Concurrency Control

Instructor : Nitesh Kumar Jha

ITER,S'O'A(DEEMED TO BE UNIVERSITY)

Oct 2018

Concurrency Control

- It Is a mechanism to ensure isolation in a concurrent execution scenario
- Achieved using the concept of mutual exclusion
  - i.e. while one transaction is accessing a data item, no other transaction is allowed to modify that data item.
- Mutual exclusion is achieved using logical locks.
  - Locks are granted/revoked by a central concurrency control manager.
  - i.e. Transactions request it to grant a lock

Locks

- Data items can be locked in two modes :
  - *Shared:* Data item can only be read. Requested using **lock-S** instruction. It can be shared with other transactions
  - *Exclusive*: Data item can be both read as well as written. It is requested using **lock-X** instruction. It can't be shared with other transactions
- Lock-compatibility matrix
  - Shared locks may be granted to multiple transactions simultaneously.
  - Exclusive locks can't be granted to multiple transactions simultaneously

	S	X
S	true	false
X	false	false

Locks

- Let A and B are two accounts that are accessed by transactions T_1 and T_2 . Transaction T_1 transfers $50 from account B to account A. Transaction T_2 displays the total amount of money in accounts A and B i.e., sum A+B.
- If these transaction are executed serially , T_1 T_2 or T_1 T_2 , It will be consistent.
- What happens if these are executed concurrently???

T_1	T_2
Lock-X(B)	Lock-S(A)
Read(B)	Read(A)
B=B-50	Unlock(A)
Write(B)	Lock-S(B)
Unlock(B)	Read(B)
Lock-X(A)	Unlock(B)
Read(A)	Display (A+B)
A=A+50	
Write(A)	
Unlock(A)	

Locking Example

T_1	T_2	CC Manager
Lock-X(B)	A=500	
	B=1000	Grant-X(B, T_1)
Read(B)		
	1000	
B=B-50		
	950	
Write(B)		
Unlock(B)		
	Lock-S(B)	
		Grant-S(B,T_2)
	Read(B)	
	Unlock(B)	
	Lock-S(A)	

T_1	T_2	CC Manager
		Grant-S(A, T2)
	Read(A)	
	Unlock(A)	
	Display (A+B)	
		1450
Lock-X(A)		
		Grant-X(A,T1)
Read(A)		
A=A+50		
Write(A)		
Unlock(B)		

- Produces inconsistent result
- Transactions must hold the lock on a data item till it access that item.
- It is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that item, since the serializability may not be ensured.

Delayed Unlocking

- Unlocking is delayed to the end of the transactions
- This schedule produces consistent result

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).

- But this technique may lead to an undesirable scenario called deadlock

Deadlock due to Hold-and-Wait

T_5	T_6	CC Manager
Lock-X(B)		
		Grant-X(B, T_1)
Read(B)		
B=B-50		
Write(B)		
	Lock-S(A)	
		Grant-S(A,T_2)
	Read(A)	
	Lock-S(B)	
Lock-X(A)		

Deadlock

- When a transaction (T_5) delays unlocking on its locked data items (B) and requests to acquire a lock on new data items (A) that is already locked by another transaction (T_6)
- This is called a Hold-and-Wait situation

Deadlock due to Hold-and-Wait

T_5	T_6	CC Manager
Lock-X(B)		
		Grant-X(B, T_1)
Read(B)		
B=B-50		
Write(B)		
	Lock-S(A)	
		Grant-S(A,T_2)
	Read(A)	
	Lock-S(B)	
Lock-X(A)		

Deadlock

- Deadlock is a state where neither of these transactions can ever proceed with its normal execution. It can be resolved by forcibly rolling back one or more participating transactions.
- Lock based concurrency control needs the transaction to follow a set of rules called locking protocol
- 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.
- It requires the transactions execute in two phases
- 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
- This protocol assures serializability.

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.

Automatic Acquisition of Locks

Not IMP

- 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.)

Not IMP

- **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

The Two-Phase Locking Protocol

T_1

Lock-X(B)
Read(B)
B=B-50
Write(B)
Lock-X(A)

Growing Phase

Lock Point

Unlock(B)
Read(A)
A=A+50
Write(A)
Unlock(A

Shrinking Phase

- The point in the schedule where transaction has obtained its final lock(end of its growing phase) is called **lock point** of the transaction.
- Now transactions can be ordered according to their lock points.

Two Phase Locking Example

T_1	T_2	CC Manager
Lock-X(B)		
		Grant-X(B, T_1)
Read(B)	**Not proceed further So T_1 Will proceed first then T_2**	
B=B-50		
Write(B)		
	Lock-S(B)	
		Grant-S(B,T_2)
	Read(B)	
	Lock-S(A)	Grant-S(A,T_2)

T_1	T_2	CC Manager
	Read(A)	
	Display (A+B)	1450
	Unlock(B)	
	Unlock(A)	
Lock-X(A)		
		Grant-X(A,T1)
Read(A)		
A=A+50		
Write(A)		
Unlock(B)		
Unlock(A)		

- Produces consistent result as transaction will be serial T_1 -> T_2
- Transactions holds the lock on a data item till the last.

Properties of 2PL

- 2PL ensures conflict serializability. The contributing transactions are isolated w.r.t. the lock point. (point at which growing phase ends and shrinking phase starts)
- It does not ensure deadlock free execution. In the event of deadlock participating transactions are rolled back. Consider

  Previous Example here ----->

T_5	T_6	CC Manager
Lock-X(B)		
		Grant-X(B, T_1)
Read(B)		
B=B-50		
Write(B)		
	Lock-S(A)	
		Grant-S(A,T_2)
	Read(A)	
	Lock-S(B)	
Lock-X(A)		
Deadlock		

- It ensures recoverability but does not safeguard against cascading rollback.

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.)

- 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, we must follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.

Strict two Phase locking Protocol

- Strict two phase locking is an enhanced 2PL that ensures cascadeless recovery.
- Strict 2PL demands that not only the locking and unlocking be in two phases, all the exclusive mode locks must be hold by the transaction till the transaction commits.
- This requirement ensured that any data written by uncommitted transactions are locked in Exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- **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.

Deadlocks (Cont.)

- Strict 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.

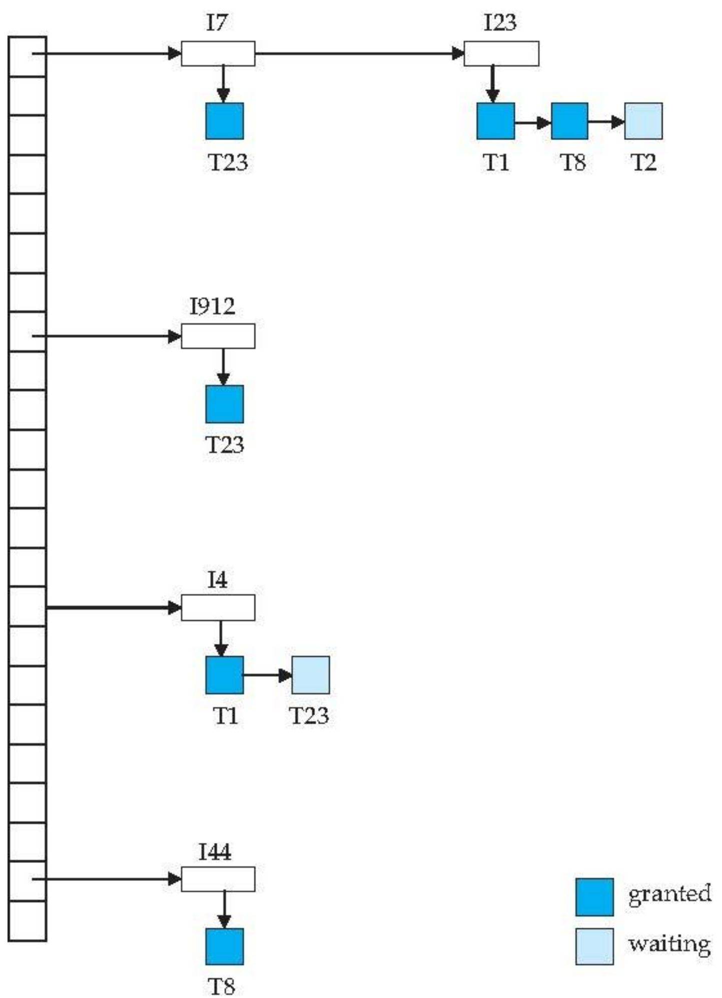
Implementation of Locking

Not IMP

- 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

Lock Table

Not IMP



- Dark blue rectangles indicate granted locks; light blue 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
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Deadlock Handling

Not IMP

- 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. **Not IMP**
- **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
  - If a younger transaction holds a data item requested by older transaction, the younger transaction is aborted and rolled back. Younger transactions may wait for older ones to release a data item.
  - may be fewer rollbacks than *wait-die* scheme.

Deadlock prevention (Cont.)

Not IMP

- 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.

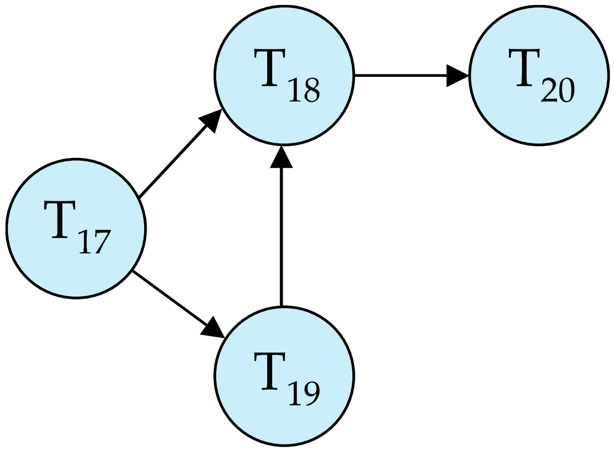
Deadlock Detection

Not IMP

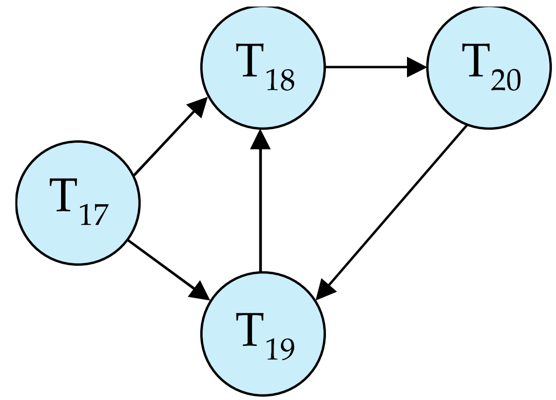
- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from 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.)

Not IMP



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

Not IMP

- 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.)

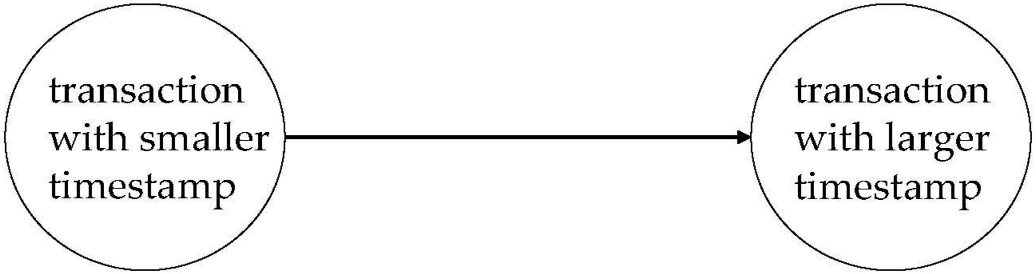
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) $\leq$ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten by a younger transaction.
     - 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).

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

Thomas' Write Rule

Not IMP

- 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 ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

Not IMP

- 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

Schedule Produced by Validation

Not IMP

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Multiversion Schemes

Not IMP

- 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.

Multiversion Timestamp Ordering

Not IMP

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont)

Not IMP

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. else a new version of Q is created.
- Observe that
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

MVCC: Implementation Issues

Not IMP

- 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

End of Lecture
Thank You