K. K. Wagh Institute of Engineering Education and Research, Nashik
(An Autonomous Institute from A. Y. 2022-23)

End-Sem Examination- Winter 2025
Model Answer

Academic Year: 2025-2026	**Semester: I**
Name of Programme: MCA	**Pattern: 2024**
Name of Course: Database System and SQL	**Course Code: 2409512**
Max. Marks: 60	**Duration: 2:30Hr.**

Q. No.	Details	Max. Marks
1	Construct ER Diagram for Library Management System: A college library maintains records of books, members, issue/return transactions, and fines. Each book has a title, ISBN, author, and category. Members can borrow multiple books, and fines apply for late returns **Answer:** 1) Entities & Attributes • Book (ISBN PK, Title, Author, Category) o *Note:* Author may be multivalued if books can have multiple authors. • Member (MemberID PK, Name, Address, Phone, Email, MembershipDate) • IssueTransaction (IssueID PK, IssueDate, DueDate, ReturnDate, FineAmount) o FineAmount can be stored or derived (derived = f(DueDate, ReturnDate)). 2) Relationships and Cardinalities • Issues (between Member and Book) — implemented as associative entity IssueTransaction. o A Member can issue 0..* IssueTransaction records (borrow many books over time). o A Book can be involved in 0..* IssueTransaction records (can be borrowed multiple times over time). o So cardinality between Member and IssueTransaction = 1 Member : * IssueTransaction; between Book and IssueTransaction = 1 Book : * IssueTransaction. • Participation: IssueTransaction totally depends on both Member and Book (each transaction must reference a member and a book). **3) ER Diagram (ASCII sketch)** +----------+ +----------+ +----------+	[6]

	| Member |1 * | IssueTransaction | * 1 | Book | |---------- |---------|---------|---------|---------| | MemberID (PK) | | IssueID (PK) | | ISBN (PK) | | Name | | IssueDate | | Title | | Address | | DueDate | | Author | | Phone | | ReturnDate | | Category | | Email | | FineAmount (derived) | | | +----------+ +----------+ +----------+ Relationship: Member ---< Issues (IssueTransaction) >--- Book Notes: - IssueTransaction is an associative (weak) entity capturing the many-to-many borrow history between Member and Book. - FineAmount can be computed at return time as max(0, (ReturnDate - DueDate) * rate).	
2	Consider the following database schema: STUDENT(StudentID, StudentName, Program, City) ENROLLMENT(EnrollID, StudentID, CourseID, Semester) COURSE(CourseID, CourseName, Credits) Write SQL queries to perform the following: a) Display the names of students enrolled in the *"Database Systems"* course. b) Retrieve the list of distinct cities from which students belong. c) Display students who are enrolled in more than 3 courses Ans: a) Display the names of students enrolled in the "Database Systems" course. SELECT S.StudentName FROM STUDENT S JOIN ENROLLMENT E ON S.StudentID = E.StudentID JOIN COURSE C ON E.CourseID = C.CourseID WHERE C.CourseName = 'Database Systems'; (b) Retrieve the list of distinct cities from which students belong. SELECT DISTINCT City FROM STUDENT; (c) Display students who are enrolled in more than 3 courses. SELECT S.StudentID, S.StudentName, COUNT(E.CourseID) AS TotalCourses FROM STUDENT S JOIN ENROLLMENT E ON S.StudentID = E.StudentID GROUP BY S.StudentID, S.StudentName HAVING COUNT(E.CourseID) > 3;	[6]
Q.3	**a)** Given a relation that violates 1**NF, explain** how you would reorganize the structure to enforce atomicity and 1NF. Demonstrate your explanation using your own simple example. (8 Marks)	[16]

Ans:
A relation violates First Normal Form (1NF) when it contains multivalued attributes, unordered sets, repeating groups, or mixed data types.
To bring such a relation into 1NF, every attribute must contain only atomic (indivisible) values, and each field must store a single value of a single type.

Steps to enforce atomicity and convert to 1NF

1. Identify non-atomic attributes
   Check for fields that contain lists, sets, or multiple values (e.g., "Java, Python", "Pen, Pencil").
2. Split multivalued attributes
   Break composite or repeated values into separate rows or separate relations.
3. Ensure each attribute contains only one value
   Each cell must represent a single value, not a collection.
4. Create new relations if required
   If an attribute holds multiple values per entity, decompose the relation.
5. Maintain primary key integrity
   Ensure the resulting tables still have a valid primary key.

Example of Violating 1NF
Consider the following relation:
COURSE_REGISTRATION

StudentID	StudentName	CoursesRegistered
101	Aarti	DBMS, Java, Networks
102	Rohan	Python, AI

The attribute CoursesRegistered contains multivalued and unordered sets → violates 1NF.
Conversion into 1NF
To enforce atomicity, we create separate rows for each course:
COURSE_REGISTRATION_1NF

StudentID	StudentName	Course
101	Aarti	DBMS
101	Aarti	Java
101	Aarti	Networks
102	Rohan	Python
102	Rohan	AI

OR

b) Differentiate between BCNF and 3NF (8 Marks)

Ans:
Third Normal Form (3NF) and Boyce–Codd Normal Form (BCNF) are

higher-level normalization forms used to remove redundancy and update anomalies in relational database design. Although both aim to eliminate transitive and partial dependencies, BCNF is stricter than 3NF.

1. Definition
   - 3NF:
     A relation is in 3NF if it is in 2NF and every non-prime attribute is non-transitively dependent on the primary key.
     OR
     For every functional dependency X → Y, either
     - X is a superkey, or
     - Y is a prime attribute (part of a candidate key).
   - BCNF:
     A relation is in BCNF if for every functional dependency X → Y, X must be a superkey.
     No exceptions allowed.
2. Strictness
   - 3NF is less strict because it allows dependencies where the determinant is *not* a superkey if the dependent attribute is prime.
   - BCNF is stricter because the determinant must *always* be a superkey.
3. Handling of Anomalies
   - 3NF reduces most anomalies, but some anomalies may still remain if a dependency violates BCNF conditions.
   - BCNF eliminates all redundancy-based anomalies, providing a more robust design.
4. Functional Dependency Condition
   - 3NF: Allows $X \rightarrow Y$ even if X is not a superkey, provided Y is a prime attribute.
   - BCNF: Does not allow any dependency where X is not a superkey.
5. Decomposition
   - 3NF decomposition is always possible without loss and often dependency-preserving.
   - BCNF decomposition is always lossless, but may not preserve all functional dependencies.

c) Given the relation:
R2(ProjectID, EmployeeID, EmpName, ProjectName, HoursWorked)
Functional Dependencies:
- ProjectID, EmployeeID → HoursWorked
- EmployeeID → EmpName
- ProjectID → ProjectName

Candidate Key: (ProjectID, EmployeeID)
Determine the highest normal form of relation R2. (8 marks)

Ans:
Assume attributes are atomic → R2 is in 1NF.
2NF requires no partial dependency of a non-prime attribute on a proper subset

of a candidate key.

- EmployeeID → EmpName — EmployeeID is a part of the key and determines non-prime attribute EmpName → partial dependency (violates 2NF).
- ProjectID → ProjectName — ProjectID (part of key) determines ProjectName → partial dependency (violates 2NF).

Because of these partial dependencies, R2 is NOT in 2NF.
Since R2 is in 1NF but fails 2NF, the highest normal form satisfied by R2 is 1NF.

OR

d) Demonstrate Domain, Referential Integrities and Enterprise Constraints with suitable example.
Ans:
1. Domain Integrity
Domain integrity ensures that each attribute in a relation has a valid and permissible value according to its data type, range, or format.
Example
Consider the relation:
STUDENT(StudentID, StudentName, Age, Email)

Domain Rule	Attribute	Valid Example	Invalid Example
Age must be between 18–30	Age	21	45 (violates domain)
Email must contain "@"	Email	raju@gmail.com	rajugmail.com
StudentName must be text	StudentName	Priya	123 (invalid type)

Valid insertion:
INSERT INTO STUDENT VALUES (101, 'Anita', 20, 'anita@gmail.com');
Invalid insertion violating domain:
INSERT INTO STUDENT VALUES (102, 'Rohan', 35, 'rohan123.com');

- Age = 35 (out of allowed range)
- Email missing "@"

Thus, domain integrity is violated.
2. Referential Integrity
Referential integrity ensures that a foreign key in one table must refer to an existing primary key in another table.
Example
DEPARTMENT(DeptID, DeptName)
EMPLOYEE(EmpID, EmpName, DeptID)
DEPARTMENT Table:

DeptID	DeptName
10	CS
20	IT

EMPLOYEE Table:

	EmpID EmpName DeptID 501 Meera 10 502 Rahul 30 INSERT INTO EMPLOYEE VALUES (502, 'Rahul', 30); Since 30 is not present in DEPARTMENT, the foreign key is invalid → referential integrity violation. 3. Enterprise Constraints Enterprise constraints (business rules) are organization-specific rules that the database must enforce. These are not defined by the DBMS structure but by business policies. Example Business Rule "An employee cannot work on more than one full-time project at the same time." Tables: PROJECT_ASSIGN(EmpID, ProjectID, AssignmentType) (AssignmentType = 'Full-Time' or 'Part-Time') Valid Records: EmpID ProjectID AssignmentType 501 P01 Full-Time Invalid Insertion: INSERT INTO PROJECT_ASSIGN VALUES (501, 'P02', 'Full-Time'); Employee 501 is already assigned to a Full-Time project → Assigning another Full-Time project violates the enterprise rule. To enforce this constraint: • Use triggers, or • Application-level validation.	
Q.4	a) Describe the ACID properties of a transaction. How does each property contribute to maintaining database consistency and reliability? Ans: **ACID Properties of a Transaction** A transaction in a database system is a sequence of operations performed as a single logical unit of work. The correctness and reliability of transactions are ensured by the ACID properties, namely Atomicity, Consistency, Isolation, and Durability. 1. **Atomicity** Atomicity ensures that a transaction is treated as an indivisible unit. Either all the operations of the transaction are successfully executed, or none of them are applied to the database. If any failure occurs during transaction execution, the database is rolled back to its previous consistent state. This prevents partial updates and helps maintain database reliability. 2. **Consistency** Consistency ensures that a transaction brings the database from one	[16]

valid (consistent) state to another valid state. All integrity constraints, rules, and triggers defined on the database must be satisfied before and after the execution of a transaction. This property guarantees that the database never violates its integrity constraints.

3. **Isolation**
   Isolation ensures that the concurrent execution of multiple transactions does not interfere with each other. Each transaction is executed as if it is the only transaction in the system. Intermediate results of one transaction are not visible to others until the transaction is committed, thereby preventing anomalies such as dirty reads, lost updates, and uncommitted data access.
4. **Durability**
   Durability ensures that once a transaction is successfully committed, its effects are permanently recorded in the database, even in the presence of system failures such as power outages or crashes. This is typically achieved through logging and recovery mechanisms, ensuring long-term reliability of the database.

OR

b) Explain the concept of lock-based concurrency control. Why are locks required to manage concurrent transactions in a database system?

Ans:

Lock-Based Concurrency Control

Lock-based concurrency control is a technique used in database management systems to control the concurrent execution of transactions by regulating access to data items through **locks**. Before a transaction can read or write a data item, it must first obtain an appropriate lock on that item. This ensures that only authorized transactions can access the data at a given time.

Locks are mainly of two types:

- **Shared (S) lock**: Allows a transaction to read a data item. Multiple transactions can hold shared locks on the same data item simultaneously.
- **Exclusive (X) lock**: Allows a transaction to read and write a data item. Only one transaction can hold an exclusive lock on a data item at a time.

Why Locks Are Required in Concurrent Transactions

Locks are required to manage concurrent transactions for the following reasons:

1. **To prevent data inconsistency**
   Without locks, multiple transactions may simultaneously read or write the same data item, leading to problems such as lost updates, dirty reads, and uncommitted data access.
2. **To maintain isolation**
   Locks ensure that transactions execute as if they are isolated from each

other. Intermediate results of one transaction are not visible to other transactions until the transaction is completed.

3. **To ensure serializability**
   Lock-based protocols help ensure that the effect of concurrent transactions is equivalent to some serial execution, thereby preserving database correctness.
4. **To handle conflicting operations**
   Locks control access to data items when read–write or write–write conflicts occur, ensuring correct execution order.
5. **To support ACID properties**
   By enforcing isolation and consistency, lock-based concurrency control plays a key role in maintaining the ACID properties of transactions.

c) Explain what is meant by a transaction schedule. Differentiate between serial and non-serial schedules with suitable explanation.

Ans:

Transaction Schedule

A **transaction schedule** is the chronological order in which the operations (read, write, commit, abort) of one or more transactions are executed by the database management system. It shows how multiple transactions are interleaved when they are executed concurrently, while preserving the order of operations within each individual transaction.

Serial Schedule

A **serial schedule** is one in which transactions are executed one after another, without any interleaving of operations. One transaction completes all its operations before the next transaction starts.

Explanation:

If there are two transactions T1 and T2, in a serial schedule either:

- T1 executes completely first, followed by T2, or
- T2 executes completely first, followed by T1.

Example:

T1: Read(A) → Write(A) → Commit
T2: Read(B) → Write(B) → Commit
Here, T2 starts only after T1 finishes.

Characteristics:

- Simple and easy to understand
- Always maintains database consistency
- Low concurrency and poor performance

Non-Serial Schedule

A **non-serial schedule** is one in which the operations of multiple transactions are interleaved. Transactions execute concurrently, but the internal order of operations of each transaction is preserved.

Explanation:

Operations of T1 and T2 are mixed to improve system performance and resource utilization.

Example:
T1: Read(A)
T2: Read(B)
T1: Write(A)
T2: Write(B)
T1: Commit
T2: Commit

Characteristics:

- Allows concurrent execution of transactions
- Improves system throughput and resource utilization
- May cause inconsistency if not properly controlled
- Requires concurrency control techniques

Difference Between Serial and Non-Serial Schedules

Aspect	Serial Schedule	Non-Serial Schedule
Execution	One transaction at a tim	Transactions execute concurrentl
Interleavin	No interleaving	Interleaving of operations
Consistenc	Always consistent	Consistent only if serializable
Performanc	Low	High
Complexity	Simple	Requires concurrency control

OR

d) Explain what a deadlock is in a database system. How does deadlock handling ensure smooth execution of transactions?

Ans:

A deadlock in a database system occurs when two or more transactions are waiting indefinitely for each other to release locks on data items. In this situation, none of the transactions can proceed because each transaction holds a lock that the other transaction needs, resulting in a circular wait condition.

Example:

- Transaction T1 holds a lock on data item A and waits for a lock on data item B.
- Transaction T2 holds a lock on data item B and waits for a lock on data item A.
  This circular waiting leads to a deadlock.

Deadlock Handling and Smooth Transaction Execution

Deadlock handling ensures smooth execution of transactions by identifying and resolving deadlock situations so that the database system does not remain blocked. This is achieved through the following approaches:

1. **Deadlock Prevention**
   Deadlock prevention techniques avoid deadlocks by ensuring that at

	least one of the necessary conditions for deadlock (such as circular wait) never occurs. This is done using methods like timestamp-based ordering (wait-die and wound-wait). 2. **Deadlock Detection** In deadlock detection, the system periodically checks for deadlocks using structures like a wait-for graph. If a cycle is detected, it indicates a deadlock. 3. **Deadlock Recovery** Once a deadlock is detected, the system resolves it by aborting one or more transactions (victims) involved in the deadlock and releasing their locks so that other transactions can continue.	
Q.5	a) Analyze the differences between centralized database architecture and client–server database architecture in terms of data access, performance, scalability, and fault tolerance. Ans: Centralized and client–server database architectures differ significantly in how data is accessed, how the system performs under load, how it scales, and how it handles failures. The key differences are analyzed below: 1. Data Access In a centralized database architecture, all data and the database management system reside on a single central machine. Users access data through terminals or thin clients, and all processing is performed at the central server. This creates a single point of data access. In a client–server database architecture, the database server stores and manages the data, while multiple client machines handle user interfaces and some application logic. Clients send queries to the server, which processes them and returns results. This distributed access allows better utilization of system resources. Analysis: Client–server architecture provides more flexible and efficient data access by distributing workload between clients and server, whereas centralized systems rely entirely on one machine. 2. Performance In a centralized architecture, performance may degrade as the number of users increases because all requests are processed by a single system. Resource contention for CPU, memory, and I/O can become a bottleneck. In a client–server architecture, performance is generally better since client machines handle presentation and partial processing, reducing the server's	[16]

	workload. Concurrent processing and better resource utilization improve overall throughput. Analysis: Client–server systems offer higher performance for multi-user environments, while centralized systems are suitable only for small-scale applications. 3. Scalability Centralized systems have limited scalability. Scaling often requires upgrading the central server (vertical scaling), which can be costly and has physical limits. Client–server systems are more scalable. Additional clients or servers can be added to handle increased load (horizontal scaling), making it suitable for growing organizations. Analysis: Client–server architecture supports better scalability due to its distributed nature, whereas centralized architecture is rigid and difficult to scale. 4. Fault Tolerance In a centralized architecture, failure of the central server results in complete system downtime, as it is a single point of failure. In a client–server architecture, failure of a client does not affect other clients. Although server failure impacts the system, backup servers and replication techniques can be used to improve fault tolerance. Analysis: Client–server architecture provides better fault tolerance and system availability compared to centralized architecture. OR b) Analyze the need for parallel databases. How does parallel database architecture improve query performance and throughput compared to traditional database systems? Ans: With the rapid growth of data volume and the increasing number of users, traditional (single-processor) database systems often fail to meet performance and scalability requirements. Parallel databases are designed to overcome these limitations by using multiple processors, disks, and memory units to execute database operations concurrently. Need for Parallel Databases 1. Handling Large Data Volumes	

Modern applications generate massive amounts of data. Traditional databases process queries sequentially, leading to long response times. Parallel databases divide data across multiple disks and nodes, enabling faster data access.

2. Improved Query Response Time
   Complex queries involving joins, aggregations, and scans are time-consuming in traditional systems. Parallel databases execute different parts of a query simultaneously, significantly reducing execution time.
3. Higher Transaction Throughput
   In multi-user environments, traditional systems become bottlenecks due to limited processing resources. Parallel databases allow multiple transactions to be processed concurrently, increasing overall system throughput.
4. Scalability
   Parallel databases support scalable architectures where additional processors or nodes can be added to handle increased workload, unlike traditional systems that rely mainly on vertical scaling.

How Parallel Database Architecture Improves Performance and Throughput

1. Data Partitioning
   Data is divided (horizontally or vertically) across multiple disks or nodes. Parallel processing allows simultaneous access to these partitions, reducing I/O time compared to accessing a single storage unit.
2. Parallel Query Execution
   Query operations such as selection, projection, join, and aggregation are executed in parallel on different processors. Each processor works on a subset of data, and results are combined, improving query performance.
3. Inter-Query and Intra-Query Parallelism
   - *Inter-query parallelism* allows multiple queries to execute at the same time.
   - *Intra-query parallelism* allows a single query to be divided into multiple tasks and executed concurrently.
     Both types increase system efficiency and throughput.
4. Reduced Resource Contention
   By distributing workload across multiple processors and disks, parallel databases minimize contention for CPU, memory, and I/O resources, leading to better utilization.

Comparison with Traditional Database Systems
Traditional databases process tasks sequentially on a single processor, resulting in limited performance and scalability. In contrast, parallel database architectures leverage concurrent processing and distributed storage, leading to faster query execution, higher throughput, and better scalability.

c) Analyze the need for distributed databases over centralized database systems. How do distributed databases address data availability and performance requirements? (8 marks)

Ans:
Centralized database systems store all data at a single location and process all requests on one central server. While simple to manage, such systems face serious limitations when used in large-scale, geographically distributed, or high-availability applications. Distributed databases overcome these limitations by distributing data across multiple sites connected through a network.
Need for Distributed Databases over Centralized Systems

1. Geographical Distribution of Users
   Modern organizations operate across multiple locations. A centralized database causes high access latency for remote users, whereas distributed databases place data closer to users, reducing response time.
2. Improved Data Availability
   Centralized systems suffer from a single point of failure. If the central server fails, the entire system becomes unavailable. Distributed databases replicate data across multiple sites, ensuring continued availability even if one site fails.
3. Scalability Requirements
   Centralized systems have limited scalability and often rely on expensive hardware upgrades. Distributed databases allow horizontal scaling by adding new sites or nodes as data and workload increase.
4. Local Autonomy and Reliability
   Distributed databases allow each site to manage its local data independently while still participating in a global database, improving system reliability and administrative flexibility.

How Distributed Databases Address Availability and Performance

1. Data Replication
   Copies of data are stored at multiple sites. If one site becomes unavailable due to failure, another site can continue serving requests, thereby improving fault tolerance and data availability.
2. Data Fragmentation and Allocation
   Data is divided into fragments and stored at different sites based on usage patterns. This enables faster local access and reduces unnecessary data transfer over the network.
3. Parallel Query Processing
   Queries can be executed simultaneously at multiple sites. Each site processes its local data fragment, and the results are combined, leading to faster query execution.
4. Load Balancing
   Workload is distributed across multiple nodes, preventing any single node from becoming a performance bottleneck and improving overall system throughput.

Comparison with Centralized Systems
In centralized databases, all requests compete for the same resources, leading to performance degradation and downtime risks. Distributed databases, by contrast, eliminate single points of failure, reduce access latency, and provide

higher throughput through parallelism and data distribution.

OR

d) Analyze the top-down and bottom-up distributed database design processes. How does each approach affect data fragmentation, allocation, and system scalability?

Ans:

Distributed database design defines how data is fragmented, allocated, and managed across multiple sites. Two major design approaches are top-down and bottom-up, each affecting data organization and system scalability in different ways.

1. Top-Down Design Approach

In the top-down approach, the design starts from a global perspective. A global conceptual schema is first created based on overall system requirements, and then it is progressively refined for distribution.

Process

- Define global conceptual schema
- Identify user views and access patterns
- Perform data fragmentation (horizontal, vertical, or hybrid)
- Allocate fragments to different sites
- Define replication and local schemas

Impact on Design Factors

- Data Fragmentation:
  Fragmentation is systematic and well-planned, based on global access requirements. This leads to optimal fragmentation that minimizes remote data access.
- Data Allocation:
  Allocation decisions consider global workload, communication cost, and data locality, resulting in efficient placement of fragments.
- System Scalability:
  The approach supports high scalability because data distribution is planned in advance. However, redesign may be required if new sites or major requirements are added later.

Strengths

- Better global optimization
- High data consistency
- Suitable for new distributed database systems

2. Bottom-Up Design Approach

In the bottom-up approach, design starts from existing local databases at different sites, which are then integrated into a global distributed database.

Process

- Identify existing local schemas
- Integrate schemas into a global schema
- Resolve schema conflicts (naming, structure, data types)

- Define global views and mappings

Impact on Design Factors

- Data Fragmentation:
  Fragmentation already exists implicitly due to separate local databases. It may not be optimal and often reflects local needs rather than global efficiency.
- Data Allocation:
  Allocation is fixed based on existing data locations, offering limited flexibility for redistribution.
- System Scalability:
  Scalability is easier when adding new sites, as new local databases can be integrated without redesigning the entire system. However, performance optimization may be limited.

Strengths

- Suitable for integrating legacy systems
- Less initial redesign effort
- Supports organizational autonomy

3. Comparative Analysis

Aspect	Top-Down Approach	Bottom-Up Approach
Design Start	Global schema	Local schemas
Fragmentatio	Planned and optimized	Pre-existing, less optimized
Allocation	Flexible and workload-based	Fixed, location-dependent
Scalability	High, but redesign may be needed	Easy site addition, limited optimization
Suitability	New systems	Existing/legacy systems