Unit 5: Transactions and Concurrency Control

Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- Consider a simple bank application consisting of several accounts and a set of transactions that access and update those accounts.
- Transactions access data using two operations:
  - **read(*X*):** which transfers the data item *X* from the database to a local buffer variable belonging to the transaction that executed the read operation.
  - **write(*X*):** which transfers the of data from local buffer of the transaction that executed the write back to the database.
- E.g. transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. $A := A - 50$
  3. **write**(*A*)
  4. **read**(*B*)
  5. $B := B + 50$
  6. **write**(*B*)

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure 4 properties:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** When a transaction is completed, the database must be in a consistent state; if any of the transaction parts violates an integrity constraint, the entire transaction is aborted.
- **Isolation.** Most of the database system allow concurrent execution of transactions.

  Isolation means that the data item used during the execution of a transaction cannot be used by a second transaction until the first one is completed. In other words, if a transaction T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2 ... T*n) until T1 ends.*
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

- **T1** **T2**

  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)

     read(A), read(B), print(A+B)

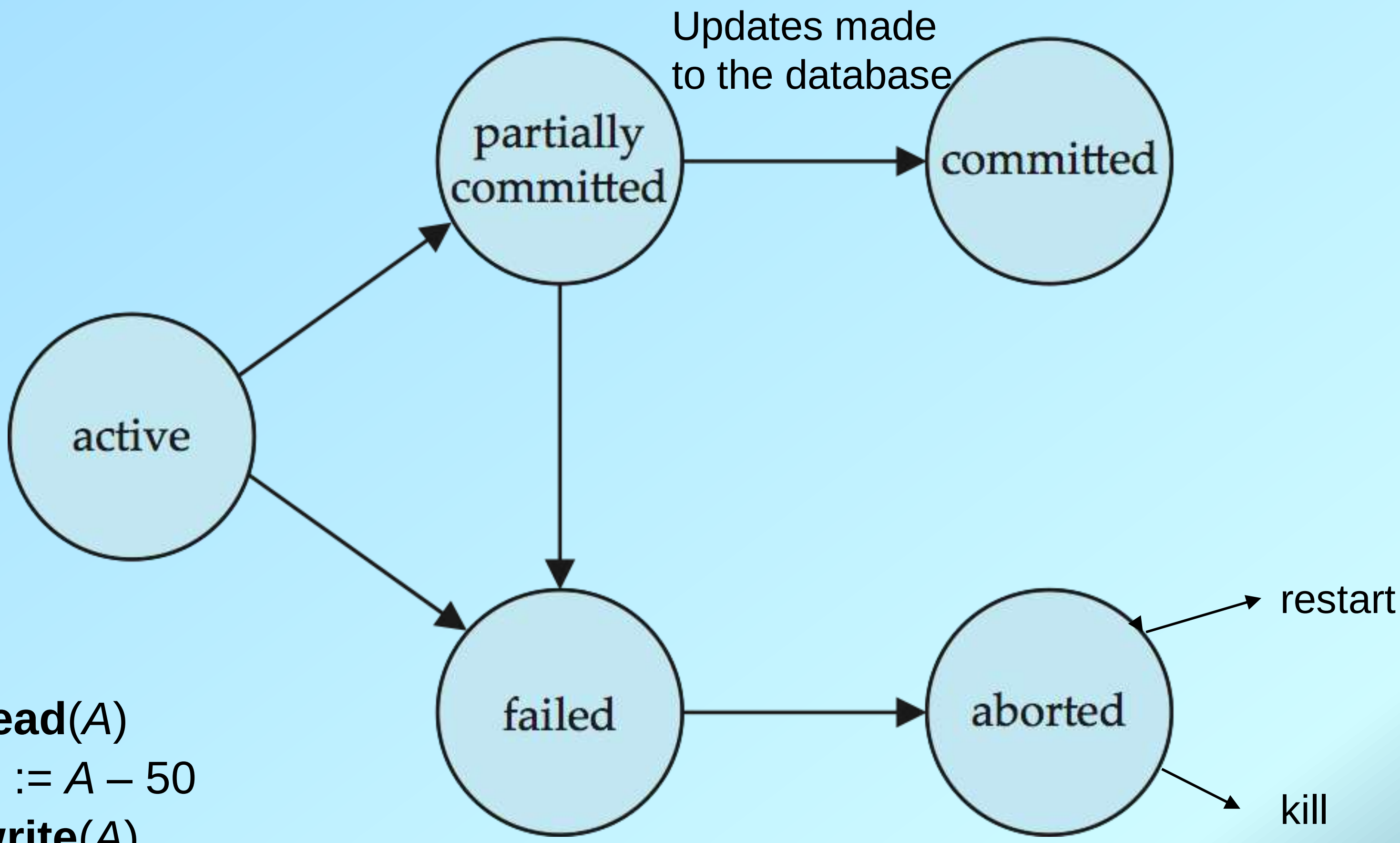
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Transaction State (Model)

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



T1:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Improved throughput and increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
    - All of this increases the **throughput** of the system—that is, increase the number of transactions executed in a given amount of time.
    - Correspondingly, the processor and disk **utilization** also increase
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Concurrent Executions

Transaction *T1 transfers $50 from account A to account B. It is defined as:*

T1: read(A);

A := A − 50;

write(*A);*

read(*B);*

B := B + 50;

write(*B).*

Transaction *T2 transfers 10 percent of the balance from account A to account B.*

T2: read(A);

*temp := A * 0.1;*

A := A − temp;

write(*A);*

read(*B);*

B := B + temp;

write(*B).*

Schedule 1

- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of transactions are executed in the system.
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- *Serial Schedules:* These schedules are **serial:** Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of *n* transactions, there exist *n!* different valid serial schedules.
- *Concurrent Schedule:* If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 3 -- A Concurrent Schedule equivalent to schedule 1

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of $(A + B)$.

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4 -- A Concurrent Schedule

Serializability

- **Basic Assumption** – Each transaction must preserve database consistency.
- The serial execution of a set of transactions always preserves database consistency.
- A concurrent schedule preserves database consistency if it is serializable.
- A concurrent schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
  - We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Let us consider a schedule S in which there are two consecutive instructions *I* and *J* of transactions T_i and T_j respectively. If *I and J* refer to different data items, then we can swap *I and J* without affecting the results of any instruction in the schedule. However, if *I and J* refer to the same data item *Q,* then the order of the two steps may matter.
- 1. *I* = **read**(*Q), J* = **read**(*Q*). *I and J* don't conflict.
  2. *I*= **read**(*Q), J*= **write**(*Q*). They conflict.
  3. *I* = **write**(*Q), J* = **read**(*Q*). They conflict
  4. *I* = **write**(*Q), J* = **write**(*Q*). They conflict
- *So, I and J* ***conflict*** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

Conflict Serializability (Cont.)

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

T_1	T_2
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Schedule 5 -- Schedule 3 After Swapping A Pair of Instructions

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 6 -- A Serial Schedule That is Equivalent to Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Conflict Serializability

- If a concurrent schedule *S* can be transformed into a serial schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a concurrent schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule.
- Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

- Let S and $S´$ be two schedules with the same set of transactions. S and $S´$ are **view equivalent** if the following three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must read the initial value of Q in schedule $S´$ also.
2. For each data item Q if transaction T_i executes **read**(Q) in schedule S, and that value was produced by a **write**(Q) operation of transaction T_j (if any), then in schedule $S´$ also the **read**(Q) *operation of* transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule $S´$.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

- Schedule 1 and schedule 2 are not view serializable

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

- Schedule 1 and schedule 2 are view serializable

View Serializability (Cont.)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Schedule 9 — a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

T_3	T_4	T_6
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

- Every conflict serializable schedule is also view serializable. But every view serializable schedule that is not conflict serializable

Testing for Conflict Serializability

- Consider a schedule S of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — We construct a directed graph, called a precedence graph, from S.
- 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 participating in the schedule.
- The set of edges consists of all edges Ti → Tj for which one of three conditions holds:

  1. Ti executes write(Q) before Tj executes read(Q).
  2. Ti executes read(Q) before Tj executes write(Q).
  3. Ti executes write(Q) before Tj executes write(Q).
- If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S' equivalent to S, Ti must appear before Tj .

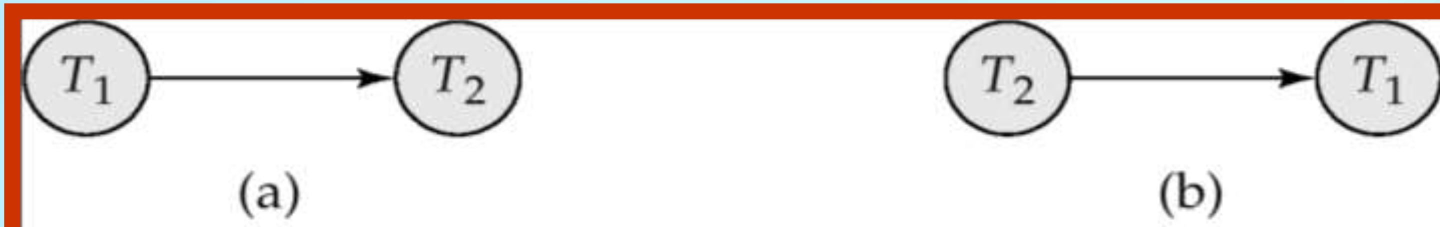
- **Example 1:** Precedence Graph for (a) Schedule 1 and (b) Schedule 2

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

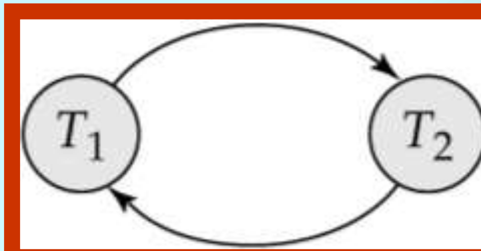


Precedence Graph for Schedule 3 and 4

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

(a)



Test for Conflict Serializability

- If the precedence graph for S has a cycle, then schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.
- Cycle-detection algorithms (based on DFS) exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph. For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.

Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
- No efficient algorithm is available to test for view serilizability.
- However, concurrency-control schemes can still use sufficient conditions for view serializability.
- That is, if the sufficient conditions are satisfied, the schedule is view serializable, but there may be view-serializable schedules that do not satisfy the sufficient conditions.

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC,   connection.setAutoCommit(false);

Concurrency Control

Concurrency Control

- Lock-Based Protocols
  - Locks
  - Granting of Locks
  - Two-Phase Locking Protocol
  - Graph-Based Protocols
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols

Concurrency Control

- Most of the database systems allowed concurrent execution of transaction for following reason:
  - Improve through put
  - Increase resource utilization
  - Minimize average waiting time
- When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.
- To ensure that it is, 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.

Lock-Based Protocols

- One way to ensure isolation 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.
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a ***lock*** on that item.
- A ***lock*** is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes:
  1. ***shared*** *(S) mode*: Data item can only be read. Shared lock is requested using **lock-S** instruction.
  2. ***exclusive*** *(X) mode*: Data item can be both read as well as written. Exclusive lock is requested using **lock-X** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols

- Every transaction **request** a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*.
- The transaction makes the request to the concurrency-control manager.
- The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.
- A transaction unlock a data item Q by the **unlock(Q)** instruction.
- The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	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 shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any 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. The lock is then granted.

Lock-Based Protocols (Cont.)

Transaction $T1$ transfers $50 from account B to account A

T_1: 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 $T2$ displays the total amount of money in accounts A and B

- T_2:**lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A+B*)

- 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 *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300.

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)		

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

Pitfalls of Lock-Based Protocols

Example of a transaction unlocking delay to the end of transaction:

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: **lock-S**(*A*);
read (*A*);
lock-S(*B*);
read (*B*);
display(*A+B*);
unlock(*A*);
unlock(*B*).

Pitfalls of Lock-Based Protocols

- Consider the 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.
  - ★ 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.

Pitfalls of Lock-Based Protocols (Cont.)

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting 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 same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation 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 incompatible with *M.*
2. There is no other transaction that is waiting for a lock on *Q, and that made its* lock request before *Ti.*

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain 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.

Example of two phase-locking protocol

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: **lock-S**(*A*);
read (*A*);
lock-S(*B*);
read (*B*);
display(*A+B);*
unlock(*A*);
unlock(*B).*

The protocol assures conflict serializability. 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 (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.

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)	

The Two-Phase Locking Protocol (Cont.)

- Cascading roll-back 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)

The Two-Phase Locking Protocol (Cont.)

- To avoid this, follow a modified protocol called
- **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Lock Conversions

- Consider the following two transactions, for which we have shown only some of the significant read and write operations:

  T8: read(a1);
  read(*a2);*
  . . .
  read(*an);*
  write(*a1).*

  T9: read(a1);
  read(*a2);*
  display(*a1 + a2).*

- We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock.
- We denote **lock conversion** from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**.

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase (growing):
    - 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 (shrinking):
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Incomplete Schedule With a Lock Conversion

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

if T_i has a lock on D

then

read(D)

else begin

if necessary wait until no other transaction has a **lock-X** on D

grant T_i a **lock-S** on D;

read(D)

end

Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

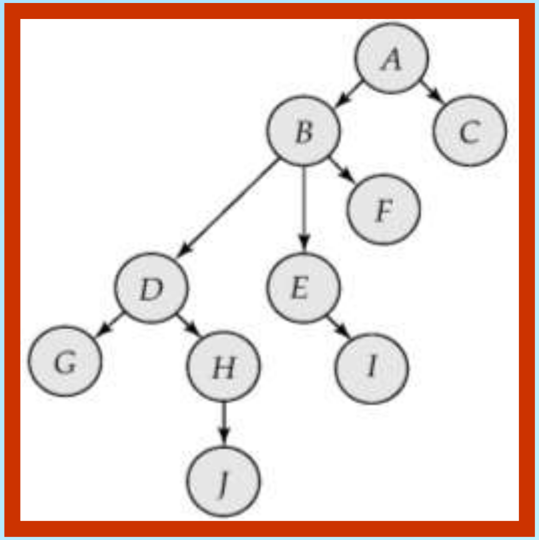
  if T_i has a **lock-X** on *D*
  then
  write(*D*)
  else begin
  if necessary wait until no other transaction has any lock on *D*,
  if T_i has a **lock-S** on *D*
  then
  upgrade lock on *D* to **lock-X**
  else
  grant T_i a **lock-X** on *D*
  write(*D*)
  end;

- All locks are released after commit or abort

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- It impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - This partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

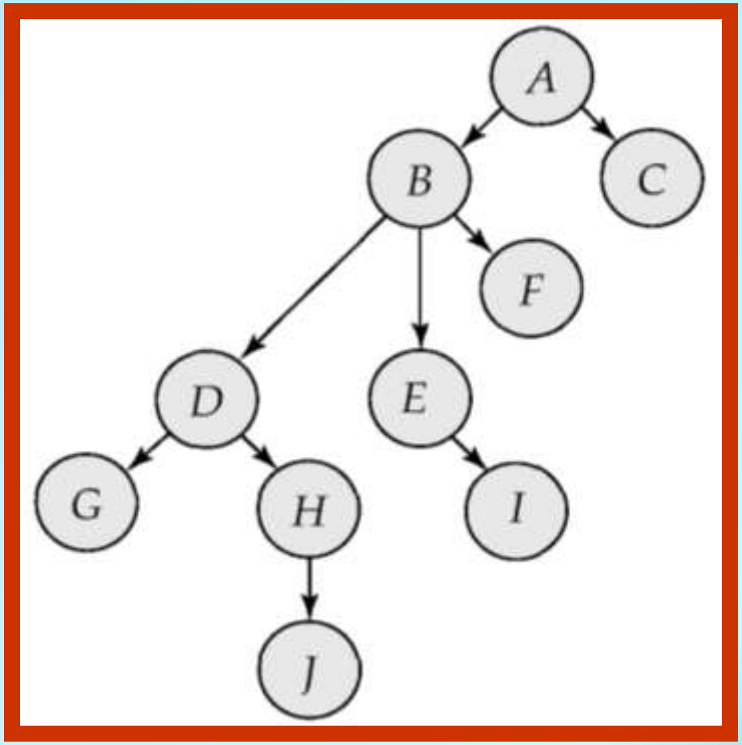
Tree Protocol



- In the **tree protocol,** the only lock instruction allowed is lock-X. Each transaction *Ti* can lock a data item at most once, and must observe the following rules*:*
- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item.
- Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

Example of Tree protocol

- *T10: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G);* unlock(*D*)*; unlock(G).*
- *T11: lock-X(D); lock-X(H); unlock(D); unlock(H).*
- *T12: lock-X(B); lock-X(E); unlock(E); unlock(B).*
- *T13: lock-X(D); lock-X(H); unlock(D); unlock(H).*



Serializable Schedule Under the Tree Protocol

T_{10}	T_{11}	T_{12}	T_{13}
lock-x (B)			
	lock-x (D)		
	lock-x (H)		
	unlock (D)		
lock-x (E)			
lock-x (D)			
unlock (B)			
unlock (E)			
		lock-x (B)	
		lock-x (E)	
	unlock (H)		
lock-x (G)			
unlock (D)			
			lock-x (D)
			lock-x (H)
			unlock (D)
			unlock (H)
		unlock (E)	
		unlock (B)	
unlock (G)			

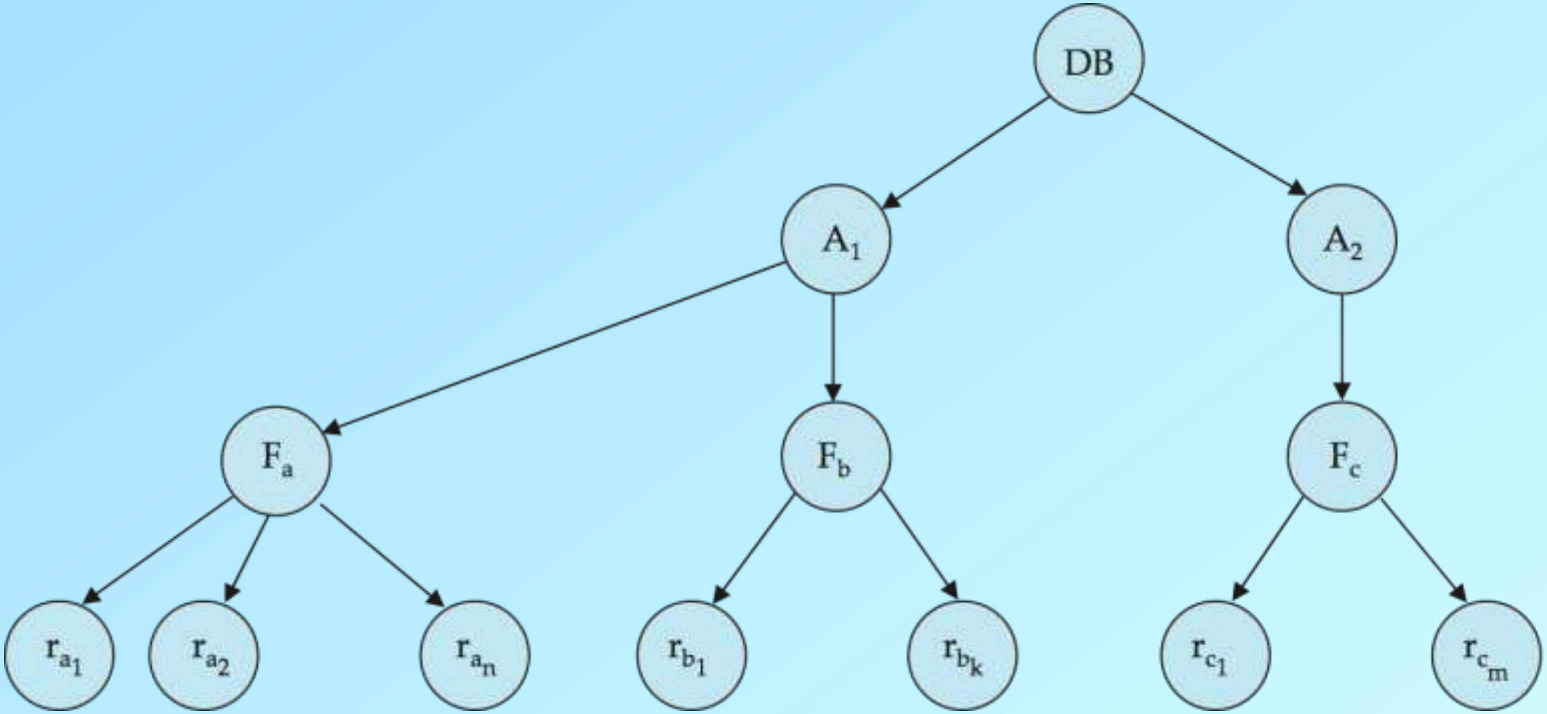
Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- **Granularity of locking** (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are:

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- Each node in the tree can be locked individually.
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

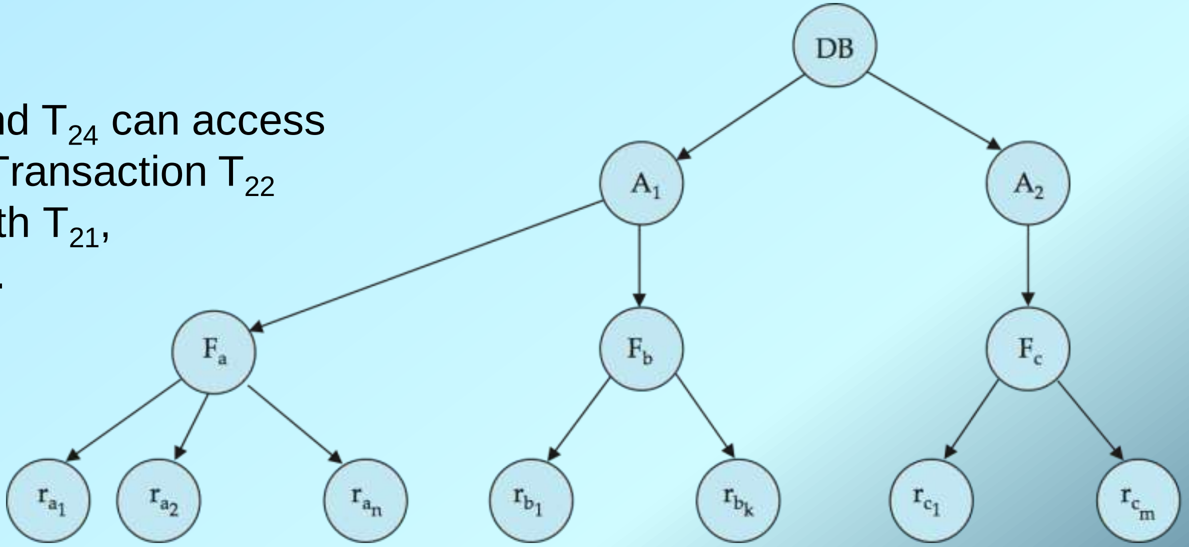
Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order (top-down), whereas they are released in leaf-to-root order (bottom-up).

Examples

- Suppose that transaction T_{21} reads record r_{a2} in file F_a . Then, T_{21} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a2} in S mode.
- Suppose that transaction T_{22} modifies record r_{a9} in file F_a . Then, T_{22} needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and finally to lock r_{a9} in X mode.
- Suppose that transaction T_{23} reads all the records in file F_a . Then, T_{23} needs to lock the database and area A_1 (and in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{24} reads the entire database. It can do so after locking the database in S mode.

The transactions T_{21}, T_{23}, and T_{24} can access the database concurrently. Transaction T_{22} can execute concurrently with T_{21}, but not with either T_{23} or T_{24}.



Timestamp-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.
- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- 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.

  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.

Timestamp-Based Protocols

- 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* .
- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q):
  1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back. **(Read operation comes before write).**
  2. If TS(T_i) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)). **(Read operation comes after write).**

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, and the system assumed that that value would never be produced.
     - 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 *Ti* is rolled back by the concurrency-control scheme, the system assigns it a new timestamp and restarts it.

Example

- Transaction T25 displays the contents of accounts A and B and
- Transaction T26 transfers $50 from account B to account A, and then displays the contents of both:

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($A + B$).

Schedule 3

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

TS(*T25)*< *TS(T26),* and the schedule is possible under the timestamp protocol.

Thomas' Write Rule

T_{27}	T_{28}
read (Q)	
	write (Q)
write (Q)	

- Assume that $TS(T_{27}) < TS(T_{28})$.
- The read(Q) operation of T_{27} succeeds, as does the write(Q) operation of T_{28}.
- R-timestamp = $TS(T_{27})$ and W-timestamp = $TS(T_{28})$.

When T_{27} attempts its write(Q) operation, we find that $TS(T_{27})$ < W-timestamp(Q), thus, the write(Q) by T_{27} is rejected (obsolete **write)** and transaction T_{27} must be rolled back.

- Modified version of the time stamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.

Thomas’ Write Rule

- The modification to the timestamp-ordering protocol, called **Thomas’ write rule,** is this: 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, and it had been assumed that the value would never be produced. 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).*

- This protocol is the same as the timestamp ordering protocol for read operation.
- Thomas' Write Rule allows greater potential concurrency.
- Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.

  1. Read and execution phase: During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database

  2. Validation phase: Transaction T_i performs a ``validation test" to determine if local variables can be copy to the database without violating serializability. If a transaction fails the validation test, the system aborts the transaction.

  3. Write phase: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but **each transaction must go through the three phases in that order**.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps:
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i):): the time when *Ti* finished its read phase and started its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j and T_i completes its write phase before T_j starts its validation phase, then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

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)

End of Chapter