TRANSACTION MANAGEMENT

Illustrate testing of Conflict Serializability with appropriate example.

To illustrate the testing of Conflict Serializability, we can use the concept of a precedence graph. Conflict Serializability is a property of a schedule in a database system that ensures that the schedule is equivalent to some serial execution of transactions.Here is an example to demonstrate the testing of Conflict Serializability using a precedence graph:Consider two transactions T1 and T2 with the following operations:

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

1. **Constructing the Precedence Graph**:
    - For each pair of conflicting operations (Write-Read or Write-Write), draw an edge from the transaction performing the earlier operation to the transaction performing the later operation.
    - In this case, we have conflicts between T1 and T2 on variables A and B, leading to the following precedence graph:

Precedence Graph

Figure 14.10 Precedence graph for (a) schedule 1 and (b) schedule 2.

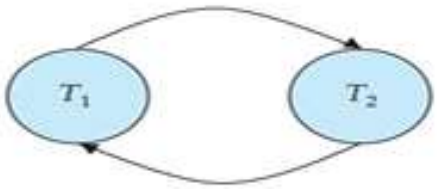


Figure 14.11 Precedence graph for schedule 4.

2. **Testing for Conflict Serializability**:
    - If the precedence graph contains a cycle, the schedule is not conflict serializable.
    - In the example above, the precedence graph forms a cycle (T1 -> T2 -> T1), indicating that the schedule is not conflict serializable.
3. **Resolution**:
    - To make the schedule conflict serializable, you can introduce appropriate transaction orderings or serializations to remove the cycles in the precedence graph.
    - Adjusting the order of operations in the transactions can ensure that conflicts are resolved, making the schedule conflict serializable.

Write the rules for Thomas Write Rule. Elaborate how Thomas' Write Rule allows greater potential concurrency as compared to Timestamp based protocol.

Thomas' Write Rule is a protocol for managing concurrent access to a database in a multi-user environment. It is designed to ensure that the database remains consistent and that all users see the same version of the data, even in the presence of concurrent updates.**Rules for Thomas' Write Rule**:

1. **Write-Write Conflict Resolution**: When two transactions attempt to write to the same data item, the transaction with the higher timestamp wins. If the timestamps are the same, the transaction that started first wins.
2. **Read-Write Conflict Resolution**: When a transaction attempts to read a data item that is being written to by another transaction, the read operation is blocked until the write operation is complete.
3. **Write-Read Conflict Resolution**: When a transaction attempts to write to a data item that is being read by another transaction, the write operation is blocked until the read operation is complete.

How Thomas' Write Rule allows greater potential concurrency as compared to Timestamp based protocol:

1. **Reduced Blocking**: Thomas' Write Rule reduces the amount of blocking that occurs due to concurrent updates. In a timestamp-based protocol, a write operation may be blocked by a read operation, even if the read operation is not accessing the same data item. Thomas' Write Rule only blocks a write operation if it is accessing the same data item as the read operation.
2. **Improved Concurrency**: Thomas' Write Rule allows for greater concurrency because it only blocks a write operation if it is accessing the same data item as the read operation. This means that multiple transactions can access different data items concurrently, even if they are updating the same relation.

3. **Faster Recovery**: Thomas' Write Rule allows for faster recovery from failures because it only requires the database to be rolled back to the last committed state, rather than the last consistent state. This reduces the amount of work that needs to be redone after a failure.
4. **Simplified Implementation**: Thomas' Write Rule is simpler to implement than a timestamp-based protocol because it does not require the database to maintain a timestamp for each data item. Instead, the database only needs to maintain a timestamp for each transaction.

List and explain the variants of Two Phase Lock Protocol

1. **Strict Two-Phase Locking (Strict 2PL)**:
    - In Strict 2PL, a transaction must hold all its locks until it commits or aborts.
    - This means that a transaction cannot release any locks until it has completed all its operations and is ready to commit or abort.
    - Strict 2PL ensures that the schedule produced is conflict serializable and recoverable.
    - It also ensures that the schedule is cascadeless, meaning that a transaction does not read data written by an uncommitted transaction.
    - Strict 2PL is commonly used in database systems to ensure data consistency and isolation.
2. **Rigorous Two-Phase Locking (Rigorous 2PL)**:
    - Rigorous 2PL is a variant of Strict 2PL that is even more conservative in its locking behavior.
    - In Rigorous 2PL, a transaction must hold all its locks until it commits.
    - This means that a transaction cannot release any locks, even if it has completed all its operations, until it has committed.
    - Rigorous 2PL ensures that the schedule produced is not only conflict serializable and recoverable, but also avoids the possibility of cascading aborts.
    - Cascading aborts occur when the abort of one transaction leads to the abort of other dependent transactions.
    - Rigorous 2PL is often used in systems where data consistency and isolation are of the utmost importance, even at the cost of some performance overhead.

Both Strict 2PL and Rigorous 2PL are widely used in database management systems to ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions. The choice between the two variants depends on the specific requirements of the application and the trade-offs between performance and data consistency.

Explain with appropriate example the following terms

a. Recoverable Schedules

b. Cascadeless Schedules

1. Recoverable Schedules:

A recoverable schedule is a schedule that can be rolled back to a previous state in case of a failure. This means that the schedule must ensure that all transactions are executed in a way that allows the database to be restored to a consistent state in case of a failure.

- **Example:**

Suppose we have two transactions T1 and T2 that execute concurrently. T1 reads a value from a database, updates it, and then writes it back. T2 reads the same value, updates it, and then writes it back. If T2 fails before writing its updated value, the database will be left in an inconsistent state. However, if T1 writes its updated value before T2 starts executing, the schedule is recoverable because the database can be rolled back to the state before T1 wrote its updated value.

2. Cascadeless Schedules:

A cascadeless schedule is a schedule that does not cause a cascade of aborts. This means that if a transaction aborts, it does not cause other transactions to abort.

- **Example:**

Suppose we have two transactions T1 and T2 that execute concurrently. T1 reads a value from a database, updates it, and then writes it back. T2 reads the same value, updates it, and then writes it back. If T1 aborts before writing its updated value, T2 will not abort because it has already read the original value. However, if T2 aborts before writing its updated value, T1 will not abort because it has already written its updated value.

What is transaction? Explain its ACID properties of transaction

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

A transaction in a database represents a unit of work performed against the database. It is an atomic operation that either completes in its entirety or is rolled back if it fails, ensuring data integrity and consistency. Transactions typically consist of multiple steps or operations that are executed as a single, indivisible unit.

ACID Properties of Transactions:

1. **Atomicity:**
    - Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully, or none of them are.
    - **Example:** Consider a transaction that transfers funds from one account to another. If any part of the transaction fails (e.g., due to insufficient funds), the entire transaction is rolled back, and the funds transfer is undone.
2. **Consistency:**
    - Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database remains in a valid state both before and after the transaction is executed.
    - **Example:** In a banking system, if a transaction deducts funds from one account, it should also credit the same amount to another account, ensuring that the total balance remains unchanged.
3. **Isolation:**

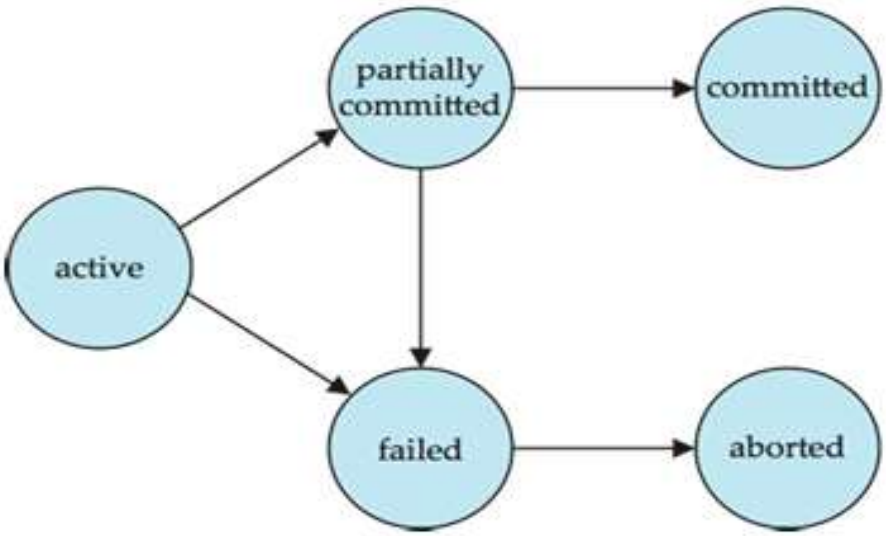
- Isolation ensures that multiple transactions can execute concurrently without interfering with each other. Each transaction is isolated from other transactions until it is completed.
- **Example:** Suppose two users concurrently update the same bank account balance. Isolation prevents one user from seeing the intermediate, inconsistent state of the account during the other user's transaction.

4. **Durability:**

- Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The changes made by the transaction are stored in non-volatile memory.
- **Example:** After a successful funds transfer transaction, the updated account balances are durably stored in the database, even if the system crashes. Upon recovery, the changes are still intact.

Draw and explain the Transaction State Diagram

1. **Active**: This is the initial state of a transaction when it begins execution. In this state, the transaction is performing its operations and accessing the database.
2. **Partially Committed**: When a transaction completes all its operations and is ready to commit, it enters the Partially Committed state. In this state, the transaction's changes are written to the database, but the transaction is not yet fully committed.
3. **Committed**: If the transaction successfully completes all its operations and the commit process is successful, the transaction enters the Committed state. In this state, the transaction's changes are permanently recorded in the database.
4. **Failed:** The transaction has failed and its effects have been rolled back, but the failure was due to a system error rather than a logical error.
5. **Aborted**: If a transaction encounters an error or a conflict during its execution, it enters the Aborted state. In this state, all the changes made by the transaction are rolled back, and the database is restored to its previous consistent state.



The transitions between these states are as follows:

- From Active to Partially Committed: This transition occurs when the transaction completes all its operations and is ready to commit.
- From Partially Committed to Committed: This transition occurs when the commit process is successful, and the transaction's changes are permanently recorded in the database.
- From Active to Failed: This transition occurs when a transaction encounters an error or conflict during its execution, but before it is fully aborted.
- From partially executed to Failed : The transition from the "Prepared" state to the "Failed" state represents a scenario where a transaction has been partially executed and prepared for commit, but the commit operation fails due to a system error. In this case, the transaction is moved to the "Failed" state, and its effects are rolled back.
- From Failed to Aborted: This transition occurs when the transaction is ultimately aborted after the Failed state.

The Transaction State Diagram represents the lifecycle of a transaction and the possible states it can go through during its execution. This diagram is crucial for understanding the behavior of transactions in a database system and ensuring the ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions.

List and elaborate the Pitfalls of Lock-Based Protocols

The pitfalls of lock-based protocols in database systems can lead to issues such as deadlocks, reduced concurrency, and performance bottlenecks. Here are some common pitfalls:

1. **Deadlocks**:
    - Deadlocks occur when two or more transactions are waiting for locks held by each other, resulting in a cyclic dependency where none of the transactions can proceed.
    - Deadlocks can lead to system halts and require intervention to resolve, impacting system availability and performance.
2. **Reduced Concurrency**:
    - Lock-based protocols can lead to reduced concurrency as transactions may need to wait for locks to be released by other transactions.
    - This waiting time can increase, leading to lower throughput and slower transaction processing.
3. **Lock Contention**:
    - Lock contention arises when multiple transactions are contending for the same locks, causing delays and reducing overall system performance.
    - High lock contention can result in increased waiting times for transactions, impacting system responsiveness.
4. **Performance Bottlenecks**:
    - Lock-based protocols can introduce performance bottlenecks, especially in systems with high transaction volumes.
    - The overhead of acquiring and releasing locks, as well as managing lock queues, can impact system performance and scalability.

5. **Complexity and Maintenance**:
    - Lock-based protocols can introduce complexity in managing locks, especially in systems with a large number of transactions and shared resources.
    - Ensuring proper lock management, avoiding deadlocks, and optimizing lock acquisition/release can be challenging and require ongoing maintenance.
6. **Lock Granularity**:
    - Inappropriate lock granularity, such as using locks at a coarse level (e.g., table-level locks), can lead to reduced concurrency and performance.
    - Fine-grained locks can increase overhead due to lock management, while coarse-grained locks can lead to contention and reduced parallelism.
7. **Resource Starvation**:
    - Lock-based protocols can result in resource starvation, where certain transactions are unable to acquire the necessary locks due to contention with other transactions.
    - Resource starvation can lead to delays in transaction processing and impact system performance.

List and elaborate the Intention Lock Modes in Multiple Granularity? Draw the Compatibility Matrix with Intention Lock Modes

Intention Lock Modes in Multiple Granularity

In a database system with multiple levels of data granularity (e.g., database, table, row), the intention lock modes are used to indicate the intention to acquire locks at lower levels of the hierarchy. The intention lock modes are:

1. **Intention Shared (IS)**
   - Indicates the intention to set shared locks at lower levels of the hierarchy.
   - Allows other transactions to set shared locks at the same level or lower levels.
   - Does not allow other transactions to set exclusive locks at the same level or lower levels.
2. **Intention Exclusive (IX)**
   - Indicates the intention to set exclusive locks at lower levels of the hierarchy.
   - Allows other transactions to set shared or exclusive locks at lower levels.
   - Does not allow other transactions to set any locks at the same level.
3. **Shared with Intention Exclusive (SIX)**
   - Indicates that a shared lock is held at the current level, and there is an intention to set exclusive locks at lower levels.
   - Allows other transactions to set shared locks at the same level or lower levels.
   - Does not allow other transactions to set any locks at the same level.

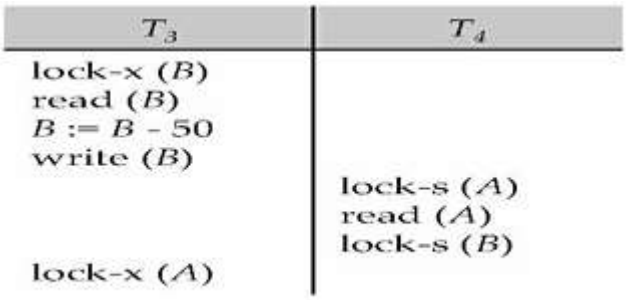
Compatibility Matrix with Intention Lock Modes

Lock Mode	S	X	IS	IX	SIX
S	Yes	No	Yes	No	Yes
X	**No**	No	No	No	No
IS	Yes	No	Yes	Yes	Yes
IX	No	No	Yes	Yes	No
SIX	Yes	No	Yes	No	No

- The matrix shows the compatibility between different lock modes.
- 'Yes' indicates the lock modes are compatible, while 'No' indicates they are not compatible.
- For example, a transaction holding an Intention Shared (IS) lock can acquire a Shared (S) lock or another Intention Shared (IS) lock, but cannot acquire an Exclusive (X) lock.

The intention lock modes allow for more concurrency in the system by allowing transactions to acquire locks at higher levels of the hierarchy, while still ensuring that the appropriate locks are held at the lower levels when accessing data.

Consider the above partial schedule. Check if the schedule is following the rules of 2PL. Also predict the state of execution of the given schedule.



T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

1. **Two-Phase Locking (2PL)**:
    - 2PL is a concurrency control method that ensures conflict serializable schedules.
    - It divides the execution phase of a transaction into two parts:
        - **Growing Phase**: During this phase, new locks on data items may be acquired, but none can be released.
        - **Shrinking Phase**: In this phase, existing locks may be released, but no new locks can be acquired.
    - If read and write operations introduce the first unlock operation in the transaction, then it is said to follow the Two-Phase Locking Protocol.
2. **Given Schedule**:
    - Let's break down the schedule based on the provided information:
3. T3:

    lock-x (B)

    read (B)

    B := B – 50

    write (B)

    lock-x (A)

T4:

lock-s (A)

read (A)

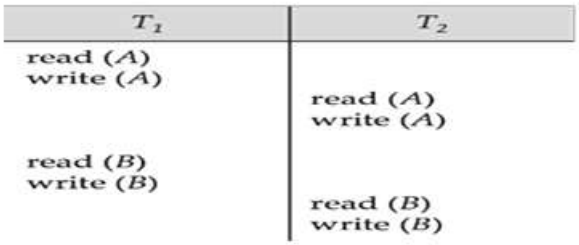
lock-s (B)

4. **Analysis**:
    - Transaction T3:
        - The growing phase for T3 starts with the acquisition of the exclusive lock on B (step 1) and continues until the last operation.
        - At last the T3 want to acquire the exclusive lock on the A .
        - The lock point for T3 (when it acquires the exclusive lock on A).
    - Transaction T4:
        - The growing phase for T4 starts with the shared lock on A and continues until the last operation
        - First It want to acquire the shared lock on the A and at last stage it want to acquire the shared lock on the B
5. **State of Execution**:
    - Considering the lock points, we can determine the state of execution:
        - T3 first acquired the exclusive lock on the B and does his operations
        - After that the T4 acquires the shared lock on the A and does his operation
        - Then at last the T4 want to acquire the shared lock on B but the T3 already acquired that using the exclusive lock cause of that the T4 won't be able to access the B variable

- On the other hand the T4 has initiated his lock on the variable A with a shared lock but the T3 want the exclusive lock of the A variable

6. **Drawbacks of 2PL**:
    - Cascading Rollback: Possible due to dependencies between transactions.
    - Deadlocks: Possible when transactions wait indefinitely for each other's locks
    - Cause of all this the deadlock might happen

Given the above schedule, check if it is conflict serializable. If yes show serial schedule to which the above schedule is conflict equilevent, if no state why?



Certainly! Let's analyze the given schedule to determine if it is conflict serializable. If it is, we'll find a serial schedule that is conflict equivalent to it. If not, I'll explain why.

1. **Given Schedule**:
    - The provided schedule consists of two transactions, T1 and T2, performing read and write operations on data items A and B.
2. **T1,T2:**

    1. Read (A)
    2. Write (A)
    5. Read (A)
    6. Write (A)

3. **T1,T2:**

    7. Read (B)
    8. Write (B)
    3. Read (B)
    4. Write (B)

4. **Conflict Serializability**:
    - A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

- Two operations are considered conflicting if they meet the following conditions:
  - They belong to separate transactions.
  - They operate on the same data item.
  - At least one of them is a write operation.

5. **Analysis**:
   - Let's check for conflicts:
     - T1 reads A and then writes A (conflict).
     - T2 reads A and then writes A (conflict).
     - T1 reads B and then writes B (conflict).
     - T2 reads B and then writes B (conflict).
6. **Conflict Equivalent**:
   - For the schedule to be conflict serializable, we need to find a serial schedule that is equivalent to it.
   - Swapping non-conflicting operations:
     - We can transform T1 and T2 as follows:
       - Swap T1's read (A) with T2's read (A).
       - Swap T1's write (A) with T2's write (A).
       - Swap T1's read (B) with T2's read (B).
       - Swap T1's write (B) with T2's write (B).
7. Serial Schedule (Equivalent to Given Schedule):

   T1: Read (A)

   T1: Write (A)

   T2: Read (A)

   T2: Write (A)

   T1: Read (B)

T1: Write (B)

T2: Read (B)

T2: Write (B)

Because there is no conflicts here so their no need to change or swap their positions