



The picture can't be displayed.

Concurrency Control

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Lock-Based Protocols

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



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.)

- 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)
```

- 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.
- 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 protocol 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).



The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
 - Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.



Lock Conversions

- Two-phase locking with lock conversions:
 - 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 assures serializability. But still relies on the programmer to insert the various locking instructions.



Deadlocks

- 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.



Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks, there is a possibility of **starvation**.
- **Starvation** occurs if the 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.



Deadlocks (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- 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.



Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **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.



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. (older means smaller timestamp) Younger transactions never wait for older ones; they are rolled back instead.
 - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
 - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
 - may be fewer rollbacks than *wait-die* scheme.



Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **Timeout-Based Schemes:**
 - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
 - Thus, deadlocks are not possible
 - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

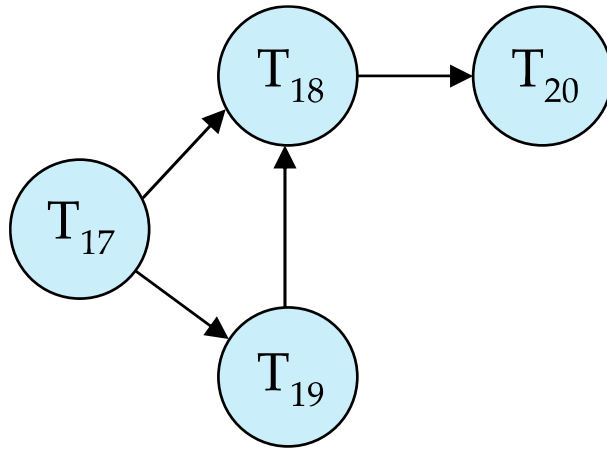


Deadlock Detection

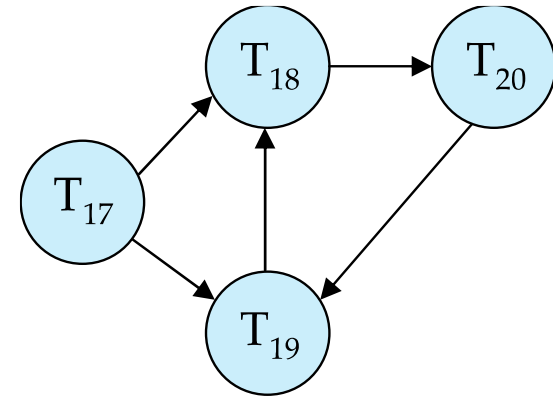
- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
 - V is a set of vertices (all the transactions in the system)
 - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E , then there is a directed edge from T_i to T_j , implying that T_i is waiting for T_j to release a data item.
- 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. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle



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



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 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 $TS(T_i) \leq \mathbf{W}$ -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.
 2. If $TS(T_i) \geq \mathbf{W}$ -timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), $TS(T_i)$).



Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
 1. If $TS(T_i) < R\text{-timestamp}(Q)$, 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 $TS(T_i) < W\text{-timestamp}(Q)$, 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 $W\text{-timestamp}(Q)$ 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) abort			read (Z)
read (X)		write (W) abort	read (W)	
				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.



Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
 - Suppose T_i aborts, but T_j has read a data item written by T_i
 - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
 - Further, any transaction that has read a data item written by T_j must abort
 - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
 - A transaction is structured such that its writes are all performed at the end of its processing
 - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
 - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability



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.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
 - Assume for simplicity that the validation and write phase occur together, atomically and serially
 - ▶ I.e., only one transaction executes validation/write at a time.
- 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
 - $\text{Start}(T_i)$: the time when T_i started its execution
 - $\text{Validation}(T_i)$: the time when T_i entered its validation phase
 - $\text{Finish}(T_i)$: the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time; this is done to increase concurrency.
 - Thus, $\text{TS}(T_i)$ is given the value of $\text{Validation}(T_i)$.
- 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.



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) $\langle \text{validate} \rangle$ display ($A + B$)	$\langle \text{validate} \rangle$ write (B) write (A)



Deadlocks

- Consider the following two transactions:

T_1 : write (X)
 write (Y)

T_2 : write(Y)
 write(X)

- 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	