ENSEM IMP DATABASE MANAGMENT SYSTEM UNIT -4

Q.1] State and explain the ACID Properties. During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain the situations when each state transition occurs.

ANS: The ACID properties stand for Atomicity, Consistency, Isolation, and Durability, which are crucial for ensuring the reliability of transactions within a database system.

1. **Atomicity:**
    - **This property ensures that a transaction is all or nothing. It means that either the entire transaction is completed successfully or none of its changes are applied to the database. There is no partial execution.**
2. **Consistency:**
    - **This property ensures that the database remains in a consistent state before and after the transaction. It means that the execution of a transaction should bring the database from one consistent state to another.**
3. **Isolation:**
    - **Isolation ensures that the execution of multiple transactions concurrently will result in the same outcome as if they were executed sequentially. It prevents interference between transactions.**
4. **Durability:**
    - **Once a transaction is committed, its changes are permanent and will persist even in the case of system failure. The changes become a permanent part of the database and are not lost.**

Regarding the states a transaction passes through:
A transaction typically passes through different states during its execution:

1. **Active:**
    - **The initial state of the transaction where it's executing and making changes to the database.**
2. **Partially Committed:**
    - **The transaction has executed all its operations successfully and is about to be finalized. It is at this point where the changes are not visible to other transactions yet.**
3. **Committed:**
    - **The transaction has completed successfully, and its changes are now visible to other transactions.**
4. **Failed:**
    - **The transaction has encountered an error or has been aborted due to some reason. This might happen at any point during its execution.**
5. **Aborted:**
    - **The transaction has been rolled back or reversed to its initial state due to a failure or intentional abort. The changes made by this transaction are discarded.**

State transitions occur due to various reasons:

- **Active to Partially Committed: This transition occurs when all the operations within the transaction have been executed successfully, and the transaction is ready to commit.**

- **Partially Committed to Committed: At this point, the changes made by the transaction are now finalized and visible to other transactions.**
- **Active to Aborted or Failed: If a transaction encounters an error during execution or is intentionally rolled back, it moves to the aborted or failed state.**
- **Any state to Aborted: This transition occurs when a failure is detected or when a rollback is initiated, reverting the changes made by the transaction.**

Q.2] Check whether following schedule is view serializable or not. Justify your answer. (Note : T1 & T2 are transactions). Also explain the concept of view equivalent schedules and conflict equivalent schedule considering the example schedule given below : T1 read (A) A := A – 50 write (A) read (B) B := B + 50 write (B) , T2 read (A) A := A – 50 read (A) temp :=A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B)

ANS: To determine if the given schedule is view serializable, I'll first analyze the schedule. View serializability determines if the end result (or the final state) of concurrent transactions could be achieved by serial execution of those transactions.

The given schedule is:

```
T1: read(A)
    A := A - 50
    write(A)
    read(B)
    B := B + 50
    write(B)
T2: read(A)
    A := A - 50
    read(A)
    temp := A * 0.1
    A := A - temp
    write(A)
    read(B)
    B := B + temp
    write(B)
```

Let's break it down into its corresponding operations:

1. **T1:**
    - **Reads A**
    - **Modifies A**
    - **Writes A**
    - **Reads B**
    - **Modifies B**
    - **Writes B**
2. **T2:**
    - **Reads A**
    - **Modifies A**
    - **Reads A again**
    - **Computes temp**
    - **Modifies A**
    - **Writes A**
    - **Reads B**
    - **Modifies B**
    - **Writes B**

Certainly! Imagine two people, T1 and T2, doing tasks on two items, A and B.

T1 starts by taking A, changes it, then also deals with B. Meanwhile, T2 also messes with A, then computes something based on A and changes it again, and finally, handles B too.

For things to work smoothly (be view serializable), if T1 does its job and then T2 does its tasks, the end result would be the same as if T2 went first and then T1 followed. But here, because T2's action depends on a temporary value (let's call it 'temp') that's derived from A, the final outcome depends on the order they work. So, the order they do things matters, and that's not good for view serializability.

As for conflict serializability, it's like making sure they work in an order that avoids stepping on each other's toes. But in this case, they're stepping on each other's toes a bit too much with conflicting actions, so it needs a better plan to avoid clashes between their tasks on A and B.

Q.3] Suppose a transaction Ti issues a read command on data item Q.How time-stamp based protocol decides whether to allow the operation to be executed or not using time-stamp based protocol of concurrency control. Explain in detail time stamp based protocol.

ANS: The timestamp-based concurrency control protocol ensures data consistency by assigning timestamps to transactions and data items, allowing the system to make decisions based on these timestamps to maintain transaction order and consistency.

Basic Components:

1. **Timestamps: Each transaction and data item has associated timestamps. A transaction T_i is assigned a unique timestamp (TS(T_i)), usually representing the order in which it began. Each data item Q also has a timestamp (TS(Q)) showing the last time it was read or written.**
2. **Read and Write Operations:**
    - **Read(Q): Transaction T_i wants to read data item Q.**
    - **Write(Q): Transaction T_i wants to write to data item Q.**

How Timestamp-based Protocol Works:

1. **Read Operation:**
    - **When a transaction T_i issues a read command on data item Q, the system compares the timestamp of the transaction with the timestamp of the data item.**
    - **If TS(T_i) > TS(Q), T_i can read Q because it means the data was last written before the transaction started.**
    - **If TS(T_i) <= TS(Q), there might be a conflict:**
        - **If the operation is allowed, the system might need to roll back T_i if a more recent transaction has already written Q.**
2. **Write Operation:**
    - **When a transaction T_i wants to write to a data item Q, the system verifies the timestamp to ensure it's allowed to modify Q.**
    - **If TS(T_i) > TS(Q), it's allowed to write because T_i's timestamp is more recent.**
    - **If TS(T_i) <= TS(Q), there might be a conflict:**
        - **The system decides whether to allow the write based on its conflict resolution mechanism (such as waiting or aborting a transaction).**

Decision Making:

- **Reads: Based on the comparison of timestamps, a read operation is allowed if the transaction's timestamp is greater than the timestamp of the data item.**
- **Writes: Writing is permitted if the transaction's timestamp is greater than the timestamp of the data item.**

Handling Conflicts:

- **In case a transaction's operation is not allowed due to timestamp comparison (read or write), the system employs conflict resolution strategies. This could involve delaying the operation, rolling back the transaction, or waiting until the conflicting transaction completes.**

Q.4] Explain the concept of conflict serializability with suitable example. Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

ANS:

Conflict Serializability:

Conflict serializability refers to the property of a schedule in a database system where it's equivalent to some serial execution of transactions, preserving the order of conflicting operations. Conflicts typically occur with shared data items when multiple transactions perform read and write operations on the same data.

To determine conflict serializability, we focus on the conflicting operations within and between transactions, particularly read-write and write-write conflicts. If a schedule can be rearranged into a serial order where conflicting operations maintain their relative order, it's considered conflict serializable.

Example of Conflict Serializability:

Let's consider two transactions, T1 and T2, operating on two data items, A and B:

Schedule:

T1:

- **Read(A)**
- **Write(A)**
- **Read(B)**
- **Write(B)**

T2:

- **Read(A)**
- **Write(A)**
- **Read(B)**
- **Write(B)**

In this schedule, T1 and T2 execute operations on A and B in the same order. There are no conflicting operations because each transaction completes its work on A before moving to B. This schedule is conflict serializable as it's equivalent to a serial execution, preserving the order of operations without conflicts.

Why Emphasize Conflict Serializability over View Serializability:

While it's true that every conflict-serializable schedule is view serializable, the emphasis on conflict serializability arises due to the practical advantages and computational complexity considerations:

1. **Practicality: Conflict serializability deals explicitly with conflicting operations that can cause inconsistencies in a database. By ensuring these conflicts are properly managed, it guarantees that transactions won't interfere with each other when accessing shared data. It's a more direct and focused approach to maintaining consistency.**
2. **Computation Complexity: Determining view serializability involves considering all possible view equivalent schedules, which can be computationally intensive for larger databases. On the other hand, conflict serializability primarily focuses on conflicting operations, making it more practical to verify and implement in real systems.**
3. **Performance: Conflict serializability directly addresses conflicts, promoting better performance by reducing the chances of transactions waiting or being aborted due to conflicts. This leads to fewer rollbacks and less contention for shared data, resulting in better system efficiency.**

Q.5] Write a short note on : [8] i) Log based recovery ii) Shadow Paging
ANS:
i) Log-Based Recovery:
In database systems, log-based recovery is a technique used to ensure data integrity and recover from failures. It involves maintaining a sequential record of all the modifications made to the database, known as the transaction log.

- **Transaction Log: This log contains a chronological record of all the database changes made by transactions. It typically includes information like the transaction identifier, operation type (such as commit, abort, write, etc.), the before and after values of modified data, and other relevant details.**
- **Recovery Process: In the event of a system crash or failure, the transaction log is utilized to bring the database back to a consistent state. This is done using the process of redo and undo:**
    - **Redo: Reapplies the changes logged in the transaction log to ensure that all committed transactions' modifications are re-executed to restore the database to the most recent state.**
    - **Undo: Rolls back any incomplete or uncommitted transactions by reversing the changes recorded in the log, ensuring that the database doesn't retain the effects of transactions that were not completed.**

Log-based recovery provides a robust mechanism for recovering from system failures and maintaining data consistency by leveraging the recorded history of transactions and their changes.
ii) Shadow Paging:
Shadow paging is a technique used for implementing database recovery and concurrency control. It's based on maintaining multiple versions of the database during transactions, allowing for a consistent snapshot of the database to be available at all times.

- **Shadow Page Table: In shadow paging, a shadow page table is utilized to keep track of the active and stable versions of the database. This table maintains pointers or references to the pages in both the current (active) version and the stable (committed) version of the database.**
- **Transaction Execution: When a transaction performs modifications, it works on a shadow copy of the database, maintaining the original database's stability. The changes made during the transaction aren't reflected in the main database until the transaction is committed.**
- **Commit Process: Once the transaction is committed, the pointers in the shadow page table are updated to reflect the changes made by the transaction in the stable version of the database. This updating process involves swapping page pointers to make the shadow copy the new stable copy, ensuring atomic commitment.**

Shadow paging allows for a consistent view of the database to be maintained throughout the transaction's execution and provides a method for quick recovery by simply reverting to the last stable copy in case of failure.

Q.6] A transaction may be waiting for more time for an Exclusive (X) lock on an item, while a sequence of other transactions request and are granted as Shared (S) lock on the same item. What is this problem? How is it solved by two phase lock protocol?

ANS: The problem described, where a transaction is waiting for an Exclusive (X) lock while other transactions are continuously granted Shared (S) locks on the same item, is known as the "Deadlock" problem in the context of concurrency control in databases.

The Two-Phase Locking (2PL) protocol is a mechanism designed to prevent deadlocks and ensure serializability. It consists of two distinct phases, the growing phase and the shrinking phase, to manage lock acquisition and release in a controlled manner.

Solving the Deadlock Problem with Two-Phase Locking Protocol (2PL):

1. **Growing Phase:**
    - **Acquiring Locks: Transactions can acquire locks but cannot release any lock once it's released.**
    - **Shared (S) Locks: Multiple transactions can hold Shared locks simultaneously on the same item.**
    - **Exclusive (X) Locks: Transactions can convert a Shared lock to an Exclusive lock, but once an Exclusive lock is acquired, no other transaction can acquire any lock on that item until the Exclusive lock is released.**
2. **Shrinking Phase:**
    - **Releasing Locks: Once a transaction releases any lock (either Shared or Exclusive), it cannot acquire any further locks on any item.**
    - **Ensuring Serializability: This phase prevents a transaction from releasing locks until it has completed its execution, thereby ensuring the consistency and isolation of the transaction's operations.**

How Two-Phase Locking Addresses the Deadlock Problem:

1. **Preventing Circular Wait:**
    - **By following the strict rules of 2PL, transactions cannot release a lock once it's released. This prevents a circular wait, where a transaction is waiting indefinitely for a lock that another transaction is holding.**
2. **Ensuring No Conflicting Locks:**
    - **As per 2PL, Exclusive locks prevent any other transaction from acquiring any lock on that item until the Exclusive lock is released. This prevents the scenario where conflicting locks (S and X) could create a deadlock situation.**
3. **Guaranteeing Serializability:**
    - **The strict rules of 2PL ensure that the transactions are executed in a manner that maintains the serializability of the system, avoiding potential inconsistencies or deadlocks that might arise due to overlapping operations on shared items.**
4. **Controlled Lock Release:**
    - **By not allowing lock releases until the end of a transaction, 2PL ensures a controlled and systematic approach to managing locks, preventing scenarios where a transaction gets stuck waiting for a resource indefinitely.**

Q.7] What is conflict serializability? How to check schedule is conflict serializable schedule. Give one example.
ANS: here's a simple and easy explanation of conflict serializability along with steps to check if a schedule is conflict serializable:

1. **What is Conflict Serializability?**
    - **Conflict serializability is a property of schedules in database transactions that ensures the transactions could have been executed in a serial order without resulting in a different final state.**
    - **It ensures that the transactions are executed in a way that the final result is the same as if they were executed one after another in some order.**
2. **How to Check for Conflict Serializability:**
    - **Step 1: Identify the Transactions: Identify all the transactions in the schedule. Each transaction is usually represented by a letter (e.g., T1, T2, T3).**
    - **Step 2: Construct a Conflict Graph: For each pair of conflicting operations (i.e., operations from different transactions that access the same data item, where at least one of the operations is a write), create a node in the conflict graph. Draw an edge from the transaction performing the earlier operation to the transaction performing the later operation. This graph is called the conflict graph.**
    - **Step 3: Check for Cycle: Check if the conflict graph contains any cycles. If it does not contain any cycles, then the schedule is conflict serializable. If it contains a cycle, then the schedule is not conflict serializable.**
3. **Example: Let's consider the following schedule:**

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

- **Step 1: Identify the transactions: T1 and T2.**
- **Step 2: Construct the Conflict Graph:**
    - **There is a conflict between T1's Write(B) and T2's Read(B).**
    - **There is a conflict between T2's Write(A) and T1's Read(A).**
    - **There is no conflict between T1's Read(C) and T2's Write(C).**
- **Step 3: Check for Cycle:**
    - **No cycle is present in the conflict graph.**
- **Conclusion: Since there is no cycle in the conflict graph, the schedule is conflict serializable.**

Q.8] During execution, a transaction passes through several states, until it commits or aborts. List all possible sequence of states through which transaction may pass. Explain the situation when each state transition occurs.

ANS: here's a simple list of the possible sequence of states a transaction may pass through during execution, along with explanations for each state transition:

1. **Active:**
    - **The transaction starts executing.**
    - **It's actively accessing and possibly modifying data.**
2. **Partially Committed:**
    - **The transaction has finished executing.**
    - **It's marked for committing but hasn't completed the process yet.**
    - **This state indicates that the transaction has reached a point where it cannot abort without causing inconsistency.**
3. **Committed:**
    - **The transaction has successfully completed and all its changes have been permanently saved to the database.**
    - **It's now durable and won't be rolled back.**
4. **Aborted:**
    - **The transaction has encountered an error or explicitly issued an abort command.**
    - **Any changes made by the transaction are discarded, and the database is restored to its state before the transaction began.**
    - **This could happen due to conflicts with other transactions, constraint violations, or system failures.**
5. **Failed:**
    - **The transaction has encountered a problem that prevents it from completing successfully.**
    - **Unlike the aborted state, the failure might occur before the transaction has made any changes to the database.**
    - **It could be due to hardware failures, deadlock situations, or violations of integrity constraints.**
6. **Terminated:**
    - **The transaction has completed its execution and been removed from the system.**
    - **This state occurs after either committing or aborting the transaction.**
    - **Resources used by the transaction are released, and any locks held by it are released.**

Q.9] Consider the following two transactions:
T31: read(A);
read(B);
if A = 0 then B:=B+1;
Write (B)
T32: read(B);
read(A);
if B= 0 then A: = A+1;
write (A).
Add lock and unlock instructions to transactions T31 and T32, so that they observe the two phase locking protocol. Can the execution of these transactions result in a deadlock
ANS: here's a step-by-step guide to adding lock and unlock instructions to transactions T31 and T32 to adhere to the two-phase locking protocol, along with an explanation of whether deadlock can occur:

1. **Transaction T31:**
    - **Before reading variable A, acquire a lock on A.**
    - **Before reading variable B, acquire a lock on B.**
    - **After writing to variable B, release the lock on B.**

T31:
Lock(A);
Lock(B);
read(A);
read(B);
if A = 0 then B:=B+1;
Write(B);
Unlock(B);

2. **Transaction T32:**

- **Before reading variable B, acquire a lock on B.**
- **Before reading variable A, acquire a lock on A.**
- **After writing to variable A, release the lock on A.**
  T32:
  Lock(B);
  Lock(A);
  read(B);
  read(A);
  if B= 0 then A: = A+1;
  write(A);
  Unlock(A);

Deadlock Possibility:
Yes, there is a possibility of deadlock in this scenario. Deadlock can occur if both transactions acquire locks on resources in a different order and then wait for each other's locks. For example:

- **If T31 acquires a lock on A and then tries to acquire a lock on B.**
- **At the same time, T32 acquires a lock on B and then tries to acquire a lock on A.**

If this happens, both transactions will be waiting for each other to release the locks they need, resulting in a deadlock. To prevent deadlocks, transactions should acquire locks on resources in a consistent order, such as always acquiring locks in alphabetical order or based on some other predetermined order.

Q.10] To ensure atomicity despite failures we use Recovery Methods. Explain in detail log based recovery method.
ANS: here's a simple and easy-to-understand explanation of the log-based recovery method:

1. **What is Log-Based Recovery?**
    - **Log-based recovery is a technique used in database management systems (DBMS) to ensure data consistency and atomicity even in the event of system failures like crashes or power outages.**
2. **Transaction Logging:**
    - **Every time a transaction is performed in the database, the details of that transaction are recorded in a log file.**
    - **The log file maintains a sequential record of all changes made to the database, including updates, inserts, and deletes.**
3. **Types of Log Records:**
    - **There are typically three types of log records:**
        - **Start: Marks the beginning of a transaction.**
        - **Update: Records the changes made by a transaction to the database.**
        - **Commit/Abort: Indicates the successful completion or failure of a transaction.**
4. **Writing Log Entries:**
    - **Before making changes to the actual database, the DBMS first writes the corresponding log entries to the log file on disk.**
    - **This ensures that even if a failure occurs during the transaction, the log entries remain intact and can be used for recovery.**
5. **Transaction Commit:**
    - **When a transaction successfully completes, a commit record is written to the log.**
    - **This indicates that all changes made by the transaction are permanent and can be applied to the database.**
6. **Transaction Abort:**
    - **If a transaction fails or needs to be rolled back, an abort record is written to the log.**
    - **This indicates that the changes made by the transaction should be undone or reverted.**
7. **Recovery Process:**
    - **In the event of a system failure, the recovery process begins when the system restarts.**
    - **The DBMS checks the log file to determine the transactions that were in progress and those that were committed but not yet applied to the database.**
8. **Redo and Undo Operations:**
    - **Redo operation: Transactions that were committed but not yet applied to the database are reapplied using the log entries to ensure all changes are reflected in the database.**
    - **Undo operation: Transactions that were in progress or aborted are undone by applying the reverse of their changes recorded in the log.**

9. **Database Consistency:**
    - **By using the log-based recovery method, the database can be restored to a consistent state, ensuring that all transactions either fully commit or are fully rolled back, even in the event of failures.**
10. **Conclusion:**
    - **Log-based recovery is a critical technique in ensuring the durability and consistency of data in a database, providing a reliable mechanism for recovering from system failures while maintaining data integrity.**

Q.11] Explain the two-phase lock protocol for concurrency control. Also explain its two versions: strict two-phase lock protocol and rigorous two-phase lock protocol.

ANS: here's a simple and easy-to-understand explanation of the two-phase lock protocol for concurrency control, along with its two versions: strict two-phase lock protocol and rigorous two-phase lock protocol:

Two-Phase Lock Protocol:

1. **Acquiring Phase:**
   - **Transactions can acquire locks on data items before using them.**
   - **Two types of locks: shared locks (read locks) and exclusive locks (write locks).**
   - **A transaction can request a lock on an item only if it does not conflict with existing locks held by other transactions.**
   - **If a transaction cannot acquire a lock immediately, it waits until the requested lock becomes available.**
2. **Releasing Phase:**
   - **Once a transaction has finished using a data item, it releases the lock on that item.**
   - **This allows other transactions to acquire locks on the released items.**

Strict Two-Phase Lock Protocol:

1. **Acquiring Phase:**
   - **Similar to the basic two-phase lock protocol.**
   - **Transactions acquire all the locks they need in the acquiring phase and hold them until the end of the transaction.**
2. **Releasing Phase:**
   - **Unlike the basic protocol, locks are released only at the end of the transaction.**
   - **This ensures that no other transaction can access the locked items until the transaction holding the locks has completed.**

Rigorous Two-Phase Lock Protocol:

1. **Acquiring Phase:**
   - **Similar to the strict two-phase lock protocol.**
   - **Transactions acquire all the locks they need in the acquiring phase.**
2. **Validation Phase:**
   - **After acquiring all locks, the transaction checks if it can complete without violating serializability.**
   - **If validation is successful, the transaction continues. Otherwise, it releases all locks and restarts.**
3. **Releasing Phase:**
   - **Locks are released only at the end of the transaction, similar to the strict protocol.**

In summary, the two-phase lock protocol ensures that transactions acquire and release locks in two distinct phases, helping to prevent issues like deadlocks and ensuring data consistency. The strict and rigorous versions add additional restrictions and validation steps to ensure stronger concurrency control.

Q.12] What is R-timestamp(Q) and W-timestamp(Q) Explain the necessary condition used by time stamp ordering protocol to execute for a read / write operation.
ANS: here's a simple explanation of R-timestamp(Q) and W-timestamp(Q) along with the necessary condition used by the timestamp ordering protocol:

1. **R-timestamp(Q):**
   - **It represents the timestamp of the last read operation on item Q.**
   - **Essentially, it's the timestamp of the most recent read operation that accessed item Q.**
2. **W-timestamp(Q):**
   - **It represents the timestamp of the last write operation on item Q.**
   - **Essentially, it's the timestamp of the most recent write operation that modified item Q.**

Necessary condition for time stamp ordering protocol:

- **For a read operation to execute:**
  - **The timestamp of the read operation (let's call it T_read) must be greater than or equal to the W-timestamp(Q) of the item being read.**
  - **This means that the read operation can only proceed if no write operation with a greater timestamp has modified the item since the last read operation.**
- **For a write operation to execute:**
  - **The timestamp of the write operation (let's call it T_write) must be greater than the R-timestamp(Q) and W-timestamp(Q) of the item being written.**
  - **This means that the write operation can only proceed if no read or write operation with a greater timestamp has accessed or modified the item since the last write operation.**

These conditions ensure that read and write operations are executed in a consistent and orderly manner, preventing conflicts and maintaining data integrity.

Q.13] To ensure atomicity despite failures we use Recovery Methods Explain in detail following Log-Based Recovery methods with example.
i) Deferred Database Modifications
ii) Immediate Database Modifications
ANS: let's break down the two log-based recovery methods:
i) Deferred Database Modifications:

1. **Logging Changes:**
   - **When a transaction begins, all its modifications to the database are logged but not applied immediately to the database itself.**
   - **These logs contain the before and after images of data items that are modified by the transaction.**
2. **Database Modification After Commitment:**
   - **After the transaction successfully commits, the changes recorded in the log are then applied to the actual database.**
   - **If a transaction fails before committing, its changes are simply undone by ignoring its log entries.**
3. **Example:**
   - **Suppose a bank transaction involves transferring $100 from account A to account B.**
   - **Initially, the log records the state of both accounts (before the transaction).**
   - **If the transaction commits successfully, the log updates the state of both accounts to reflect the transfer.**
   - **If the transaction fails before committing (e.g., due to a system crash), the log entries for this transaction are ignored, leaving both accounts in their initial state.**

ii) Immediate Database Modifications:

1. **Logging and Applying Changes Simultaneously:**
   - **In this method, modifications made by a transaction are applied immediately to the database, and then logged.**
   - **The log primarily serves as a record of the changes made.**
2. **Undo Operations in Case of Failure:**
   - **If a transaction fails before committing, the changes made by the transaction are undone by using the log.**
   - **The log contains information about how to reverse the modifications (undo operations) that were applied to the database.**
3. **Example:**
   - **Using the same bank transaction scenario, in this method, when the $100 transfer from account A to account B is initiated, the database is immediately updated to reflect this change.**
   - **Simultaneously, a log entry is made indicating the details of this transaction.**
   - **If the transaction fails before committing (e.g., due to a system crash), the log entry for this transaction is used to reverse the $100 transfer, restoring both accounts to their initial states.**