CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assessment Test 2 – Mar 2025

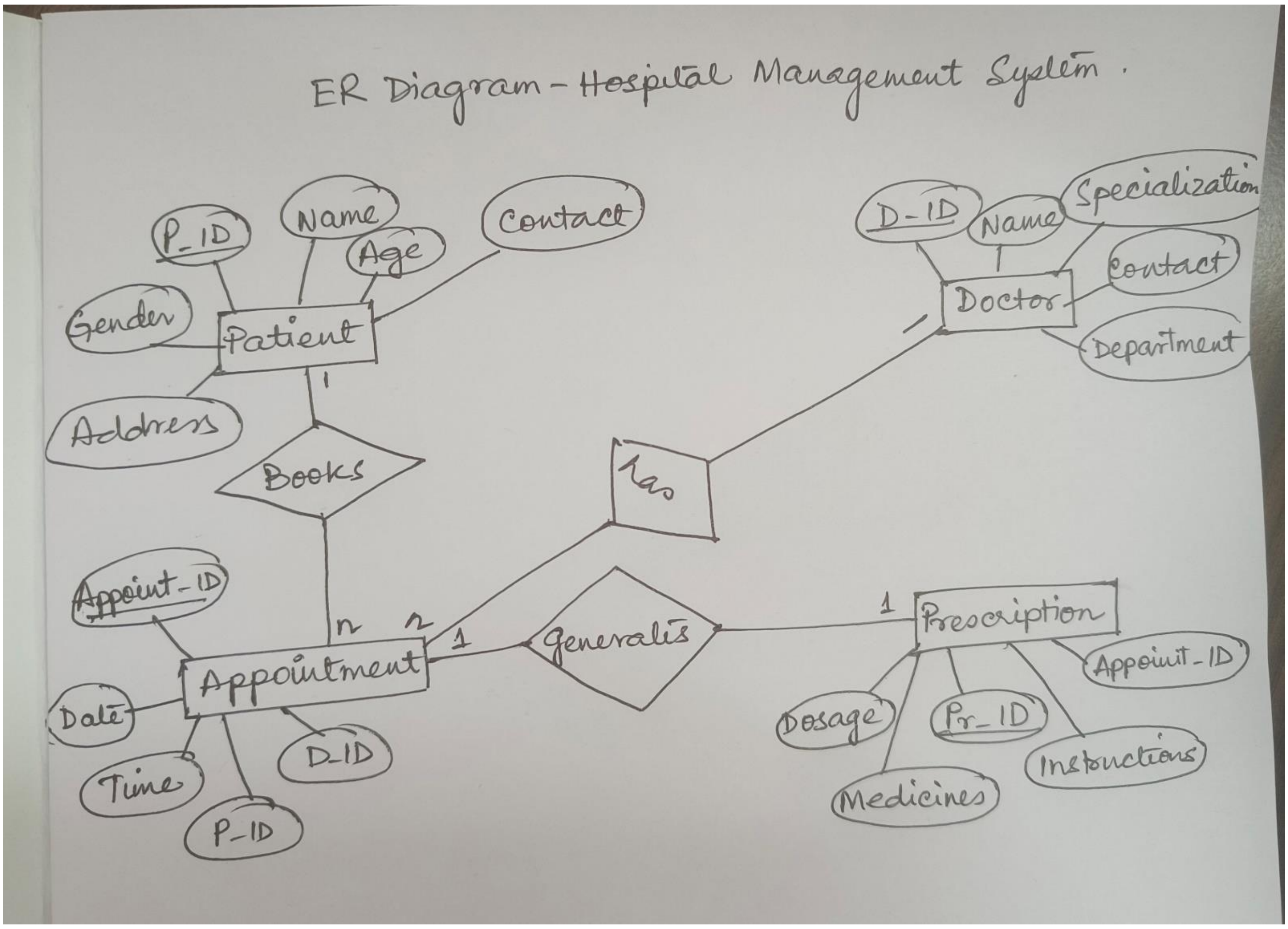
Sub:	**Database Management Systems**							**Sub Code:**	**MMC103**
Date:	**19.03.25**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**I**	**Branch:**	**MCA**

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	Draw a complete ER diagram for a Hospital Management System (Entities are Patient, Doctor, Appointment and Prescription). **OR**	[10]	CO1	L3
2	What is relational algebra? Explain each of its operations with an example.	[2+8]	CO1	L2
3	**PART II** Write a trigger that fires before an INSERT operation in the orders table when order amount exceeds $10,000, else raise an error or prevent the insertion to enforce the business rule. **OR**	[10]	CO3	L3
4	a. What is nested subquery? What are the types of subqueries? Explain with example. b. A table has customer data (Customer_No, Name, Age, Phone, Email, Address). Write a query to print the customer's name having minimum age.	[2+4+4]	CO3	L3
5	**PART III** What are functional dependencies? Explain trivial, semi non-trivial and transitive functional dependencies with examples. **OR**	[2+2+3+3]	CO2	L2
6	What is lossless decomposition? Discuss its advantages and disadvantages.	[4+6]	CO1	L1
7	**PART IV** Discuss ACID properties. Explain the transaction isolation levels **OR**	[5+5]	CO1	L1
8	What is serializability? How do you detect it? What are its types? Explain with examples	[10]	CO1	L2
9	**PART V** What is checkpoint? Explain its benefits. How can a log file help in recovering a database? **OR**	[2+3+5]	CO1	L1
10	How does recovery with concurrent transactions work? How can we deal with failure with loss of non-volatile storage	[5+5]	CO1	L1

Answers:

1. Draw a complete ER diagram for a Hospital Management System (Entities are Patient, Doctor, Appointment and Prescription).



2. What is relational algebra? Explain each of its operations with an example.

Relational Algebra is a procedural query language used to retrieve data from relational databases. It defines a set of operations that take one or more relations (tables) as input and produce a new relation as output.

Basic Relational Algebra Operations

1. Selection (σ)

- **Definition:** Filters rows (tuples) based on a given condition.
- **Notation:** σ_condition (Relation)
- **Example:** Retrieve students with age greater than 20.
  σ Age > 20 (Student)
- **Result:** Returns only those rows where Age > 20.

2. Projection (π)

- **Definition:** Selects specific columns (attributes) from a relation.
- **Notation:** π_column1, column2, ... (Relation)
- **Example:** Retrieve only Student Names and Ages.
  π Name, Age (Student)
- **Result:** Returns a table with only the selected columns.

3. Union (∪)

- **Definition:** Combines tuples from two relations, removing duplicates.
- **Notation:** Relation1 ∪ Relation2
- **Example:** Retrieve students enrolled in either Course A or Course B.
  Enrolled_A ∪ Enrolled_B
- **Result:** Returns unique tuples present in either relation.

4. Set Difference (−)

- **Definition:** Retrieves tuples from the first relation that are not in the second relation.
- **Notation:** Relation1 - Relation2
- **Example:** Find students in Course A but not in Course B.
  Enrolled_A − Enrolled_B

- **Result:** Returns tuples present in Enrolled_A but not in Enrolled_B.

5. Cartesian Product (×)

- **Definition:** Combines all tuples of one relation with all tuples of another.
- **Notation:** Relation1 × Relation2
- **Example:** Combine Student and Course relations.
  Student × Course
- **Result:** Produces all possible combinations of students and courses.

6. Join (⋈)

- **Definition:** Combines two relations based on a related attribute.
- **Notation:** Relation1 ⋈ condition Relation2
- **Example:** Retrieve students along with the courses they are enrolled in.
  Student ⋈ Student.StudentID = Enrollment.StudentID Enrollment
- **Result:** Returns only matching tuples from both relations.

7. Division (÷)

- **Definition:** Finds tuples from the first relation that are related to all tuples in the second relation.
- **Notation:** Relation1 ÷ Relation2
- **Example:** Find students who have taken all required courses.
  Enrolled ÷ Required_Courses
- **Result:** Returns students who are enrolled in every course listed in Required_Courses.

3. Write a trigger that fires before an INSERT operation in the orders table when order amount exceeds $10,000, else raise an error or prevent the insertion to enforce the business rule.

```
CREATE TRIGGER before_order_insert
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
   -- Check if the order amount exceeds $10,000
   IF NEW.order_amount > 10000 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Order amount exceeds the permitted limit of $10,000.';
   END IF;
END
```

4. a. What is nested subquery? What are the types of subqueries? Explain with example.
b. A table has customer data (Customer_No, Name, Age, Phone, Email, Address). Write a query to print the customer's name having minimum age.

a. A nested subquery is a subquery (a query inside another query) that is executed before the outer query and provides results to it. The inner query runs first and its output is used by the outer query to perform further operations.

Example of Nested Subquery

Let's consider two tables:

- Employees (EmpID, Name, Salary, DeptID)
- Departments (DeptID, DeptName)

If we want to find employees who earn more than the **average salary** of all employees, we can use a nested subquery:

```
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

 The **inner query** (SELECT AVG(Salary) FROM Employees) calculates the average salary.
 The **outer query** selects employees whose salary is greater than this average.

Types: correlated and uncorrelated.

Correlated example:

```
SELECT Name, Salary, DeptID
FROM Employees E1
WHERE Salary > (SELECT AVG(Salary) FROM Employees E2 WHERE E1.DeptID = E2.DeptID);
```

b.
```
SELECT Name
FROM Customers
WHERE Age = (SELECT MIN(Age) FROM Customers);
```

5. What are functional dependencies? Explain trivial, semi non-trivial and transitive functional dependencies with examples.

A **functional dependency** is a constraint that specifies the relationship between attributes in a relational database. It expresses how one set of attributes uniquely determines another set.
Definition:
If X and Y are two sets of attributes in a relation R, then X → Y (X determines Y) means that for every unique value of X, there is only one corresponding value of Y.

Types of Functional Dependencies
1. Trivial Functional Dependency
A functional dependency X → Y is **trivial** if Y is a subset of X.
Example:
Consider a relation **Student (RollNo, Name, Age)**
- {RollNo, Name} → Name is a **trivial** FD because Name is already part of {RollNo, Name}.

General Rule:
A functional dependency is trivial if it **does not add new information** to the relation.

2. Semi Non-Trivial Functional Dependency
A **semi non-trivial** functional dependency X → Y is where Y is not a subset of X, but part of Y is contained in X.
Example:
Consider a relation **Employee (EmpID, Name, Dept, Salary)**
- {EmpID, Dept} → {Dept, Salary}
  - Here, {Dept} is already in {EmpID, Dept}, but {Salary} is new.

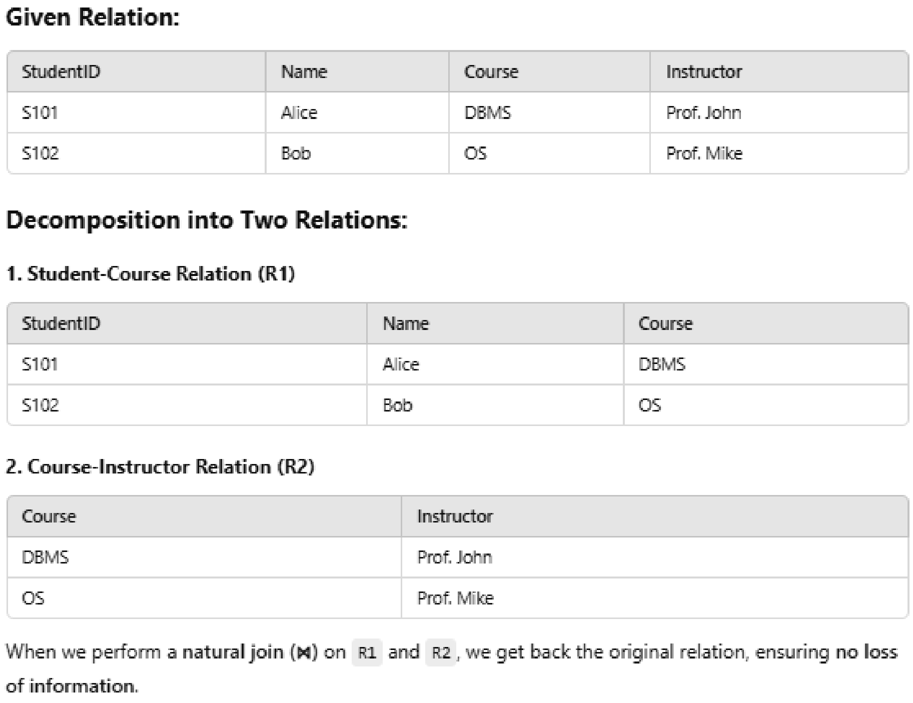
Thus, this dependency is **semi non-trivial** because it partially includes existing attributes.

3. Transitive Functional Dependency
A **transitive functional dependency** occurs when an attribute indirectly depends on another through a third attribute.
Example:
Consider a relation **Employee (EmpID, DeptID, DeptName)** where:
1. EmpID → DeptID (EmpID determines DeptID)
2. DeptID → DeptName (DeptID determines DeptName)

Since EmpID determines DeptID and DeptID determines DeptName, we get:
EmpID → DeptName (Transitive Dependency)

6. What is lossless decomposition? Discuss its advantages and disadvantages.

Lossless decomposition is a process in **database normalization** where a relation is decomposed into two or more smaller relations **without losing any information**. It ensures that when the smaller relations are joined back using a **natural join (⋈)**, they **reproduce the original relation** without any data loss or spurious tuples.
Definition
A decomposition of a relation R into R1 and R2 is **lossless** if:
$R1 \cap R2 \rightarrow R1$ or $R1 \cap R2 \rightarrow R2$
This means that the common attributes **must be a candidate key** in at least one of the decomposed relations.



Given Relation:

StudentID	Name	Course	Instructor
S101	Alice	DBMS	Prof. John
S102	Bob	OS	Prof. Mike

Decomposition into Two Relations:

1. Student-Course Relation (R1)

StudentID	Name	Course
S101	Alice	DBMS
S102	Bob	OS

2. Course-Instructor Relation (R2)

Course	Instructor
DBMS	Prof. John
OS	Prof. Mike

When we perform a **natural join (⋈)** on `R1` and `R2`, we get back the original relation, ensuring **no loss of information.**

7. Discuss ACID properties. Explain the transaction isolation levels

In database systems, **ACID** properties ensure **data integrity, consistency, and reliability** during transactions. **ACID** stands for:

1. Atomicity (All or Nothing)

- A transaction must be **fully completed or fully rolled back** if any part of it fails.
- **Example:** If you transfer ₹500 from Account A to Account B, both the **debit from A** and **credit to B** must happen together. If the system crashes after debiting A but before crediting B, the transaction must be **rolled back** to ensure no money is lost.

2. Consistency (Data Integrity)

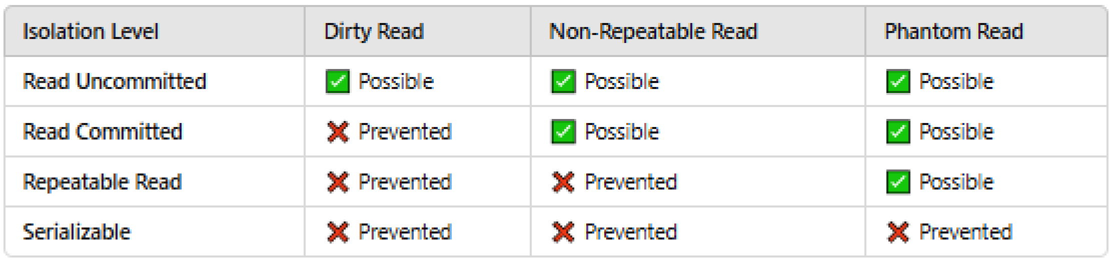
- The database must remain in a **valid state** before and after a transaction.
- **Example:** If a transaction violates a foreign key constraint, it must be **aborted** to maintain consistency.

3. Isolation (Concurrent Transactions Should Not Interfere)

- Transactions should execute **independently** without affecting each other.
- **Example:** If two users withdraw money from the same account simultaneously, isolation ensures that the final balance remains correct.

4. Durability (Permanent Changes)

- Once a transaction is **committed**, its changes are **permanently stored**, even in case of system failures.
- **Example:** If a power outage occurs after a successful ticket booking, the booking details must be retained after system recovery.



Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	✅ Possible	✅ Possible	✅ Possible
Read Committed	❌ Prevented	✅ Possible	✅ Possible
Repeatable Read	❌ Prevented	❌ Prevented	✅ Possible
Serializable	❌ Prevented	❌ Prevented	❌ Prevented

8. What is serializability? How do you detect it? What are its types? Explain with examples

Serializability is a **concept in database concurrency control** that ensures the execution of multiple transactions is **equivalent to some serial (one-by-one) execution** of those transactions.
It ensures **data consistency** in a **concurrent execution environment** by preventing anomalies such as dirty reads, lost updates, and inconsistent retrievals.

Types of Serializability
1. Conflict Serializability
A schedule is **conflict serializable** if it can be **transformed into a serial schedule** by swapping **non-conflicting** operations.
Conflicting Operations:
Two operations in different transactions **conflict** if:

1. They **operate on the same data item**.
2. At least one of them is a **write operation**.

Example of Conflict Serializability:
Schedule S:

T1: Read(A)
T2: Write(A)
T1: Write(A)

Here, T1 and T2 are **conflicting** because both modify A. If we swap T1 and T2, we may get a **different result**, making it **non-conflict serializable**.
✅ **Conflict Serializable Schedule:**

T1: Read(A), Write(A)
T2: Read(B), Write(B)

Since T1 and T2 operate on different data (A and B), they are **not conflicting** and can be swapped.

2. View Serializability
A schedule is **view serializable** if it **produces the same final result** as a serial schedule, even if operations are not conflict-swappable.
Conditions for View Serializability:
A schedule is view serializable if:

1. **Initial Reads Match:** The first read in both schedules must be from the same transaction.
2. **Intermediate Reads Match:** Every transaction reads the same data as in a serial schedule.
3. **Final Writes Match:** The last write in both schedules must be from the same transaction.

Example of View Serializability:
T1: Read(A), Write(A)
T2: Read(A), Write(A)

Even if swapping is **not possible**, if the **final output is the same as a serial execution**, it is **view serializable**.
Note: Every **conflict serializable** schedule is **view serializable**, but not every view serializable schedule is conflict serializable.

How to Detect Serializability?
1. Precedence Graph (Serializability Graph)

- **Steps to construct a precedence graph:**
    1. **Create a node** for each transaction.
    2. **Draw a directed edge (Ti → Tj)** if:
        - Ti writes a data item before Tj reads/writes it.
        - Ti reads a data item before Tj writes it.
    3. If the **graph has a cycle**, the schedule is **not serializable**.

Example:
Schedule:
T1: Read(A)
T2: Write(A)
T1: Write(A)

T1 → T2 → T1 (Cycle detected ✗)
Since there's a **cycle**, it is **not serializable**.

✅ **Serializable Schedule:**

T1: Read(A), Write(A)
T2: Read(B), Write(B)

Graph: T1 → T2 (No cycle ✅)
This schedule is **serializable**.

9. What is checkpoint? Explain its benefits. How can a log file help in recovering a database?

A **checkpoint** is a mechanism in a database management system (DBMS) used to minimize the time required for **recovery** in case of a system crash. It is a point where the DBMS ensures that all **modified data (dirty pages) in memory are written to disk**, and all log records before this point are **safely stored**.

Benefits of Checkpoints

1. **Reduces Recovery Time**
    - Instead of scanning the entire log, the system can **start recovery from the latest checkpoint**, making the process much faster.
2. **Minimizes Data Loss**
    - Ensures that committed transactions are safely written to disk, reducing the chances of data loss in case of a crash.
3. **Improves System Performance**
    - Reduces the overhead of maintaining large log files, improving overall system efficiency.
4. **Efficient Log Management**
    - Older log records before the checkpoint can be **discarded**, preventing log files from growing indefinitely.
5. **Prevents Redo of Committed Transactions**
    - Transactions committed before the checkpoint do not need to be **redone**, reducing recovery workload.

How Log Files Help in Database Recovery
A **log file** maintains a sequential record of all changes made to the database. During recovery, log records help restore the database to a **consistent state**.
Types of Logging

1. **UNDO Logging** (Rollback)
    - If a transaction fails, the system uses the log to revert uncommitted changes.
2. **REDO Logging** (Reapply Changes)
    - Ensures committed transactions are reapplied if they were lost before being written to disk.

Steps in Database Recovery Using Log Files

1. **Identify the Last Checkpoint**
    - Find the most recent checkpoint in the log.
2. **REDO Phase (Reapply Committed Changes)**
    - Scan the log forward from the checkpoint and **reapply committed transactions** to restore lost updates.
3. **UNDO Phase (Rollback Uncommitted Changes)**
    - Rollback uncommitted transactions by using the **before-values** stored in the log.

Example of Log-Based Recovery

Before Crash (Log File)

T1: Start
T1: Write(A=500 → 600)
T2: Start
T2: Write(B=100 → 150)
Checkpoint
T1: Commit
T2: Write(B=150 → 200)
Crash!

Recovery Process

1. **Start from Checkpoint**
2. **Redo T1** (Since it was committed)
3. **Undo T2** (Since it was incomplete at the crash)

After recovery, the **database is restored to a consistent state**.

10. How does recovery with concurrent transactions work? How can we deal with failure with loss of non-volatile storage

Recovery with Concurrent Transactions

When multiple transactions execute **concurrently**, recovery must ensure that the database remains **consistent** and transactions follow **ACID properties** (Atomicity, Consistency, Isolation, Durability). The recovery system must handle concurrent transactions efficiently using **log-based recovery** and **synchronization techniques**.

Steps for Recovery with Concurrent Transactions

1. **Identify the Last Checkpoint**
   - Recovery starts from the most recent **checkpoint** in the log file.
2. **REDO Phase (Reapply Committed Changes)**
   - Transactions that were **committed before the crash** are reapplied to ensure no data is lost.
3. **UNDO Phase (Rollback Uncommitted Changes)**
   - Transactions that were **not committed before the crash** are rolled back to maintain consistency.

Handling Concurrent Transactions in Recovery

- **Write-Ahead Logging (WAL)**
  - Log changes **before** applying them to the database to ensure data is not lost.
- **Strict Two-Phase Locking (2PL)**
  - Ensures transactions do not interfere with each other during recovery.
- **ARIES (Algorithm for Recovery and Isolation Exploiting Semantics)**
  - A widely used recovery method that ensures proper handling of concurrent transactions using **log records** and **redo/undo logging**.

Failure with Loss of Non-Volatile Storage

Non-volatile storage (like HDDs or SSDs) stores the database **permanently**. If this storage fails, the database must be **restored from backups** and logs.

Ways to Handle Loss of Non-Volatile Storage

1. **Backup & Restore Mechanism**

    - **Periodic backups** (full, incremental, differential) ensure recovery even if storage is completely lost.
    - **Cloud-based backups** provide redundancy.
2. **Replication & Mirroring**
    - **Database replication** maintains copies of data in **different locations**.
    - **RAID (Redundant Array of Independent Disks)** provides **data redundancy**.
3. **Shadow Paging**
    - Maintains a **copy of data** in a separate storage location, reducing dependency on a single disk.
4. **Redo Log from Remote Storage**
    - If logs are stored on a **different non-volatile storage system**, transactions can be **redone** after restoration.

Recovery Process After Storage Loss

1. **Restore from the latest backup**
2. **Apply all REDO logs** stored externally
3. **Rollback any incomplete transactions using UNDO logs**