Concurrency Control

Query Processing

- When several transactions execute concurrently in the database, the isolation property may no longer be preserved.
- The system must control the interaction among the concurrent transactions, this control is achieved through one of a variety of mechanisms called **Concurrency Control Schemes**
- There are a variety of concurrency-control schemes.
- No one scheme is clearly the best; each one has advantages.
- In practice, the most frequently used schemes are ***two-phase locking*** and ***snapshot isolation***.

Lock Based Protocols

- In this case data items should be accessed in **mutually exclusive manner**
- **While one transaction is accessing a data item, no other transaction can modify that data item**
- A transaction can access a data item only if it is currently holding a **lock** on that item.

Locks

There are various modes in which a data item may be locked, we restrict our attention to two modes:

1. **Shared**: 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**: If a transaction Ti has obtained an **exclusive-mode lock** (denoted by **X**) on item *Q*, then ***Ti* can both read and write Q**

•Every transaction has to **request** a lock to the concurrency-control manager in an appropriate mode on data item *Q*

• The transaction can proceed only after the concurrency-control manager **grants** the lock to the transaction.

COMPATIBILITY FUNCTION

Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* (*Ti* ≠ *Tj*) currently holds a lock of mode *B*. If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*.

	S	X
S	true	false
X	false	false

Lock – Compatibility Matrix

An element comp(*A*, *B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

LOCK REQUEST & GRANTING

- A transaction requests a shared lock on data item *Q* by executing the **lock-S(Q)** instruction. Similarly, a transaction requests an exclusive lock through the **lock-X(Q)** instruction. A transaction can unlock a data item *Q* by the **unlock(Q)** instruction.

- Consider the banking example. Let *A* and *B* be two accounts that are accessed by transactions T_1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A* (Figure 1). Transaction *T*2 displays the total amount of money in accounts *A* and *B* (*A* + *B*) *as shown in Figure 2.*

Lock Request & Granting

***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)
```

Figure 1 Transaction T1.

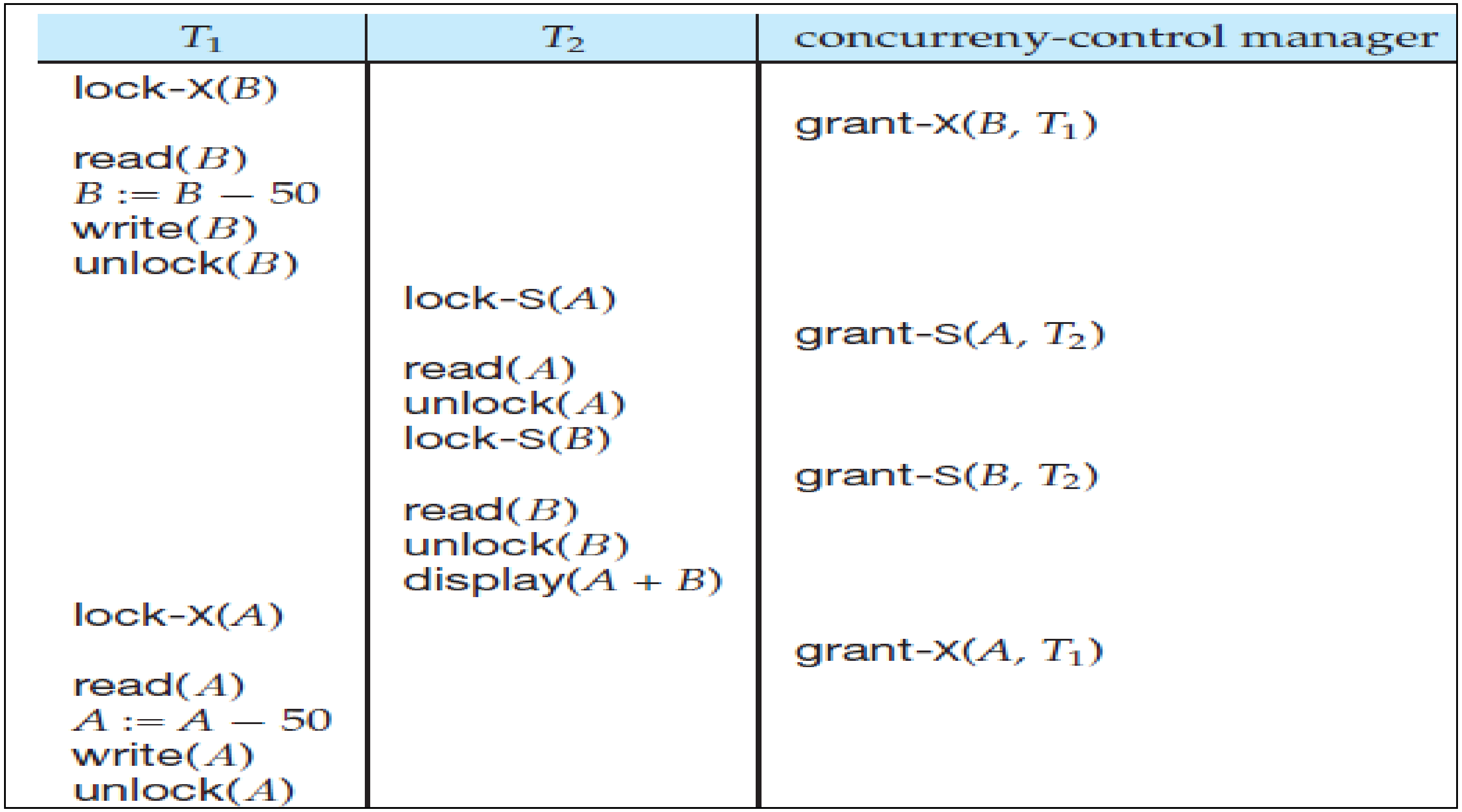
***T2*:**

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

Figure 2 Transaction T2

Suppose that the 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, then schedule 1, in Figure 3, is possible.

Lock Request & Granting



T_1	T_2	concurreny-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_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure 3 Schedule 1

In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction **T1 unlocked data item B too early**, as a result of which T2 saw an inconsistent state.

Lock Request & Granting

Suppose now that unlocking is delayed to the end of the transaction. Transaction *T*3 corresponds to *T*1 with unlocking delayed (Figure 4). Transaction *T*4 corresponds to *T*2 with unlocking delayed (Figure 5).

```
T3: lock-X(B);
read(B);
B := B – 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A)
```

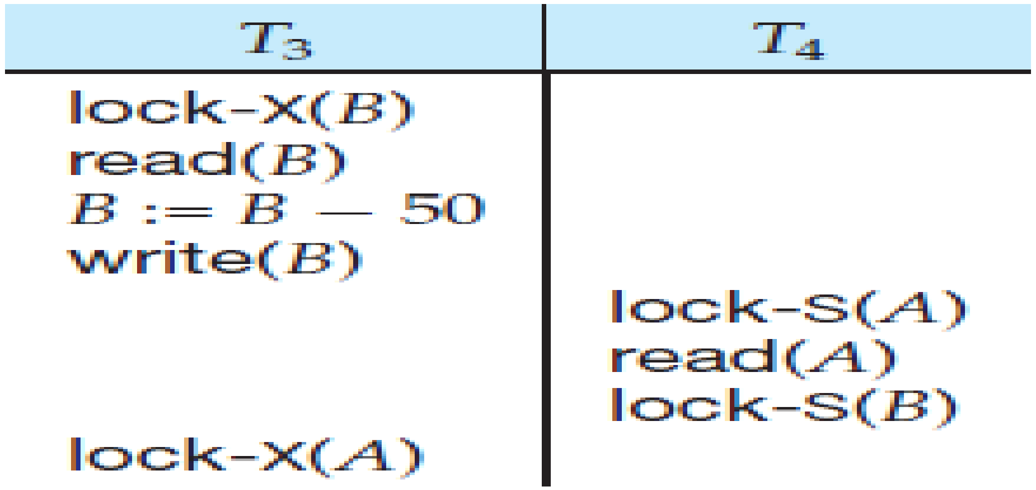
```
T4: lock-S(A);
read(A);
lock-S(B);
read(B);
display(A + B);
unlock(A);
unlock(B).
```

Figure 5 Transaction T_4(transaction *T*2 with unlocking delayed)

Figure 4 Transaction T_3 (transaction *T*1 with unlocking delayed).

Lock Request & Granting

- ➢The locking can lead to an undesirable situation. Consider the partial schedule of Figure 6 for *T*3 and *T*4.



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)	

Figure 6 Schedule 2

- ➢Neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**.
- ➢When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

STARVATION

Suppose a transaction *T*2 has a shared-mode lock on a data item, and another transaction *T*1 requests an exclusive-mode lock on the data item. Clearly, *T*1 has to wait for *T*2 to release the shared-mode lock. Meanwhile, a transaction *T*3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock. At this point *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish. But again, there may be a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 never gets the exclusive-mode lock on the data item. The transaction *T*1 may never make progress, and is said to be **starved.**

Avoiding Starvation

We can avoid starvation of transactions by granting locks in the following manner: When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that:

1. There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.

2. There is no other transaction that is waiting for a lock on *Q* and that made its lock request before *Ti* .

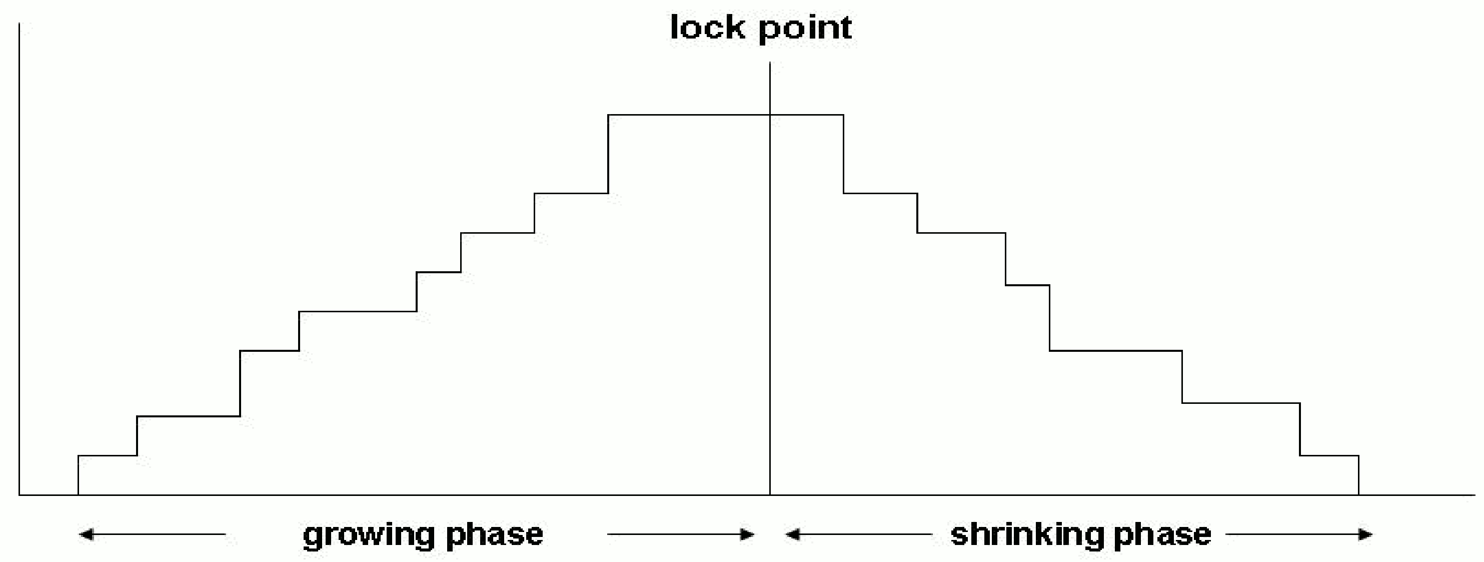
Thus, a lock request will never get blocked by a lock request that is made later.

THE TWO-PHASE LOCKING PROTOCOL

One protocol that ensures serializability is the **two-phase locking protocol**. 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.

Deadlock Handling

- Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2 (Figure 6), they are deadlocked.
- 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. More precisely, there exists a set of waiting transactions {*T*0, *T*1, . . . , *Tn*} such that *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . . , and *Tn*−1 is waiting for a data item that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds. None of the transactions can make progress in such a situation.

DEADLOCK HANDLING

➢The only remedy to deadlock is rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial (a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock)

➢ There are two principal methods for dealing with the deadlock problem:

1. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state.
2. We can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme

DEADLOCK PREVENTION

First Approach:

- The simplest scheme requires that each transaction locks all its data items before it begins execution. Disadvantages:

(1) hard to predict, before the transaction begins, what data items need to be locked (2) many of the data items may be locked but unused for a long time.

- <u>Other Scheme</u>: to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.

Deadlock Prevention

Second Approach:

- Uses **preemption** and transaction **rollbacks**.
- To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins.
- The system uses these timestamps only to decide whether a transaction should wait or roll back.
- Locking is still used for concurrency control.
- If a transaction is rolled back, it retains its *old* timestamp when restarted.
- Two different deadlock-prevention schemes using timestamps have been proposed:

Deadlock Prevention

1. 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).

Example:

Suppose that transactions *T*14, *T*15, and *T*16 have timestamps 5, 10, and 15, respectively. If *T*14 requests a data item held by *T*15, then *T*14 will wait. If *T*16 requests a data item held by *T*15, then *T*16 will be rolled back.

Deadlock Prevention

2. 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*)

Returning to our <u>example</u>, with transactions *T14*, *T15*, and *T16*, if *T14* requests a data item held by *T15*, then the data item will be preempted from *T15*, and *T15* will be rolled back. If *T16* requests a data item held by *T15*, then *T16* will wait

Deadlock Prevention

Another simple approach is based on lock timeouts:

➢A transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.

➢Easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks.

Issues:

✓Too long a wait results in unnecessary delays once a deadlock has occurred.

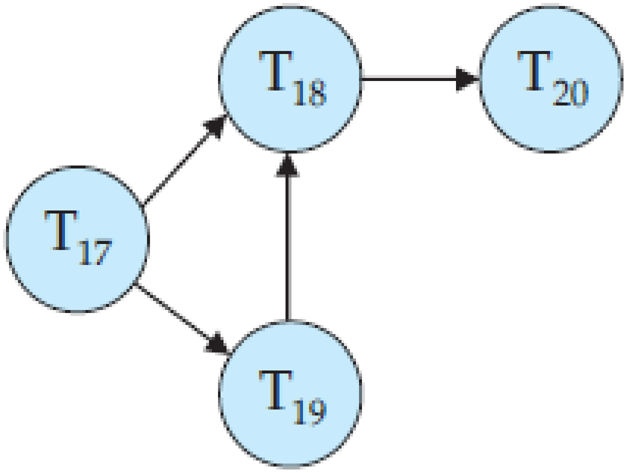
✓Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.

DEADLOCK DETECTION

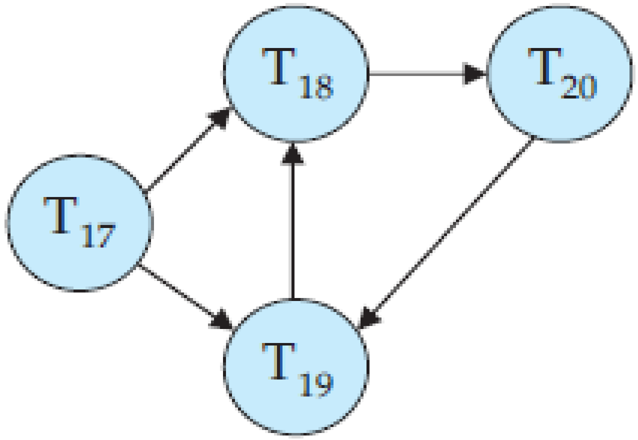
- Deadlocks can be described precisely in terms of a directed graph called a wait-for graph.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system.
- Each element in the set E of edges is an ordered pair Ti → Tj. If Ti → Tj is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs. This edge is inserted in the wait-for graph.
- This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti

Deadlock Detection

- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked



Wait-For graph with no cycle



Wait-For graph with cycle

- When should we invoke the detection algorithm? The answer depends on two factors:
  1. How often does a deadlock occur?
  2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

DEADLOCK RECOVERY

➢When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **<u>Selection of a victim:</u>**

✓ 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. Many factors may 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 the transaction has used.

Deadlock Recovery

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:

✓ How far the transaction should be rolled back. The simplest solution is a total rollback: Abort the transaction and then restart it.

✓ However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks.

DEADLOCK RECOVERY

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 a transaction can be picked as a victim only a (small) finite number of times.

Time Stamp Based Protocols

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. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*). There are two simple methods for implementing this scheme:

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. (i.e. T_1 comes at time 2:00:00 and T_2 comes at 2:00:25. This arrival time is acting as a timestamp.)

2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. (i.e. Counter value can be one from 1,2,3,4,5,….,n)

Time Stamp Based Protocols

➢ To implement this scheme, we associate with each data item *Q* two timestamp values:

• **W-timestamp**(*Q*) : denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
• **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.

Time Stamp Based Protocols

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*).

Time Stamp Based Protocols

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, and the system assumed that that value would never be produced. 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 result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Time Stamp Based Protocols

T1	T2	T3
100	200	300
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

	A	B	C
RTS	0	0	0
WTS	0	0	0

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, and the system assumed that that value would never be produced. 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*).

Validation Based Protocols

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, no checking is done while the transaction is executing.
- In this scheme, updates in the transaction are not applied directly to the database items until the transaction reaches its end.
- During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction.
- At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

Validation Based Protocols

There are three phases for this concurrency control protocol:

1. **Read phase:** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase:** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase:** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted

Validation Based Protocols

Three different timestamps are associated with each transaction *Ti* :

1. **Start**(*Ti*), the time when *Ti* started its execution.
2. **Validation**(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. **Finish**(*Ti*), the time when *Ti* finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(*Ti*). Thus, the value TS(*Ti*) = Validation(*Ti*)

Validation Based Protocols

The **validation test** for transaction *Ti* requires that, for all transactions *Tk* with TS(*Tk*) < TS(*Ti*), one of the following two conditions must hold:

1. Finish(*Tk*) < Start(*Ti*). Since *Tk* completes its execution before *Ti* started, the serializability order is indeed maintained.

2. The set of data items written by *Tk* does not intersect with the set of data items read by *Ti* and *Tk* completes its write phase before *Ti* starts its validation phase (Start(*Ti*) < Finish(*Tk*) < Validation(*Ti*)).

This condition ensures that the writes of *Tk* and *Ti* do not overlap. Since the writes of *Tk* do not affect the read of *Ti* , and since *Ti* cannot affect the read of *Tk* , the serializability order is indeed maintained.

Validation Based Protocols

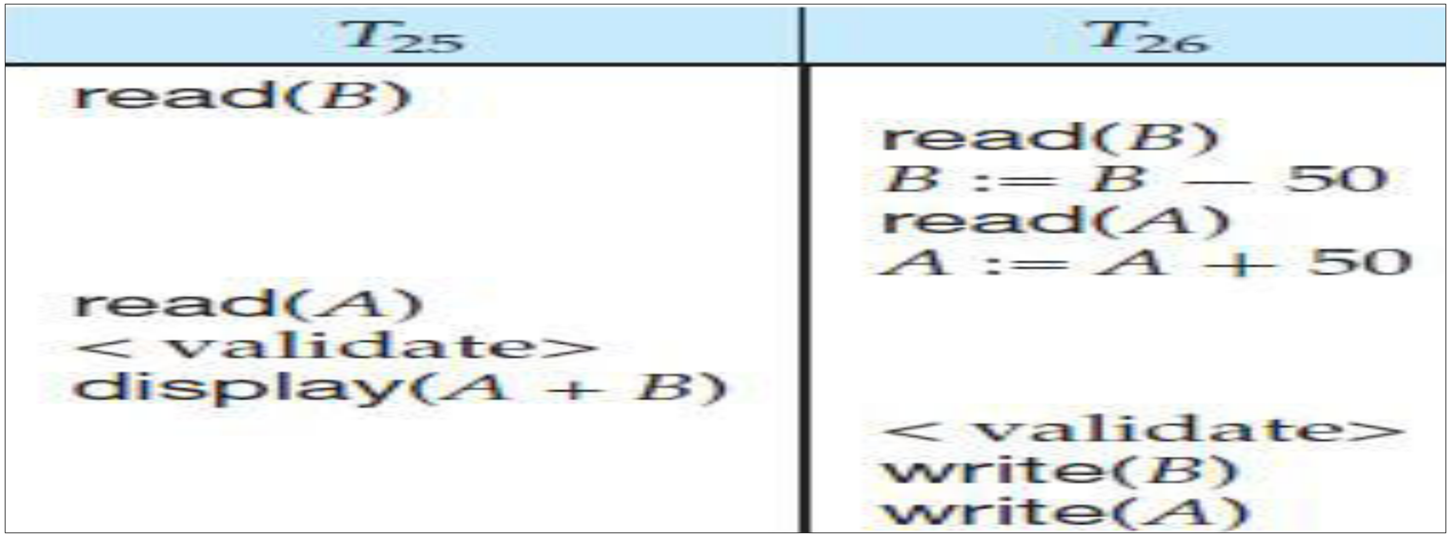
*T*25:
read(*B*);
read(*A*);
display(*A* + *B*)

*T*26:
read(*B*);
B := *B* – 50;
write(*B*);
read(*A*);
A := *A* + 50;
write(*A*);
display(*A* + *B*)

Transaction *T*26 transfers $50 from account *B* to account *A*, and then displays the contents of both:

VALIDATION BASED PROTOCOLS

Suppose that TS(*T*25) < TS(*T*26)



T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

Note: The writes to the actual variables are performed only after the validation phase of *T*26. Thus, *T*25 reads the old values of *B* and *A*, and this schedule is serializable.