Chapter 16 : Concurrency Control

Database System Concepts 5th Ed.

© Silberschatz, Korth and Sudarshan, 2005
See www.db-book.com for conditions on re-use

Chapter 16: Concurrency Control

- Concurrency Control
- Problems of concurrency control
- Concurrency Control Schemes
  - Lock-based Protocols
  - Timestamp-based Protocols

Concurrency Control

- Concurrency control is the procedure in DBMS for managing simultaneous operations of transactions which execute without conflicting with each another
- Concurrent access is quite easy if all users are just reading the data
- But, any practical database would have a mix of READ and WRITE operations and hence, the concurrency control is a challenge
- Concurrency control is used to address such conflicts which mostly occur in a multi-user system and it helps us in assuring that database transactions are performed concurrently without violating the consistency of the database system

Problems of Concurrency Control

- Several problems can occur when concurrent transactions are executed in an uncontrolled manner.
- The following are the three problems that may occur during concurrent execution of the transactions:
  - **Temporary Update Problem**
  - **Incorrect Summary Problem**
  - **Lost Update Problem**

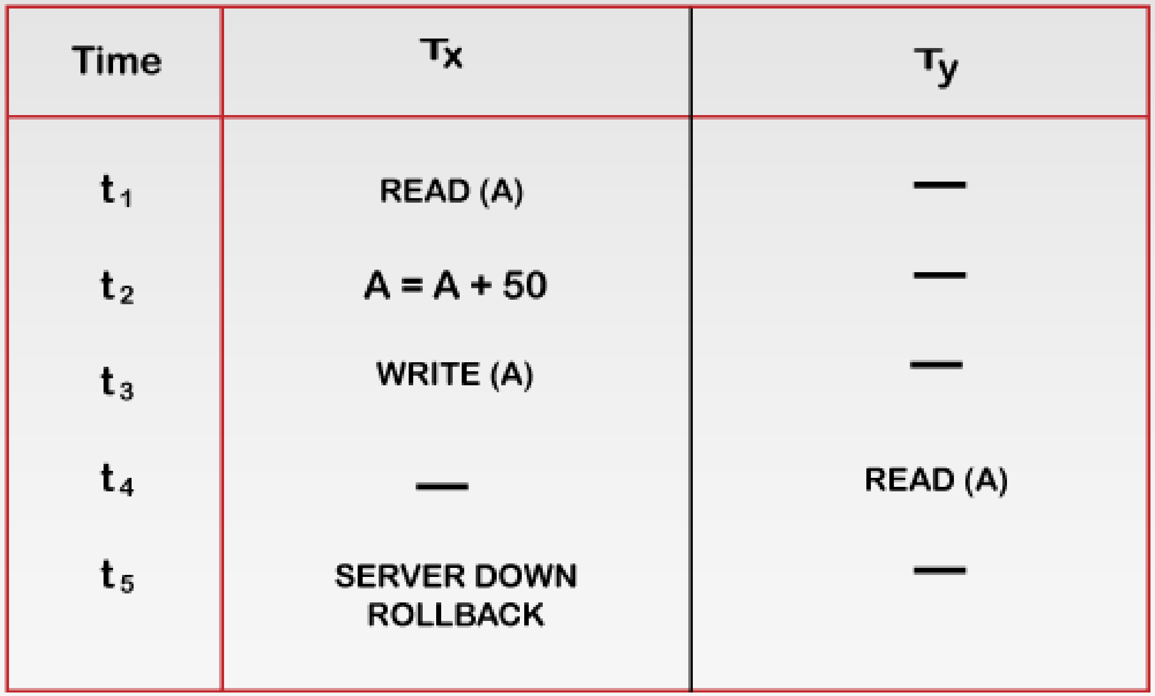
Problems of Concurrency Control

- **Temporary Update Problem/Dirty Read Problem**
  - Suppose one transaction updates a data item but fails due to some reason before committing.
  - But the updated data item is accessed by another concurrent transaction before the data item value is reverted back to its earlier value (roll back).
  - Thus, the **dirty read problem** occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rolled back, the updated database item is accessed by another running transaction.*

Problems of Concurrency Control

An Example of Dirty Read Problem

- Consider two transactions T_X and T_Y in the diagram below performing read/write operations on account A where the initial balance in account A is suppose $300:



Time	Tx	Ty
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

Problems of Concurrency Control

Incorrect Summary Problem

- Consider a situation, where one transaction is applying a aggregate function on some records while another concurrent transaction is updating these records
- The **incorrect summary problem** happens when the aggregate function read some values before the values have been updated and others after they are updated and calculate the aggregate value accordingly.

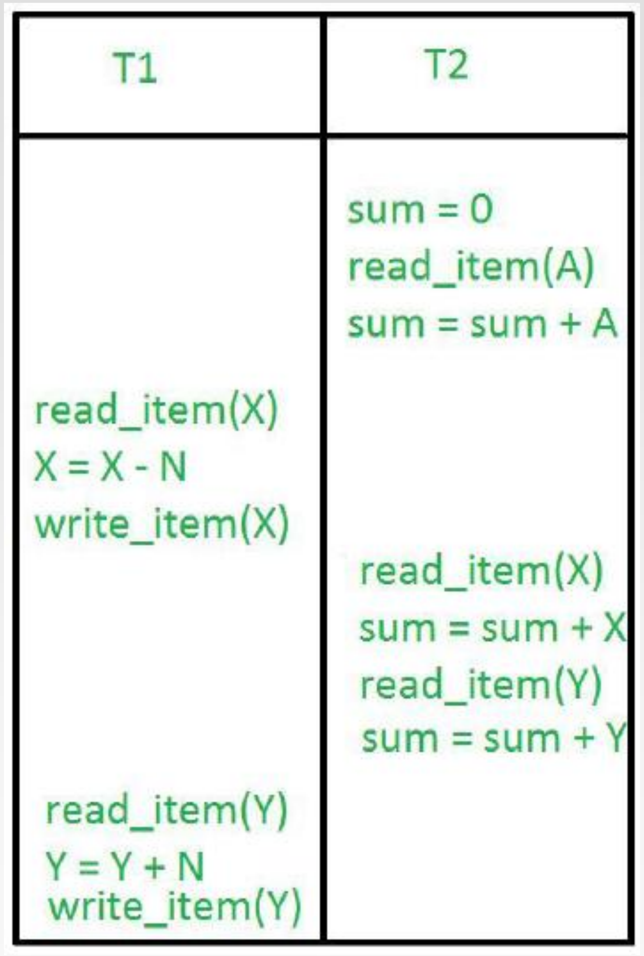
Problems of Concurrency Control

An Example of Incorrect Summary Problem

- Transaction T2 is calculating the sum of some records while transaction T1 is updating them.
- Therefore, the aggregate function may calculate some values before they have been updated and others after they have been updated.

Problems of Concurrency Control

- **Incorrect Summary Problem**



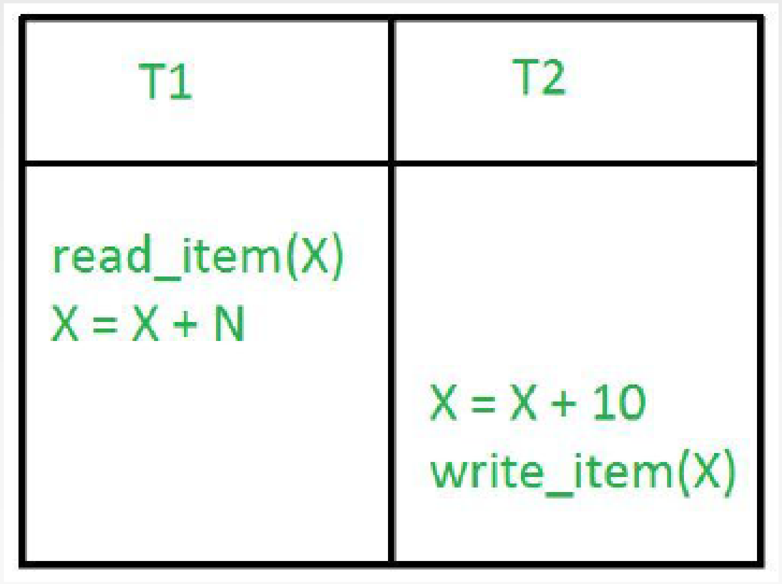
T1	T2
	sum = 0
	read_item(A)
	sum = sum + A
read_item(X)	
X = X - N	
write_item(X)	
	read_item(X)
	sum = sum + X
	read_item(Y)
	sum = sum + Y
read_item(Y)	
Y = Y + N	
write_item(Y)	

- In this example, T2 is calculating the sum of some attribute values while T1 is updating them.
- The aggregate function may read some values before they have been updated and others after they have been updated and calculate accordingly.

Problems of Concurrency Control

Lost Update Problem

- In the lost update problem, update done to a data item by a transaction is lost as it is overwritten by the update done by another concurrently running transaction.
- **Example:** Transaction T1 changes the value of X but it is overwritten by the update done by transaction T2 on X. Therefore, the update done by transaction T1 is lost.



T1	T2
read_item(X)	
X = X + N	
	X = X + 10
	write_item(X)

CONCURENCY CONTROL SCHEMES

- Lock-based Protocols
- Timestamp-based Protocols

Lock-Based Protocols

- A **lock** is a mechanism to control concurrent access to a data item by various transactions
- A lock is expressed as a variable associated with a data item that describes the status of the item w.r.t the possible operations that can be applied to it.
- Data items can be locked in the following two modes:
  1. **exclusive (X) mode:** Once X-mode lock is obtained on a data item, it can be both read as well as written.
  2. **shared (S) mode:** Once S-mode lock is obtained on a data item, it can only be read.
- Locks on data items can be obtained by transactions by placing requests, which are made to the **concurrency-control manager**.
- A transaction can proceed with its execution only after the request is granted.

Lock-Compatibility Matrix

- Given a set of lock modes, a **compatibility function** can be defined on them as follows:

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold S-locks on an item.
- But if any transaction holds an exclusive on the item, then no other transaction can hold any lock on the same item till the first transaction does not releases its X-lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Thereafter, the lock is granted.

Lock-Based Protocols

- To access a data item, a transaction *Ti* must first lock that item.
- If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.
- Thus, *Ti* is made to **wait** until all incompatible locks held by other transactions have been released.

Lock Instructions

- Lock instructions in concurrency control are used to prevent uncontrolled data access and ensure that only one user or application can modify a data resource at a time.
- The types of lock instructions are as follows:
  - **lock-S(Q)**: Instruction to request for a **shared mode lock** on a data item Q. This lock allows multiple transactions to share a data item but not modify it.
  - **lock-X(Q)**: Instruction to request for a **exclusive mode lock** on a data item Q. This lock allows a transaction to read and write data items.
  - **unlock(Q):** A transaction can unlock a data item Q.
- A transaction must ahold a lock on a data item as long as it access that data item

Example of Transactions that Perform Locking

```
T1: lock-X(B);
    read(B);
    B := B – 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Transaction T1

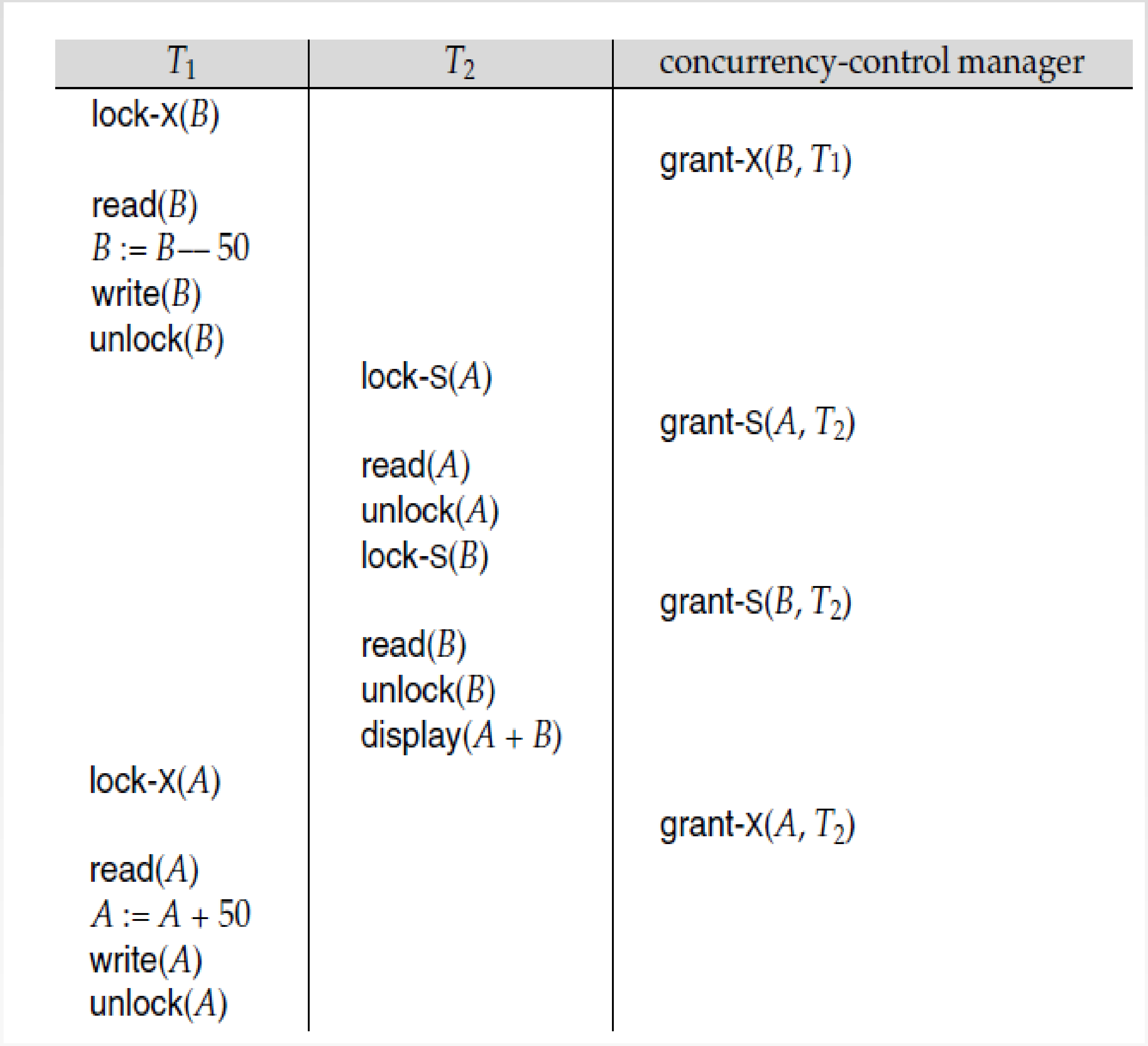
```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Transaction T2

Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. *T*1 transfers $50 from account *B* to account *A*. *T*2 displays the total amount of money in accounts *A* and *B.*

Role of Concurrency-Control Manager

Schedule 1



T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

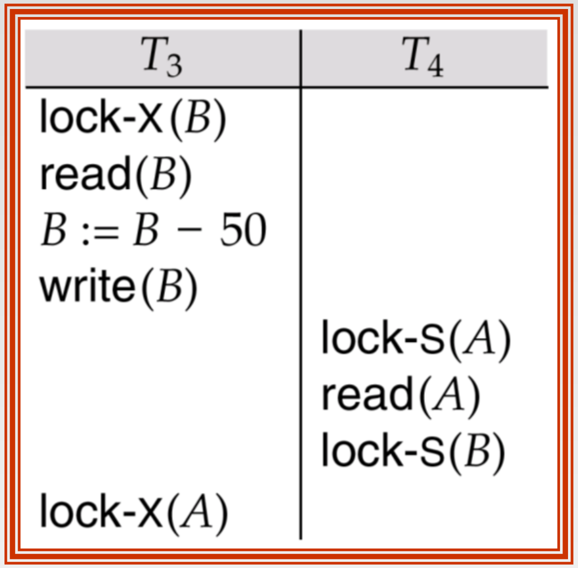
- The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks.
- The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock.
- Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.

Lock-Based Protocols (Cont.)

- Suppose that the initial values of accounts *A* and *B* are \$100 and \$200, respectively.
- If these two transactions are executed serially, either in the order <T1, T2> or the order <T2, T1>, then transaction T2 will display the value \$300.
- If, however, these transactions are executed concurrently, as shown in Schedule 1 then in this case, transaction T2 displays \$250, which is incorrect.
- Thus, only locking in this manner is not sufficient to guarantee serializability.

Pitfalls of Lock-Based Protocols

- Consider the following partial 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)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

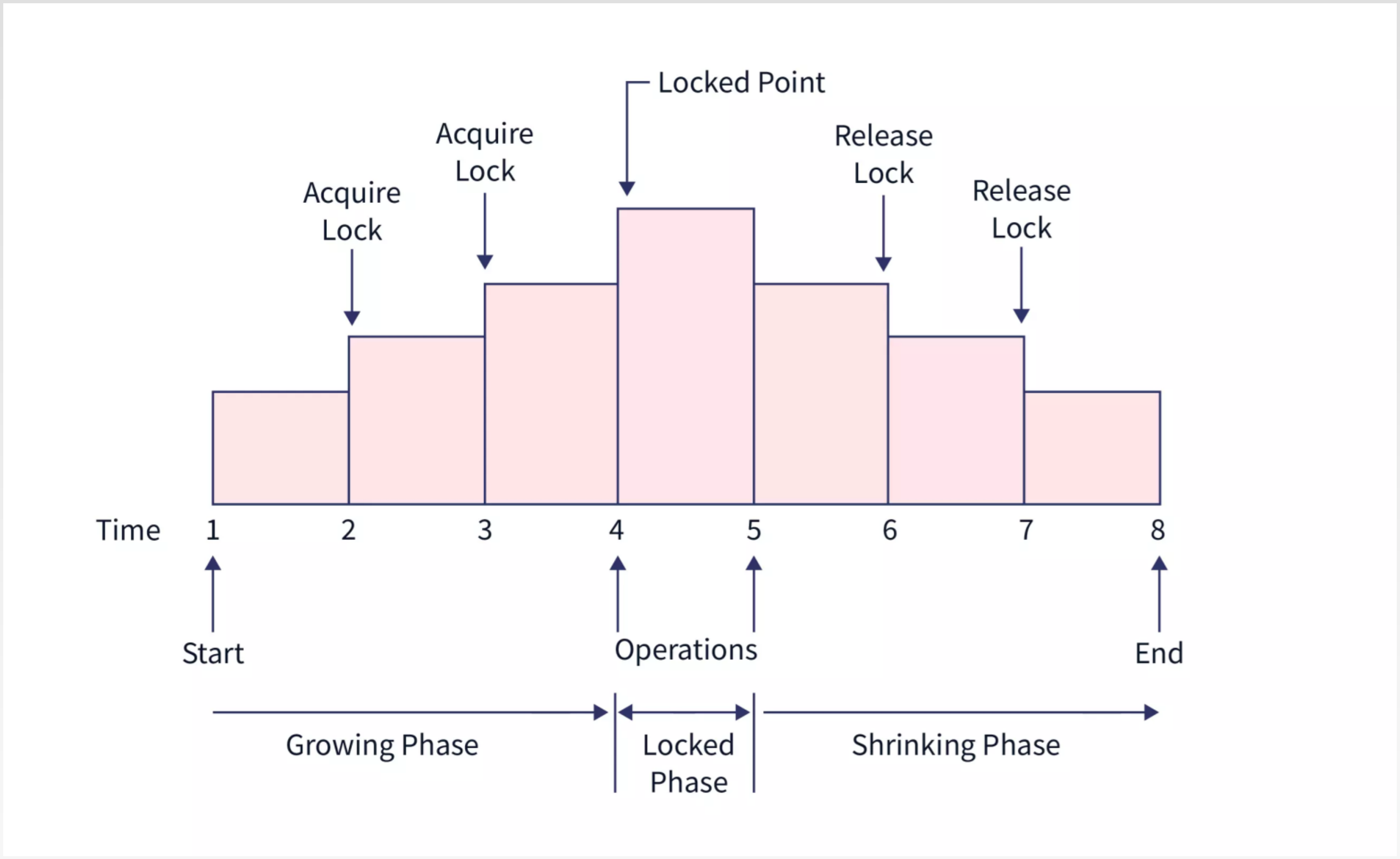
Pitfalls of Lock-Based Protocols (Cont.)

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - Suppose, a transaction is requesting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The transaction requesting for X-lock on the item is said to be **starved** as it has to wait for all the transactions to release their S-locks.
- Concurrency control schemes can be designed properly to prevent starvation.

The Two-Phase Locking Protocol

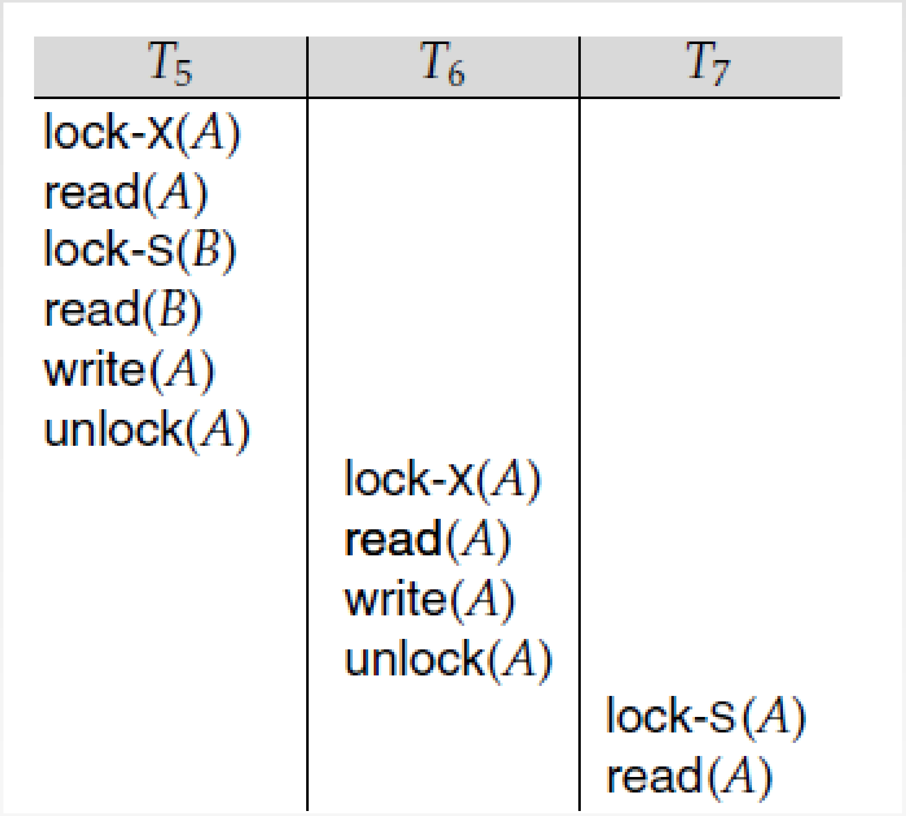
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- This is a protocol which ensures the generation of **conflict-serializable schedules**. Thus, locking protocols restrict the set of possible schedules.
- The Two-phase Locking (2PL) protocol works in two phases:
- **Phase 1: Growing Phase**
  - A transaction can obtain locks
  - A transaction do not release locks
- **Phase 2: Shrinking Phase**
  - A transaction can release locks
  - A transaction cannot obtain locks
- It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol



Disadvantages of Two-Phase Locking Protocol

- Two-phase locking does not ensure freedom from **deadlocks**
- **Cascading rollback** is possible under two-phase locking.



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)

Consider the partial schedule as shown in the figure.
Each transaction observes the 2PL protocol. But, the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

Modified 2PL Protocols

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**.
- This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which is even stricter. It requires that all locks be held until the transaction commits.

Modified 2PL Protocols

Two-Phase Locking Protocol with Lock Conversions:

- This is a refinement of the basic 2PL protocol in which lock conversions are allowed.
  - **First Phase:**
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - **Second Phase:**
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol ensures serializability.

Timestamp-Based Protocols

- Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously.
- This approach relies on **timestamps** to manage and coordinate the execution order of transactions.
- Each transaction T_i is issued a unique timestamp **TS(T_i)** when it enters the system.
- The timestamp is assigned by the DBS before the transaction starts its execution.
- If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) then, **TS(T_i) < TS(T_j)**.

Timestamps

- There are two simple methods for assigning the timestamp:
  1. Use the value of the **system clock** as the timestamp. A transaction's timestamp is equal to the value of the clock when the transaction enters the system.
  2. Use a **logical counter** that is incremented after a new timestamp has been assigned. The transaction's timestamp is equal to the value of the counter when the transaction enters the system.
- The timestamps of the transactions determine their 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* .

Timestamps

- In order to implement this concurrency-control scheme, the protocol maintains two timestamp values for each data *Q* :
  - **W-timestamp(*Q*):** It is the largest timestamp of any transaction that executed **write(*Q*)** successfully.
  - **R-timestamp(*Q*):** It is 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.

Timestamp-Based Protocols

- The **timestamp-ordering protocol** ensures that any **conflicting read** and **write** operations are executed in timestamp order.
- Whenever some Transaction T_i tries to issue a read(Q) or a write(Q) operation, the algorithm compares the timestamp of T_i i.e., **TS(T_i)** with **W-timestamp(*X*)** and **R-timestamp(*X*)** to ensure that the timestamp order of transactions is not violated.

Steps of Basic Timestamp-Based Protocol

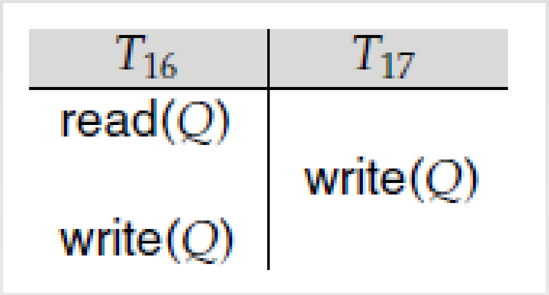
- The timestamp ordering protocol operates as follows:
- Suppose that transaction ***Ti*** issues **read(*Q*)**:
  - a. If **TS(*Ti*) < W-timestamp(*Q*)**, then *Ti* needs to read a value of *Q* that was already overwritten.
    - Hence, the read operation is rejected, and *Ti* is rolled back.
  - b. If **TS(*Ti*) ≥ W-timestamp(*Q*)**, then the read operation is executed, and R-timestamp(*Q*) is updated to the maximum of current R-timestamp(*Q*) value and TS(*Ti*).
  - c. => R-timestamp(*Q*) = max(Current R-timestamp(*Q*), TS(*Ti*))

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write(*Q*)**:
  1. If **TS(T_i) < R-timestamp(*Q*)**, then the value of Q that T_i is producing was needed previously. Thus, the system assumed that that value would never be required.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If **TS(T_i) < W-timestamp(*Q*)**, then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).
- If a transaction is rolled back by the concurrency-control scheme, then the system assigns it a new TS and restarts it.

Modified Version of the Timestamp-ordering Protocol

- Let us consider the following **Schedule 4** and apply the timestamp-ordering protocol on it.



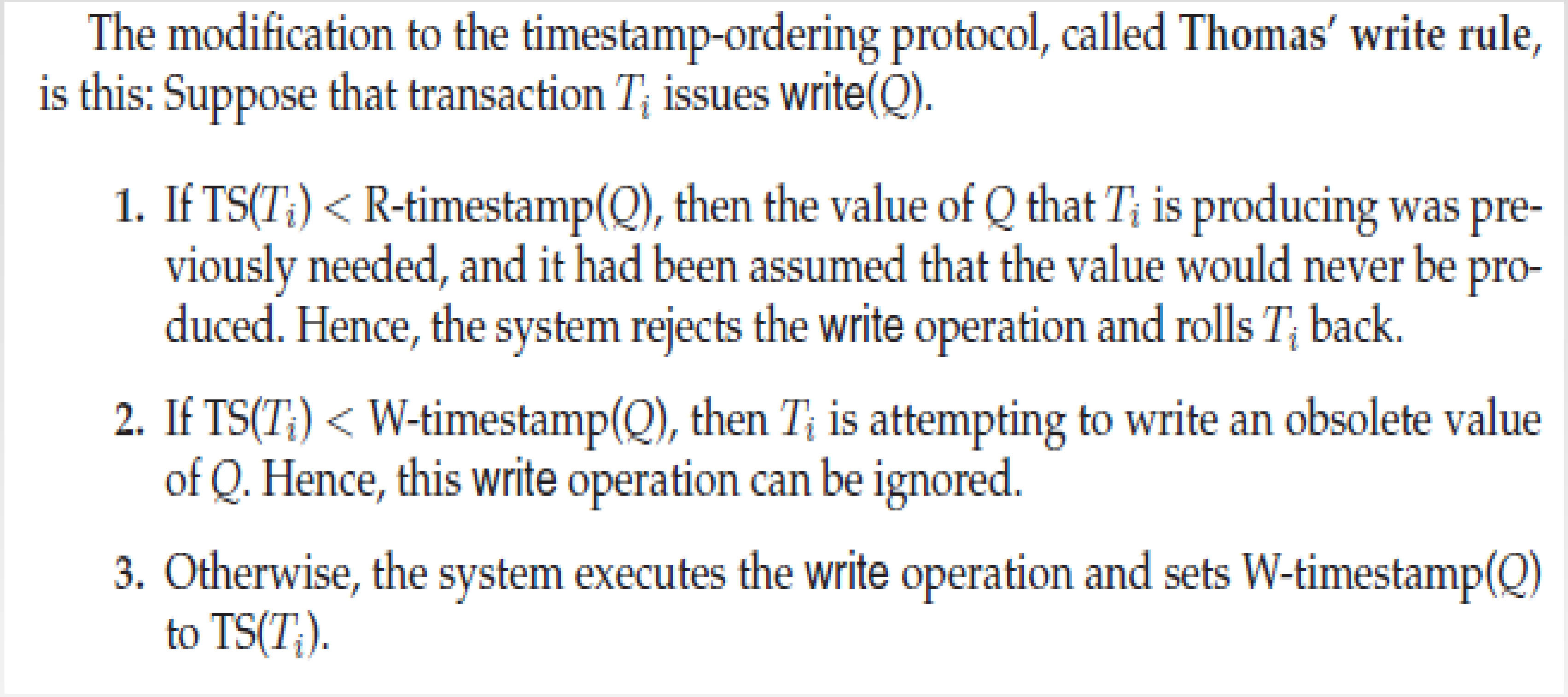
T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- 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*), since W-timestamp(*Q*) = TS(*T*17).
- Thus, the write(*Q*) by *T*16 is rejected and transaction *T*16 must be rolled back.

Modified Version of the Timestamp-ordering Protocol

- 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 that will never be read.
- 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 protocol rules for read operations remain unchanged.
- The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.

Thomas' Write Rule



The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction T_i issues write(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
2. If TS(T_i) < W-timestamp(Q), then T_i 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(T_i).