Concurrency Control Mechanisms

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Concurrency Control

- 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.

©Silberschatz, Korth and Sudarshan

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_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).

T_2:

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

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking – Cont.

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)		

- 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.

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.

Pitfalls of Lock-Based Protocols (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

- 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).

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. 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(a n);

write(a1)

T9 : read(a1);

read(a2);

display(a1 + a2)

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.

©Silberschatz, Korth and Sudarshan

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 trans. 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

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

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***
  - ***Deadlock detection***
  - ***Deadlock recovery***

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).
  - Deadlock prevention by ordering usually ensured by careful programming of transactions

©Silberschatz, Korth and Sudarshan

Deadlock Prevention Schemes

- Two different deadlock-prevention schemes using timestamps have been proposed:
  1. The **wait–die** scheme is a non-preemptive technique.

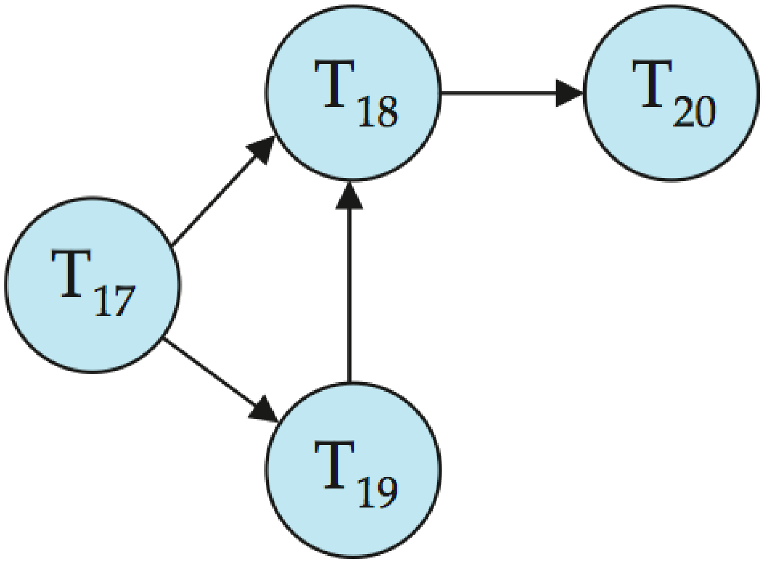
     For example, suppose that transactions T14 , T15 , and T16 have timestamps 5, 10, and 15, respectively. If T14 requests a data item held by T15 , then T14 will wait. If T24 requests a data item held by T15 , then T16 will be rolled back.
  2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme.

Deadlock Prevention Schemes

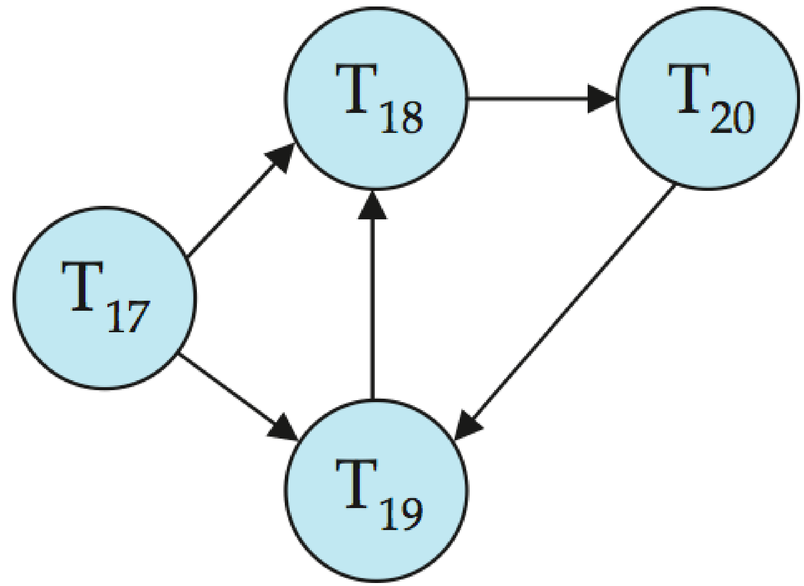
- Another simple approach to deadlock prevention is based on **lock timeouts**.
- In this approach, a transaction that has requested a lock waits for at most a specified amount of time.
- If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.

Deadlock Detection

- ***Deadlock detection*** algorithms used to detect deadlocks



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

 ©Silberschatz, Korth and Sudarshan

Locking Extensions

- **Multiple granularity locking:**
  - idea: instead of getting separate locks on each record
    - lock an entire page explicitly, implicitly locking all records in the page, or
    - lock an entire relation, implicitly locking all records in the relation

©Silberschatz, Korth and Sudarshan

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 $TS(T_i) \leq$ **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 $TS(T_i) \geq$ **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-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).

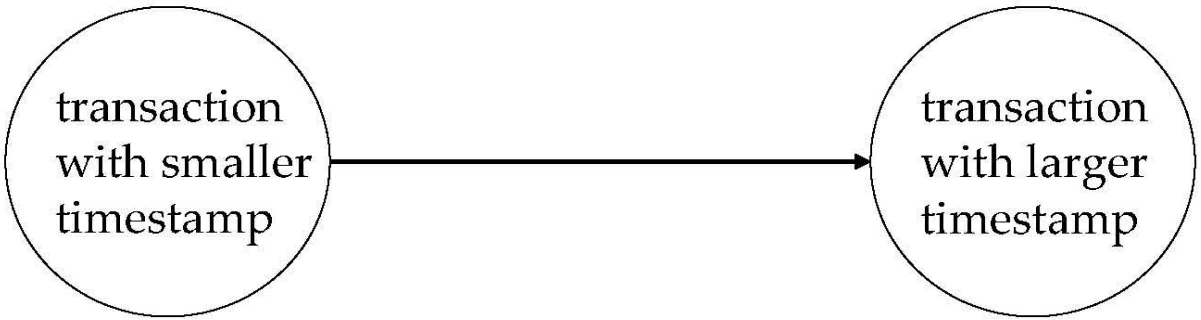
Example Use of the Protocol

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.

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

 ©Silberschatz, Korth and Sudarshan

Validation-Based Protocols

- 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 Protocols (Cont.)

- To perform the validation test, we need to know when the various phases of transactions took place.
- Therefore, associate three different timestamps 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.

THANK YOU

eBOX

AN AMPHISOFT PRODUCT