Chapter 15 : Concurrency Control

Database System Concepts, 6th Ed.

Chapter 15: Concurrency Control

- Lock-Based Protocols
  - 2PL
  - Graph-Based Protocols
  - Deadlock Prevention/Detection/Recovery
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Lock requests are made to concurrency-control manager.
- Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Data items can be locked in two modes:
  1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- **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.

Lock-Based Protocols (Cont.)

- Data items can be locked in two modes:
  1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

read(A)

read(A)

- 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.

Lock-Based Protocols (Cont.)

- Data items can be locked in two modes:
  1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

lock-S(A)
read(A)

lock-S(A)
read(A)

- 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.

Lock-Based Protocols (Cont.)

- Data items can be locked in two modes:
  1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

lock-S(A)
read(A)
write(A)
lock-S(A)
read(A)
read(A)

- 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.

Lock-Based Protocols (Cont.)

- Data items can be locked in two modes:

  1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

lock-S(A)
read(A)

lock-X(A)
write(A)

lock-S(A)
read(A)

lock-S(A)
read(A)

- 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.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

T_2: lock-S*(A)*;

read *(A)*;

unlock*(A)*;

lock-S*(B)*;

read *(B)*;

unlock*(B)*;

display*(A+B)*

- Is the above safe?

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: lock-S*(A)*;

  read *(A)*;

  unlock*(A)*;

  lock-S*(B)*;

  read *(B)*;

  unlock*(B)*;

  display*(A+B)*

- Is the above safe?
- Locking as above is not sufficient to guarantee serializability — if *A* or *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- How then can we fix it? Use one of **locking protocols** (e.g., 2PL) that ensure serializability.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule. Is it Okay?

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)	

Pitfalls of Lock-Based Protocols

- Consider the partial schedule. Is it Okay?

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*)	

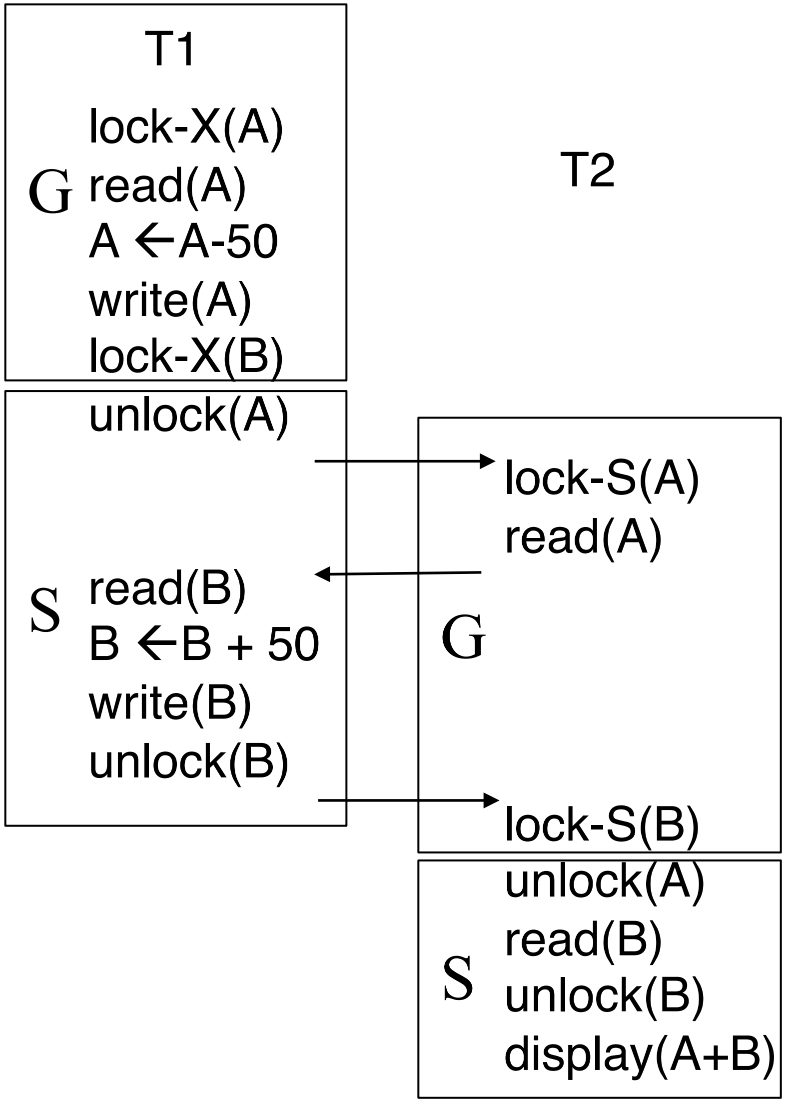
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must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

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**
- The protocol assures 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).
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.



The Two-Phase Locking Protocol (Cont.)

- risk of deadlocks.
- may not be recoverable
- Cascading roll-back is possible. 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.

T1	T2	T3
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)
<xaction fails>		

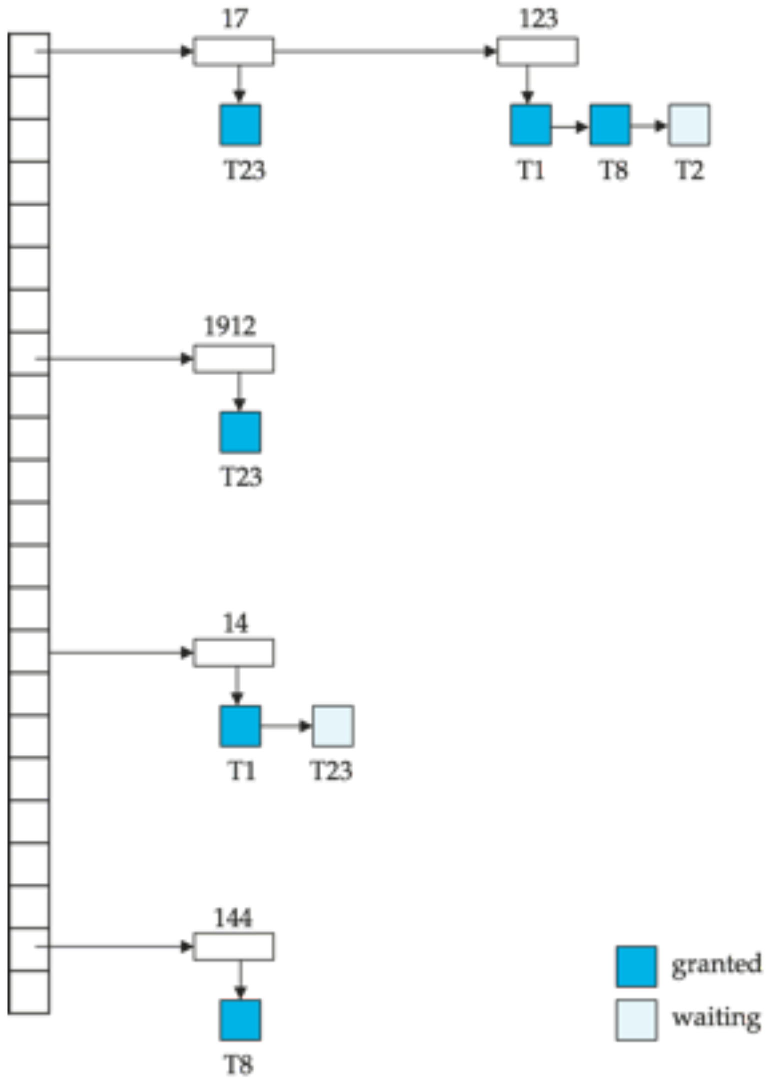
Strict 2PL will not allow that

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests.
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests.

Lock Table

- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.



- Blue rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted

Deadlock

- Consider the following two transactions:

  T_1: write (A) $\quad$ T_2: write(B)

  write(B) $\quad$ write(A)

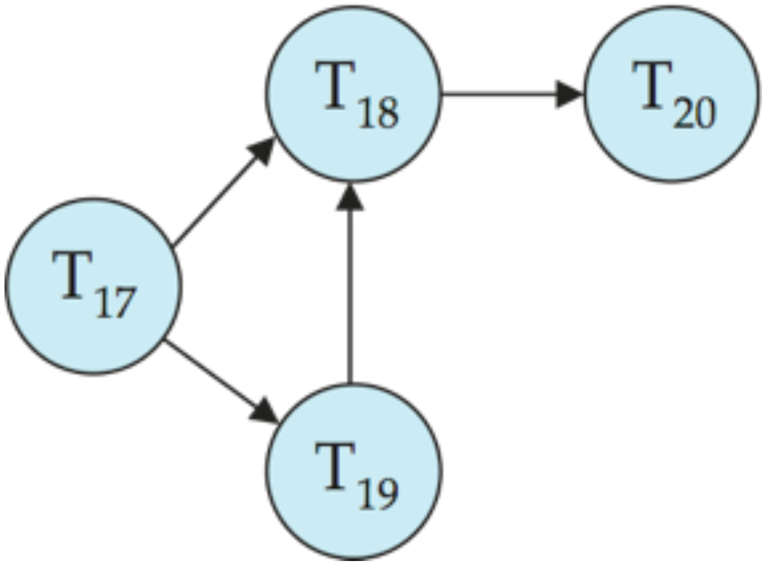
- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

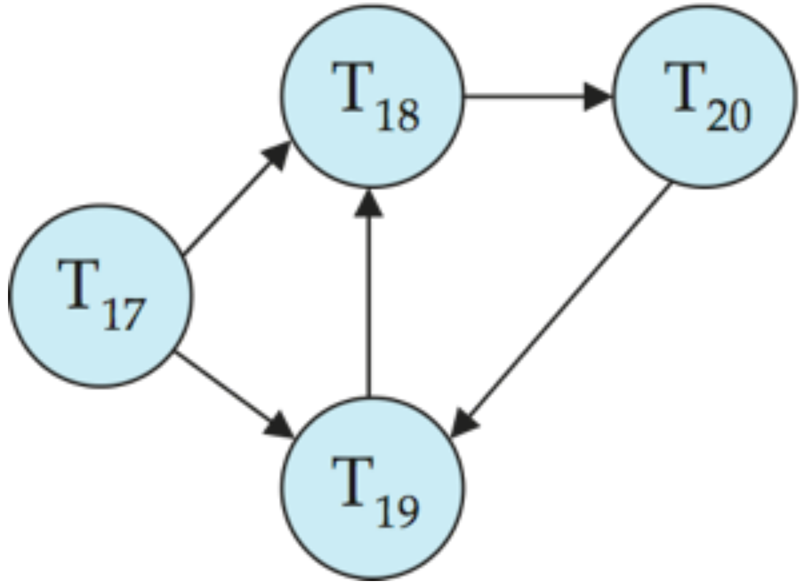
Deadlock Detection

- Deadlocks can be described as a ***wait-for*** *graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Prevention

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- Following schemes use **transaction timestamps** for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item.
  - younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item (**starvation**)
- **wound-wait** scheme — preemptive
  - younger transactions may wait for older ones.
  - older transaction *wounds* (forces rollback of) younger transaction instead of waiting for it.
  - may be fewer rollbacks than *wait-die* scheme

Deadlock prevention (Cont.)

- **Timeout-Based Schemes**:
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

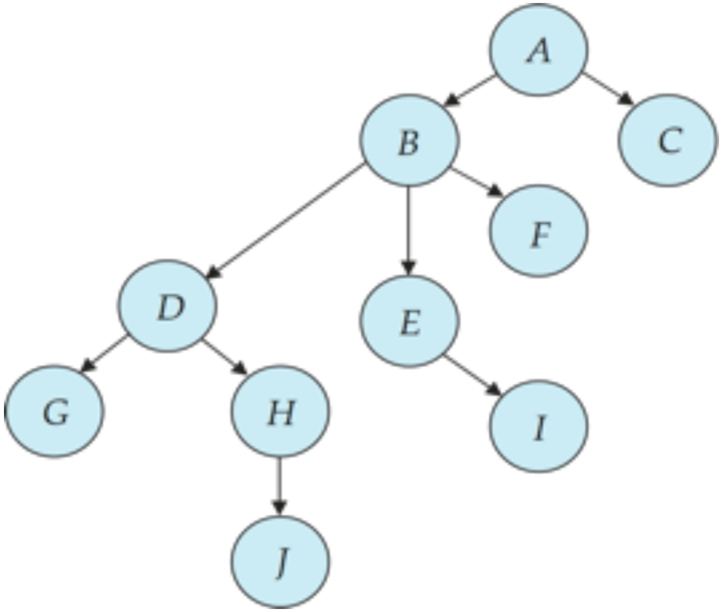
Deadlock Recovery

- When deadlock is detected:
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- 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.**
  - 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



1. Only exclusive locks are allowed.
2. 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.
3. Data items may be unlocked at any time after the relevant children are locked.

- Example: T_1 and T_2 both on A and D, T_1 goes first

T_1	T_2
lock-X(A)	
lock-X(B)	
unlock(A)	
	lock-X(A)
lock-X(D)	
unlock(B)	
	lock-X(B)
	unlock(A)
unlock(D)	
	lock-X(D)
	unlock(B)
	unlock(D)

Graph-Based Protocols (Cont.)

- ensures conflict serializability
- **free from deadlock** (no rollbacks).
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
- Drawbacks
  - 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.

Chapter 15: Concurrency Control

- Lock-Based Protocols
  - 2PL
  - Graph-Based Protocols
  - Deadlock Prevention/Detection/Recovery
- **Timestamp-Based Protocols**
- Validation-Based Protocols
- Multiversion Schemes

Timestamp-Based Protocols

- 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).
- The protocol manages concurrent execution such that *the time-stamps determine the* ***serializability order***.
- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest timestamp 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) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).
  2. If TS(T_i) $<$ **W**-timestamp(Q), then the **read** operation is rejected, and T_i is rolled back **(late read)**.

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) $<$ R-timestamp(Q), then the **write** operation is rejected, and T_i is rolled back **(late write)**.
  2. If TS(T_i) $<$ W-timestamp(Q), then this **write** operation is rejected, and T_i is rolled back **(late write)**.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

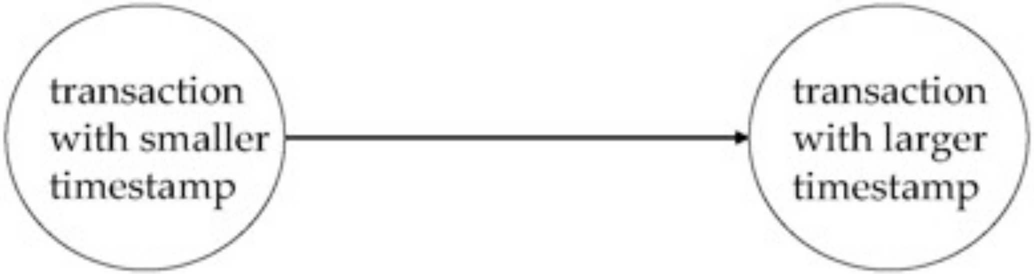
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph.

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which **obsolete write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be safely ignored.

T1	T2
R(Q)	
	W(Q)
W(Q)	

Chapter 15: Concurrency Control

- Lock-Based Protocols
  - 2PL
  - Graph-Based Protocols
  - Deadlock Prevention/Detection/Recovery
- Timestamp-Based Protocols
- **Validation-Based Protocols**
- Multiversion Schemes

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a ``validation test'' to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Chapter 15: Concurrency Control

- Lock-Based Protocols
  - 2PL
  - Graph-Based Protocols
  - Deadlock Prevention/Detection/Recovery
- Timestamp-Based Protocols
- Validation-Based Protocols
- **Multiversion Schemes**

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

w(A)	
	w(A)
r(A)	

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

Chapter 15: Concurrency Control

- Lock-Based Protocols
  - 2PL
  - Graph-Based Protocols
  - Deadlock Prevention/Detection/Recovery
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- If you are really interested in concurrency control, consider reading this free book:
  - http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx