Chapter 18 : Concurrency Control

Database System Concepts, 7th Ed.

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

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 concurrency-control manager. 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.

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*

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
-

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_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Deadlock

- 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)	

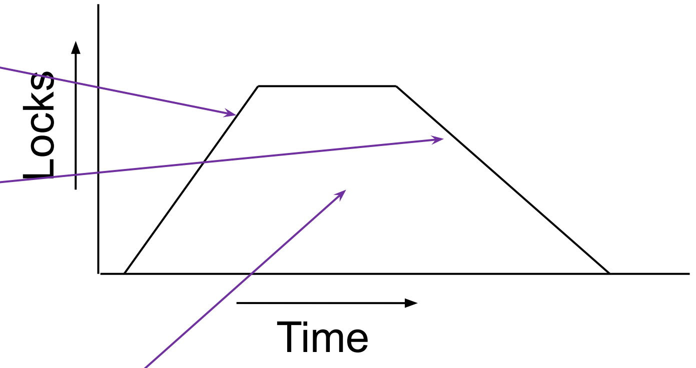
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must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

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **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

- 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).



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary 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.*

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A + 50$	
write(A)	
unlock(A)	

Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing 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**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

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.

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)	

Deadlock Handling

- ***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 (pre-declaration).
  - 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

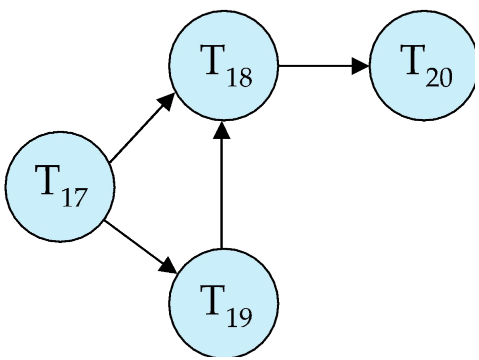
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Deadlock prevention (Cont.)

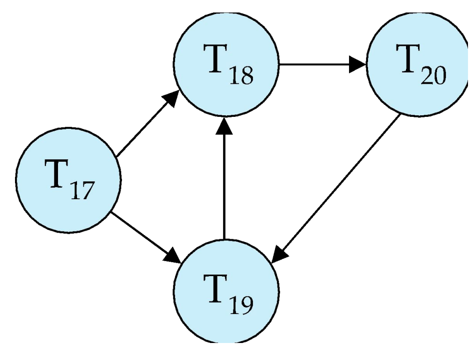
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



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 cycle.
    - 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.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

End of Chapter 18