CBCS SCHEME

USN 1CR24MC097

MMC103

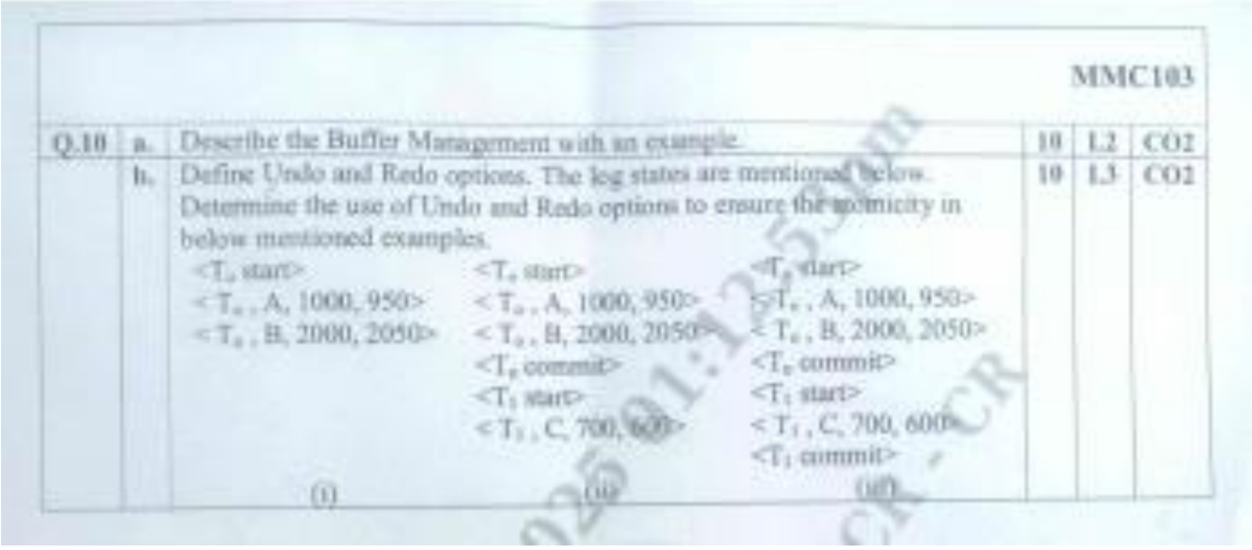
First Semester MCA Degree Examination, Dec.2024/Jan.2025
Database Management Systems (DBMS)

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. DBMS ii. Entity & Attribute iii. Relational data model iv. Schema and Schema Diagram v. Primary key and Foreign key	10	L1	CO1
	b.	Discuss the different applications of DBMS.	5	L2	CO1
	c.	Explain three schema architecture with neat diagram.	5	L2	CO1
		OR			
Q.2	a.	Explain components of DBMS.	10	L2	CO1
	b.	Discuss the different types of Relationship types.	5	L2	CO1
	c.	Draw ER-Diagram for Company database which contains entity type Employee, Department, Project and Dependent.	5	L3	CO1
		Module – 2			
Q.3	a.	Explain the following relational algebra operations i. Selection ii. Projection	10	L2	CO2
	b.	Describe the following DDL and DML commands i. Create ii. Insert iii. Delete iv. Update v. Drop	10	L2	CO2
		OR			
Q.4	a.	Explain the following clauses i. select ...From... Where clause ii. Group by and Having clause	10	L2	CO3
	b.	Elaborate the importance of views.	5	L2	CO3
	c.	Discuss about Procedures.	5	L2	CO3
		Module – 3			
Q.5	a.	Explain 1NF and 2NF with an example.	10	L2	CO3
	b.	Discuss 3NF and Boyce codd with an example.	10	L2	CO3
		OR			
Q.6	a.	Explain 4NF and 5NF with an example.	10	L2	CO3
	b.	Discuss the following with an Example. i. Functional dependency ii. Dependency Preservation Property	10	L2	CO3
		Module – 4			
Q.7	a.	Describe the following i. ACID Properties of transaction ii. Different states for transaction execution	10	L2	CO2
	b.	Discuss two-phase locking system with an example.	10	L2	CO2
		OR			
Q.8	a.	Explain the implementation of Isolation level.	10	L2	CO2
	b.	Discuss Multiple Granularity with an example.	10	L2	CO2
		Module – 5			
Q.9	a.	How the log can be used to recover from a system crash and to roll back transactions during normal operation?	10	L3	CO2
	b.	Illustrate Checkpoints and Fuzzy Check pointing with an example.	10	L3	CO2
		OR			



MMC103

Q.10	a.	Describe the Buffer Management with an example.	10	L2	CO2
	b.	Define Undo and Redo options. The log states are mentioned below. Determine the use of Undo and Redo options to ensure the atomicity in below mentioned examples.	10	L3	CO2

(i)	(ii)	(iii)
<T_0 start>	<T_0 start>	<T_0 start>
< T_0 , A, 1000, 950>	< T_0 , A, 1000, 950>	< T_0 , A, 1000, 950>
< T_0 , B, 2000, 2050>	< T_0 , B, 2000, 2050>	< T_0 , B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	< T_1 , C, 700, 600>	< T_1 , C, 700, 600>
		<T_1 commit>

Solution:

1. a) i. **DBMS** stands for **Database Management System**. It is software that is used to create, manage, and manipulate databases. A DBMS provides an interface between users and the database, enabling users to store, retrieve, update, and manage data efficiently and securely.
ii. An **entity** is a real-world object or concept that can be distinctly identified and stored in a database. It can be a **person, place, thing, or event** about which data is collected.
Example: In a student database, a **Student** is an entity.
An **attribute** is a **property or characteristic** of an entity. It defines the information that is to be stored about the entity.
Example: For the entity **Student**, possible attributes are: StudentID, Name, Age, Course,
iii. The **Relational Data Model** is a type of data model used in relational databases, where **data is organized into tables (called relations)**. Each table consists of **rows (tuples)** and **columns (attributes)**.
iv. A **schema diagram** is a visual representation of the database schema. It shows:
Tables (entities), Attributes (columns), Primary keys and foreign keys, Relationships between tables. Example:

Students	Courses	Enrollments
---------	--------	-----------
StudentID (PK)	CourseID (PK)	EnrollmentID (PK)
Name	CourseName	StudentID (FK)
Age		CourseID (FK)

v. A **Primary Key** is a **unique identifier** for each record (row) in a database table. It ensures that no two rows can have the same value for the primary key and that the value is **never NULL**.

- **Purpose:** To uniquely identify each record.
- **Rules:**
    - Must be unique.
    - Cannot be NULL.
    - A table can have only **one** primary key (which can be made up of one or more columns — called a composite key).

Example: StudentID in Student Table

A **Foreign Key** is a field (or a set of fields) in one table that refers to the **primary key** in another table. It is used to **establish and enforce a link** between the data in two tables.

- **Purpose:** To maintain referential integrity between related tables.
- **Can have duplicate and NULL values** (unlike primary keys).
  Example: StudentID in Course Table.

b) **1. Banking and Finance**

- **Use:** Managing customer accounts, transactions, loans, and financial records.
- **Example:** ATM transaction management, online banking, fraud detection.

---

2. Education

- **Use:** Storing student information, course registrations, results, attendance, and faculty records.
- **Example:** University student portals, exam result systems.

---

3. Healthcare

- **Use:** Managing patient records, doctor schedules, lab reports, billing, and prescriptions.
- **Example:** Electronic Medical Records (EMRs), hospital management systems.

---

4. Retail and E-commerce

- **Use:** Managing inventory, sales, customer orders, and supplier data.
- **Example:** Online shopping platforms like Amazon use DBMS for order processing and product tracking.

---

5. Telecommunications

- **Use:** Handling call records, customer data, billing, and service management.
- **Example:** Mobile network providers use DBMS for storing call logs and user plans.

---

6. Government

- **Use:** Storing citizen data, tax records, identity management, and public services.
- **Example:** National ID databases, passport systems, vehicle registration.

---

7. Airlines and Railways

- **Use:** Reservation systems, schedules, passenger data, and ticketing.
- **Example:** Online flight and train booking systems.

---

8. Social Media Platforms

- **Use:** Managing user profiles, posts, messages, and friend relationships.
- **Example:** Facebook, Instagram use large-scale DBMS to manage massive data volumes.

---

9. Manufacturing

- **Use:** Managing supply chains, inventory, production schedules, and quality control.
- **Example:** ERP systems in factories.

---

10. Scientific Research

- **Use:** Storing and analyzing experimental data, simulations, and results.
- **Example:** Weather forecasting, genome databases.

c) The three-schema architecture is a framework for database management systems (DBMS) that separates the database into three different levels: internal, conceptual, and external. This separation helps in abstraction, security, and flexibility in managing databases.

Three Schema Architecture

Internal Schema (Physical Level)

- It defines the physical storage structure of the database.
- Concerned with data storage, indexing, and optimization.

Example: Storing customer data as binary files on disk.

Conceptual Schema (Logical Level)

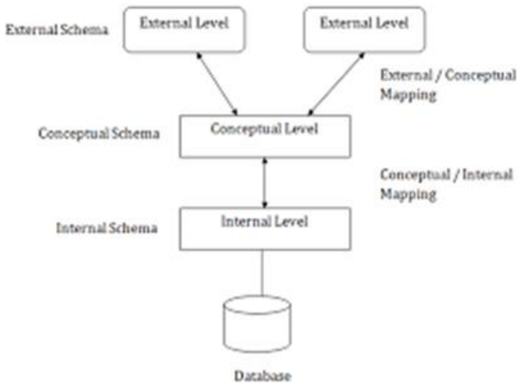
- It provides a unified logical view of the entire database.
- Defines tables, relationships, constraints, and security rules.

Example: A relational model defining tables like Customers (ID, Name, Address).

External Schema (View Level)

- It defines different views of the database for users or applications.
- Ensures data security by restricting access to only necessary data.

Example: A bank's customer portal only shows account details but not backend transaction logs.



Importance of Each Schema Layer

Internal Schema (Storage Optimization & Performance)

- Ensures efficient data storage and retrieval.

Example: Uses indexing to speed up search queries in an e-commerce database.

Conceptual Schema (Logical Data Independence)

- Helps in defining and maintaining database integrity.

Example: If a new attribute Email is added to Customers, applications remain unaffected.

External Schema (Security & Customization)

- Enables different user views without exposing unnecessary data.

Example: A sales department sees only Customer_Name, Purchase_History, while HR sees Employee_ID, Salary.

This architecture provides data abstraction, security, and independence, making it a vital design principle in DBMS.

2. A) A DBMS is made up of several key components that work together to store, manage, and retrieve data efficiently. Here's a breakdown of the **main components**:

1. Database Engine

Role: The core service for accessing and processing data.

Function: Handles storage, retrieval, and update of data.

Acts as: The interface between low-level data and higher-level DBMS operations.

2. Data Definition Language (DDL) Compiler

Role: Processes DDL commands (like CREATE, ALTER, DROP).

Function: Defines the structure of the database schema (tables, fields, constraints).

Stores results in: The **data dictionary** (also called system catalog).

3. Data Manipulation Language (DML) Compiler

Role: Interprets DML queries (like SELECT, INSERT, UPDATE, DELETE).

Function: Converts high-level queries into low-level instructions the DBMS engine can execute.

4. Query Processor

Role: Interprets and optimizes SQL queries.

Function: Ensures queries are executed efficiently by creating the best execution plan.

5. Transaction Manager

Role: Manages transactions (a group of operations).

Function: Ensures **ACID properties** (Atomicity, Consistency, Isolation, Durability) are maintained.

Protects against: Data loss, system crashes, or concurrent access conflicts.

6. Storage Manager

Role: Manages how data is stored on physical devices (e.g., hard disks).

Function: Handles file management, buffering, indexing, and data access methods.

7. Buffer Manager (Cache Manager)

Role: Handles memory buffers for data being read/written.

Function: Minimizes disk I/O by storing frequently accessed data in main memory.

8. Authorization and Integrity Manager

Role: Controls access and enforces security rules.

Function: Manages user permissions, roles, and checks data integrity constraints.

9. Data Dictionary (System Catalog)

Role: A centralized repository of metadata (data about data).

Function: Stores information like table definitions, constraints, user roles, etc.

2B) **1. One-to-One (1:1) Relationship**

Definition: A single record in **Table A** is related to **only one** record in **Table B**, and vice versa.

Example:

One person has **one passport**.

Person(ID, Name)

Passport(ID, PassportNumber, PersonID)

Use Case: When data is split across tables for privacy or security.

2. One-to-Many (1:N) Relationship

Definition: A single record in **Table A** can be related to **many** records in **Table B**, but each record in **Table B** relates to **only one** in **Table A**.

Example:

One teacher teaches **many students**.

Teacher(TeacherID, Name)

Student(StudentID, Name, TeacherID)

Most common type of relationship in relational databases.

3. Many-to-Many (M:N) Relationship

Definition: Multiple records in **Table A** can relate to **multiple records** in **Table B**.

Example:

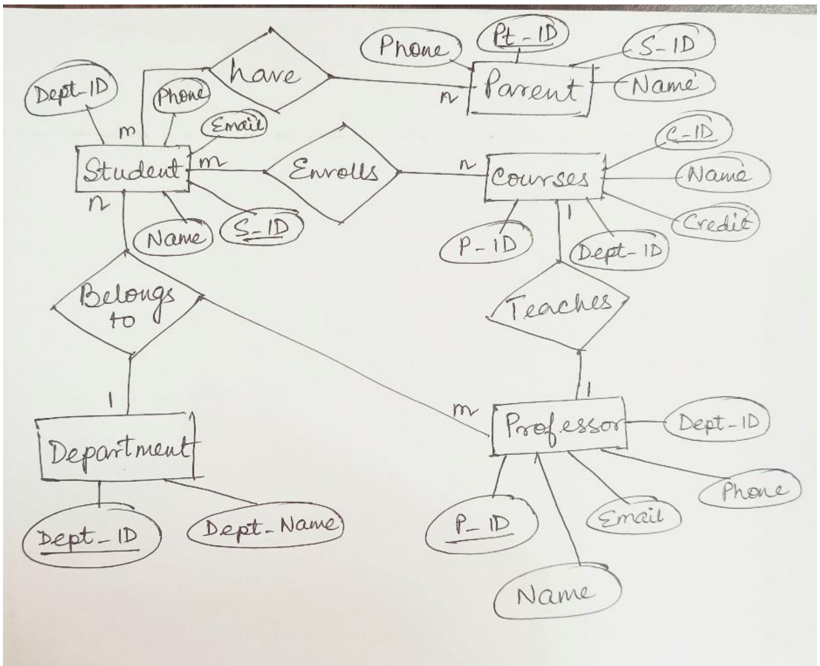
Students can enroll in **many courses**, and each course can have **many students**.

Student(StudentID, Name)

Course(CourseID, CourseName)

Junction Table: Enrollment(StudentID, CourseID)

Note: This relationship is implemented using a **third table** (junction or associative table) to link both sides.

2C) 

3a) **1. Selection (σ)**

Definition: The Selection operation retrieves rows (tuples) from a relation (table) that satisfy a specified condition.

Notation: σ<condition>(Relation)

Purpose: Filters rows based on a condition.

Let Employee be a relation:

EmpID	Name	Age	Dept
101	Alice	25	HR
102	Bob	30	IT
103	Carol	22	HR

Query: Get all employees from the HR department.

Relational Algebra:

σ Dept = 'HR' (Employee)

Result:

EmpID	Name	Age	Dept
101	Alice	25	HR
103	Carol	22	HR

2. Projection (π)

Definition: The **Projection** operation retrieves **specific columns** (attributes) from a relation, removing duplicates.

Notation: π<attribute list>(Relation)

Purpose: Filters **columns**, keeping only the specified ones.

Example: Query: Get the names and departments of all employees.

Relational Algebra: π Name, Dept (Employee)

Result:

Name	Dept
Alice	HR
Bob	IT
Carol	HR

3b) **1. DDL Commands (Data Definition Language)**

These commands are used to **define or modify the structure** of database objects such as **tables, schemas, indexes, and views**.

Common DDL Commands:

Command	Description
CREATE	Creates a new table, view, database, or other database object.
ALTER	Modifies an existing table structure (e.g., adding or deleting a column).
DROP	Deletes an existing database object like a table or database permanently.
TRUNCATE	Removes all records from a table, but keeps the structure. Faster than DELETE.
RENAME	Changes the name of a database object (e.g., a table).

Example: CREATE TABLE Employee (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Age INT);

2. DML Commands (Data Manipulation Language)

These commands are used to **manipulate the data** stored in database tables.

Common DML Commands:

Command	Description
SELECT	Retrieves data from one or more tables.
INSERT	Adds new data (rows) into a table.
UPDATE	Modifies existing data in a table.
DELETE	Removes specific rows from a table.

Example:

```
INSERT INTO Employee (EmpID, Name, Age)

VALUES (101, 'Alice', 25);

SELECT * FROM Employee;

UPDATE Employee

SET Age = 26

WHERE EmpID = 101;

DELETE FROM Employee

WHERE EmpID = 101;
```

Summary Table:

Category	Commands	Purpose
DDL	CREATE, ALTER, DROP, TRUNCATE, RENAME	Define or change database structure
DML	SELECT, INSERT, UPDATE, DELETE	Add, view, or change data

4a) **i. SELECT ... FROM ... WHERE**

This is the **basic structure of an SQL query** used to **retrieve data** from a table, with an optional condition to **filter** the rows.

Syntax:

```
SELECT column1, column2, ...

FROM table_name

WHERE condition;
```

Purpose:

- SELECT: Specifies which columns to retrieve.
- FROM: Specifies the table.
- WHERE: Filters rows based on a condition.

Example:

SELECT Name, Age

FROM Employee

WHERE Age > 25;

This will retrieve the names and ages of employees who are older than 25.

ii. GROUP BY and HAVING

These clauses are used together to **group rows** based on column values and then **filter groups** based on a condition.

GROUP BY

- Used to group rows that have the same values in specified columns.
- Often used with **aggregate functions** like SUM(), AVG(), COUNT(), MAX(), etc.

HAVING

- Used to filter groups (not individual rows).
- Similar to WHERE, but **applied after grouping**.

Syntax:

SELECT column, AGG_FUNC(column)

FROM table

GROUP BY column

HAVING condition;

Example:

SELECT Dept, COUNT(EmpID) AS EmployeeCount

FROM Employee

GROUP BY Dept

HAVING COUNT(EmpID) > 2;

This query:

- Groups employees by department,
- Counts employees in each department,
- Returns only those departments that have **more than 2 employees**.

Summary Table:

Clause	Applies To	Purpose
WHERE	Rows	Filters individual records
GROUP BY	Groups of rows	Groups data based on column(s)
HAVING	Groups (after GROUP BY)	Filters groups after aggregation

b) A **view** is a **virtual table** in a database. It does **not store data physically** but presents data from one or more tables through a **predefined SQL query**. Views are very important for security, simplicity, and data abstraction in a database system.

Why Views Are Important:

1. Data Security

- Views can **restrict access** to sensitive columns or rows.
- Users can access only specific data defined in the view, not the entire table.
- Example: A view for HR may show employee names and departments, but not salaries.

2. Data Abstraction / Simplification

- Views **hide complex queries** and present a simpler interface.
- Users don't have to write long JOIN queries; they can query the view like a table.

3. Logical Data Independence

- You can change the structure of base tables (e.g., add columns) without affecting users who use views.
- Views act as a buffer between the **physical data structure** and the **users/applications**.

4. Reusability

- Once a view is created, it can be reused in multiple queries just like a table.
- Saves time and ensures consistency in business logic.

5. Query Optimization

- In some cases, views can help the DBMS engine optimize query performance, especially when the view uses indexed columns or aggregates.

6. Consistency

- Views ensure that **all users** see **the same result** from a common logic or business rule (e.g., tax calculations, data formatting).

Example:

Given a table Employee(EmpID, Name, Salary, Dept)

You can create a view:

```sql
CREATE VIEW HR_View AS
SELECT Name, Dept
FROM Employee;
```

4c) A **procedure** (more accurately, a **stored procedure**) is a **precompiled set of SQL statements** stored in the database. It is used to **perform a specific task**, like inserting data, updating tables, or handling business logic.

What is a Stored Procedure?

- A **stored procedure** is a **named block of code** written in SQL (or procedural SQL like PL/SQL, T-SQL).
- It can **accept parameters**, perform operations, and **return results**.
- Stored once and **reused** multiple times.

Advantages of Stored Procedures:

1. Improved Performance

- Stored procedures are **precompiled**, so they run faster than regular SQL queries.

2. Reusability

- Once written, the procedure can be **called repeatedly**, reducing code duplication.

3. Security

- Users can be granted permission to execute a procedure without giving direct access to the underlying tables.

4. Modularity

- Complex logic can be **broken into smaller procedures**, making code easier to maintain.

5. Reduces Network Traffic

- Multiple SQL statements are sent to the server **as one procedure call**, reducing communication overhead.

6. Error Handling

- Many procedural SQL languages support **exception handling** inside procedures.

Syntax Example (MySQL Style):

```sql
DELIMITER //
CREATE PROCEDURE GetEmployeesByDept(IN deptName VARCHAR(50))
BEGIN
  SELECT Name, Salary
  FROM Employee
  WHERE Dept = deptName;
```

END //

DELIMITER ;

Calling the procedure: CALL GetEmployeesByDept('HR');

5a) **1NF – First Normal Form**

Definition:

A relation (table) is in **1NF** if:

1. All attributes (columns) contain only **atomic (indivisible) values**
2. There are **no repeating groups or arrays**

In Simple Words:

- Every cell should hold **a single value**, not a list.
- Every record (row) should be unique.

Example (Not in 1NF):

StudentID	Name	Subjects
101	Alice	Math, Science
102	Bob	English

- Subjects column contains multiple values — **not atomic**

Convert to 1NF:

StudentID	Name	Subject
101	Alice	Math
101	Alice	Science
102	Bob	English

Now every field contains **only one value** → ✅ In **1NF**

2NF – Second Normal Form

Definition:

A relation is in **2NF** if:

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

In Simple Words:

- Remove **partial dependency**

- A non-key column should not depend on **part of a composite key**

Example (1NF but Not 2NF):

StudentID	CourseID	CourseName	StudentName
101	C01	Math	Alice
101	C02	Science	Alice

- Primary Key: (StudentID, CourseID)
- StudentName depends only on StudentID (partial dependency)

Convert to 2NF by splitting:

Table 1: Student

StudentID	StudentName
101	Alice

Table 2: CourseEnrollment

StudentID	CourseID	CourseName
101	C01	Math
101	C02	Science

5b) **3NF – Third Normal Form**

Definition:

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

1. It is already in **2NF**
2. **No transitive dependency** exists between non-key attributes

Transitive Dependency:

If **A → B** and **B → C**, then **A → C** is a **transitive dependency**.

In relational terms:

- A non-prime attribute depends on another non-prime attribute.

Example (2NF but not 3NF):

StudentID	StudentName	Department	HOD
101	Alice	Science	Dr. Mehta
102	Bob	Commerce	Dr. Sharma

- StudentID → Department
- Department → HOD

So:
StudentID → HOD (transitive dependency)

This violates 3NF because HOD depends on Department, not directly on the key.

Convert to 3NF:

Student Table:

StudentID	StudentName	Department
101	Alice	Science
102	Bob	Commerce

Department Table:

Department	HOD
Science	Dr. Mehta
Commerce	Dr. Sharma

Now, no transitive dependencies → ✅ In **3NF**

BCNF – Boyce-Codd Normal Form

Definition:

A table is in **BCNF** if:

1. It is in **3NF**
2. For every non-trivial functional dependency **X → Y**, **X must be a super key**

In simple words: If a **non-primary key column** determines another column, it's not in BCNF.

Example (3NF but not BCNF):

Course	Instructor	Room
DBMS	John	A101
OS	Alice	B202

Assume:

- A course is taught by **one instructor**.
- An instructor always teaches in the **same room**.

Functional dependencies:

- Course → Instructor

- Instructor → Room

Here, Instructor is **not** a super key, but it determines Room → Violates **BCNF**

Convert to BCNF:

Table 1: CourseInstructor

Course	**Instructor**
DBMS	John
OS	Alice

Table 2: InstructorRoom

Instructor	**Room**
John	A101
Alice	B202

Now all determinants are super keys → ✅ In **BCNF**

6a) **4NF – Fourth Normal Form**

Definition:

A table is in **4NF** if:

1. It is in **Boyce-Codd Normal Form (BCNF)**
2. It has **no multi-valued dependencies**.

Multi-Valued Dependency (MVD):

A **multi-valued dependency** occurs when one attribute determines multiple values of another attribute (or a set of attributes), but they are **independent of each other**.

In other words, a column in a table determines multiple independent values in other columns.

Example (Not in 4NF):

StudentID	**Subject**	**Language**
101	Math	English
101	Science	French
102	History	Spanish

Here:

- StudentID → Subject (Student determines the subject)
- StudentID → Language (Student determines the language)

The problem is that **Subject** and **Language** are independent of each other, but both depend on StudentID, which leads to **redundancy**.

Convert to 4NF:

To convert this to **4NF**, you can **split** the multi-valued dependencies into two separate tables:

Table 1: StudentSubjects

StudentID	Subject
101	Math
101	Science
102	History

Table 2: StudentLanguages

StudentID	Language
101	English
101	French
102	Spanish

Now, there are no multi-valued dependencies in either table → ✅ In **4NF**

5NF – Fifth Normal Form

Definition:

A table is in **5NF** if:

1. It is in **4NF**
2. It has **no join dependencies** and joining the table does not result in **loss of information**.

Join Dependency:

A **join dependency** occurs when a table can be decomposed into multiple smaller tables, but when joined back together, the resulting data contains redundant information, leading to loss of information.

In 5NF, the goal is to eliminate any redundancy that arises from decomposing a table into smaller tables and joining them.

Example (Not in 5NF):

Consider a table with data about **projects**, **employees**, and their **skills**:

ProjectID	EmployeeID	Skill
P1	E1	Java

ProjectID	EmployeeID	Skill
P1	E1	SQL
P1	E2	Python
P2	E1	Java
P2	E3	Java

The issue here is that we have a **join dependency** — the combination of ProjectID, EmployeeID, and Skill can be split in several ways. The table can be decomposed into:

- **Projects Table**: (ProjectID, EmployeeID)
- **EmployeeSkills Table**: (EmployeeID, Skill)

However, when you join these tables, you will **still get the same information**, but there's **redundancy** because EmployeeID-Skill combinations could be replicated in multiple rows.

Convert to 5NF:

To convert to **5NF**, we can decompose the table in such a way that no information is lost when the tables are joined:

Table 1: ProjectEmployee

ProjectID	EmployeeID
P1	E1
P1	E2
P2	E1
P2	E3

Table 2: EmployeeSkill

EmployeeID	Skill
E1	Java
E1	SQL
E2	Python
E3	Java

6b) i) If we have two attributes (or sets of attributes) in a relation, say X and Y, we say that **Y is functionally dependent on X** if:

- **For every value of X, there is exactly one corresponding value of Y**.

This is represented as: X → Y

Where:

- X is the **determinant** (the attribute or set of attributes that determines other attributes).
- Y is the **dependent** (the attribute or set of attributes whose value depends on X).

Example:

Let's consider a simple table of employees:

EmployeeID	Name	Department	Salary
101	Alice	HR	5000
102	Bob	IT	6000
103	Charlie	HR	5500

Functional Dependency Examples:

1. **EmployeeID → Name**
   - The EmployeeID uniquely determines the Name of the employee.
   - In other words, for each EmployeeID, there is exactly one Name.
2. **EmployeeID → Department**
   - Each EmployeeID determines a specific Department.
3. **EmployeeID → Salary**
   - Each EmployeeID determines a specific Salary.

In this case, **EmployeeID** is a **determinant** for **Name**, **Department**, and **Salary**

ii) In the context of database normalization, **dependency preservation** is an important concept related to **decomposition** of relations (tables). It ensures that after decomposing a table into smaller tables (during normalization), the **functional dependencies (FDs)** are still **preserved** or can be **reconstructed** without losing any of the original constraints.

In simple terms, when we decompose a table into multiple smaller tables, the **functional dependencies** from the original table should either:

1. Be **inherited** by the new smaller tables, or
2. Be able to be **recreated** by the new set of smaller tables.
3. Let a relation `R` have a set of functional dependencies `F`. If a decomposition of `R` into smaller relations `R1`, `R2`, ..., `Rn` results in a set of functional dependencies `F1`, `F2`, ..., `Fn`, the decomposition is **dependency-preserving** if:
4. The union of the functional dependencies in `F1`, `F2`, ..., `Fn` must contain all the functional dependencies in `F`. In other words, the functional dependencies in the original relation should be **either directly or indirectly** captured by the decomposed relations.

Given Table:

Consider a table Student with the following attributes and functional dependencies:

StudentID	Name	Department	DepartmentHead
101	Alice	HR	Dr. Mehta
102	Bob	IT	Dr. Sharma

Functional Dependencies:

- StudentID → Name
- StudentID → Department
- Department → DepartmentHead

Decomposition into Two Tables:

1. **Table 1: Student**

StudentID	Name	Department
101	Alice	HR
102	Bob	IT

2. **Table 2: Department**

Department	DepartmentHead
HR	Dr. Mehta
IT	Dr. Sharma

After decomposing the original Student table into two tables:

- In **Table 1**, we can preserve StudentID → Name and StudentID → Department.
- In **Table 2**, we preserve Department → DepartmentHead.

So, all original functional dependencies are **preserved** in the decomposed tables. Thus, the decomposition is **dependency-preserving**.

7a) i) The **ACID properties** are a set of properties that guarantee that database transactions are processed reliably and ensure the integrity of the database, even in cases of system failures, errors, or crashes.

ACID stands for:

1. **Atomicity**
2. **Consistency**
3. **Isolation**
4. **Durability**

Each of these properties ensures that database transactions are handled correctly, ensuring that data remains consistent, accurate, and reliable.

1. Atomicity

Atomicity means that a transaction is treated as a **single unit**, which either **completes in full** or **does not execute at all**. In other words, the transaction is **atomic** — it is indivisible and irreducible.

- If a transaction involves multiple operations (e.g., updating multiple records), all operations must either complete successfully, or none of them should be applied (in case of failure).

Example:

Consider a transaction where you are transferring money from one account to another:

- Step 1: Deduct money from Account A.
- Step 2: Add money to Account B.

If **Step 1** is completed, but **Step 2** fails, the system will **roll back** the transaction so that **no money is deducted** and the database remains consistent.

Atomicity ensures that even if there is a system failure between these two steps, the transaction will either complete entirely or leave the database unchanged.

2. Consistency

Consistency ensures that a transaction transforms the database from one valid state to another valid state, adhering to all the **rules**, **constraints**, and **triggers** defined in the database schema.

- Before and after the transaction, the database must satisfy all **integrity constraints** (e.g., **primary keys**, **foreign keys**, **unique constraints**, etc.).
- If a transaction violates any of these constraints, it is **rolled back**.

Example:

Let's say you have a constraint that ensures no negative balances in bank accounts. If a transaction tries to make the balance negative (e.g., withdrawal exceeds balance), it will not be allowed to complete.

Consistency guarantees that the database always maintains valid data after each transaction.

3. Isolation

Isolation ensures that the operations of a transaction are not visible to other transactions until the transaction is complete (committed). Even though multiple transactions can be executing concurrently, **each transaction** must be executed as if it were the **only transaction**.

- **Isolation** prevents **race conditions** where two transactions access and modify the same data simultaneously, causing inconsistencies.

There are different **isolation levels**:

- **Read Uncommitted**: Transactions can read data that has been modified but not yet committed by other transactions.
- **Read Committed**: Transactions can only read committed data.

- **Repeatable Read**: Transactions can read the same data multiple times, ensuring no other transaction modifies the data in the interim.
- **Serializable**: The highest isolation level, where transactions are executed in such a way that the result is equivalent to executing them sequentially.

Example:

Consider two transactions:

- Transaction 1: Transfer $100 from Account A to Account B.
- Transaction 2: View the balance of Account B.

If **Transaction 1** is still in progress (but not yet committed), **Transaction 2** should not see the incomplete transfer; instead, it should either wait for **Transaction 1** to complete or read only committed data.

Isolation ensures that data remains accurate even when multiple transactions are processed concurrently.

4. Durability

Durability ensures that once a transaction is committed, its changes are **permanent** and will not be lost, even if there is a system failure, power loss, or crash. Once a transaction is successfully committed, its effects are saved to the database and will persist.

- **Transaction logs** are maintained to recover the database state in case of a crash. When a transaction is committed, the changes are written to the disk, ensuring the data survives any failure.

Example:

If you transfer money between accounts and the system confirms that the transaction is complete (committed), even if there is a power failure immediately afterward, the database will be able to **recover** the transaction during the next startup and ensure the money has been correctly transferred.

Durability guarantees that once a transaction is completed, its changes are safe and will not be undone.

ii) New → Active → Partially Committed → Committed

↑ ↓

Failed ←-------------- Aborted

7b) **Two-Phase Locking (2PL)** is a concurrency control protocol used in database systems to ensure that **transactions** are executed in such a way that they preserve the **ACID properties**, especially **Isolation**. It is widely used to prevent **race conditions**, **lost updates**, and **dirty reads**, which can arise when multiple transactions are executed concurrently.

In the **Two-Phase Locking Protocol**, each transaction follows two distinct phases:

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

The goal of **2PL** is to ensure that transactions do not interfere with each other in ways that would violate **isolation** (one of the ACID properties).

Phases of Two-Phase Locking:

1. **Growing Phase:**
    - In the **growing phase**, a transaction can **acquire locks** (read or write locks) on data items.
    - **No locks are released** in this phase, only new locks can be **obtained**.
    - The transaction keeps acquiring locks until it reaches the **end of the growing phase**.
2. **Shrinking Phase:**
    - In the **shrinking phase**, a transaction can only **release locks**.
    - No new locks can be acquired in this phase; the transaction is now **restricted to releasing locks** only.
    - Once a transaction starts releasing locks, it cannot go back to acquiring new locks.

Basic Working of Two-Phase Locking:

The idea behind **2PL** is to ensure that each transaction follows these two phases strictly, preventing **deadlocks** and ensuring **serializability** — meaning that the results of executing transactions concurrently are the same as if they were executed sequentially.

1. **Growing Phase**:
    - The transaction **acquires locks** (either **shared** or **exclusive**) on data it is accessing.
    - The transaction can only acquire locks in this phase but **cannot release** any locks.
    - The goal is to accumulate all the necessary locks to perform the intended operations.
2. **Shrinking Phase**:
    - Once the transaction has acquired all the locks it needs, it enters the **shrinking phase**, where it **releases locks**.
    - No new locks can be acquired in this phase.
    - The transaction continues releasing locks as it completes its operations.

The key point is that the **growing phase must finish before the shrinking phase begins**, meaning that once a transaction starts releasing locks, it cannot go back and acquire any more locks.

Example:

Imagine two transactions, T1 and T2, on a database with two records, **A** and **B**.

Transaction T1:

- **Step 1**: T1 starts and acquires a **write lock** on record **A** (growing phase).
- **Step 2**: T1 proceeds to modify **A** and acquires a **read lock** on record **B** (still growing phase).

- **Step 3**: T1 finishes its operations and starts releasing its locks (shrinking phase).
- **Step 4**: T1 releases the lock on **B** and then releases the lock on **A**.

Transaction T2:

- **Step 1**: T2 starts and tries to acquire a **write lock** on record **B**.
- **Step 2**: However, since T1 already has a **read lock** on **B**, T2 must **wait** until T1 releases it.
- **Step 3**: T2 then proceeds to acquire the lock and make its changes.

This way, the protocol ensures **serializability**, meaning that T1 and T2 are not interfering with each other in a way that would lead to an inconsistent state.

Advantages of Two-Phase Locking:

1. **Ensures Serializability**:
   - **2PL guarantees serializability**, which is the highest level of isolation in a DBMS. It ensures that the concurrent execution of transactions is equivalent to some serial execution (i.e., transactions are executed one after another).
2. **Prevents Anomalies**:
   - It helps prevent various concurrency anomalies such as **dirty reads**, **non-repeatable reads**, and **phantom reads** by ensuring that no transaction can access data that another transaction is in the process of modifying.
3. **Deadlock Prevention (with some extensions)**:
   - While **2PL** doesn't inherently prevent deadlocks, it can be combined with deadlock detection or prevention mechanisms to avoid them.

Disadvantages of Two-Phase Locking:

1. **Deadlock**:
   - Although **2PL** guarantees **serializability**, it **does not prevent deadlocks**. Transactions may block each other, leading to situations where they wait for each other's locks indefinitely.
   - Deadlock detection or prevention methods are required to handle this.
2. **Reduced Concurrency**:
   - Since transactions hold locks during the entire growing phase, the protocol can lead to **reduced concurrency** because other transactions cannot access locked data until the transaction releases the lock.
3. **Complexity**:
   - Managing locks, especially in complex systems with many transactions, can increase the **complexity** of implementation. Efficient lock management and deadlock handling are important.

Variants of Two-Phase Locking:

1. **Strict Two-Phase Locking (S2PL)**:
   - In **strict 2PL**, transactions hold all their locks until they commit. This is a stricter version of the two-phase locking protocol, and it also **prevents cascading rollbacks** because once a transaction releases a lock, it has already committed or aborted.
   - It **guarantees recoverability** and **avoids cascading rollbacks** but still may cause deadlocks.
2. **Rigorous Two-Phase Locking**:
   - **Rigorous 2PL** is even stricter than strict 2PL because **no lock is released until the transaction commits**. It guarantees both **serializability** and **recoverability**.

Summary of Two-Phase Locking (2PL):

Phase	Action
Growing Phase	Acquire locks but cannot release any locks.
Shrinking Phase	Release locks but cannot acquire any new locks.

- **Advantages**: Ensures serializability and prevents anomalies like dirty reads.
- **Disadvantages**: Can lead to deadlocks, reduced concurrency, and complexity.
- **Deadlock Handling**: Deadlocks need to be detected or prevented through additional mechanisms.

8a) In a **Database Management System (DBMS)**, **isolation** is one of the key ACID properties that determines how transactions interact with each other in terms of visibility of uncommitted changes. The **isolation level** defines the degree to which one transaction must be isolated from other concurrent transactions.

There are **four standard isolation levels** defined by the SQL standard:

1. **Read Uncommitted**
2. **Read Committed**
3. **Repeatable Read**
4. **Serializable**

Each isolation level offers a trade-off between performance (concurrency) and consistency (isolation), with higher isolation levels providing stronger guarantees but potentially reducing concurrency.

Isolation Levels:

1. Read Uncommitted

- **Description**: In this isolation level, transactions are allowed to **read uncommitted changes** made by other transactions.
- **Consequences**: This is the **lowest level of isolation**, where the possibility of **dirty reads**, **non-repeatable reads**, and **phantom reads** is highest.

- **Dirty Read**: A transaction reads data that has been written by another transaction but not yet committed. If the second transaction rolls back, the data read by the first transaction becomes invalid.

Example:

- **Transaction 1** updates a record.
- **Transaction 2** reads the record **before** Transaction 1 commits.
- If Transaction 1 rolls back, the changes made by it are lost, leading to **inconsistent data** being read by Transaction 2.

Use Case: This level may be used for situations where performance is more important than consistency, and it's acceptable to read uncommitted data.

2. Read Committed

- **Description**: In **Read Committed** isolation, a transaction can only **read committed data**, meaning it cannot see the changes made by other transactions until they are committed.
- **Consequences**: This isolation level prevents **dirty reads**, but it still allows **non-repeatable reads** and **phantom reads**.
  - **Non-repeatable Read**: A transaction reads the same data twice, but the value has changed between the two reads because another transaction updated it.

Example:

- **Transaction 1** reads a record.
- **Transaction 2** updates the record and commits.
- **Transaction 1** reads the record again and sees a different value because Transaction 2 committed its changes after the first read.

Use Case: Commonly used for applications where data consistency is important, but absolute isolation (serializability) is not required. For example, querying a product's stock levels in an inventory system.

3. Repeatable Read

- **Description**: In this isolation level, **read operations within the same transaction will always return the same result**, even if other transactions update the data in the meantime.
- **Consequences**: **Non-repeatable reads** are prevented, but **phantom reads** may still occur.
  - **Phantom Read**: A transaction reads a set of rows that match a condition, but another transaction inserts or deletes rows that would match the condition. These newly inserted or deleted rows are invisible to the transaction reading the data.

Example:

- **Transaction 1** reads a set of records that satisfy a condition.
- **Transaction 2** inserts or deletes records that would match the same condition.

- **Transaction 1** still sees the same set of records as it did at the beginning of the transaction, but new rows (inserted by Transaction 2) would be "phantoms" and not visible to Transaction 1.

Use Case: Used in situations where consistency in repeated reads is essential, such as generating reports where results must not change during the transaction. However, concurrency is still allowed, meaning other transactions can insert new records, as long as they don't affect the transaction's current results.

4. Serializable

- **Description**: The **Serializable** isolation level is the highest level of isolation and ensures that transactions are executed in a way that guarantees **serializability**, meaning the results of concurrent transactions are as if they were executed sequentially (one after the other).
- **Consequences**: This isolation level prevents **dirty reads**, **non-repeatable reads**, and **phantom reads**. It essentially locks the data involved in the transaction for the duration of the transaction.
    - **Serializable** transactions behave as if they are executed **serially**, even when they are actually being executed concurrently.

Example:

- **Transaction 1** reads and locks a set of rows.
- **Transaction 2** cannot modify or even read the locked rows until Transaction 1 has finished and committed.
- The transactions are executed in a way that no other transaction can interfere, providing complete isolation.

Use Case: This isolation level is used when consistency is of utmost importance, and the application cannot tolerate any anomalies, such as financial transactions or banking systems.

Implementation of Isolation Levels

The implementation of isolation levels is usually done using **locks** and **scheduling techniques** in DBMS. Here's a brief overview of how DBMS systems implement these isolation levels:

1. Locking Protocols:

- **Read Uncommitted**: No locks or **shared locks** are placed on data. Any transaction can read any data, even if it is being modified by another transaction.
- **Read Committed**: **Shared locks** are used for reading, and **exclusive locks** are used for writing. Once a transaction commits, the locks are released.
- **Repeatable Read**: **Shared locks** are used to read data, and they are held for the duration of the transaction to prevent other transactions from modifying the data being read.
- **Serializable**: **Range locks** and **key locks** are used to ensure no other transaction can insert, delete, or modify data that could affect the outcome of the transaction.

2. Timestamp Ordering (for Serializable):

- In some DBMS implementations (especially for the **Serializable** level), a **timestamp ordering** protocol is used. Each transaction is given a unique timestamp, and transactions are ordered based on these timestamps to ensure serializability.
- A transaction is only allowed to access data that was committed before its timestamp. This prevents conflicts and guarantees serial execution order.

3. Two-Phase Locking (2PL):

- As discussed earlier, the **Two-Phase Locking protocol** is often used to ensure that transactions meet certain isolation requirements. In **Strict 2PL**, for example, a transaction holds all locks until it commits, ensuring **Serializable** isolation.

4. Optimistic Concurrency Control:

- This approach is typically used for **Read Committed** or **Repeatable Read** isolation levels. Transactions execute without acquiring locks, but before committing, the system checks whether any data has been modified by another transaction in the meantime. If so, the transaction is rolled back; otherwise, it commits.

Isolation Level Comparison:

Isolation Level	**Dirty Read**	**Non-Repeatable Read**	**Phantom Read**	**Locks**
Read Uncommitted	Allowed	Allowed	Allowed	No or very few locks
Read Committed	Not Allowed	Allowed	Allowed	Shared locks for reading
Repeatable Read	Not Allowed	Not Allowed	Allowed	Shared locks for reading
Serializable	Not Allowed	Not Allowed	Not Allowed	Strict locking, prevents changes

8b) **Multiple Granularity Locking in DBMS**

Multiple Granularity Locking is a technique used in **Database Management Systems (DBMS)** to manage **locks** at different levels of data granularity (i.e., at different "sizes" of data). Instead of locking data at just the individual data item level (such as a single record), multiple granularity locking allows locks to be placed on different levels of the database hierarchy, such as:

- **Database**
- **Table**
- **Page**
- **Tuple (Row)**
- **Field (Column)**

This approach helps balance **concurrency** and **data consistency**, as it can allow more flexible and efficient locking. By locking data at various levels, a DBMS can increase **concurrency** by allowing some transactions to access different parts of the database while still ensuring **correctness** and **serializability**.

Granularity of Locks

Locks in **multiple granularity** locking are applied to different data units, each having its own level of granularity:

1. **Database-level locks**:
   - A lock is applied to the entire database.
   - It is the **coarsest level** of granularity.
   - When a transaction holds a database lock, it has exclusive access to the entire database, preventing other transactions from accessing any part of the database.
2. **Table-level locks**:
   - A lock is applied to an entire table.
   - When a transaction holds a table lock, it prevents other transactions from accessing or modifying the data in that table.
3. **Page-level locks**:
   - A lock is applied to a page (a block of data storage that contains multiple rows).
   - This is a middle granularity level, balancing between access to specific rows and the entire table.
4. **Row-level locks**:
   - A lock is applied to a single **row** or **tuple** in a table.
   - This is a **finer** granularity than table-level or page-level locks and allows for the highest **concurrency** because it minimizes the locking scope, but it also comes with the overhead of managing many more locks.
5. **Field-level locks**:
   - A lock is applied to a specific **field** or **column** in a row.
   - This is the finest level of granularity, and it provides the most concurrent access to the database. However, it is also complex to manage and often not supported by all DBMS systems.

Locking Modes

In multiple granularity locking, a transaction can request a **lock** at different levels, and it can use various types of **lock modes** to determine what kind of access it needs. These lock modes are:

1. **Shared (S) Lock**:
   - Allows a transaction to read the data but prevents other transactions from modifying it.

- Multiple transactions can hold **shared locks** on the same data at the same time.

2. **Exclusive (X) Lock**:
   - Prevents other transactions from reading or modifying the data.
   - Only one transaction can hold an **exclusive lock** at a time on a particular data item.
3. **Intention Shared (IS) Lock**:
   - Indicates that the transaction intends to acquire shared locks on lower levels (e.g., individual rows or fields) of the data.
   - Allows transactions to share access at higher levels (like a table) while still being able to acquire more specific locks at lower levels.
4. **Intention Exclusive (IX) Lock**:
   - Indicates that the transaction intends to acquire exclusive locks on lower levels of the data.
   - This lock mode ensures that no other transactions can acquire shared or exclusive locks on the lower-level data items.
5. **Shared and Intention Exclusive (SIX) Lock**:
   - A combination of **shared** and **intention exclusive** locks. It allows a transaction to have a shared lock on a higher-level unit (like a table) while intending to acquire exclusive locks on lower-level units (like rows).
   - This is useful for balancing read and write access across the database.

Multiple Granularity Locking Protocol

The **Multiple Granularity Locking Protocol (MGLP)** defines rules for how locks can be applied at various granularities and how locks must be managed to ensure **serializability** and prevent **deadlocks**.

1. **Parent-Child Relationship**:
   - In a hierarchical structure (such as a table, page, or row), a **parent lock** (such as a table lock) must be acquired before a **child lock** (such as a row or page lock) can be requested.
   - For example, to acquire a row lock, a transaction must first acquire a **table lock** at the higher level, ensuring that the transaction will not violate the consistency of the data.
2. **Lock Compatibility**:
   - The protocol requires that locks at higher levels (e.g., table or database) may not conflict with locks at lower levels (e.g., row or page).
   - For example, a transaction that has an **intention shared (IS)** lock on a table can still acquire **shared (S)** locks on individual rows within the table, but it cannot acquire an **exclusive (X)** lock on any row in the table until it has finished with the **IS lock** on the table.

3. **Upward Propagation of Locks**:
    - A transaction that holds a lock on a lower-level object (like a row) must also hold a lock at the higher-level object (like the table or page) to avoid conflicts.
    - For instance, if a transaction holds a row lock, it must also hold an **intention exclusive (IX)** lock on the parent table.

Example of Multiple Granularity Locking

Let's consider a **database** with a table of **employees**. This table has multiple rows, and each row represents a different employee's data.

- **Transaction 1** intends to update the salary of one employee. It would first acquire a **shared (S) lock** on the **table** to ensure no other transaction can modify the structure of the table. Then, it would acquire an **exclusive (X) lock** on the specific **row** representing that employee's data.
- **Transaction 2**, meanwhile, wants to read all employee data. It could acquire a **shared (S) lock** on the **table** to read the rows but would avoid locking any individual rows with an exclusive lock, allowing more concurrency.
- **Transaction 3** wants to modify the table structure (e.g., adding a new column). It would acquire an **exclusive (X) lock** on the **table**, preventing any other transaction from accessing or modifying the table until the schema modification is complete.

Advantages of Multiple Granularity Locking

1. **Increased Concurrency**:
    - By allowing locks to be applied at different granularities (database, table, row), DBMS can achieve higher **concurrency** compared to locking everything at the most granular level (i.e., row or field).
2. **Flexibility**:
    - The protocol offers flexibility, allowing transactions to operate at the level of granularity required for their task while still ensuring that consistency is maintained.
3. **Better Resource Utilization**:
    - Coarser-grained locks (like on tables) reduce the overhead of managing a large number of locks compared to finer-grained locking schemes (like row or field-level locks), leading to more efficient resource utilization.
4. **Reduced Deadlocks**:
    - Multiple granularity locking helps in reducing the likelihood of deadlocks by imposing clear rules on lock acquisition and release, ensuring that transactions follow a hierarchical locking order.

Disadvantages of Multiple Granularity Locking

1. **Complexity**:

- Managing locks at multiple levels introduces complexity in both implementation and monitoring. The system needs to keep track of all lock acquisitions and releases, as well as handle issues like deadlock detection.

2. **Overhead**:
    - Although multiple granularity allows for more flexibility, it can lead to additional **locking overhead**, especially in systems with a large number of transactions and data items.
3. **Risk of Deadlocks**:
    - Even though the protocol reduces deadlocks, it doesn't eliminate them entirely. If transactions acquire locks at multiple levels in conflicting orders, deadlocks can still occur.

9a) In a **Database Management System (DBMS)**, **logs** are critical for ensuring **data integrity** and **recoverability** in the event of a **system crash** or a **transaction failure**. The process of using logs for recovery is a key part of **transaction management** and is tightly tied to the **ACID properties**, especially **atomicity** and **durability**.

What Is a Log?

A **log** is a sequential record maintained by the DBMS that stores information about **all the transactions** and the **changes** they make to the database.

Each entry in the log typically contains:

- **Transaction ID**
- **Type of operation** (e.g., write, commit)
- **Data item** being modified
- **Old value** (before modification)
- **New value** (after modification)
- **Timestamps**

Purpose of the Log

1. **To Recover from a System Crash** (e.g., power failure, OS crash)
2. **To Rollback Uncommitted Transactions** (e.g., due to errors or aborts)

How Logging Helps in Recovery

1. Write-Ahead Logging (WAL)

Most DBMSs use the **Write-Ahead Logging (WAL)** protocol:

Before any change is made to the database, the **log must be written to disk**.

This ensures that even if the system crashes after the log is written but **before** the data is written to disk, the log can still be used to **redo** or **undo** changes.

2. Recovery Using Logs After a System Crash

When a system crashes, the DBMS performs two types of operations during recovery:

REDO (Reapply Committed Changes)

- If a **transaction committed before the crash**, its operations must be **redone** to ensure that all its changes are reflected in the database.

UNDO (Rollback Uncommitted Changes)

- If a **transaction did not commit before the crash**, any partial changes must be **undone** using the log.

Recovery Process Using Logs: Step-by-Step

Step 1: Analyze the Log

- Scan the log **from the beginning** to identify:
  - Which transactions were committed
  - Which were active (not committed) at the time of the crash

Step 2: UNDO Active Transactions

- For each **uncommitted transaction**, perform an **UNDO** using the **old values** in the log.
- This restores the database to the state before the transaction began.

Step 3: REDO Committed Transactions

- For each **committed transaction**, reapply its changes using the **new values** from the log to ensure **durability**.

Example

Let's assume a log with the following entries:

Time	Operation	Data Item	Old Value	New Value
T1	Start Transaction			
T1	Write(A)	A	50	60
T1	Write(B)	B	20	30
T1	Commit			
T2	Start Transaction			
T2	Write(A)	A	60	70
🔥	**System Crash**			

After Recovery:

- **T1** is **committed**, so its changes to A and B are **REDONE**.
- **T2** is **not committed**, so its change to A is **UNDONE** (restored to 60).

Rollback Using Log

A **rollback** is the **manual or automatic reversal** of all actions performed by a transaction that has not completed successfully.

Rollback Steps:

1. DBMS checks the log for all **write operations** by the transaction.
2. It reverses the changes using the **old values** in the log.
3. Once all changes are undone, a **"Transaction Aborted"** log record is written.

Example:

Log Entry
T3 Start
T3 Write(X): 100→90
T3 Write(Y): 200→180
T3 Rollback

→ After rollback,

- X = 100
- Y = 200
  (from old values in log)

9b) **What is a Checkpoint in DBMS?**

A **checkpoint** is a **snapshot** of the database at a specific point in time. It is used to **minimize recovery time** after a system crash.

Purpose of a Checkpoint:

- Reduces the number of log records the system must process during recovery.
- Helps **truncate** old logs that are no longer needed.
- Improves **efficiency** of crash recovery.

How Checkpoint Works

At the time of a **checkpoint**, the DBMS:

1. **Writes all modified (dirty) pages** from memory (buffer cache) to disk.
2. **Writes a checkpoint log record** noting that all previous changes are now safely stored on disk.
3. Removes or archives old logs before the checkpoint since they're no longer needed for recovery.

Example:

Imagine the following sequence of events:

Time → T1 Start → T1 Write(A) → T2 Start → T2 Write(B) → CHECKPOINT → T2 Commit

If the system crashes **after the checkpoint**, recovery only needs to consider log entries **after** the checkpoint.

Fuzzy Checkpointing

A **fuzzy checkpoint** is a **non-blocking checkpoint** that allows **transactions to continue** executing while the checkpoint is in progress.

This is used in **modern DBMSs** to avoid halting the system during checkpointing.

Key Characteristics:

- **No transaction is paused** while the checkpoint is happening.
- The system **marks the active transactions** and **dirty pages** at the time the checkpoint starts.
- It may write data to disk in stages while allowing updates to continue.
- Ensures that recovery can still work by using **logs** to cover any pages that were not flushed yet.

Comparison: Checkpoint vs Fuzzy Checkpoint

Feature	**Regular Checkpoint**	**Fuzzy Checkpoint**
Blocking	May block transactions briefly	Non-blocking (transactions continue)
Recovery Time	Faster if frequent	Slightly more complex due to concurrency
Complexity	Simpler	More complex (needs tracking of changes)
Use Case	Basic DBMS or periodic logging	Modern, high-performance DBMS

Visualization

Memory (Buffer)   Disk   Log File

[Page A*]   ---->   [Page A]   ← Log: T1 writes A, T2 writes B

[Page B*]   ---->   [Page B]   (before checkpoint)

Checkpointing

Dirty pages written to disk

Checkpoint record added to log

[Page A]   ✓

[Page B]   ✓

Later:

→ If crash occurs, recovery starts from checkpoint log record.

→ Only transactions active after checkpoint are redone or undone.

In **fuzzy checkpointing**, some pages may not have been flushed **yet**, so recovery will:

1. Use the checkpoint record as a base.
2. Redo all committed changes **after** the checkpoint (even if some were not flushed yet).
3. Undo uncommitted changes using log.

10a) **Buffer management** is a **core component of a Database Management System (DBMS)** that manages **data transfer between disk and main memory (RAM)**. Since accessing data from disk is much slower than accessing it from memory, buffer management plays a key role in **improving performance** by minimizing direct disk I/O operations.

Why Buffer Management Is Needed

- Databases are too large to fit entirely in memory.
- Disk I/O is expensive and time-consuming.
- Frequently accessed data should be cached in memory (buffer pool).
- Buffer management ensures efficient use of memory and quick data access.

Key Concepts

1. Buffer Pool

- A reserved area in **main memory** used to cache database pages.
- Consists of fixed-size blocks (also called **frames**).
- When a page is needed, it's **loaded from disk into the buffer** if not already present.

2. Page

- A **unit of data transfer** between disk and memory (usually 4KB–16KB).
- The database is read/written in terms of pages, not individual rows or columns.

Working of Buffer Management

1. **Page Request**:
   - A transaction requests a data page.
   - Buffer manager checks if the page is already in memory.
2. **Page Hit**:
   - If the page is **in memory**, it is returned immediately (**fast access**).
3. **Page Miss**:

- If the page is **not in memory**, it is **fetched from disk** into the buffer pool.
- If the buffer pool is full, a page must be **replaced** using a **replacement policy**.

4. **Modified Pages (Dirty Pages)**:
    - If a page is **modified** in memory, it is marked as a **dirty page**.
    - Dirty pages must be **written back to disk** before being replaced to avoid data loss.

Buffer Replacement Policies

When the buffer pool is full, the DBMS must decide **which page to evict**. Common strategies include:

Policy	Description
LRU (Least Recently Used)	Replaces the page that hasn't been used for the longest time.
MRU (Most Recently Used)	Replaces the most recently used page.
FIFO (First-In First-Out)	Replaces the page that entered the buffer first.
Clock	Approximate LRU using a circular list and reference bits.

Pinning and Unpinning

- **Pinning** a page: Marks it as "in use" so it can't be replaced.
- **Unpinning**: After a transaction finishes using the page, it unpins it, allowing it to be replaced if needed.

Write Strategies

1. Write-Through:

- Immediately writes changes to disk.
- Safer but slower.

2. Write-Back (Deferred Write):

- Writes only when the page is evicted from the buffer.
- Faster but riskier (needs proper recovery mechanisms like logs).

Example

Imagine a buffer pool with 3 frames, and a transaction wants to read the following page sequence:

less

CopyEdit

Request sequence: A, B, C, A, D

- A, B, C → Loaded into buffer (3 slots used)
- A → Already in buffer (hit)
- D → Buffer full, use LRU to replace B (least recently used)

10b) In a Database Management System (DBMS), **UNDO** and **REDO** are mechanisms used to **maintain consistency and recover** from errors such as system crashes or transaction failures. They are part of the **recovery management system**, which ensures **atomicity** and **durability**—two key ACID properties.

UNDO

Definition:
UNDO refers to the process of **reversing the changes** made by a **transaction that did not commit** successfully.

Purpose:

- To **abort** a transaction.
- To **remove partial or uncommitted changes** from the database.
- To restore the database to the **state before the transaction began**.

Example:

If a transaction modifies a balance from ₹1000 to ₹1200 but crashes before committing, **UNDO** restores the balance back to ₹1000.

Uses:

- Manual rollback by user.
- Automatic rollback on system crash.
- Aborting failed or deadlocked transactions.

REDO

Definition:
REDO refers to the process of **reapplying changes** of a **committed transaction** that might not have been permanently written to disk at the time of a system crash.

Purpose:

- To **ensure durability**.
- To **replay committed transactions** and ensure their effects are reflected in the database.

Example:

If a transaction commits after updating a balance from ₹1000 to ₹1200, but the system crashes before writing it to disk, **REDO** re-applies the change so the balance becomes ₹1200.

Uses:

- Crash recovery after system failure.
- Re-establishing committed changes from logs.