Department of Computer Science & engineering

Question Bank

Concepts in Advance Database Systems (CS346)

Unit-3

1. Differentiate between centralized and distributed database with their advantages and disadvantages.

Centralized databases and distributed databases refer to two distinct architectures for storing and managing data.

A centralized database refers to a system where all data is stored in a single location and managed by a single entity. This approach is often used in small organizations or for specific applications where the data is not complex or changing frequently.

On the other hand, a distributed database is a system where data is stored across multiple computers or servers in a network, and managed by multiple entities. This approach is commonly used in large organizations or for applications that require high availability and scalability.

Advantages of Centralized Databases:

- Easier to manage as there is only one database to maintain
- Easier to ensure data consistency as all data is stored in one location
- Easier to secure as there is only one point of access
- Simpler backup and recovery processes as there is only one database to backup and recover

Disadvantages of Centralized Databases:

- Limited scalability due to the single point of access and data storage
- Higher risk of data loss in case of a system failure
- Higher risk of data corruption due to the centralization of data
- Limited availability due to the reliance on a single system

Advantages of Distributed Databases:

- Higher scalability as data is distributed across multiple servers
- Higher availability as multiple servers can serve requests in case of system failure

- Higher performance as requests can be processed locally on the server
- Reduced risk of data loss as data is distributed across multiple servers

Disadvantages of Distributed Databases:

- More complex to manage and maintain as multiple servers are involved
- More difficult to ensure data consistency as data is distributed across multiple servers
- More complex backup and recovery processes as data is distributed across multiple servers
- Higher security risks due to the distributed nature of data.

2. What is the difference between heterogeneous and homogeneous databases.

Heterogeneous and homogeneous databases are two different types of database systems based on their underlying architecture and data storage characteristics.

Homogeneous databases are databases where all the data is stored in the same format and managed by the same database management system (DBMS). In other words, a homogeneous database is a single database system that stores data in a consistent manner. This means that all the data stored in a homogeneous database has the same data type, data structure, and the same schema. An example of a homogeneous database is a database that is entirely managed by a single vendor or a single DBMS software.

In contrast, heterogeneous databases refer to databases where data is stored in different formats or managed by different DBMS software. A heterogeneous database can include data in different formats, data structures, or data models, and may be managed by different vendors. An example of a heterogeneous database is a database that includes data in different formats, such as XML, JSON, or CSV, or a database that is managed by different DBMS software, such as Oracle, MySQL, or Microsoft SQL Server.

The main difference between homogeneous and heterogeneous databases is the level of data consistency and the degree of complexity involved in managing and integrating data. In homogeneous databases, the data is consistent and easy to manage, but it may not be suitable for managing complex data types or integrating data from different sources. In contrast, heterogeneous databases can handle complex data types and data integration from different sources, but they are more complex to manage and maintain due to the variety of data formats and management systems involved.

3. What do you mean by data replication. Write the advantages and disadvantages of data replication.

Data replication is the process of creating and maintaining multiple copies of the same data in different locations, usually in distributed database systems. The primary purpose of data

replication is to improve data availability, reduce data access latency, and enhance fault tolerance in case of system failures.

Advantages of Data Replication:

1. Improved data availability: Data replication can help improve data availability and reduce data access latency by making data accessible from multiple locations. This means that users can access the data from the nearest replica, which results in faster response times and improved performance.

2. Enhanced fault tolerance: Data replication can help improve fault tolerance in case of system failures. If one replica fails, other replicas can still provide access to the data, ensuring that the system remains available.

3. Scalability: Data replication can help improve scalability in distributed database systems. By creating multiple copies of the data, the workload can be distributed across multiple nodes, which helps improve performance and scalability.

4. Reduced network traffic: Data replication can help reduce network traffic and improve response times by providing data locally, reducing the need for data to be transferred across the network.

Disadvantages of Data Replication:

1. Data consistency: Data replication can lead to inconsistencies in data across different replicas. This is because updates made to one replica may not be immediately reflected in other replicas, which can result in data inconsistencies if the replicas are not properly synchronized.

2. Increased complexity: Data replication can increase the complexity of database management, as it requires additional resources to manage and synchronize multiple replicas.

3. Increased storage requirements: Data replication can result in increased storage requirements, as multiple copies of the same data must be stored in different locations.

4. Increased maintenance: Data replication requires additional maintenance and administration, which can increase the cost and complexity of database management.

4. What do you mean by data transparency?

Data transparency refers to the ability of users to access and understand data without any barriers or limitations. It means that data is open, accessible, and understandable to all users who have the appropriate permissions to access it.Data transparency is important for ensuring accountability, trust, and confidence in data-driven decision-making. It enables users to understand how data is collected, processed, and analyzed, and to verify the accuracy and validity of the data.

Data transparency can take several forms, including:

1. Accessible data: Data transparency requires that data is easily accessible to authorized users. This means that data should be stored in a centralized location that is easily accessible and that users have the necessary permissions to access the data.

2. Clear data documentation: Data transparency requires clear documentation of data, including the metadata, schema, and data dictionaries. This enables users to understand the structure, content, and meaning of the data.

3. Open data sharing: Data transparency requires open sharing of data, either publicly or within the organization. This enables users to access and analyze data independently and to verify the accuracy and validity of the data.

4. User-friendly data visualization: Data transparency requires user-friendly data visualization that enables users to understand data easily and quickly.

In summary, data transparency means that data is open, accessible, and understandable to all authorized users. It helps to promote accountability, trust, and confidence in data-driven decision-making, and enables users to verify the accuracy and validity of the data.

5. What is the role of transaction manager and transaction coordinator?

The transaction manager and transaction coordinator are two critical components of a database management system (DBMS) that are responsible for managing transactions in a database.

The transaction manager is responsible for ensuring the consistency, isolation, and durability of transactions. It manages the life cycle of transactions, including transaction initiation, execution, and completion. The transaction manager also ensures that transactions are atomic, meaning that they are treated as a single unit of work, and either complete entirely or rollback if any part of the transaction fails. Additionally, the transaction manager is responsible for ensuring that transactions are durable, meaning that once a transaction is committed, its effects are permanent and cannot be undone.

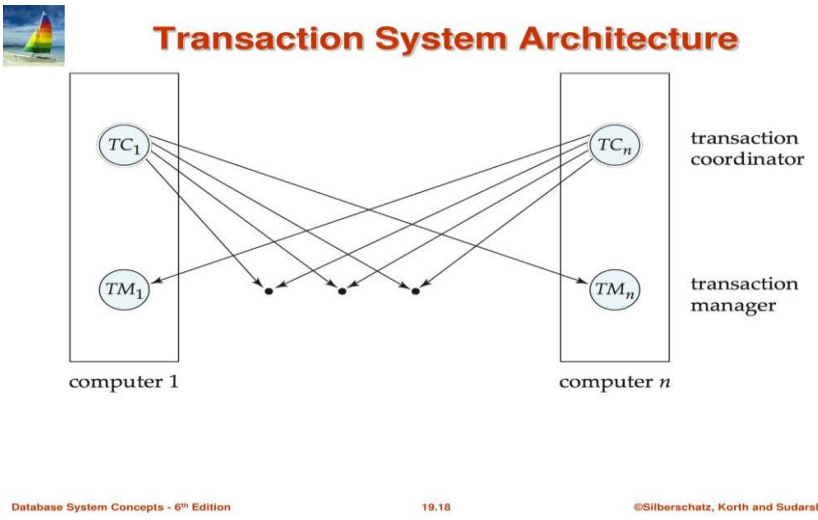
The transaction coordinator, on the other hand, is responsible for coordinating the actions of multiple transaction managers in a distributed database system. In a distributed database system, transactions can span multiple nodes, and each node may have its own transaction manager. The transaction coordinator ensures that all transaction managers agree on the outcome of the transaction and that the transaction is either committed or rolled back consistently across all nodes.

In summary, the transaction manager is responsible for managing transactions within a single node, ensuring their atomicity, consistency, isolation, and durability, while the transaction coordinator is responsible for coordinating transactions across multiple nodes in a distributed

database system, ensuring consistency across all nodes. Both components are critical to the proper functioning of a database system and ensure that data is consistent and reliable.

6. Draw and explain Transaction System Architecture.

The Transaction System Architecture (TSA) is a widely used architecture for implementing transaction processing systems. It consists of three main components: the client, the application server, and the database server.



Now, let's explain each component in more detail:

1. Client: The client is the user-facing component of the system. It is responsible for accepting user input and displaying output to the user. The client interacts with the application server to process transactions.

2. Application Server: The application server is responsible for processing transactions and providing an interface between the client and the database server. It processes requests from the client and interacts with the database server to retrieve or modify data as needed. The application server also enforces business rules and ensures that transactions are processed correctly.

3. Database Server: The database server is responsible for storing and retrieving data. It receives requests from the application server and retrieves or modifies data as needed. The database server ensures that data is consistent and maintains data integrity.

In the TSA architecture, transactions are initiated by the client, processed by the application server, and ultimately stored in the database server. The application server acts as an

intermediary between the client and the database server, ensuring that transactions are processed correctly and that data integrity is maintained.

Overall, the Transaction System Architecture is a proven approach for building transaction processing systems that can handle high volumes of concurrent transactions while maintaining data consistency and integrity.

7. Explain Two Phase Commit Protocol (2PC). How is it different from three Phase Commit Protocol (3PC).

The Two-Phase Commit Protocol (2PC) is a widely used protocol for ensuring the atomicity and durability of distributed transactions across multiple nodes in a distributed database system. It consists of two phases: the prepare phase and the commit phase.

Here is how the 2PC works:

1. Prepare Phase: In the prepare phase, the transaction coordinator sends a prepare message to all participants, asking them to vote on whether they are ready to commit the transaction. Each participant responds with either a "yes" vote, indicating that it is ready to commit, or a "no" vote, indicating that it cannot commit the transaction. If any participant votes "no," the transaction is aborted.

2. Commit Phase: If all participants vote "yes," the transaction coordinator sends a commit message to all participants, asking them to commit the transaction. Once each participant receives the commit message, it commits the transaction and sends an acknowledgement back to the transaction coordinator. If any participant fails to commit or does not respond, the transaction coordinator aborts the transaction.

The Three-Phase Commit Protocol (3PC) is an extension of the 2PC that adds an additional phase, the pre-commit phase, to handle cases where the coordinator node fails.Here is how the 3PC works:

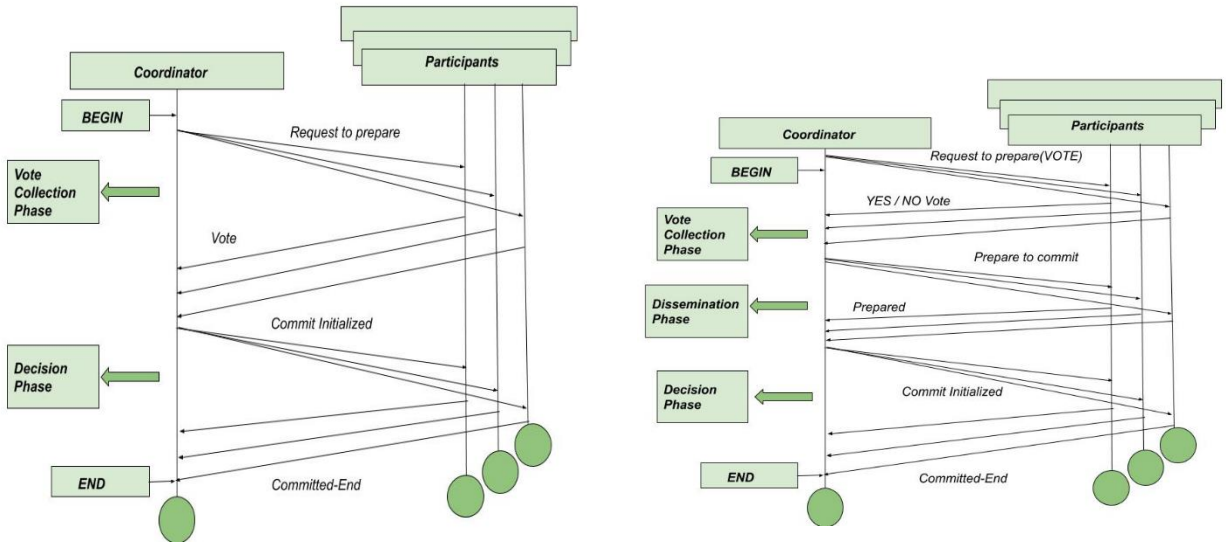
1. Prepare Phase: In the prepare phase, the transaction coordinator sends a prepare message to all participants, asking them to vote on whether they are ready to commit the transaction. Each participant responds with either a "yes" vote, indicating that it is ready to commit, or a "no" vote, indicating that it cannot commit the transaction. If any participant votes "no," the transaction is aborted.

2. Pre-Commit Phase: If all participants vote "yes," the transaction coordinator sends a pre-commit message to all participants, asking them to tentatively commit the transaction. Each participant responds with an acknowledgement.

3. Commit Phase: Once the transaction coordinator receives acknowledgements from all participants, it sends a commit message to all participants, asking them to commit the

transaction. Once each participant receives the commit message, it commits the transaction and sends an acknowledgement back to the transaction coordinator.

The key difference between 2PC and 3PC is that 3PC adds an additional phase to handle the case where the coordinator node fails during the prepare phase. In 3PC, the pre-commit phase ensures that all participants are aware of the transaction and have tentatively committed it, even if the coordinator node fails. This reduces the likelihood of a "hung" transaction, where a transaction is in limbo and cannot be completed due to a failed coordinator node. However, the additional phase also adds complexity to the protocol and can increase the latency and overhead of transaction processing.



8. What do you mean by Concurrency Control? Explain Distributed Lock Manager.

Concurrency control is a critical aspect of database management systems that ensures that multiple concurrent transactions accessing the same data do not interfere with each other, thereby maintaining data consistency and integrity.

Concurrency control is a technique used in database management systems to manage the access of multiple transactions to the same data concurrently without causing interference or inconsistency. Concurrency control is essential to ensure data consistency and accuracy, particularly in multi-user database systems.

Concurrency control mechanisms ensure that multiple transactions can access the same data item concurrently while maintaining consistency and integrity of the data. Some commonly used concurrency control techniques include locking, time-stamping, and optimistic concurrency control.

Locking: Locking is a widely used technique in concurrency control. In locking, a transaction obtains a lock on a data item before reading or modifying it. The lock prevents other transactions from accessing the same data item until the lock is released. There are two types of locks: shared locks, which allow multiple transactions to read the data, and exclusive locks, which prevent other transactions from reading or modifying the data.

Time-stamping: Time-stamping is another technique used in concurrency control. In this technique, each transaction is assigned a unique timestamp when it begins. The system checks the timestamps of transactions to determine which transactions can access the data item. The system ensures that a transaction can only access the data item if its timestamp is earlier than the timestamp of any other transaction that has modified the data.

Optimistic concurrency control: In optimistic concurrency control, transactions operate on a copy of the data, which is modified during the transaction. When the transaction completes, the system checks whether any other transaction has modified the same data item during the transaction. If not, the transaction is committed, and the modified data is written to the database. If there is a conflict, the transaction is aborted and must be retried.

Concurrency control is critical in ensuring data consistency and accuracy in multi-user database systems. By managing the access of multiple transactions to the same data concurrently, concurrency control mechanisms prevent interference and ensure that data is updated consistently and accurately.One widely used technique for concurrency control is the Distributed Lock Manager (DLM), which is used to manage locks across multiple nodes in a distributed database system.

The Distributed Lock Manager (DLM) is a component of a distributed database system that manages locks on data items to ensure that concurrent transactions do not interfere with each other. The DLM consists of two main components: the lock manager and the lock table.

Here is how the DLM works:

1. Lock Manager: The lock manager is responsible for managing locks on data items. It receives lock requests from transactions and grants or denies them based on the current state of the lock table.

2. Lock Table: The lock table is a data structure that tracks the state of locks on data items. It contains one entry for each data item in the system and tracks which transactions hold locks on each data item.

When a transaction needs to access a data item, it sends a lock request to the lock manager. The lock manager checks the lock table to see if any other transactions hold locks on the requested data item. If the data item is not currently locked, the lock manager grants the lock to the requesting transaction and updates the lock table. If the data item is already locked, the lock manager either denies the lock request or queues it until the data item is available.

When a transaction completes its work on a data item, it sends a lock release request to the lock manager. The lock manager updates the lock table to reflect the release of the lock.

In a distributed database system, the DLM is used to manage locks across multiple nodes in the system. Each node in the system has its own lock manager and lock table, but they all work together to ensure that locks are managed consistently across the system.

Overall, the Distributed Lock Manager is a critical component of concurrency control in distributed database systems. By managing locks on data items, the DLM ensures that concurrent transactions do not interfere with each other, thereby maintaining data consistency and integrity.

9. Explain Primary Copy and Quorum Consensus protocol for Concurrency Control.

Primary Copy and Quorum Consensus protocols are two widely used techniques in distributed database systems to ensure concurrency control. Here is a brief explanation of each protocol:

1. Primary Copy Protocol: In the Primary Copy Protocol, each data item is assigned a primary copy that is responsible for managing all updates to that item. When a transaction wants to update a data item, it sends the update request to the primary copy. The primary copy then applies the update and propagates it to all the other copies of the data item. The other copies of the data item are used for read-only access. The primary copy ensures that no other transactions can modify the data item while an update is being applied.

The primary copy protocol is simple to implement and provides strong consistency. However, it can suffer from performance issues and can become a bottleneck if a large number of transactions are accessing the same data item simultaneously.

2. Quorum Consensus Protocol: In the Quorum Consensus Protocol, each data item is replicated across multiple nodes or servers, and a quorum is defined as the minimum number of nodes required to access or modify a data item. For example, a quorum of two out of three nodes may be required to access or modify a data item.

When a transaction wants to access or modify a data item, it must obtain a quorum by communicating with a subset of nodes. The quorum ensures that any changes made to the data item are consistent across all nodes. The quorum consensus protocol ensures that multiple transactions can access or modify a data item simultaneously without causing conflicts.

The quorum consensus protocol is highly scalable and fault-tolerant, as it allows multiple nodes to participate in the consensus process. However, it can be more complex to implement than the primary copy protocol, and determining the optimal quorum size can be challenging.

In summary, both the primary copy and quorum consensus protocols are used in distributed database systems to ensure concurrency control. The primary copy protocol assigns a primary copy responsible for managing updates to a data item, while the quorum consensus protocol replicates data across multiple nodes and requires a quorum to access or modify a data item.

10. Explain Timestamping protocol for Concurrency Control.

Timestamping is a widely used protocol for concurrency control in database management systems. In timestamping, each transaction is assigned a unique timestamp when it begins, which is used to determine the order in which transactions are executed. Here is a detailed explanation of the Timestamping protocol:

1. Timestamp Assignment: Each transaction is assigned a unique timestamp when it begins. The timestamp can be based on the system clock or other methods to ensure uniqueness.

2. Timestamp Ordering: Transactions are ordered based on their timestamps. A transaction with a lower timestamp is considered to have started earlier than a transaction with a higher timestamp. This ordering is used to ensure that transactions are executed in the correct order.

3. Read and Write Timestamps: In timestamping, each data item in the database has two timestamps: a read timestamp and a write timestamp. The read timestamp indicates the time of the latest read operation on the data item, and the write timestamp indicates the time of the latest write operation on the data item.

4. Validation Phase: When a transaction wants to read or write a data item, it checks the read and write timestamps of the data item. If the transaction's timestamp is earlier than the read timestamp of the data item, it means that the transaction is trying to read a stale value of the data item. The transaction is aborted, and must be retried.

If the transaction's timestamp is earlier than the write timestamp of the data item, it means that another transaction has already modified the data item. The transaction is also aborted, and must be retried.

If the transaction's timestamp is later than both the read and write timestamps of the data item, it means that the transaction can safely read or modify the data item.

5. Update Phase: When a transaction wants to write a data item, it updates the write timestamp of the data item to its own timestamp. This update ensures that no other transactions can write to the data item while the current transaction is executing.

Timestamping protocol ensures serializability and provides a high degree of concurrency in database systems. It ensures that transactions access data items in a consistent order, and avoids conflicts between transactions that access the same data items. However, timestamping protocol may cause transaction aborts due to conflicts between transactions that access the same data items at the same time, which can lead to performance degradation in highly concurrent systems.

11. How deadlock handling done in distributed database. Also differentiate Local and Global Wait-For Graphs with example.

Deadlock handling is an important aspect of concurrency control in distributed databases, where multiple transactions may be accessing the same data items simultaneously. Deadlocks occur when two or more transactions are waiting for each other to release a resource, causing a circular dependency that cannot be resolved.

In distributed databases, deadlock handling can be done using two approaches: centralized deadlock detection and distributed deadlock detection.

Centralized deadlock detection involves a single server or process that monitors the transactions and resources in the entire distributed system to detect deadlocks. This approach requires a lot of communication between the centralized server and the nodes in the system, which can lead to performance overhead.

Distributed deadlock detection involves each node in the system maintaining a local wait-for graph that represents the dependencies between transactions and resources. When a transaction requests a resource that is currently being used by another transaction, the requesting transaction is added to the wait-for graph of the resource. If a cycle is detected in the graph, it means that a deadlock has occurred.

There are two types of wait-for graphs in distributed deadlock detection: local and global.

Local wait-for graphs are maintained by each node in the system, and represent the dependencies between transactions and resources within that node. For example, consider a distributed system with three nodes A, B, and C. Node A has transactions T1 and T2, node B has transactions T3 and T4, and node C has transactions T5 and T6. The local wait-for graph for node A might look like this:

T1 -> T2

T2 -> T1

And the local wait-for graph for node B might look like this:

T3 -> T4

T4 -> T3

Global wait-for graphs are constructed by merging the local wait-for graphs of all the nodes in the system. The global wait-for graph represents the dependencies between transactions and resources across the entire system. For example, if we combine the local wait-for graphs of the three nodes in the example above, the global wait-for graph might look like this:

T1 -> T2 -> T1

T3 -> T4 -> T3

In this case, the global wait-for graph shows that transactions T1 and T2 on node A, and transactions T3 and T4 on node B are involved in a deadlock. The distributed deadlock detection algorithm can then take action to resolve the deadlock, such as aborting one or more of the transactions involved.

Overall, distributed deadlock detection with local and global wait-for graphs is an effective way to handle deadlocks in distributed databases, as it allows each node to maintain its own local wait-for graph without relying on a centralized server, while also providing a global view of the dependencies between transactions and resources across the entire system.

12. Explain false cycle with proper example.

In database management systems, false cycles refer to situations where a cycle appears in a wait-for graph, leading the system to believe that a deadlock has occurred when in fact, no such deadlock exists.

A false cycle can occur when a transaction releases a resource and then immediately requests it again. This can happen when the transaction releases a shared resource, then later requests an exclusive lock on the same resource, or when the transaction releases a lock on a resource and then immediately requests another lock on the same resource.

For example, consider two transactions T1 and T2 that are accessing two data items, X and Y. T1 initially locks X and then requests a lock on Y, while T2 initially locks Y and then requests a lock on X. If T1 requests an exclusive lock on X before releasing the shared lock, and T2 requests an exclusive lock on Y before releasing the shared lock, a false cycle can occur:

1. T1 locks X (shared)

2. T2 locks Y (shared)

3. T1 requests X (exclusive)

4. T2 requests Y (exclusive)

5. T1 releases X (shared)

6. T2 releases Y (shared)

7. T2 requests X (shared)

8. T1 requests Y (shared)

At this point, a wait-for graph can be constructed that appears to show a cycle:

T1 -> X -> T2 -> Y -> T1

However, this is a false cycle, as both transactions can proceed without any deadlock occurring. In this case, T2 can release its shared lock on X, allowing T1 to acquire an exclusive lock on X and complete its transaction. Then, T1 can release its shared lock on Y, allowing T2 to acquire a shared lock on Y and complete its transaction.

False cycles can lead to unnecessary aborts and performance overhead in the system, as the distributed deadlock detection algorithm may incorrectly identify deadlocks that do not actually exist. To avoid false cycles, it is important to use concurrency control mechanisms that can detect and prevent unnecessary lock conflicts, such as optimistic concurrency control or multi-version concurrency control.

13.ACID properties

ACID properties are a set of four properties that ensure the reliability and consistency of database transactions. The acronym "ACID" stands for Atomicity, Consistency, Isolation, and Durability. Here is a brief explanation of each property:

1. Atomicity: This property ensures that a transaction is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction is rolled back to its previous state, ensuring that the database remains in a consistent state.

2. Consistency: This property ensures that a transaction brings the database from one valid state to another. The transaction must satisfy all the integrity constraints and business rules defined in the database schema, ensuring that the data remains consistent and accurate.

3. Isolation: This property ensures that multiple transactions can run concurrently without interfering with each other. Each transaction must be isolated from other transactions until it is completed. This isolation is achieved through the use of locks and other concurrency control mechanisms.

4. Durability: This property ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures, including power outages, crashes, and other catastrophic events. The database must be able to recover its state to the point of the last committed transaction after such failures.

Together, these four properties ensure that transactions in a database system are reliable, consistent, and durable. By adhering to the ACID properties, database systems can ensure that

data is stored and retrieved in a consistent and reliable manner, even in the face of multiple concurrent transactions and system failures.