AB \rightarrow D$$

we obtain

$$B^+ = \{A, B, D\}$$

Since

$$D \in B^+$$

attribute A is extraneous.

Therefore,

$$AB \rightarrow D$$

becomes

$$B \rightarrow D$$

Thus,

$$E = \{B \rightarrow A,\ D \rightarrow A,\ B \rightarrow D\}$$

Step 3: Remove Redundant Dependencies

Check whether

$$B \to A$$

is redundant.

Compute closure of B using

$$\{D \to A,\ B \to D\}$$

From

$$B \to D$$

we obtain

$$D$$

and using

$$D \to A$$

we get

$$A$$

Hence

$$B^+ = \{A, B, D\}$$

Therefore,

$$B \to A$$

is redundant and can be removed.

The remaining dependencies are

$$M = \{B \to D,\ D \to A\}$$

Minimal Cover

$$\boxed{M = \{B \to D,\ D \to A\}}$$

Conclusion

The **minimal cover** of the set of functional dependencies

$$E = \{B \to A,\ D \to A,\ AB \to D\}$$

is

$$\boxed{M = \{B \to D,\ D \to A\}}$$

This set is equivalent to the original set of dependencies and contains no redundant attributes or functional dependencies.

Q.6	**a.**	Explain the types of update anomalies in SQL with an example.	**05**	**L2**	**CO4**
	b.	Explain types of JBBC drivers.	**05**	**L2**	**CO5**
	c.	Consider the schema R = ABCD, subjected to FDs F = {A →B, B →C}, and the non-binary partition D1 = {ACD, AB, BC}. State whether D1 is a lossless decomposition? [give all steps in detail].	**10**	**L3**	**CO4**

Q.6. a Explain the types of update anomalies in SQL with an example (5M)

An **update anomaly** is an inconsistency that arises in a database due to redundancy and poor database design. When the same information is stored repeatedly in multiple tuples, modifications to the data may lead to inconsistencies and undesirable effects. Update anomalies are one of the major reasons for performing normalization.

The three types of update anomalies are:

1. **Insertion Anomaly**
2. **Deletion Anomaly**
3. **Modification (Updation) Anomaly**

Consider the following relation:

EMPLOYEE

EmpID	EmpName	DeptID	DeptName
E1	Ravi	D1	CSE
E2	Priya	D2	ISE

E3	Arjun	D1	CSE

Notice that the department name **CSE** is stored repeatedly for employees E1 and E3.

1. Insertion Anomaly

Definition

An **Insertion Anomaly** occurs when certain information cannot be inserted into the database without the presence of some other information. In other words, the database structure forces unnecessary data to be inserted along with the required data.

Insertion anomalies occur when all information about multiple entities is stored in a single relation.

Example

Suppose a new department **ECE (D3)** is established, but no employee has yet been assigned to it.

Since the EMPLOYEE table requires employee details, the department information cannot be inserted independently.

Thus, information about the ECE department cannot be stored until at least one employee joins that department.

Importance

Insertion anomaly:

- Prevents independent insertion of data.

- Results in unnecessary null values.
- Causes difficulty in maintaining new information.

2. Deletion Anomaly

Definition

A **Deletion Anomaly** occurs when deleting a tuple unintentionally results in the loss of additional valuable information. This happens because multiple facts are stored together in the same relation.

Deletion anomalies lead to accidental loss of information that should have been retained.

Example

Consider the relation:

EmpID	EmpName	DeptID	DeptName
E1	Ravi	D1	CSE
E2	Priya	D2	ISE
E3	Arjun	D1	CSE

Suppose employee E2 leaves the organization and the corresponding tuple is deleted.

After deletion:

EmpID	EmpName	DeptID	DeptName
E1	Ravi	D1	CSE

E3	Arjun	D1	CSE

The information regarding department **D2 (ISE)** is also lost because there are no other employees belonging to that department.

Thus, deleting an employee record causes the loss of department information.

Importance

Deletion anomaly:

- Causes accidental loss of useful information.
- Leads to inconsistency in the database.
- Makes data maintenance difficult.

3. Modification (Updation) Anomaly

A **Modification Anomaly** occurs when the same information is stored in several tuples and changes made to one tuple are not reflected in all the remaining tuples. As a result, inconsistent data may exist in the database.

Modification anomalies arise because of redundancy.

Example

Consider:

EmpID	**EmpName**	**DeptID**	**DeptName**
E1	Ravi	D1	CSE
E2	Priya	D2	ISE

E3	Arjun	D1	CSE

Suppose the department name **CSE** is changed to **Computer Science**.

Both tuples corresponding to employees E1 and E3 must be updated.

If only one tuple is updated, the relation becomes:

EmpID	**EmpName**	**DeptID**	**DeptName**
E1	Ravi	D1	Computer Science
E2	Priya	D2	ISE
E3	Arjun	D1	CSE

Now, two different department names exist for the same department ID, resulting in inconsistency.

Modification anomaly:

- Produces inconsistent data.
- Requires multiple updates for the same information.
- Increases maintenance cost.
- Reduces database reliability.

Q.6.b. explain types of JDBC drivers (5M)

JDBC (Java Database Connectivity) is an API that enables Java applications to communicate with databases. JDBC drivers are software components that convert JDBC calls into

database-specific calls. They act as an interface between Java applications and database servers.

Based on their architecture and method of communication with the database, JDBC drivers are classified into four types:

1. Type-1 Driver (JDBC-ODBC Bridge Driver)
2. Type-2 Driver (Native API Driver)
3. Type-3 Driver (Network Protocol Driver)
4. Type-4 Driver (Thin Driver)

1. Type-1 Driver (JDBC-ODBC Bridge Driver)

Type-1 driver is also known as the **JDBC-ODBC Bridge Driver**. It translates JDBC method calls into ODBC function calls and uses an ODBC driver to communicate with the database.

Features

- Converts JDBC calls into ODBC calls.
- Requires ODBC driver installation on the client machine.
- Supports any database having an ODBC driver.
- Processing involves multiple layers, resulting in lower performance.

2. Type-2 Driver (Native API Driver)

Type-2 driver converts JDBC calls into native database API calls. It uses the client-side libraries provided by the database vendor to communicate with the database.

Features

- Uses native database libraries.
- Part of the processing occurs on the client side.
- Provides better performance than Type-1 driver.

3. Type-3 Driver (Network Protocol Driver)

Type-3 driver converts JDBC calls into a database-independent network protocol and sends them to a middleware server. The middleware server translates the request into database-specific calls.

Features

- Uses a middleware server.
- Client does not require database-specific libraries.
- Supports communication with multiple databases.
- Performance is lower due to intermediate processing.

4. Type-4 Driver (Thin Driver)

Type-4 driver is a pure Java driver that converts JDBC calls directly into database-specific protocols. It communicates directly with the database server without using native libraries or middleware.

Features

- Written entirely in Java.
- Direct communication with the database server.

- Does not require ODBC drivers or native libraries.

	c.	Consider the schema R = ABCD, subjected to FDs F = {A →B, B →C}, and the non-binary partition D1 = {ACD, AB, BC}. State whether D1 is a lossless decomposition? [give all steps in detail].	10	L3	CO4

Q. Consider the schema $R = ABCD$, subjected to FDs

$$F = \{A \to B,\ B \to C\}$$

and the non-binary decomposition

$$D_1 = \{ACD,\ AB,\ BC\}$$

State whether D_1 is a **lossless decomposition**. Give all steps in detail.

Given

Relation:

$$R(A, B, C, D)$$

Functional Dependencies:

$$F = \{A \to B,\ B \to C\}$$

Decomposition:

$$D_1 = \{ACD,\ AB,\ BC\}$$

Step 1: Construct the Table

Create one row for each decomposed relation and one column for each attribute of R.

Relation	A	B	C	D
ACD	a1	b1	c1	d1
AB	a2	b2	c2	d2
BC	a3	b3	c3	d3

Step 2: Replace Present Attributes by Common Symbols

If an attribute is present in a relation, replace it by the attribute symbol itself (a, b, c, d).

Otherwise use distinct symbols.

Relation	A	B	C	D
ACD	a	b1	c	d
AB	a	b	c2	d2
BC	a3	b	c	d3

Step 3: Apply Functional Dependency $A \rightarrow B$

Rows having the same value of A must have the same value of B.

Rows 1 and 2 have

$$A = a$$

Therefore B values become equal:

$$b_1 \rightarrow b$$

New table:

Relation	A	B	C	D
ACD	a	b	c	d
AB	a	b	c2	d2
BC	a3	b	c	d3

Step 4: Apply Functional Dependency $B \rightarrow C$

Rows 1, 2, and 3 have the same B value:

$$B = b$$

Hence their C values must be identical.

Therefore,

$$c_2 \rightarrow c$$

Updated table:

Relation	A	B	C	D
ACD	a	b	c	d
AB	a	b	c	d2
BC	a3	b	c	d3

Step 5: Examine the Rows

The first row becomes

(a,b,c,d)(a,b,c,d)(a,b,c,d)

which consists entirely of attribute symbols and contains no subscripted entries.

Therefore, one row corresponds exactly to the original relation.

Conclusion

Since one row contains only the original attribute symbols,

$$\boxed{D_1 = \{ACD,\ AB,\ BC\}}$$

is a **Lossless-Join Decomposition**.

$$\boxed{\text{The decomposition } D_1 = \{ACD,\ AB,\ BC\} \text{ is lossless.}}$$

Reason:

After applying the functional dependencies

$$A \to B$$

and

$$B \to C$$

one row of the lossless-join test table becomes

$$(a, b, c, d),$$

which proves that the decomposition is **lossless**.

Q.7	a.	Define transaction. Discuss ACID properties.	05	L2	CO5
	b.	With a neat diagram, explain transition diagram of a transaction.	05	L2	CO5
	c.	Demonstrate working of assertion and triggers in SQL with example.	10	L3	CO5

Q.7. a. Define transaction. Discuss ACID properties (5M)

A **transaction** is a sequence of one or more database operations that performs a single logical unit of work. A transaction may consist of operations such as insertion, deletion, updation, or retrieval of data. A transaction is considered complete only when all its operations are executed successfully. If any operation fails, the entire transaction is rolled back to preserve the consistency of the database.

Thus, a transaction transforms the database from one consistent state to another consistent state.

A transaction generally consists of the following operations:

- **BEGIN TRANSACTION** – Marks the beginning of a transaction.
- **READ(X)** – Reads the value of data item X.
- **WRITE(X)** – Updates the value of data item X.
- **COMMIT** – Permanently saves the changes made by the transaction.
- **ROLLBACK (ABORT)** – Cancels all changes made by the transaction and restores the database to its previous state.

Example

Consider a bank fund transfer of ₹5000 from Account A to Account B.

Read(A)

A = A – 5000

Write(A)

Read(B)

B = B + 5000

Write(B)

Commit

Both debit and credit operations together constitute a single transaction. If the system fails after deducting money from Account A but before adding it to Account B, the transaction must be rolled back to maintain consistency.

ACID Properties

ACID properties are a set of four properties that guarantee the correctness, reliability, and consistency of database transactions. ACID stands for **Atomicity, Consistency, Isolation, and Durability**.

1. Atomicity

Atomicity states that a transaction is treated as an indivisible unit of work. Either all the operations of a transaction are executed successfully, or none of them are executed. There is no concept of partial execution.

This property ensures that if any operation within the transaction fails, the entire transaction is aborted and the database is restored to its previous state.

Example

Consider transferring ₹5000 from Account A to Account B.

A = A – 5000

B = B + 5000

If the system crashes after deducting money from Account A but before crediting Account B, both operations are rolled back. Hence, either both operations occur or neither occurs.

Importance

- Prevents partial updates.

- Maintains data reliability.
- Ensures complete execution of transactions.

2. Consistency

Consistency ensures that every transaction preserves the integrity constraints of the database and transforms the database from one valid state to another valid state.

A transaction should not violate constraints such as primary key, foreign key, or domain constraints.

Example

Suppose Account A contains ₹20,000 and Account B contains ₹10,000.

Before transfer:

Total Balance = ₹30,000

After transferring ₹5000 from A to B:

A = ₹15,000

B = ₹15,000

Total Balance remains:

₹30,000

Thus, the consistency of the database is maintained.

Importance

- Preserves database integrity.

- Ensures valid data at all times.
- Prevents constraint violations.

3. Isolation

Definition

Isolation ensures that multiple transactions executing concurrently do not interfere with each other. The intermediate results of one transaction are hidden from other transactions until the transaction is completed.

From the user's perspective, concurrent transactions appear as though they are executed serially.

Example

Suppose two users simultaneously withdraw money from the same account.

Without isolation, both transactions may access the same balance and produce incorrect results. With isolation, one transaction completes before the other accesses the updated data, thereby preventing inconsistencies.

Importance

- Prevents interference among concurrent transactions.
- Avoids problems such as dirty reads and lost updates.
- Ensures correctness in multi-user environments.

4. Durability

Durability guarantees that once a transaction is committed, its effects are permanently stored in the database. Even if a system failure or power outage occurs after the commit operation, the committed changes are not lost.

The recovery mechanism ensures that committed transactions survive failures.

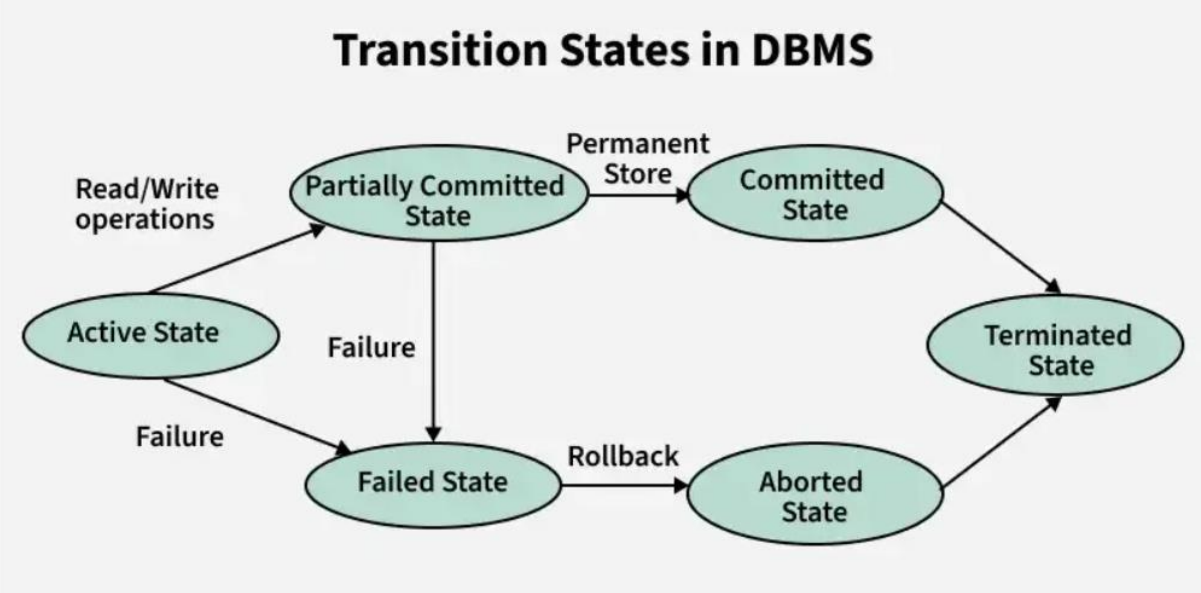
Example

Suppose a customer deposits ₹10,000 into an account and the transaction is committed. Even if the system crashes immediately afterward, the deposited amount remains stored in the database and can be recovered.

Importance

- Guarantees permanent storage of committed data.
- Protects against hardware and software failures.
- Provides reliability and fault tolerance.

Q.7. b Write a neat diagram, explain transition diagram of a transaction (5M)



1. Active State

- It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state.
- Operations such as insertion, deletion, or updation are performed during this state.
- During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

2. Partially Committed

- The transaction has finished its final operation, but the changes are still not saved to the database.
- After completing all read and write operations, the modifications are initially stored in main memory or a local buffer. If the changes are made permanent in the database then the state will change to "committed state" and in case of failure it will go to the "failed state".

3. Committed

This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation, i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.

4. Failed State

If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state. Here, the database recovery system makes sure that the database is in a consistent state.

5. Aborted State

If a transaction reaches the failed state due to a failed check, the database recovery system will attempt to restore it to a consistent state. If recovery is not possible, the transaction is either rolled back or cancelled to ensure the database remains consistent.

In the aborted state, the DBMS recovery system performs one of two actions:

- **Kill the transaction**: The system terminates the transaction to prevent it from affecting other operations.
- **Restart the transaction**: After making necessary adjustments, the system reverts the transaction to an active state and attempts to continue its execution.

6. Terminated State

It refers to the final state of a transaction, indicating that it has completed its execution. Once a transaction reaches this state, it has either been successfully committed or aborted. In this state, no further actions are required from the transaction, as the database is now stable.

Example of Transaction States
Imagine a bank transaction where a user wants to transfer $500 from **Account A** to **Account B**. The system should handle the following transaction states:

Active State

- The transaction begins. It reads the balance of Account A and checks if it has enough funds.
- **Example:** Read balance of Account A = $1000.

Partially Committed State

- The transaction performs all its operations but hasn’t yet saved (committed) the changes to the database.
- **Example:** Deduct $500 from Account A's balance ($1000 - $500 = $500) and temporarily update Account B's balance (add $500).

Committed State

- The transaction successfully completes, and the changes are saved permanently in the database.
- **Example:** Account A’s new balance = $500; Account B’s new balance = $1500. Changes are written to the database.

Failed State

- If something goes wrong during the transaction (e.g., power failure, system crash), the transaction moves to this state.

- **Example:** System crashes after deducting $500 from Account A but before adding it to Account B.

Aborted State

- The failed transaction is rolled back, and the database is restored to its original state.
- **Example:** Account A's balance is restored to $1000, and no changes are made to Account B.

These states ensure that either the transaction completes successfully (committed) or the database is restored to its original state (aborted), maintaining consistency and preventing errors.

Q.7. c Demonstrate working of assertion and triggers in SQL with example (10M)

Assertions and triggers are mechanisms provided by SQL to maintain the integrity and consistency of a database. While **assertions** are used to specify constraints that must always hold true for the database, **triggers** are procedures that are automatically executed in response to certain database events.

1. Assertion

An **assertion** is a schema-level constraint that specifies a condition that must always be satisfied by the database. Whenever an insert, delete, or update operation is performed, the DBMS checks whether the assertion condition is violated. If the condition evaluates to false, the operation is rejected.

Assertions are used to enforce complex constraints involving one or more relations.

Syntax

```
CREATE ASSERTION assertion_name
CHECK (condition);
```

Example

Consider a bank database where the balance of an account should never become negative.

ACCOUNT

Acc_No	Name	Balance
A101	Ravi	10000

A102	Priya	15000

The following assertion ensures that all account balances are non-negative.

```
CREATE ASSERTION Positive_Balance
CHECK (NOT EXISTS
       (SELECT *
        FROM ACCOUNT
        WHERE Balance < 0));
```

Working

- Whenever a tuple is inserted or updated, the DBMS checks the assertion.
- If any account balance becomes negative, the assertion fails.
- The operation causing the violation is rejected.
- Thus, assertions maintain database integrity.

Advantages

- Enforces database-wide constraints.
- Ensures consistency among relations.
- Prevents invalid data from entering the database.

2. Trigger

A **trigger** is a stored procedure that is automatically executed whenever a specified event such as INSERT, DELETE, or UPDATE occurs on a table. Triggers are used to enforce business rules, maintain audit trails, and ensure data integrity.

A trigger is associated with a particular table and is activated automatically when the triggering event occurs.

Syntax

```
CREATE TRIGGER trigger_name
BEFORE | AFTER INSERT | DELETE | UPDATE
ON table_name
FOR EACH ROW
BEGIN
   Trigger body;
END;
```

Example

Consider the relation:

EMPLOYEE

EmpID	Name	Salary
E1	Ravi	50000

E2	Priya	60000

Suppose salary should never be less than ₹20,000.

Trigger Definition

```
CREATE TRIGGER Check_Salary
BEFORE INSERT OR UPDATE
ON EMPLOYEE
FOR EACH ROW
BEGIN
   IF :NEW.Salary < 20000 THEN
      RAISE_APPLICATION_ERROR(-20001,
      'Salary cannot be less than 20000');
   END IF;
END;
```

Working

Suppose the following statement is executed:

```
INSERT INTO EMPLOYEE
VALUES('E3','Arjun',15000);
```

Before inserting the tuple, the trigger is activated.

Since

```
Salary = 15000 < 20000
```

the trigger raises an error and the insertion is rejected.

If

```
INSERT INTO EMPLOYEE
VALUES('E3','Arjun',25000);
```

the condition is satisfied, and the tuple is inserted successfully.

Types of Triggers

1. BEFORE Trigger

Executed before the triggering event occurs.

Example:

```
BEFORE INSERT
BEFORE UPDATE
BEFORE DELETE
```

2. AFTER Trigger

Executed after the triggering event has occurred.

Example:

```
AFTER INSERT
AFTER UPDATE
AFTER DELETE
```

3. ROW-Level Trigger

Executed once for each row affected by the statement.

4. STATEMENT-Level Trigger

Executed once for the entire SQL statement.

Differences Between Assertion and Trigger

Assertion	Trigger
Specifies constraints on the database.	Executes automatically when an event occurs.
Checks a condition that must always hold true.	Performs actions in response to INSERT, UPDATE, or DELETE.
Mainly used for integrity constraints.	Used for auditing, logging, and enforcing business rules.
Does not execute procedural code.	Can execute procedural statements.

Q.8	a.	Explain cursor and its properties in embedded SQL with suitable example.	05	L2	CO5
	b.	Determine if the following schedule is serializable and explain your reasoning: i) T1 : R(X)W(X) T2 : R(X)W(X) T1 : COMMIT T2 : COMMIT ii) T1 : W(X)R(Y) T2 : R(X)W(Y) T1 : COMMIT T2 : COMMIT	05	L2	CO5

	c.	Consider the tables below: Sailors (sid : integer, sname : string, rating : integer, age : real) Boats (bid : integer, bname : string, color : string); Reserves (sid : integer, bid : integer, day : date) Write SQL queries for the following: (i) Write create table statement for reserves. (ii) Find all information of sailors who have reserved boat number 101. (iii) Find the names of sailors who have reserved at least one boat. (iv) Find the names of sailors who have reserved a red boat. (v) Find the average age of sailors for each rating level.	10	L3	CO5

Q.8.a. Explain cursor and its properties in embedded SQL with suitable example (5M)

A **cursor** is a database object used in embedded SQL to retrieve and process query results one tuple (row) at a time. It acts as a pointer to the tuples returned by an SQL query and enables an application program to access each row individually.

Cursors are particularly useful when a query returns multiple rows and the application needs to process them sequentially. The cursor maintains a pointer to the current row and advances to the next row after each retrieval operation.

In embedded SQL, a cursor is associated with a query and typically involves the following operations:

1. **DECLARE CURSOR**
2. **OPEN CURSOR**
3. **FETCH**
4. **CLOSE CURSOR**

Properties of Cursor

1. Active Set
The collection of tuples produced by the query associated with the cursor is called the **active set**. The cursor traverses these tuples one by one.

Example
If the query

```
SELECT EmpID, Name
FROM EMPLOYEE;
```

returns five rows, then those five rows constitute the active set.

2. Current Row Pointer

A cursor maintains a pointer to the current tuple being processed. After each FETCH operation, the pointer automatically moves to the next row.
Thus, the cursor always knows which row is currently being accessed.

3. Sequential Processing

Cursors process tuples sequentially, one row at a time. This feature is useful when row-by-row operations are required.
For example, salaries of employees can be increased individually based on specific conditions.

4. Updatable Cursor

5. An updatable cursor allows modification or deletion of the current row being pointed to by the cursor.
6. Example:
7.
```
UPDATE EMPLOYEE
SET Salary = Salary + 5000
WHERE CURRENT OF Emp_Cursor;
```
8. Here, the current tuple accessed by the cursor is updated.

Advantages of Cursors

- Enable row-by-row processing.
- Useful when queries return multiple tuples.
- Facilitate updates and deletions on the current row.
- Provide controlled access to query results.
- Support sequential and scrollable access to data.

Q.8	**a.**	Explain cursor and its properties in embedded SQL with suitable example.	**05**	**L2**	**CO5**
	b.	Determine if the following schedule is serializable and explain your reasoning: i) T1 : R(X)W(X) T2 : R(X)W(X) T1 : COMMIT T2 : COMMIT ii) T1 : W(X)R(Y) T2 : R(X)W(Y) T1 : COMMIT T2 : COMMIT	**05**	**L2**	**CO5**

(i) Schedule

T1 : R(X) W(X)
T2 : R(X) W(X)
T1 : COMMIT T2
: COMMIT

Schedule Representation

Step	Operation
1	T1 : R(X)
2	T1 : W(X)
3	T2 : R(X)
4	T2 : W(X)
5	T1 : Commit
6	T2 : Commit

Conflicting Operations

- W1(X) → R2(X) ⇒ T1 → T2
- W1(X) → W2(X) ⇒ T1 → T2

- R1(X) → W2(X) ⇒ T1 → T2
- **Precedence Graph**
- T1 → T2

Since there is no cycle in the graph, the schedule is Conflict Serializable. Equivalent

Serial Schedule

T1 → T2

The schedule is serializable because the precedence graph contains no cycles. It is equivalent to the serial execution T1 followed by T2.

(ii) Schedule

T1 : W(X) R(Y)
T2 : R(X) W(Y)
T1 : COMMIT T2
: COMMIT

Schedule Representation

Step	Operation
1	T1 : W(X)
2	T1 : R(Y)
3	T2 : R(X)
4	T2 : W(Y)
5	T1 : Commit
6	T2 : Commit

Conflicting Operations

1. W1(X) and R2(X)

T1 → T2

2. R1(Y) and W2(Y)

T1 → T2

Precedence Graph

T1 → T2

Since there is no cycle, the schedule is Conflict Serializable. Equivalent

Serial Schedule

T1 → T2

The schedule is serializable because the precedence graph has no cycle. It is equivalent to the serial schedule T1 followed by T2

c.	Consider the tables below: Sailors (sid : integer, sname : string, rating : integer, age : real) Boats (bid : integer, bname : string, color : string); Reserves (sid : integer, bid : integer, day : date) Write SQL queries for the following: (i) Write create table statement for reserves. (ii) Find all information of sailors who have reserved boat number 101. (iii) Find the names of sailors who have reserved at least one boat. (iv) Find the names of sailors who have reserved a red boat. (v) Find the average age of sailors for each rating level.	10	L3	CO5

(i)

```
CREATE TABLE Reserves
(
   sid INTEGER,
   bid INTEGER,
   day DATE,
   PRIMARY KEY (sid, bid, day),
   FOREIGN KEY (sid) REFERENCES Sailors(sid),
   FOREIGN KEY (bid) REFERENCES Boats(bid)
);
```

(ii)

```
SELECT S.*
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
AND R.bid = 101;
```

(iii)

```
SELECT DISTINCT S.sname

FROM Sailors S, Reserves R

WHERE S.sid = R.sid;
```

(iv)

```
SELECT DISTINCT S.sname

FROM Sailors S, Boats B, Reserves R

WHERE S.sid = R.sid

AND R.bid = B.bid

AND B.color = 'red';
```

(v)

```
SELECT rating, AVG(age)

FROM Sailors

GROUP BY rating;
```

Q.9	**a.**	Explain the CAP theorem.	**05**	**L2**	**CO6**
	b.	What is NOSQL graph database? Explain Neo4j.	**05**	**L2**	**CO6**
	c.	Why concurrency control and recovery are needed in DBMS? Demonstrate with suitable examples types of problems that may occur when two simple transactions run concurrently.	**10**	**L3**	**CO5**

Q. 9. a. Explain the CAP theorem (5M)

The **CAP Theorem**, also known as **Brewer's Theorem**, states that a distributed database system can provide at most two out of the following three properties simultaneously:

1. **Consistency (C)**
2. **Availability (A)**
3. **Partition Tolerance (P)**

Thus, it is impossible for a distributed system to guarantee all three properties at the same time. When a network partition occurs, the system must choose between consistency and availability.

Components of CAP Theorem

1. Consistency (C)

Consistency ensures that all nodes in the distributed system see the same data at the same time. Whenever data is updated, all users receive the most recent value.

Example:
If a user updates his account balance, every node in the system should immediately reflect the updated balance.

2. Availability (A)

Availability guarantees that every request receives a response, even if some nodes in the system have failed. The system remains operational and accessible to users.

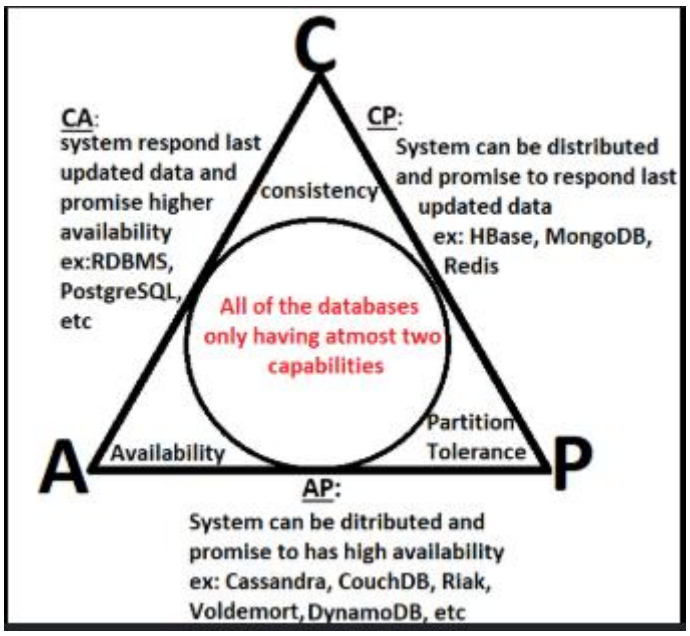
Example:
A web application should continue serving user requests even if one of its servers becomes unavailable.

3. Partition Tolerance (P)

Partition tolerance means that the system continues to function despite communication failures or network partitions between nodes.

Example:
If communication between two data centers is interrupted, the database should continue operating without complete failure.

Diagram of CAP Theorem



Possible Combinations

1. CP Systems (Consistency + Partition Tolerance)

- Maintain data consistency.
- Continue functioning during network partitions.
- May sacrifice availability.

Example: HBase, MongoDB (in certain configurations).

2. AP Systems (Availability + Partition Tolerance)

- Always provide responses to users.
- Continue operating during partitions.
- May temporarily sacrifice consistency.

Example: Cassandra, DynamoDB.

3. CA Systems (Consistency + Availability)

- Provide consistent and available data.
- Cannot tolerate network partitions.
- Suitable mainly for centralized databases.

Example: Traditional Relational Database Management Systems.

Q.9.b. What is NOSQL graph database? Explain Neo4j (5M)

A **NoSQL graph database** is a type of non-relational database that represents data in the form of **nodes, relationships, and properties**. Unlike relational databases, which store data in tables and connect them using foreign keys, graph databases directly store relationships between data items, making them highly efficient for handling interconnected data.

Graph databases are particularly suitable for applications involving complex relationships, such as social networks, recommendation systems, fraud detection, and knowledge graphs.

Components of a Graph Database

1. **Nodes**
   - Represent entities or objects.
   - Examples: Person, Product, Movie.
2. **Relationships (Edges)**
   - Represent connections between nodes.
   - Examples: FRIENDS_WITH, PURCHASED, ACTED_IN.
3. **Properties**
   - Store information about nodes and relationships in the form of key-value pairs.
   - Example: Name = "Ravi", Age = 22.

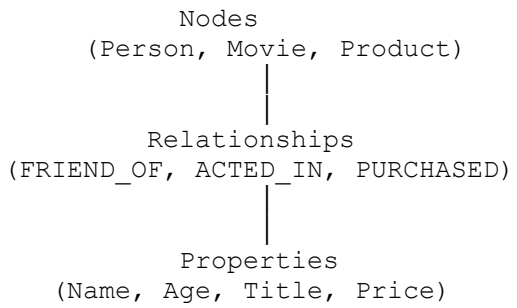
Advantages of Graph Databases

- Efficient handling of highly connected data.
- Faster traversal and querying of relationships.
- Flexible schema.
- High scalability and performance.
- Suitable for real-time applications.

Neo4j

Neo4j is an open-source, native graph database management system developed using Java. It stores data as nodes and relationships and uses the **property graph model** to represent interconnected data. Neo4j is one of the most widely used graph databases in NoSQL systems.

Architecture of Neo4j



Features of Neo4j

1. Property Graph Model

Neo4j stores data using nodes, relationships, labels, and properties, enabling efficient representation of connected data.

2. Schema Flexibility

It supports a dynamic schema, allowing data structures to evolve without requiring major modifications.

3. ACID Compliance

Neo4j supports **Atomicity, Consistency, Isolation, and Durability**, ensuring reliable transaction processing.

4. High Performance

Relationship traversal is very fast because connections are stored directly rather than computed through joins.

5. Cypher Query Language

Neo4j uses **Cypher**, a declarative query language designed for creating and querying graph structures.

Example:

```
CREATE (s:Student {name:'Ravi', age:22})
```

Applications of Neo4j

- Social networking sites.
- Recommendation engines.
- Fraud detection systems.
- Network and IT operations.
- Knowledge graphs.
- Route planning and logistics.
- Bioinformatics and healthcare applications.

Advantages of Neo4j

- Efficient storage and traversal of connected data.
- Reduced complexity compared to relational joins.
- Flexible schema design.
- Supports ACID transactions.
- Provides powerful graph query capabilities.

Q.9.c. Why concurrency control and recovery are needed in DBMS? Demonstrate with suitable examples types of problems that may occur when two simple transactions run concurrently (10M)

Need for Concurrency Control and Recovery in DBMS

In a database system, several users may access and modify the database simultaneously. Such simultaneous execution of transactions is called **concurrent execution**. Although concurrency improves system performance and resource utilization, improper coordination among transactions may lead to inconsistent and incorrect results. Therefore, concurrency control and recovery mechanisms are essential components of a DBMS.

Need for Concurrency Control

Concurrency control is the activity of coordinating the simultaneous execution of transactions in a multi-user database system so that the consistency and integrity of the database are preserved.

Concurrency control is required to:

- Maintain database consistency.
- Ensure isolation among transactions.
- Prevent interference among concurrently executing transactions.
- Avoid anomalies caused by simultaneous updates.
- Preserve data integrity.
- Ensure serializability of transaction schedules.
- Improve system throughput and resource utilization.
- Guarantee the ACID properties of transactions.

Need for Recovery

Recovery refers to the techniques used to restore the database to a consistent state after failures such as hardware failures, software errors, power failures, or transaction aborts.

Recovery is necessary because:

- Transactions may fail before completion.
- System crashes may occur unexpectedly.
- Hardware and software failures may corrupt data.
- Atomicity and durability properties must be preserved.
- The database must be restored to a consistent state.
- Committed transactions should not lose their effects.
- Uncommitted transactions should be rolled back.

Thus, concurrency control and recovery together ensure reliable and correct operation of the database system.

Problems Due to Concurrent Execution of Transactions

Consider two transactions:

Transaction T1

Transfer ₹100 from Account A to Account B.

```
Read(A)
A = A − 100
Write(A)

Read(B)
B = B + 100
Write(B)
```

Transaction T2

Add 10% interest to Account A.

```
Read(A)
A = A × 1.1
Write(A)
```

Suppose initially

```
A = 1000
B = 2000
```

When these transactions execute simultaneously, the following anomalies may occur.

1. Lost Update Problem

The lost update problem occurs when two transactions simultaneously update the same data item and one update overwrites the other. As a result, one transaction's modification is lost.

This anomaly arises because both transactions read the same old value before either of them writes its updated value.

Example

Initial value:

```
A = 1000
```

T1	T2
Read(A=1000)	
	Read(A=1000)
A=A+100=1100	
	A=A×1.1=1100
Write(A=1100)	
	Write(A=1100)

Final value:

```
A = 1100
```

However, if executed serially, the correct answer should be:

```
(1000+100)×1.1 = 1210
```

Therefore, the update performed by T1 is overwritten by T2 and is lost.

Consequences

- Data inconsistency.
- Incorrect database state.
- Violation of transaction isolation.

2. Dirty Read Problem (Temporary Update Problem)

A dirty read occurs when one transaction reads data written by another transaction before it commits. If the transaction that performed the update later aborts, the first transaction will have used invalid data.

Dirty reads involve reading uncommitted values.

Example

Suppose

```
A = 1000
```

T1	T2
Read(A=1000)	
A=A+500=1500	
Write(A=1500)	
	Read(A=1500)
Abort	
	Uses 1500

Since T1 aborts, value 1500 becomes invalid.

T2 has already used the uncommitted value, producing incorrect results.

Consequences

- Inconsistent computations.
- Cascading rollback.
- Reduced reliability.

3. Unrepeatable Read Problem

An unrepeatable read occurs when a transaction reads the same data item more than once and obtains different values because another transaction modifies the data between the two reads.

Thus, repeated reads produce different results.

Example

Initial value:

```
A = 1000
```

T1	T2
Read(A=1000)	
	Read(A=1000)
	A=A+200=1200
	Write(A=1200)
	Commit
Read(A=1200)	

Transaction T1 first reads

```
A=1000
```

and later reads

```
A=1200
```

Thus, the same transaction obtains two different values for the same data item.

Consequences

- Inconsistent reports.
- Incorrect calculations.
- Lack of repeatability.

4. Incorrect Summary Problem

The incorrect summary problem occurs when one transaction computes aggregate values while another transaction simultaneously updates some of the values involved in the computation.

As a result, the aggregate calculation becomes incorrect.

Example

Suppose

```
A = 1000
B = 2000
```

Transaction T1 calculates

```
SUM = A + B
```

Transaction T2 transfers ₹500 from A to B.

T1	T2
Read(A=1000)	
	Read(A=1000)
	A=500
	Write(A=500)
	Read(B=2000)
	B=2500
	Write(B=2500)
Read(B=2500)	

T1 computes:

```
SUM = 1000 + 2500 = 3500
```

But the actual total balance is

```
500 + 2500 = 3000
```

Hence, an incorrect summary is obtained.

Consequences

- Wrong statistical information.
- Incorrect reports and analysis.
- Violation of consistency.

5. Phantom Read Problem

A phantom read occurs when a transaction re-executes a query and finds additional tuples inserted or deleted by another transaction.

The rows returned by the same query differ between executions.

Example

Suppose Transaction T1 executes

```
SELECT * FROM EMPLOYEE
WHERE Salary > 50000;
```

and obtains five records.

Before T1 executes the query again, Transaction T2 inserts a new employee with salary 60000 and commits.

When T1 repeats the query, six records are returned.

The newly appearing row is called a **phantom tuple**, and the phenomenon is known as a phantom read.

Consequences

- Inconsistent query results.
- Unreliable reporting.

Q.10	**a.**	Explain basic operations CRUD in MongoDB. BANGALORE - 560 037
	b.	Explain deadlock prevention protocols.
	c.	Briefly discuss the two-phase looking techniques f_0 concurrency control.

Q.10 . a. Explain basic operations CRUD in MongoDB. (5M)

CRUD stands for **Create, Read, Update, and Delete**. These are the fundamental operations used to manipulate documents in MongoDB. MongoDB is a **NoSQL document-oriented database** that stores data in the form of **BSON (Binary JSON) documents** inside collections. CRUD operations provide mechanisms to insert, retrieve, modify, and remove documents efficiently.

1. Create Operation

The **Create** operation is used to add new documents into a collection. If the collection does not exist, MongoDB automatically creates it when the first document is inserted.

Operations Used

- `insertOne()`
- `insertMany()`

Syntax

```
db.collection_name.insertOne(document)
```

Example

```
db.Student.insertOne(
{
sid:101,
name:"Ravi",
branch:"CSE",
age:21
}
)
```

Working

- A new document is inserted into the collection.
- MongoDB automatically generates a unique `_id` field.
- Multiple documents can be inserted simultaneously using `insertMany()`.

2. Read Operation

Definition

The **Read** operation is used to retrieve documents stored in a collection. MongoDB provides various methods to query data based on specified conditions.

Operations Used

- `find()`
- `findOne()`

Syntax

```
db.collection_name.find(condition)
```

Example

Display all students:

```
db.Student.find()
```

Display students belonging to CSE branch:

```
db.Student.find(
{
branch:"CSE"
```

```
}
)
```

Working

- Searches the collection for matching documents.
- Returns all documents satisfying the specified condition.
- If no condition is given, all documents are returned.

3. Update Operation

Definition

The **Update** operation modifies existing documents in a collection. It allows changing one or more fields of selected documents.

Operations Used

- `updateOne()`
- `updateMany()`

Syntax

```
db.collection_name.updateOne(
condition,
{$set:{field:value}}
)
```

Example

Update age of student 101:

```
db.Student.updateOne(
{
sid:101
},
{
$set:{age:22}
}
)
```

Working

- Finds the document matching the condition.
- Modifies only the specified fields.
- Other fields remain unchanged.

4. Delete Operation

Definition

The **Delete** operation removes documents from a collection. MongoDB provides methods to delete one or multiple documents.

Operations Used

- `deleteOne()`
- `deleteMany()`

Syntax

```
db.collection_name.deleteOne(condition)
```

Example

Delete student whose SID is 101:

```
db.Student.deleteOne(
{
sid:101
}
)
```

Working

- Searches for documents satisfying the condition.
- Removes the selected documents permanently.
- `deleteMany()` removes all matching documents.

Advantages of CRUD Operations in MongoDB

- Provide simple and efficient data manipulation.
- Support flexible schema design.
- Handle large volumes of data efficiently.
- Enable fast retrieval and modification of documents.
- Suitable for scalable and distributed applications.

Q.10. b. Explain deadlock prevention protocols (5M)

A **deadlock** is a situation in which two or more transactions are waiting indefinitely for one another to release locks on data items, and hence none of the transactions can proceed further.

Deadlocks may occur in lock-based concurrency control mechanisms when transactions hold some locks and simultaneously request additional locks held by other transactions.

For example, if transaction **T1** holds a lock on data item **A** and requests a lock on **B**, while transaction **T2** holds a lock on **B** and requests a lock on **A**, both transactions will wait indefinitely, resulting in a deadlock.

To avoid such situations, several **deadlock prevention protocols** are employed.

Deadlock Prevention Protocols

Deadlock prevention protocols ensure that the system never enters a deadlock state by imposing restrictions on how transactions acquire locks. These protocols eliminate one or more of the necessary conditions for deadlock.

The important deadlock prevention protocols are:

1. Wait-Die Scheme
2. Wound-Wait Scheme
3. No-Wait Scheme
4. Cautious Waiting Scheme
5. Timeout-Based Scheme

1. Wait-Die Scheme

The **Wait-Die scheme** is a non-preemptive deadlock prevention protocol based on timestamps. Each transaction is assigned a unique timestamp when it begins execution. Older transactions are given smaller timestamps.

According to this protocol:

- If an older transaction requests a resource held by a younger transaction, the older transaction is allowed to wait.
- If a younger transaction requests a resource held by an older transaction, the younger transaction is rolled back (dies) and restarted later with the same timestamp.

Thus, waiting is allowed only from older transactions to younger transactions.

Example

Suppose:

- T1 has timestamp 5 (older)
- T2 has timestamp 10 (younger)

Case 1:

T1 requests a lock held by T2.

Since T1 is older,

```
T1 waits.
```

Case 2:

T2 requests a lock held by T1.

Since T2 is younger,

```
T2 dies (rollback).
```

Advantages

- Prevents deadlocks.
- Simple to implement.

Disadvantages

- Younger transactions may suffer repeated rollbacks.
- Possibility of starvation.

2. Wound-Wait Scheme

The **Wound-Wait scheme** is a preemptive deadlock prevention protocol based on timestamps.

According to this protocol:

- If an older transaction requests a resource held by a younger transaction, the older transaction wounds the younger transaction. The younger transaction is rolled back and the older transaction acquires the lock.
- If a younger transaction requests a resource held by an older transaction, the younger transaction waits.

Thus, waiting is allowed only from younger transactions to older transactions.

Example

Suppose:

- T1 has timestamp 5 (older)
- T2 has timestamp 10 (younger)

Case 1:

T1 requests a lock held by T2.

Since T1 is older,

```
T2 is rolled back.
T1 acquires the lock.
```

Case 2:

T2 requests a lock held by T1.

Since T2 is younger,

```
T2 waits.
```

Advantages

- Prevents deadlocks effectively.
- Fewer rollbacks compared to Wait-Die scheme.

Disadvantages

- Requires transaction rollback.
- Additional overhead for restarting transactions.

3. No-Wait Scheme

In the **No-Wait scheme**, a transaction is never allowed to wait for a lock.

If a requested lock is unavailable, the transaction is immediately rolled back and restarted later.

Example

Suppose transaction T1 holds a lock on data item A.

If T2 requests the same lock,

```
T2 is immediately aborted.
```

It is restarted later.

Advantages

- Deadlocks are completely eliminated.
- Easy to implement.

Disadvantages

- High number of rollbacks.
- Reduced system throughput.

4. Cautious Waiting Scheme

In the **Cautious Waiting protocol**, a transaction requesting a lock waits only if the transaction holding the lock is not itself waiting for another lock.

If the lock-holding transaction is already waiting, the requesting transaction is rolled back.

This prevents circular waiting and thereby avoids deadlocks.

Example

Suppose:

- T1 holds a lock and is not waiting for any resource.
- T2 requests that lock.

Then:

```
T2 waits.
```

However, if T1 itself is waiting for another resource,

```
T2 is aborted.
```

Advantages

- Reduces unnecessary rollbacks.
- Prevents circular wait.

Disadvantages

- More complex than other schemes.

5. Timeout-Based Scheme

Definition

In the timeout-based scheme, a transaction is allowed to wait only for a specified period. If the waiting time exceeds the predefined limit, the transaction is rolled back and restarted.

Example

Suppose T1 waits for 30 seconds for a lock held by T2.

If the lock is not released within that time,

```
T1 is aborted and restarted.
```

Advantages

- Simple to implement.
- Suitable for practical systems.

Disadvantages

- Choosing an appropriate timeout value is difficult.
- May result in unnecessary rollbacks.

Q.10.c. Briefly Discuss the Two-Phase Locking Technique for Concurrency Control (10M)

Two-Phase Locking (2PL) is a lock-based concurrency control protocol used to ensure the serializability and consistency of transactions in a database system. In this protocol, a transaction must acquire appropriate locks before accessing data items and release locks according to specific rules.

The name **Two-Phase Locking** arises because the execution of every transaction is divided into two distinct phases:

1. **Growing Phase**
2. **Shrinking Phase**

By following these two phases, transactions can execute concurrently without violating database consistency.

Need for Two-Phase Locking

Two-phase locking is used to:

- Ensure serializability of concurrent transactions.
- Maintain database consistency.
- Prevent lost updates and dirty reads.
- Coordinate simultaneous access to shared data.
- Enforce isolation among transactions.

Phases of Two-Phase Locking

1. Growing Phase

Definition

The growing phase is the phase during which a transaction acquires locks on data items but is not allowed to release any lock.

In this phase:

- New locks can be acquired.
- No lock can be released.

The transaction continues to obtain all required locks until it reaches a point called the **lock point**.

Example

Transaction T1:

```
Lock-X(A)
Read(A)
Write(A)

Lock-X(B)
Read(B)
Write(B)
```

During this period, T1 acquires locks on A and B but does not release any lock.

Characteristics

- Only lock acquisition is allowed.
- Lock release is prohibited.
- Ends when the transaction obtains its final lock.

2. Shrinking Phase

Definition

The shrinking phase begins after the transaction releases its first lock. During this phase, the transaction releases locks but is not allowed to acquire any new lock.

In this phase:

- Locks are released.
- No new lock can be acquired.

Example

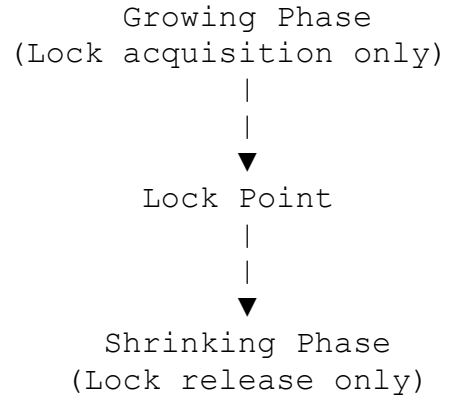
```
Unlock(A)
Unlock(B)
```

Once a transaction starts releasing locks, it enters the shrinking phase and cannot request additional locks.

Characteristics

- Only lock release is allowed.
- Acquisition of new locks is prohibited.
- Continues until all locks are released.

Diagram of Two-Phase Locking



Example of Two-Phase Locking

Consider transaction T1:

```
Lock-X(A)
Read(A)
Write(A)

Lock-X(B)
Read(B)
Write(B)

Unlock(B)
Unlock(A)
```

Here,

- Acquiring locks on A and B represents the **Growing Phase**.
- Releasing locks on B and A represents the **Shrinking Phase**.

Hence, the transaction follows the Two-Phase Locking protocol.

Types of Two-Phase Locking

1. Basic Two-Phase Locking

In Basic 2PL:

- Transactions follow growing and shrinking phases.
- Locks may be released before the transaction commits.
- Guarantees conflict serializability.

Disadvantage

- Deadlocks may occur.

2. Conservative (Static) Two-Phase Locking

In Conservative 2PL:

- A transaction acquires all required locks before execution begins.
- If all locks are unavailable, the transaction waits.

Advantages

- Eliminates deadlocks.

Disadvantages

- Reduced concurrency.
- Requires prior knowledge of all required data items.

3. Strict Two-Phase Locking

In Strict 2PL:

- Exclusive locks are held until the transaction commits or aborts.
- Locks are released only after completion of the transaction.

Advantages

- Prevents cascading rollbacks.
- Ensures recoverable schedules.
- Widely used in commercial DBMSs.

Disadvantages

- Deadlocks are still possible.

4. Rigorous Two-Phase Locking

In Rigorous 2PL:

- Both shared and exclusive locks are retained until the transaction commits or aborts.

Advantages

- Produces strict and serializable schedules.
- Simplifies recovery mechanisms.

Disadvantages

- Reduces concurrency.

Advantages of Two-Phase Locking

- Ensures serializability.
- Preserves database consistency.
- Supports concurrent execution of transactions.
- Prevents anomalies such as lost updates and dirty reads.
- Simple and widely implemented.

Disadvantages of Two-Phase Locking

- May lead to deadlocks.
- Transactions may experience waiting delays.
- Lock management introduces overhead.
- Reduced concurrency in strict forms of 2PL.