Unit - V
Concurrency Control

1. Concurrent Execution of Transactions

- Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications. However, there are two good reasons for allowing concurrency:
  - **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU I/O activities can be done in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase.
  - **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**.

2. Anomalies Due to Concurrent Executions

- If all the transactions in DBMS systems are doing read operation on the Database then no problem will arise. When the read and write operations are done simultaneously, then there is a possibility of some type of anomalies. These are classified into three categories.
    - Write–Read Conflicts (WR Conflicts)
    - Read–Write Conflicts (RW Conflicts)
    - Write–Write Conflicts (WW Conflicts)

WR Conflicts (or Dirty Read)

- This happens when the transaction T_2 read the data item A that has been modified by another transaction T_1, which has not yet committed. Such a read is called a *dirty read*.
- As an example, consider the following set of transactions (T_1 & T_2) and the schedule for executing T_1 & T_2.

  T_1: Transfers $100 from account A to account B

  T_2: Increments the balances of account A and account B by 10%

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)
	Commit
Read(B)	
Write(B)	
Commit	

Dirty read

Reading uncommitted data

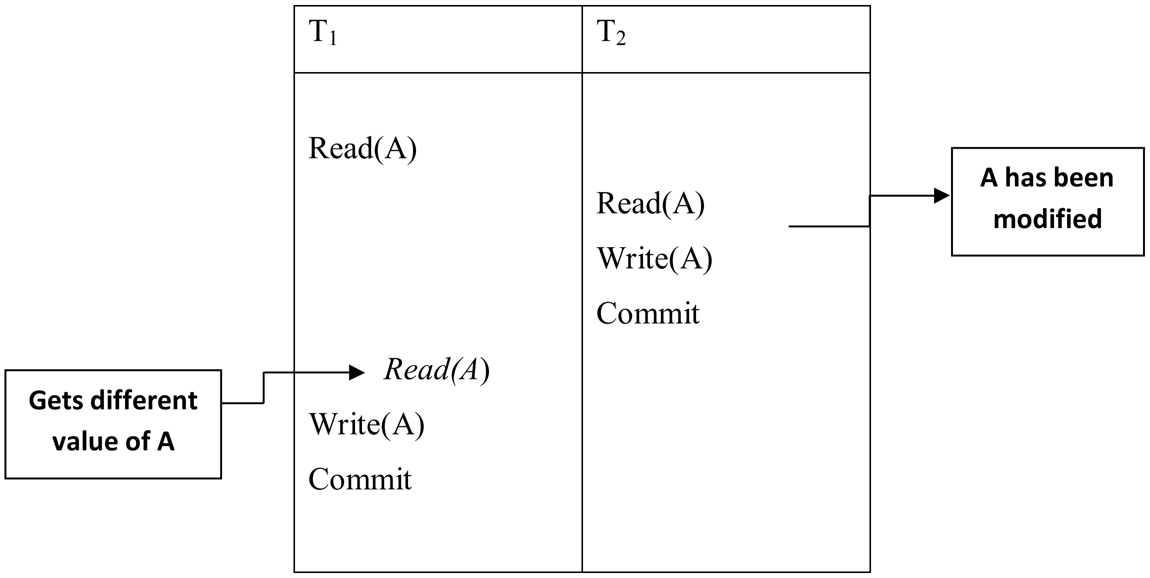
If the two transactions are allowed to execute as per the schedule shown above, then the outcome of this execution will be different from the normal execution like if the two instructions are executed one after another. This type of anomalies leaves the database in an inconsistency state.

RW Conflicts (or Unrepeatable Read)

- This happens when the transaction T_2 has modified the data item A that has been read by another transaction T_1, while T_1 is still in progress. If T_1 tries to read the value of A again, it will get a different value, even though it has not modified A in the mean time. This is called *unrepeatable read.*
- As an example, consider the following set of transactions (T_1 & T_2) and the schedule for executing T_1 & T_2.

  T_1: Reduce account A by $100

  T_2: Reduce account B by $100



RW conflict

Again, the execution of transactions T_1 & T_2 using the above schedule leads to database inconsistency.

WW Conflicts (or Blind Write)

- This happens when the transaction T_2 could overwrite the value of data item A, which has already been modified by another transaction T_1, while T_1 is still in progress. Such a write is called *blind write.*

- As an example, consider the following set of transactions (T_1 & T_2) and the schedule for executing T_1 & T_2.

  T_1: Sets the salaries of X and Y to $1000

  T_2: Sets the salaries of X and Y to $2000

T_1	T_2
Write(X)	
	Write(Y)
Write(Y)	
	Write(X)
	Commit
Commit	

Blind write

WW conflict

Again, the execution of transactions T_1 & T_2 using the above schedule leads to database inconsistency.

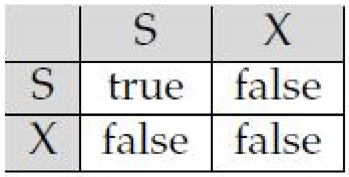
3. Lock-Based Protocols

- One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item no other transaction can modify that data item. A DBMS typically uses a *locking protocol* to achieve this.

Locks

- There are various modes in which a *data item may be locked.*
  1. **Shared Lock:** If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.
  2. **Exclusive Lock:** If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

Following figure shows compatibility of locks.



	S	X
S	true	false
X	false	false

- From the figure it is clear that shared lock is compatible with shared lock, but not with exclusive lock. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

The Two-Phase Locking Protocol (2PL)

- It ensures serializability.
- This protocol requires that each transaction issue lock and unlock requests in two phases:
  1. **Growing phase.** A transaction may obtain locks, but may not release any lock.
  2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.
- Initially, a transaction is in the *growing phase*. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the *shrinking phase*, and it can issue no more lock requests.

T_1	T_2
Lock-X(A)	
Read(A)	
Write(A)	
Lock-X(B)	
Unlock(A)	
	Lock-X(A)
	Read(A)
	Write(A)
Read(B)	
Write(B)	
Unlock(B)	

	Lock-X(B)
	Unlock(A)
	Read(B)
	Write(B)
	Unlock(B)

Schedule – 1

- As an example, consider the following set of transactions (T_1 & T_2)

  T_1: Transfers $100 from account A to account B

  T_2: Increments the balances of account A and account B by 10%

  T_1 & T_2 can be executed using **schedule – 1** under Two-phase locking protocol.

Advantages

- It ensures serializability
- It is a simple and straightforward protocol.

Disadvantages

- Deadlocks may occur.

  Consider the following partial schedule. This schedule is under Two-phase locking protocol. However it leads to a deadlock since *T*3 is holding an exclusive-mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*. Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**.

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)	

- Cascading rollbacks may occur.

  Consider the following partial schedule. Each transaction in this schedule is observing Two-phase locking protocol. However it leads to cascading rollbacks. If T_5 aborts after *read(A)* of T_7 then it leads to the rollback of T_5 which in turn leads to the rollback of T_6 and T_7 as T_6 has read the modified value of A by T_5 and T_7 has read the modified value of A by T_6. If rollback of a transaction causes rollback of a series of transactions then such rollbacks are called as cascading rollbacks.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Strict two-phase locking protocol (Strict 2PL)

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**.

- It also ensures serializability.
- It has the following rules;

  Rule 1: It is similar to two-phase locking protocol.

  Rule 2: All exclusive locks held by a transaction are released only when the transaction commit/abort.
- As an example, consider the following two transactions;

  T_1: Transfers $100 from account A to account B
  T_2: Increments the balances of account A and account B by 10%

  The following schedule results for executing T_1 & T_2 under Strict Two-phase locking protocol.

T_1	T_2
Lock-X(A)	
Read(A)	
Write(A)	
Lock-X(B)	
Read(B)	
Write(B)	
commit	
Unlock(A)	
Unlock(B)	
	Lock-X(A)
	Read(A)
	Write(A)
	Lock-X(B)
	Read(B)
	Write(B)
	commit
	Unlock(A)
	Unlock(B)

Rigorous two-phase locking protocol

- Another variant of two-phase locking is the **rigorous two-phase locking protocol.**
- It also ensures serializability.
- It has the following rules;

  Rule 1: It is similar to two-phase locking protocol.
  Rule 2: All locks held by a transaction are released only when the transaction commit/abort.
- With rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.
- As an example, consider the following two transactions;

  T_1: Transfers \$100 from account A to account B
  T_2: Increments the balances of account A and account B by 10%

  The following schedule results for executing T_1 & T_2 under Strict Two-phase locking protocol.

T_1	T_2
Lock-X(A)	
Read(A)	
Write(A)	
Lock-X(B)	
Read(B)	
Write(B)	
commit	
Unlock(A)	
Unlock(B)	
	Lock-X(A)
	Read(A)
	Write(A)
	Lock-X(B)
	Read(B)
	Write(B)
	commit
	Unlock(A)

	Unlock(B)

4. Timestamp-Based Protocols

- Another method for determining the serializability order is a **timestamp-ordering** scheme.

Timestamps

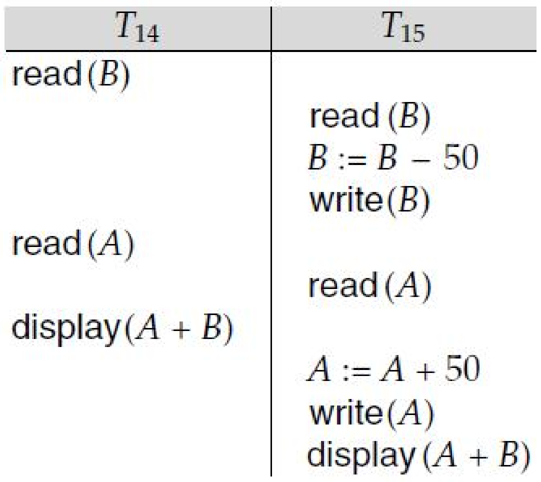
- With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution.
- Usually, the value of the system clock is used as the timestamp; that is, a transaction's timestamp is equal to the value of the system clock when the transaction enters the system.
- The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj .
- To implement this scheme, we associate with each data item Q two timestamp values:
  a) **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully
  b) **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.
- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The Timestamp-Ordering Protocol

- The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:
  1. Suppose that transaction *Ti* issues read(*Q*).
     a. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the read operation is rejected, and *Ti* is rolled back.
     b. If TS(*Ti*) ≥ W-timestamp(*Q*), then the read operation is executed, and R-timestamp(*Q*) is set to the maximum of R-timestamp(*Q*) and TS(*Ti*).

2. Suppose that transaction *Ti* issues write(*Q*).
   a. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously. Hence, the system rejects the write operation and rolls *Ti* back.
   b. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, the system rejects this write operation and rolls *Ti* back.
   c. Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

- If a transaction *Ti* is rolled back by the concurrency-control scheme as a result of issuance of either a read or write operation, then the system assigns it a new timestamp and restarts it.
- To illustrate this protocol, we consider the following two transactions.
  T_{14}: Displays the contents of accounts A and B
  T_{15}: Transfers $50 from account A to account B, and then displays the contents of both.
- Following schedule results for executing T_{14} and T_{15} under Timestamp-based protocol.

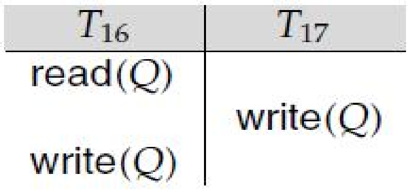


T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

- The main advantage of Timestamp-based protocol is that it is free from deadlocks.

Thomas' Write Rule

- Consider the following schedule and apply the timestamp-ordering protocol.



T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- Since T_{16} starts before T_{17}, we shall assume that $TS(T_{16}) < TS(T_{17})$. The read(Q) operation of T_{16} succeeds, as does the write(Q) operation of T_{17}.When T_{16} attempts its write(Q) operation, we find that $TS(T_{16})$ < W-timestamp(Q). Thus, the write(Q) by T_{16} is rejected and transaction T_{16} must be rolled back.
- Although the rollback of T_{16} is required by the timestamp-ordering protocol, it is unnecessary. Since T_{17} has already written Q, the value that T_{16} is attempting to write is one that will never need to be read. Any transaction Ti with TS(Ti) < $TS(T_{17})$ that attempts a read(Q) will be rolled back, since TS(Ti) < W-timestamp(Q). Any transaction Tj with TS(Tj) > $TS(T_{17})$ must read the value of Q written by T_{17}, rather than the value written by T_{16}.
- This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances.
- The modification to the timestamp-ordering protocol is called Thomas' write rule and is given as;
- Suppose that transaction Ti issues write(Q)
  1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed. Hence, the system rejects the write operation and rolls Ti back.
  2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
  3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).
- The difference between these rules and the timestamp-ordering protocol lies in the second rule. The timestamp-ordering protocol requires that *Ti* be rolled back if *Ti* issues write(*Q*) and TS(*Ti*) < W-timestamp(*Q*). However, here, in Thomas' Write Rule, we ignore the obsolete write.

- Thomas' write rule makes use of view serializability and increases concurrency.

6. Deadlock Handling

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- For example there exists a set of waiting transactions {T0, T1, . . ., Tn} such that T0 is waiting for a data item that T1 holds, and T1 is waiting for a data item that T2 holds, and . . ., and Tn−1 is waiting for a data item that Tn holds, and Tn is waiting for a data item that T0 holds. None of the transactions can make progress in such a situation.
- There are two principal methods for dealing with the deadlock problem.
  1. deadlock prevention
  2. deadlock detection and deadlock recovery scheme.

Deadlock Prevention

- Two different deadlock prevention schemes using timestamps have been proposed:

  a. The **wait–die** scheme is a non preemptive technique. When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj (that is, Ti is older than Tj). Otherwise, Ti is rolled back (dies).

     For example, suppose that transactions T22, T23, and T24 have timestamps 5, 10, and 15, respectively. If T22 requests a data item held by T23, then T22 will wait. If T24 requests a data item held by T23, then T24 will be rolled back.

  b. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).

Example, with transactions T22, T23, and T24, if T22 requests a data item held by T23, then the data item will be preempted from T23, and T23 will be rolled back. If T24 requests a data item held by T23, then T24 will wait.

- The major problem with both of these schemes is that *unnecessary rollbacks may occur.*

Deadlock Detection and Recovery

- An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. This uses a directed graph called a **wait-for graph** for Deadlock Detection.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.
- The set of **vertices** represents the **transactions** in the system and there is an **edge form T_i to T_j if T_i waits for T_j**.
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle**.

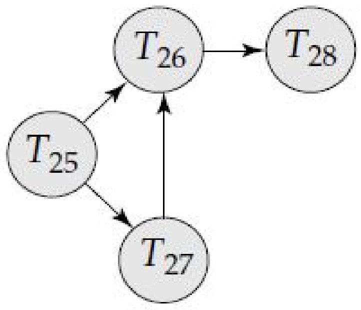


Fig. 3: Waits-for graph

- To illustrate these concepts, consider the wait-for graph in Figure 3, which depicts the following situation:
  - Transaction T25 is waiting for transactions T26 and T27.
  - Transaction T27 is waiting for transaction T26.
  - Transaction T26 is waiting for transaction T28.
- Since the graph has no cycle, the system is not in a deadlock state.

- Suppose now that transaction T28 is requesting an item held by T27. The edge T28 → T27 is added to the wait-for graph, resulting in the new system state in Figure 4. This time, the graph contains the cycle

  T26 → T28 →T27 →T26 implying that transactions T26, T27, and T28 are all deadlocked.

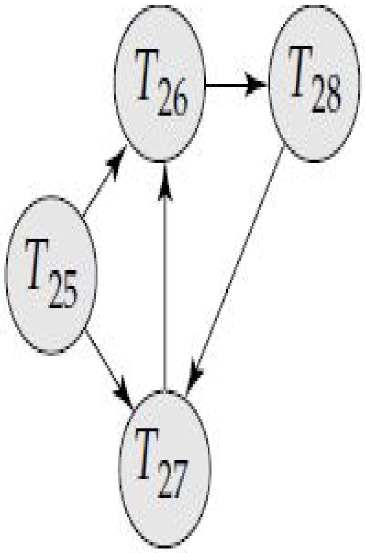


Fig. 4: Waits-for graph

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution for deadlock recovery is to roll back one or more transactions to break the deadlock. Three actions need to be taken:
  1. **Selection of a victim**: Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Factors that determine the cost of a rollback, including
     a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
     b. How many data items that the transaction has used.
     c. How many more data items the transaction needs for it to complete.

   d. How many transactions will be involved in the rollback.

2. **Rollback:** Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   a. **Total rollback**: Abort the transaction and then restart it.
   b. **Partial rollback:** roll back the transaction only as far as necessary to break the deadlock.
3. **Starvation:** In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

UNIT-V

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. If a transaction acquires a shared lock, then it can perform_____________operation. [ ]
   A) Read B) Write C) Read and Write D) Update
2. If a transaction obtains an exclusive lock on a row, it means that the transaction wants to _______________that row. [ ]
   A) Select B) Update C) View D) Read
3. In a two-phase locking protocol, a transaction release locks in _____________phase. [ ]
   A) Shrinking phase B) Growing phase C) Running phase D) Initial phase
4. In time stamp based protocol, transactions are executed based on their____________
5. ____________protocol ensure that the system will never enter into a deadlock state.
6. Deadlocks can be described precisely in terms of a directed graph called _________
7. In strict 2PL [ ]
   1. Locking be in 2PL
   2. All exclusive locks must be held until transaction commits
   3. All shared and exclusive locks must be held until transaction commits
   A) Both 1 and 2 B) Only 2 C) Only 2 D) All of the above
8. In rigorous 2PL [ ]
   1. Locking be in 2pl
   2. All shared and exclusive locks must be held until transaction commits
   A) Both 1 and 2 B) Only 2 C) Only 1 D) None of the above
9. Two phase locking doesn't ensure [ ]

A) Freedom from deadlock B) Cascading rollbacks

C) Both a and b D) Either a or b

10. Test whether the following schedule observes i) 2PL ii) Strict 2pl iii) Rigorous 2PL [ ]

Lock- S(A)

R(A)

Lock –X(B)

R(B)

Unlock(A)

W(B)

Unlock(B)

A) Only I B) I & II C) I, II, & III D) None

11. Test whether the following schedule observes i) 2PL ii) Strict 2pl iii) Rigorous 2PL [ ]

Lock- S(A)

R(A)

Lock –X(B)

Unlock (A)

R(B)

W(B)

Commit

Unlock (B)

A) Only I B) I & II C) I, II, & III D) None

12. Test whether the following schedules observes i) 2PL ii) Strict 2pl iii) Rigorous 2PL [ ]

Lock- S(A)

R(A)

Lock –X(B)

Write(B)

Unlock(A)

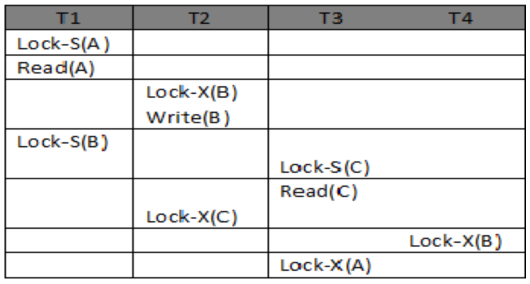
Unlock(B)

A) Only I B) I & II C) I, II, & III D) None

SECTION-B

Descriptive Questions

1. Why concurrency control is needed? Explain the problems that would arise when concurrency control is not provided by the database system.
2. Identify the anomalies due to concurrent execution of transactions. (Dirty Read, Unrepeatable Read, Blind Write)
3. What is a lock? List the types of lock.
4. Define 2-Phase Locking. Differentiate 2PL, Strict 2PL and Rigorous 2PL.
5. Discuss in detail about Time Stamp Based Protocol and Thomas Write Rule.
6. What is deadlock? Illustrate different deadlock handling techniques.
7. Outline the actions to be taken to recover from a deadlock.
8. Draw the waits-for graph for the following schedule and test whether this schedule leads to a deadlock?



T1	T2	T3	T4
Lock-S(A)			
Read(A)			
	Lock-X(B) Write(B)		
Lock-S(B)		Lock-S(C)	
	Lock-X(C)	Read(C)	
			Lock-X(B)
		Lock-X(A)	

9. Describe wait/die and wound/wait deadlock protocols.
10. Consider the following schedules;

    S1: T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)

    S2: T1:R(X),T1:R(Y),T1:W(X),T2:R(Y),T3:W(Y),T1:W(X),T2:R(Y)

For each of the above schedules, state whether timestamp-based protocol allows the actions to occur in exactly the order shown.

SECTION-C

GATE Questions

1) Which of the following concurrency control protocols ensure both conflict serialzability and freedom from deadlock? [GATE 2010]

I) 2-phase locking

II) Time-stamp ordering [ ]

A) I only B) II only C) Both I and II D) Neither I nor II