Concurrency Control

Chapter - 16

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.
  - 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.

- To access a data item, transaction Ti must first lock that item. If the data item is
- already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.

T1: lock-X(B);
read(B);
B := B − 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A)

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*)

- Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A
- Transaction T2 displays the total amount of money in accounts A and B
- Schedule -1

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_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

200

B=150

200

A=100

A=150

A=100

B=150

A+B=250

- 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.
- If, however, these transactions are executed concurrently, given in schedule 1, in
- In this case, transaction T2 displays $250, which is incorrect. It should be 300.
- The reason for this mistake is that the
- transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state

– Suppose now that unlocking is delayed to the end of the transaction in T1 transcation. Then t2 is not displaying incosistant result.
– In figure t1 with modification consider as a T3 and T2 is consider as T4 in following

T3	T4
Lock-x(B) read(*B*) *B* := *B* -50 write(*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) Unlock(B)	

can not be granted as exclusive lock on B by transaction t4.
T4 is in waiting mode

After T3 unlock (B) then T4 can Resume and excute next instruction.
So A+B in consistant state

Pitfalls of Lock-Based Protocols

- Sometime locking can lead to an undesirable situation.
- 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)	

- since T3 is holding an exclusive-mode lock on B
- and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B.
- Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock 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.
-

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)	

- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked and other transaction can use that

Pitfalls of Lock-Based Protocols (Cont.)

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - 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 and
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks but
  - transaction may not obtain locks
- Once transaction release a lock it enters the shrinking phase, i.e it does not acquire any lock
- The protocol assures serializability.

Transaction T3 in two-phase and T4 in Two phase

T3 : lock-x(b)
read(b)
B:=b-50
Write(b)
Lock-x(a)
Read(a)
A:=a+50
Wirite(a)
Unlock(b)
Unloack(a)

T4 : lock-s(a)
read(a)
Lock-s(b)
Read(b)
Display a+b
Unlock(a)
Unloack(b)

alternate T3 : lock-x(b)
read(b)
B:=b-50
Write(b)
Lock-x(a)
Unlock(b)
Read(a)
A:=a+50
Wirite(a)
Unloack(a)

alternate T3 is also in two-phase rule

The Two-Phase Locking Protocol

- Two-phase locking *does not* ensure freedom from deadlocks. T3 and t4 are in deadlock situation.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called .
- Transaction T5 is fail after the read(A) of T7 transaction then cascading rollback of T6 and T7.
- **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.

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)	

T_5	T_6	T_7
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)

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).
- Two method for implemeting scheme.
  1. Use the system clock as the timestamp.
  2. Use a logical counter that is increment after new timestamp has been assigned.
     - Transaction's timestamp is value of counter .
- 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) ≤ **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)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
- 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.
- 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.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

This protocol free form deadlock
Timestamp of T14 is less than T15. RTS(A) & RTS(B) =0 WTS(A) & WTS(b) =0

10 20

T14	T15	
Read(B)		Rts(B)=10
	Read(B) B: =B-50; Write (B)	Rts(B)=20 20<20 no(1), 20<0 no(2), execute and WTS(B)=20
Read(A)		RTS(A)=10 WTS(A)=0 , 20>0((2) execute RTS{10,20} RTS(A) =20
	Read(A)	
Display (A+B)		
	A=A+50 Write(A) Display (A+B)	RTS(A)=20 20<20 no(1), 20<0 no(2), 20>0 yes Execute Write(A) WTS(A)=max{ 0,20}

#

– If T_i rolled back then the system assigns it a new timestamp.

Deadlock Handling

– Consider the following two transactions:

T_1: write (X) T_2: write(Y)
write(Y) write(X)

– Schedule with deadlock

T_1	T_2
lock-X on X	
write (X)	
	lock-X on Y
	write (y)
	wait for **lock-X** on X
wait for **lock-X** on Y	

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. That is transaction (t1,t2,…tn) where t1 is waiting for data item that t2 hold and t2 waiting for data item that t3 hold and so on…
- Two main method for dealing with deadlock is deadlock prevention and deadlock detection and recovery.

Deadlock prevention

– ***Deadlock prevention is a*** protocols to ensure that the system will *never* enter into a deadlock state.

– Two approaches to deadlock prevention is

1. ensure that no cyclic wait
2. Requiring all lock acquired together.
   - Its hard to predict all its data items before it begins execution.

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
  - When Ti request data item held by Tj , Ti is allowed to wait only if Ti is older than Tj . …. o.w Ti is roll bak.
  - E.g t1,t2,t3 has time stamp 5,10,15. now t1 request a data ,held by t2 then t1 will wait.
  - If t3 request data item held by t2 then t3 will be rollback

- **T1 will Wait T3 will rollback**
- **T1 ————————————-> T2 <————————————- T3**
- **(5) (10) (15)**

- 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.
  - when Ti request data item held by Tj then Ti will wait if it Ti is younger than Tj.
  - O.w Tj will roll back.
  - In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource.

- After the minute delay, the younger transaction is restarted but with the same timestamp.
- E.g t1,t2,t3 has time stamp 5,10,15. now t1 request a data ,held by t2 then data hold by t2 is preempted and t2 will roll back i.e t2(younger transaction force to rollback).
- T3 is request a data held by t2 then t3 will wait.

- **T1 access T2 will be**
- **Data item rollback**
- **T1 ——————> T2**
- **(5) (10)**
  - **Data item with T2 wait for T2 to finish**
  - **T2 <—————— T3**
  - **(10) (15)**

Deadlock prevention (Cont.)

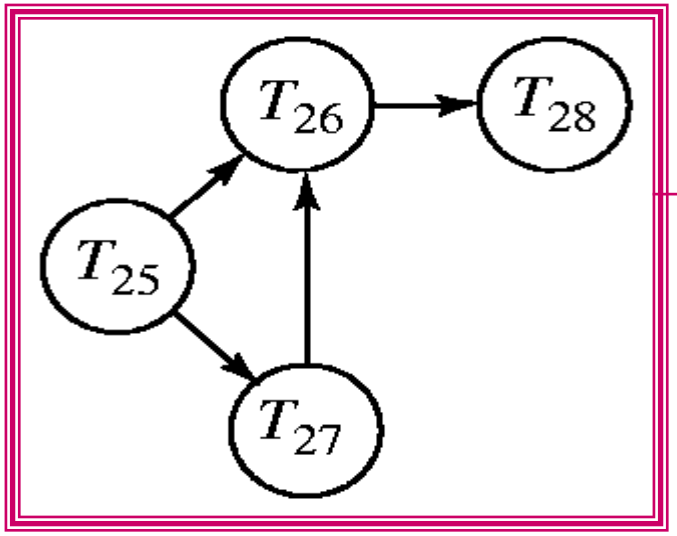
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

- Timeout-Based Schemes :
  - Deadlock handling is based on lock timeout.
  - 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 Detection

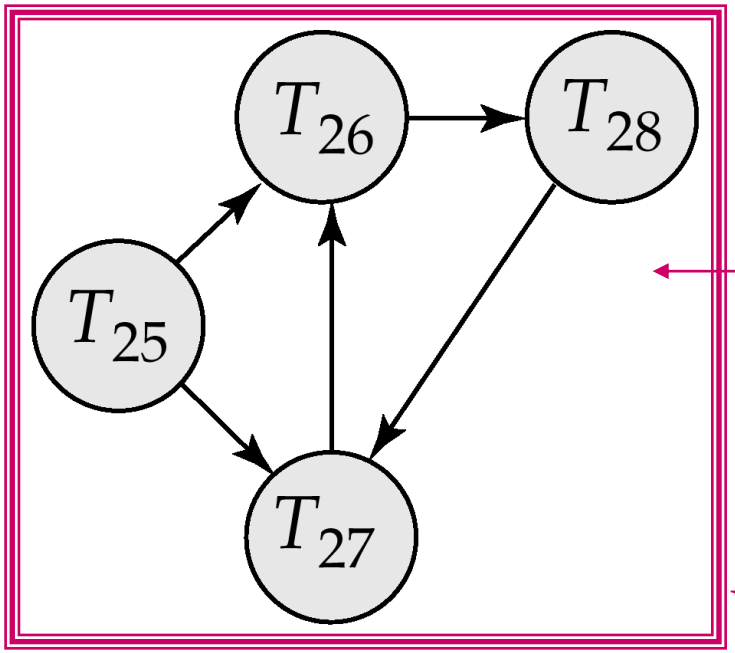
- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \ T_j$ 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.)



- Transaction *T*25 is waiting for transactions *T*26 and *T*27.
- • Transaction *T*27 is waiting for transaction *T*26.
- • Transaction *T*26 is waiting for transaction *T*28.
- Since the graph has no cycle, the system is not in a deadlock state.

Wait-for graph without a cycle



- now that transaction *T*28 is requesting an item held by *T*27. The edge
- *T*28 → *T*27 is added to the wait-for graph,
- *T*26 → *T*28 →*T*27 →*T*26 contain cycle.
- implying that transactions *T*26, *T*27, and *T*28 are all deadlocked.

Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
- Some transaction will have to rolled back
- Three action to be taken.

- Selection of victim :
  - (made a victim) to break deadlock.
  - Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.
  - Select that transaction as victim that will incur minimum cost.
  - And cost of rollback including
    - How many data items the transaction has used.
    - How many more data items needs for complition
    - How long transaction has computed and how longer it will computed to finish the task

- Rollback :
  - Once we have decided that a particular transaction must be rolled back,
  - we must determine how far this transaction should be rolled back.determine how far to roll back transaction.
  - Total rollback: Abort the transaction and then restart it.
  - Partial rollback :
    - More effective to **roll back transaction only as far as necessary to break deadlock.**
    - Additional information about the running transaction are required.
    - the sequence of lock requests/grants and updates performed by the transaction needs to be recorded.

- The selected transaction must be rolled back to the which may create deadlock. point where it obtained the first of these locks
- , undoing all actions it took after that point.

- Starvation :
  - IT happens if same transaction is always chosen as victim. , this transaction never completes its designated task, thus there is starvation.
  - We must ensure that transaction can be picked as a victim only a (small) finite number of times
  - TO AVOID the starvation.

Insert and Delete Operations

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple

- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation (e.g., find all accounts in Perryridge) and a transaction that inserts a tuple in the relation (e.g., insert a new account at Perryridge) may conflict in spite of not accessing any tuple in common
- the scan transaction may not see the new account, yet may be serialized before the insert transaction.

Delete operations

- Deletion : Delete(Q)
  - delete instructions affects concurrency control,
  - we must decide when a delete instruction conflicts with another instruction
  - . Let Ii and Ij instructions of Ti and Tj , respectively
  - r. Let Ii = delete(Q).and we consider several instruction of Ij
  - 1. Ij= read(Q). I_i and I_j conflict. If I_i comes before I_j , Tj will have a logical error.(as records are already deleted by the Ii) If Ij come before Ii then read execute.

- 2. Ij = write(Q). Ii and Ij conflict. If Ii comes before Ij (as record already delete by the Ii and Ij try to update), Tj will have a logical error.
- If Ij comes before Ii, Tj can execute the write operation successfully.
- 3 Ij = delete(Q). Ii and Ij conflict. If Ii comes before Ij , Tj will have a logical error. If Ij comes before Ii, Ti will have a logical error.
- 4. Ij = insert(Q). Ii and Ij conflict. Suppose that data item Q did not exist prior to the execution of Ii and Ij . Then, if Ii comes before Ij , a logical error results for Ti. If Ij comes before Ii, then no logical error results

Insertion operation

- an insert(Q) operation conflicts with a delete(Q) operation.
- insert(Q) conflicts with a read(Q) operation or a write(Q) operation; no read or write can be performed on a data item before it exists

End of Chapter