REG.NO.:

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

NAME OF THE SCHOOL
CONTINUOUS ASSESSMENT TEST - I
WINTER SEMESTER 2024-2025

SLOT: C1+TC1

Programme Name & Branch : M. Tech. Integrated (CSE Core and Data Science)
Course Code and Course Name : CSI 2004, Advanced Database Management Systems
Faculty Name(s) : Prof. JAYASHREE J, Prof. SIVAKUMAR V, Prof. NITHYA N S, Prof. SAURABH AGRAWAL, Prof. ANURADHA D
Class Number(s) : VL2024250503174, 3184, 3169, 2053, 3177
Date of Examination : 29 January 2025
Exam Duration : **90 minutes** **Maximum Marks: 50**

General instruction(s):

- **Assumptions outside of what is stated in the question must be explained.**
- **Any assumptions made should not alter the given question.**
- **Answer All Questions**
- M - Max mark; CO – Course Outcome;
- **CO1: Acquire the concept of physical database design and tuning**
- **CO2: Learn the concept of parallel and distributed database**

Q. No	Question	M	CO	BL
1.	Consider the following relational schema for a library database: **Book** (Title, Author, Catalog_no, Publisher, Year, Price, bookCoverType, contractDate) **Collection** (Title, Author, Catalog_no) Assume {Author, Title} is the key for both relations, and following are additional functional dependencies: I. Title,Author --> Catalog_no II. Catalog_no --> Publisher, Year, bookCoverType III. Publisher, bookCoverType --> Price IV. Author --> contractDate **a.** Explain what normal form the relation is in. **(2 Marks)** **b.** Apply normalization until the 3rd NF. State reasons behind each normalization. **(8 Marks)**	10	CO1	5
Sol 1.	a. It's in 1 NF, because no multi valued/composite attribute and no nested relations. b. It is not in 2NF as there is partial dependency due to FD IV, therefore normalizing to 2NF: Collection(Title,Author,Catalog_no) Book(Title, Author, Catalog_no, Publisher, Year, Price, bookCoverType) Author_Info(Author, ContractDate) Now normalizing to 3NF as there is still transitive dependency in "Book" table due to Fd II and III. Collection(Title,Author,Catalog_no) Author_Info(Author, ContractDate) Book(Title, Author, Catalog_no) Catalog(Catalog_no, Publisher, year, bookcovertype)			

	Price_info(Publisher, bookcovertype, price)			
2.	Consider the BANK Entity Relationship schema and suppose that it is necessary to keep track of different types of **ACCOUNTS (SAVINGS_ACCTS, CHECKING_ACCTS, ...)** and **LOANS (CAR_LOANS, HOME_LOANS, ...)**. Suppose that it is also desirable to keep track of each account's **TRANSACTIONS (deposits, withdrawals, checks, ...)** and each loan's **PAYMENTS**; both of these include the amount, date, time,... Modify the BANK schema, using Extended ER concepts of specialization and generalization using following assumptions. 1. **Accounts** can be of different types, such as SAVINGS_ACCTS, CHECKING_ACCTS, etc. We need to track transactions like deposits, withdrawals, and checks. 2. **Loans** can also be of different types, such as CAR_LOANS, HOME_LOANS, etc. We need to track payments made towards these loans. 3. **Transactions** and **Payments** have common attributes like amount, date, and time. These should be generalized into an entity that can track both types of activities (transaction and loan payment). 4. **Entities** that are specialized from a general type (such as account or loan) will inherit the common attributes and relationships but may also have specialized attributes.	10	CO1	4
Sol. 2	**1. Generalization for ACCOUNTS and LOANS** We can create a general ACCOUNT and LOAN entity, and then apply specialization to define the specific types of accounts and loans. • **Generalized ACCOUNT** entity could have the following attributes: o account_id (Primary Key) o balance o creation_date o account_holder (relationship to customer) • **Specialized subtypes of ACCOUNT**: o SAVINGS_ACCTS o CHECKING_ACCTS o Other account types as needed (e.g., BUSINESS_ACCTS). • **Generalized LOAN** entity could have the following attributes: o loan_id (Primary Key) o loan_amount o interest_rate o start_date o borrower (relationship to customer) • **Specialized subtypes of LOAN**: o CAR_LOANS o HOME_LOANS o Other loan types (e.g., STUDENT_LOANS). **2. Specialization for TRANSACTIONS and PAYMENTS**			

Since both **TRANSACTIONS** (for accounts) and **PAYMENTS** (for loans) share common attributes, we can create a generalized entity called FINANCIAL_ACTIVITY that will capture both types of activities.

- **Generalized FINANCIAL_ACTIVITY** entity could have the following attributes:
  - activity_id (Primary Key)
  - amount
  - date
  - time
  - activity_type (either 'transaction' or 'payment', depending on the activity)
- **Specialized subtypes of FINANCIAL_ACTIVITY**:
  - TRANSACTIONS (includes deposits, withdrawals, checks)
    - Relationships to ACCOUNT (this entity records the account involved in the transaction).
  - PAYMENTS (includes loan payments)
    - Relationships to LOAN (this entity records the loan involved in the payment).

3. Relationships

- **Account-Transaction Relationship**:
  - Each ACCOUNT can have many TRANSACTIONS.
  - The TRANSACTIONS entity would include a foreign key reference to ACCOUNT to establish this relationship.
- **Loan-Payment Relationship**:
  - Each LOAN can have many PAYMENTS.
  - The PAYMENTS entity would include a foreign key reference to LOAN to establish this relationship.

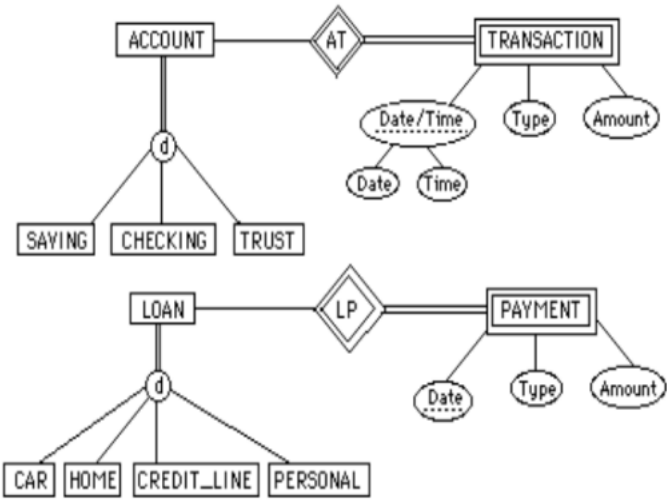
4. Modified ER Diagram (Conceptually)

Here's a high-level conceptual structure for the modified schema:

- **Entities**:
  - ACCOUNT (generalized entity)
    - Specialized: SAVINGS_ACCTS, CHECKING_ACCTS, ...
  - LOAN (generalized entity)
    - Specialized: CAR_LOANS, HOME_LOANS, ...
  - FINANCIAL_ACTIVITY (generalized entity)
    - Specialized: TRANSACTIONS, PAYMENTS
- **Relationships**:
  - ACCOUNT ↔ TRANSACTIONS (1 to many)
  - LOAN ↔ PAYMENTS (1 to many)

Diagram Description:

- ACCOUNT and LOAN will be connected to their respective specialized types through a **generalization hierarchy**.
- FINANCIAL_ACTIVITY will have two specializations: TRANSACTIONS and PAYMENTS.
- TRANSACTIONS and PAYMENTS will each have foreign keys pointing to their parent entities (ACCOUNT or LOAN).

				
3.	Explain the three types of parallel database architectures: Shared Memory, Shared Disk, and Shared Nothing. Describe the key features and characteristics of each architecture. For the financial institution, which parallel database architecture would be most suitable for the following purposes: • Real-time transaction processing • High availability and fault tolerance • Scalability for future growth in data and transactions	10	CO2	2
Sol 3	**Types of Parallel Database Architectures:** **a. Shared Memory Architecture:** • **Key Features:** o In a **Shared Memory** architecture, multiple processors or CPUs share a common memory space (RAM). All processors have access to the same memory, and the system is tightly coupled. o Data is stored in a central memory location, and processors communicate via shared memory for access to data. o The system uses memory access control mechanisms (e.g., locks, semaphores) to ensure that multiple processors do not access the same memory location simultaneously in conflicting ways. • **Characteristics:** o **Tightly coupled** system, with a small number of processors accessing a shared memory. o **Low latency** since all processors access the same memory, and data retrieval is fast. o Limited scalability because as more processors are added, contention for memory increases, leading to performance bottlenecks. • **Example Use Case:** A small to medium-sized system that requires high-speed access to data and has fewer processing nodes.			

b. Shared Disk Architecture:

- **Key Features:**
  - In a **Shared Disk** architecture, multiple processors or nodes share the same disk storage. Each processor has its own memory, but all processors access a common disk subsystem.
  - The disk is the central storage point for data, and processors communicate with the shared disk system to read/write data.
- **Characteristics:**
  - **Moderately coupled** system, with processors connected via a shared storage medium.
  - **Scalable** to some extent by adding more processors and disks, but scalability is limited by the shared disk system.
  - **High availability** because data is accessible to all nodes, and if one processor fails, others can continue to access the data on the shared disk.
  - Performance can degrade if too many processors compete for access to the shared disk.
- **Example Use Case:** Systems where high availability and fault tolerance are more critical than absolute scalability, such as in high-availability transaction systems.

c. Shared Nothing Architecture:

- **Key Features:**
  - In a **Shared Nothing** architecture, each processor or node has its own local memory and local storage (disk). There is no shared storage or memory between nodes.
  - Each processor operates independently, and communication between nodes is typically done through a network.
  - Data is partitioned across nodes, and each node processes only the data it owns. If a query requires data from multiple nodes, it is sent to the relevant nodes for processing.
- **Characteristics:**
  - **Loosely coupled** system, where each node is independent and operates on its own data.
  - **Highly scalable**, as new nodes can be added to the system with minimal impact on existing nodes. No contention for shared resources.
  - **Fault tolerance** is built-in because each node operates independently, and the failure of one node does not affect others.
  - **Network latency** can be an issue when nodes need to communicate, especially in large distributed systems.
- **Example Use Case:** Large-scale distributed systems that require horizontal scalability, such as big data analytics platforms, cloud-based applications, and systems that handle large volumes of data.

Suitability of Each Architecture for the Financial Institution:

a. Real-time Transaction Processing:

For real-time transaction processing, the financial institution needs low-latency data access and high transaction throughput. This is critical because financial systems handle many transactions per second, and delays can be costly.

- **Shared Memory**: Suitable for real-time processing with a small number of processors and minimal data contention. However, it is not ideal for large-scale systems due to scalability limitations.

	• **Shared Disk**: Offers a balance between scalability and real-time processing, but performance can degrade if too many processors compete for access to the shared disk. • **Shared Nothing**: Best suited for large-scale real-time transaction processing, as it allows for horizontal scalability and independent operation of each node. However, it may introduce network latency when processing transactions across multiple nodes. **Recommended**: **Shared Memory** for small-scale real-time systems, but for larger-scale systems, **Shared Nothing** would be ideal due to scalability and fault tolerance.			
4.	You are working on a large-scale database system where the data needs to be partitioned for better performance and scalability. The database contains a **Customer** table with the following attributes: • **CustomerID** (Primary Key) • **Name** • **Address** • **PhoneNumber** • **Email** There are 10,00000 records in the Customer table, and the database is facing issues related to query performance (specifically SELECT and UPDATE queries). To improve the performance, you decide to partition the table. You need to determine an appropriate partitioning strategy for the table based on the following requirements: 1. The queries often filter customers by **CustomerID**. 2. The data is relatively evenly distributed by **CustomerID**. 3. You want to minimize the number of records involved in a query operation. **What partitioning strategy would you use for this Customer table, and how would you partition the data? Also briefly explain the Round-Robin, Hash, and Range partitioning strategies?**			
Sol. 4	To solve this problem, we will consider **range partitioning** as the most suitable strategy for the **Customer** table, based on the given requirements. **1. Partitioning Strategy:** • **Range Partitioning**: This strategy divides the table into ranges based on a specific column, which in this case will be **CustomerID**. o Range partitioning works well when the data is evenly distributed and the queries often filter on the partitioning column (CustomerID). o The partitioning can be done by dividing the **CustomerID** into specific ranges (for example, 1 to 250,000 for one partition, 250,001 to 500,000 for another, and so on). **2. Why Range Partitioning?**			

- The **CustomerID** is evenly distributed, and it is a primary key, so range partitioning will allow the database to quickly narrow down which partition to query, reducing the number of records involved.
- Queries filtering by **CustomerID** will only need to search the relevant partition, improving query performance.
- **UPDATE** operations can be optimized by restricting the changes to the partition containing the record, reducing the amount of data that needs to be locked or modified.

3. How to Partition the Data:

Since there are **1 million records** in the **Customer** table, we can partition the data into 4 ranges as follows:

Partition Name	CustomerID Range	Number of Records
Partition 1	1 to 250,000	250,000
Partition 2	250,001 to 500,000	250,000
Partition 3	500,001 to 750,000	250,000
Partition 4	750,001 to 1,000,000	250,000

5.	What are the similarities and differences between centralized and distributed database in context with architecture, functioning, and applications.	10	CO2	2

Sol. 5

Centralized Database

Architecture
1. Single Server: All data is stored on a single server.
2. Centralized Control: A single database management system (DBMS) controls access to the data.

Functioning
1. Data Storage: All data is stored in a single location.
2. Data Retrieval: Clients send requests to the central server, which processes and responds to queries.
3. Data Management: The central DBMS manages data consistency, security, and backup.

Applications
1. Small-Scale Applications: Suitable for small-scale applications with limited users and data.
2. Simple Transactional Systems: Suitable for simple transactional systems, such as banking or retail.

Distributed Database

Architecture
1. Multiple Servers: Data is split across multiple servers, each storing a portion of the data.
2. Decentralized Control: Each server may have its own DBMS, and data is managed in a decentralized manner.

Functioning

	1. Data Distribution: Data is split across multiple servers using various distribution techniques (e.g., horizontal partitioning). 2. Data Retrieval: Clients send requests to any server, which may redirect the request to another server or process the query locally. 3. Data Management: Each server manages its local data, and a global DBMS may be used to manage data consistency and integrity across servers. Applications 1. Large-Scale Applications: Suitable for large-scale applications with many users and massive amounts of data. 2. Complex Transactional Systems: Suitable for complex transactional systems, such as financial trading platforms or e-commerce websites. 3. Big Data Analytics: Suitable for big data analytics, as distributed databases can handle massive amounts of data and scale horizontally. **Similarities** 1. Data Management: Both centralized and distributed databases require data management, including data consistency, security, and backup. 2. Query Processing: Both types of databases process queries, although distributed databases may require more complex query processing due to data distribution. **Differences** 1. Scalability: Distributed databases are more scalable than centralized databases, as they can handle increasing amounts of data and users by adding more servers. 2. Fault Tolerance: Distributed databases are more fault-tolerant than centralized databases, as data is replicated across multiple servers, ensuring availability even in the event of server failure. 3. Complexity: Distributed databases are more complex than centralized databases, requiring more sophisticated data management and query processing techniques. 4. Data Consistency: Distributed databases may face challenges in maintaining data consistency across servers, whereas centralized databases have a single point of control for data consistency.

****