DATABASE MANAGEMENT SYSTEM

Subash Manandhar

Chapter 8 : Transactions and Concurrency Control

- **Transaction System:**
  - Group of operations that form a single logical unit of work is called transaction.
  - A transaction is a logical unit of work that must be either entirely completed or aborted.
  - A transaction is a unit of program execution that access and possibly updates various data items.
  - A transaction is the DBMS abstract view of a user program that consists of sequence of read and writes.
  - Transaction access data using two operations
    - Read(x)
    - Write(x)

Chapter 8 : Transactions and Concurrency Control

- **Transaction System:**
  - Transaction processing is information processing that is derived into individual, indivisible operations called transactions.
  - Each transaction must succeed or fail as a complete unit, but can't remain in intermediate state.
  - Transaction processing maintains a system in a consistent state.
  - E.g. T1 : read(A)

    A = A-1000

    write(A)

    read(B)

    B = B+1000

    write(B)

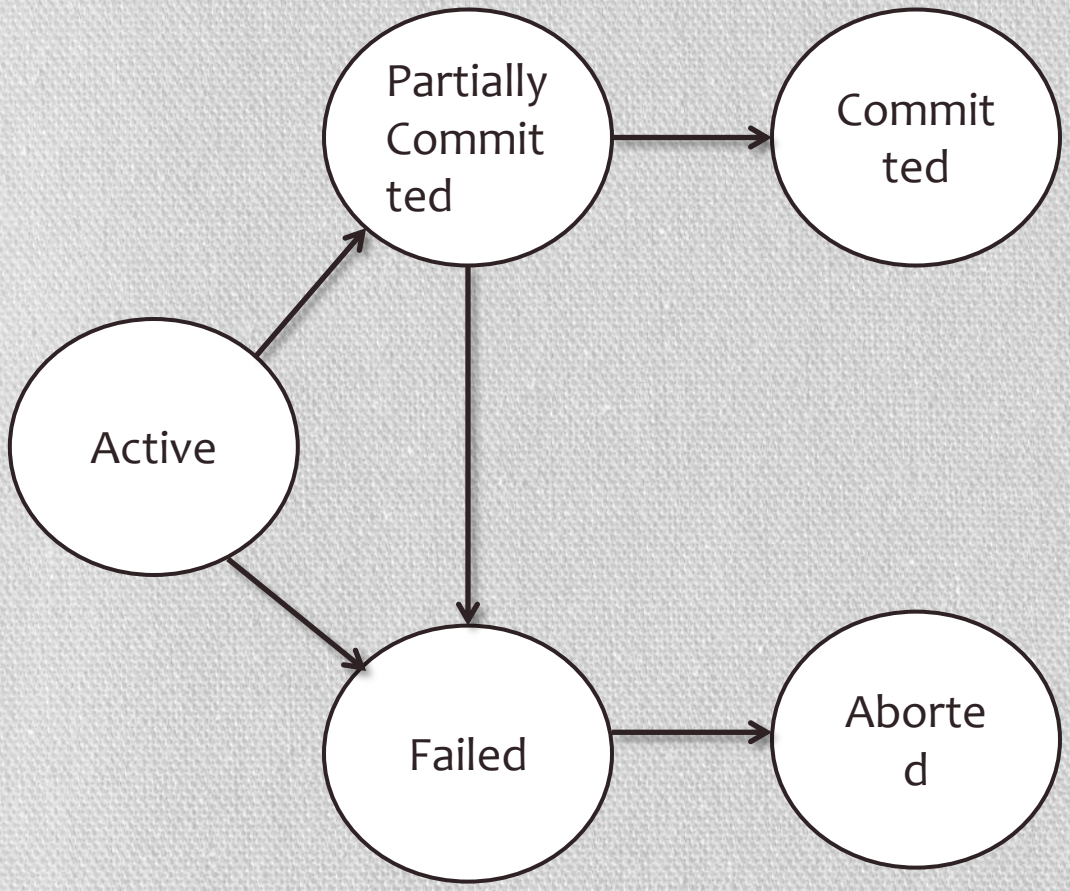
Chapter 8 : Transactions and Concurrency Control
ACID Properties of Transaction

- **ACID Properties:**
- **Atomicity**
  - Either all operations of transactions are reflected properly in database or none are.
- **Consistency**
  - Execution of a transaction in isolation preserves the consistency of database.
- **Isolation**
  - Even though multiple transactions may execute concurrently, the system guarantees that for every pair of transactions T_i and T_j , for T_i either T_j finished execution before T_i started or T_j started execution after T_i finished.
- **Durability**
  - After successful completion of transaction, the changes in database persist, even if there are system failures.

Chapter 8 : Transactions and Concurrency Control

- **Transaction State:**
- **Active State**
  - It is the initial state.
  - Transaction stays in this state while it is executing.
- **Partially Committed**
  - Transaction is in this state after the final statement has been executed.
- **Failed**
  - Transaction is in this state after normal execution can no longer proceed.
- **Aborted**
  - Transaction is in this state after it has been rolled back and the database has been restored to its state prior to the start of transaction.
- **Committed**
  - Transaction is in this state after successful completion

Chapter 8 : Transactions and Concurrency Control



- A transaction has committed only if it enter committed state.
- A transaction has aborted only if it enter abort state.
- A transaction has terminated if either committed or aborted.
- After transaction being in aborted state, system has two options either it can restart transaction or it can kill transaction.

FIG: State diagram of a transaction

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- Transaction processing system usually allow multiple transactions to run concurrently.
- This cause several complications with consistency of data.
- Though transactions can be run serially to ensure consistency, concurrent executions of transaction has following advantages
  - Improved throughput and resource utilization
  - Reduced waiting time
- Database system control the interaction among the concurrent transactions to prevent from destroying consistency of database through variety of mechanisms called **concurrency control scheme**.
- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operation from various transaction is known as **schedule**.
- *A schedule S specifies the chronological order in which the operations of concurrent transactions are executed.*

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- E.g. let T1 transaction transfer Rs.1000 from account A to B. T2 transaction transfer 20% of balance from A to B. Assume , A= 2000 , B=1000

T1: read(A)
A = A – 1000
write(A)
read(B)
B = B + 1000
write(B)

T2: read(A)
temp = A * 0.2
A = A – temp
write(A)
read(B)
B = B + temp
write(B)

T1	T2
read(A)	
A = A – 1000	
write(A)	
read(B)	
B = B + 1000	
write(B)	
	read(A)
	temp = A * 0.2
	A = A – temp
	write(A)
	read(B)
	B = B + temp
	write(B)

Fig 1: Serial Schedule

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

T1	T2
	read(A)
	temp = A * 0.2
	A = A – temp
	write(A)
	read(B)
	B = B + temp
	write(B)
read(A)	
A = A – 1000	
write(A)	
read(B)	
B = B + 1000	
write(B)	

Fig 2: Serial Schedule

T1	T2
read(A)	
A = A – 1000	
write(A)	
	read(A)
	temp = A * 0.2
	A = A – temp
	write(A)
read(B)	
B = B + 1000	
write(B)	
	read(B)
	B = B + temp
	write(B)

Fig 3: Non-Serial Schedule

Fig 3 Non-Serial Schedule or concurrent execution results correct state. But not all concurrent execution results correct state.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

T1	T2
read(A)	
A = A – 1000	
	read(A)
	temp = A * 0.2
	A = A – temp
	write(A)
	read(B)
Write (A)	
read(B)	
B = B + 1000	
write(B)	
	B = B + temp
	write(B)

Fig 4: Non-Serial Schedule

Fig 4 Non-Serial Schedule or concurrent execution does not results correct state.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Serializability:**
  - Basic assumption is each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserve database consistency.
  - Main objective of serializability is to search non serial schedules that allow transaction to execute concurrently without interfering one another transaction and produce the result of database state that could be produced by serial execution.
  - we can conclude that a non serial schedule is correct if it produce same result as serial execution.
  - A schedule is serializable if it is equivalent to a serial schedule.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Serializability:**
  - **Rules:**
    - Ordering of read/write is important
    - If two transactions only read data item they do not conflict and order is not important.
    - If two transactions either read or write completely separate data items, they do not conflict and order is not important.
    - If one transaction write a data item and another reads or writes same data item, order of execution is important.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Serializability:**
- E.g.

T1	T2	T3
R(a)		
	R(b)	
		R(c)
W(a)		
	W(b)	
		W(c)

Fig 5 : schedule 1

T1	T2	T3
R(a)		
W(a)		
	R(b)	
	W(b)	
		R(c)
		W(c)

Fig 6 : schedule 2

Here, actions of transactions in schedule 1 are not executed as same as in schedule 2 (i.e. serial schedule) but at the end schedule 1 gives same result as that of schedule 2. Thus schedule 1 is considered as ***serializable****.*

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Conflict Serializability:**
  - Instructions I_i and I_j of transactions T_i and T_j respectively conflicts
    - If and only if there exists same item Q accessed by both I_i and I_j and at least one of these instruction wrote Q
  - If a schedule S can be transformed into a schedule S' by a series of swapping of non conflicting instructions, then S and S' are conflict equivalent.
  - A schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

Instruction I_i	**Instruction I_j**	**Result**
Read(Q)	Read(Q)	No Conflict
Read(Q)	Write(Q)	Conflict
Write(Q)	Read(Q)	Conflict
Write(Q)	Write(Q)	Conflict

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **E.g .1** Let we have schedule A and schedule B as follows:

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Fig: schedule A

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Fig: schedule B

Here, schedule A can be transformed into serial schedule B by series of swaps of non conflict instructions.
Hence, schedule A is ***conflict serializable.***

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **E.g .2** Let we have schedule X and schedule Y as follows:

T1	T2
Read(Q)	
	Write(Q)
Write(Q)	

Fig: schedule X

T1	T2
Read(Q)	
Write(Q)	
	Write(Q)

Fig: schedule Y

Here, we can't transform schedule X into schedule Y by swapping of non conflict instructions.
Hence, schedule X is ***non conflict serializable.***

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **View Serializability:**
  - Two schedules are said to be view equivalent if following three conditions hold:
    - For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i in schedule S' must also read initial value of Q.
    - For each data item Q, if T_i executes read(Q) in S and if value was produced by write(Q) of T_i then read(Q) of T_i in S' must also read values of Q produced by some write(Q) if T_i.
    - For each data item Q, the transaction that performs the final write(Q) operation in S must perform the final write(Q) operation in S'.
  - A schedule S is view serializable if it is view equivalent to a serial schedule.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **E.g .1** Let we have schedule A and schedule B as follows:

T1	T2	T3
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)

Fig: schedule A

T1	T2	T3
Read(Q)		
Write(Q)		
	Write(Q)	
		Write(Q)

Fig: schedule B

- *Here, schedule A is **view serializable schedule** , because it is view equivalent to serial sechedule B, as transaction T1 perform Read(Q) i.e. reads the initial value of Q in both schedules. And also transaction T3 performs final Write(Q) in both schedules.*
- *Here, Write(Q) of T2 and T3 are called blind writes because it is performed without having performed Read(Q).*
- *Every conflict serializable schedule is also view serializable but not vice-versa.*

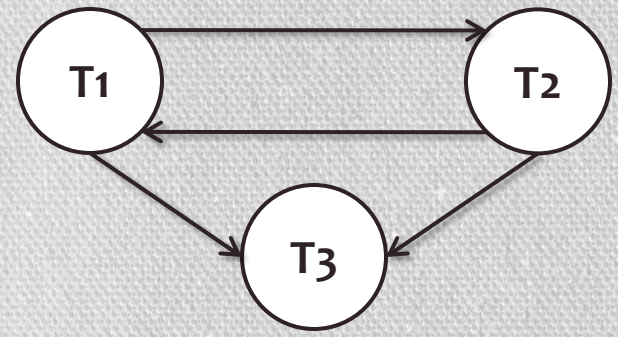
Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Testing for Serializability:**
  - Construct a directed graph called precedence graph from S.
  - Directed graph consists of a pair G=(V,E) where
    - V is set of vertices ; it consist of all transaction in the schedule.
    - E is a set of edges. Set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds.
      - T_i executes write (Q) before T_j executes read(Q)
      - T_i executes read(Q) before T_j executes write(Q)
      - T_i executes write(Q) before T_j executes write(Q)
  - If an edge $T_i \rightarrow T_j$ exists in precedence graph, then in any serial schedule S' equivalent to S ,T_i must appear before T_j.
  - If the precedence graph has a cycle then the schedule is not conflict serializable.
  - If the precedence graph has not any cycle then schedule is conflict serializable.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

E.g. 1

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

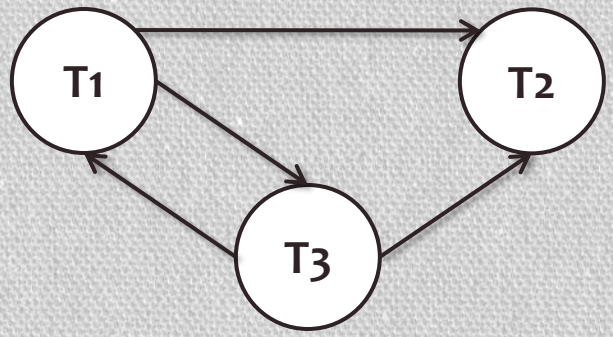


precedence graph

Here, as graph contains cycle, above schedule is non conflict serializable.

- **E.g. 2**

T1	T2	T3
Write(A)		
	Read(A)	
		Write(B)
Write(B)		
		Write(B)
	Write(A)	
		Read(B)
	Read(B)	



precedence graph

Here, as graph contains cycle, above schedule is non conflict serializable.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Recoverability:**
  - Address the effect of transaction failures during concurrent execution.
  - If a transaction T_i fails, we have to undo the effect of this transaction to ensure atomicity property of the transaction.
  - In concurrent execution, if T_i fails, then any transaction T_j that depends on T_i also have to be aborted.
  - A schedule is recoverable schedule where for each pair of transaction T_i and T_j such that T_j reads data written by T_i, the commit operation of T_i appears before commit operation of T_j.
  - A schedule is unrecoverable schedule where for each pair of transaction T_i and T_j such that T_j is dependent on T_i, T_j commits before T_i commits and then T_i fails.

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **E.g .** Let we have schedule A and schedule B as follows:

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Commit	
	Commit

Fig: schedule A

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Abort	
	Abort

Fig: schedule B

*Here, schedule A and schedule B are **recoverable schedules**.*

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **E.g .** Let we have schedule C as follows:

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
	Commit
Abort	

Fig: schedule C

Here, schedule C is ***unrecoverable schedule.***

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Cascadeless Schedules:**
  - To recover correctly from failure of transaction in recoverable schedule several transactions have to be rolled back.
  - The phenomenon in which a single transaction failure leads to series of transaction rollbacks is called cascading rollback.
  - E.g . Let we have schedule A as follows:

T1	T2	T3
Read(A)		
Read(B)		
Write(A)		
	Read(A)	
	Write(A)	
		Read(A)

Fig: schedule A

- *Here, transaction T2 is dependent on T1 and transaction T3 is dependent on T2.*
- *So, if T1 fails, T1 must be rolled back.*
- *Since T2 is dependent on T1 , T2 must be rolled back.*
- *Again as T3 is dependent on T2 , T3 mustalso be rolled back.*
- *This is **cascading rollback.***

Chapter 8 : Transactions and Concurrency Control
Schedules and Serializability

- **Cascadeless Schedules:**
  - Cascading rollback is undesirable as loads of works have to be undo.
  - We can restrict the schedule where cascading rollback cannot occur, such schedules are called cascadeless schedules.
  - Formally, a schedule is cascadeless schedule where for each pair of transaction T_i and T_j such that T_j reads data written by T_i, the commit operation of T_i appears before the read operation of T_j.
  - Every cascadeless schedules are recoverable.
  - Transaction in SQL:
    - Transactions are ended by one of these SQL state:
      - Commit → commits current transaction and begin new.
      - Rollback → cause current transaction to abort.

Chapter 8 : Transactions and Concurrency Control

- **Concurrency Control:**
  - Ensures that database transactions are performed concurrently without violating data integrity of database.
  - Different concurrency control schemes can be used to ensure the isolation property is ensured when multiple transactions are executed in parallel.
  - DBMS must guarantee that only serializable recoverable schedule are generated and that no effect of committed transaction is lost and no effect of aborted transaction remains in database.
  - Different methods like locking method, timestamp methods, validation based techniques are used.

Chapter 8 : Transactions and Concurrency Control

- **Lock Based Protocol:**
  - A lock is a mechanism to control concurrent access to a data item.
  - Data items can be locked in two modes
    - **Exclusive(X) mode**
      - Data item can be both read as well as written.
      - X-lock is requested using Lock-X instruction.
    - **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 only be processed once request is granted.
  - 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.
  - 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 Protocol:**

	S	X
S	TRUE	FALSE
X	FALSE	FALSE
Lock Compatibility Matrix		

- Shared mode is compatible only with shared mode. i.e. several shared mode locks can be held simultaneously by different transactions on any data item.
- If any transaction holds an exclusive lock on the item, no other transaction may hold any lock on the item.
- A transaction can unlock a data item Q by instruction Unlock(Q)
- Transaction must hold a lock on a data item as long as it access item.

- **E.g.**

T:	Lock-S(A)
	read(A)
	Unlock(A)
	Lock-S(B)
	read(B)
	Unlock(B)
	display(A+B)

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocol restricts the set of possible schedule.
- Locking protocol must ensure serializability.
- ***Let , value of account A and B is 1000 and 1000 respectively.***
- ***Transaction T1 transfer 200 from A to B***
- ***Transaction T2 display sum of A and B***

- Now,

T1:	
	Lock-X(A)
	read(A)
	A=A-200
	write(A)
	Unlock(A)
	Lock-X(B)
	read(B)
	B=B+200
	write(B)
	Unlock(B)

T2:	
	Lock-S(A)
	read(A)
	Unlock(A)
	Lock-S(B)
	read(B)
	Unlock(B)
	display(A+B)

- If T1 and T2 are executed serially, we get correct result.
- But, concurrent execution of T1 and T2 may result incorrect value like in schedule X .

T1	T2	Concurrency Control Manager
Lock-X(A)		
		grant-X(A,T1)
read(A)		
A=A-200		
write(A)		
Unlock(A)		
	Lock-S(A)	
		grant-S(A,T2)
	read(A)	
	Unlock(A)	
	Lock-S(B)	
		grant-S(B,T2)
	read(B)	
	Unlock(B)	
	display(A+B)	
Lock-X(B)		
		grant-X(B,T1)
read(B)		
B=B+200		
write(B)		
Unlock(B)		

Schedule X

- Assume that unlocking is delayed at the end of transactions then T1 becomes T3 and T2 becomes T4 as follows:

T3:	Lock-X(A) read(A) A=A-200 write(A) Lock-X(B) read(B) B=B+200 write(B) Unlock(A) Unlock(B)

T4:	Lock-S(A) read(A) Lock-S(B) read(B) display(A+B) Unlock(A) Unlock(B)

- With T3 and T4 we can't get schedule X and for any other schedules, it produce correct value like in schedule Y

T3	T4	Concurrency Control Manager
Lock-X(A)		
		grant-X(A,T3)
read(A)		
A=A-200		
write(A)		
Lock-X(B)		
		grant-X(B,T3)
read(B)		
B=B+200		
write(B)		
Unlock(A)		
	Lock-S(A)	
		grant-S(A,T4)
	read(A)	
Unlock(B)		
	Lock-S(B)	
		grant-S(B,T4)
	read(B)	
	display(A+B)	
	Unlock(A)	
	Unlock(B)	

Schedule Y

- **Pitfalls of Lock Based Protocol:**
- Consider partial schedule as follows

T3	T4
Lock-X(B)	
read(B)	
B=B-50	
write(B)	
	Lock-S(A)
	read(A)
	Lock-S(B)
Lock-X(A)	

Here,

- Neither T3 not T4 can make progress.
- Executing Lock-S(B) causes T4 to wait for T3 to release its lock on B, while executing Lock-X(A) causes T3 to wait for T4 to release its lock on A.
- Such a situation is called a **deadlock**.
- To handle a deadlock one of transactions T3 or T4 must be rolled back and its lock must be released.

- if we do not use locking, we may get inconsistent state.
- Similarly, if we do not unlock data item before requesting lock on another data item, deadlock occurs.

- **Pitfalls of Lock Based Protocol:**
  - However, deadlocks are necessary evil, more preferable than inconsistent states.
  - The potential for deadlock exists in most locking protocol.
  - Starvation is also possible if control manager is badly designed like
    - A transaction may be waiting for an X-Lock on an item, while a sequence of other transactions request S-Lock and are granted on same item.
    - Same transaction is repeatedly rolled back due to deadlock.
  - Starvation can be avoided by granting locks in following manner:
    - Let T_i request a lock on a data item Q in particular mode M, the CCM grants lock provided
      - There is no other transactions holding lock on Q in a mode that conflicts with M
      - There is no other transactions waiting for lock on Q and that made its lock request before T_i.

Chapter 8 : Transactions and Concurrency Control

- **Two Phase Locking Protocol:**
  - Ensures serializability.
  - Here, transaction issue lock and unlock requests in two phases
    - Growing Phase:
      - Transaction may obtain locks
      - Transaction may not release locks
    - Shrinking Phase
      - Transaction may release locks
      - Transaction may not obtain locks
  - Initially transaction is in growing phase.
  - After transaction release a lock it enters shrinking phase.
  - E.g. T3 and T4 are two phase . But not T1 and T2
  - It can be proved that transactions can be serialized in order of their lock points (i.e. the point where a transaction acquired its final lock.)
  - Two phase locking does not ensure freedom from deadlocks.
  - Cascading rollback is possible under two phase locking.

- **Two Phase Locking Protocol:**
  - In schedule X: if Ta fails after read(A) of Tc then there must be cascading rollback of Tb and Tc.

T1	T2	T3
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)

Fig: partial schedule X under Two phase locking

- **Two Phase Locking Protocol:**
  - **Strict Two Phase Locking Protocol**
    - Here, transaction must hold all its exclusive mode locks till it commits.
    - This prevents any other transaction from reading the data.
    - No cascading rollback.
  - **Rigorous Two Phase Locking Protocol**
    - Here, all locks (S and X modes) are hold till commits.
    - No cascading rollback.
    - Here, transaction 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.
  - Let T1 : Write(X) Write(Y) T2 : Write(Y) Write(X)
  - Now , schedule with deadlock

T1	T2
Lock-X(X)	
Write(X)	
	Lock-X(Y)
	Write(Y)
	{wait for Lock-X on X}
	Write(X)
{wait for Lock-X on Y}	
Write(Y)	

- *To deal with deadlock we can use **1. Deadlock Prevention Protocol** (which ensures that system will never enter a deadlock state.) **2. Deadlock Detection and Recovery Scheme** (which try to recover system once it entered deadlock state.)*
- *Both methods may result in transaction rollback.*
- *If probability of system entering deadlock state is relatively high, prevention is used. Otherwise detection and recovery are more efficient*

- **Deadlock Prevention:**
  - Deadlock prevention protocol ensure that the system will never enter into deadlock state.
  - Some prevention strategies:
    - Require that each transaction locks all data item before it begins execution.
    - Impose partial ordering of all data items. Require that a transaction can lock data items only in order specified by partial order (e.g. graph based protocol).
    - Timeout based schemes: a transaction waits for a lock only for specified amount of time. After the wait time is out then transaction is roll back.
      - Simple to implement but starvation is possible.
      - Also difficult to determine good value of time out interval.

- **Deadlock Prevention:**
  - Following schemes use transaction timestamps for the sake of deadlock prevention:
    - **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.
    - **Wound-Wait Scheme**
      - Preemptive
      - Older transactions wounds(forces rollback) younger transactions instead of waiting for it.
      - Younger transactions may wait for older ones.
      - May be fewer rollbacks than wait die scheme.
  - In both schemes, a rolled back transaction is restarted with its original timestamp.
  - Older transaction thus have precedence over newer ones in these schemes and starvation is hence avoided.

- **Deadlock Detection and Recovery:**
  - Is used if no protocol is used to ensure deadlock freedom.
  - Here, to determine whether deadlock occurs or not, some algorithms to check must be implemented.
  - If deadlock occurs, then must recover from deadlock.
  - **Deadlock Detection**
    - Deadlocks can be described as a wait for graph which consists of a pair G=(V,E) where V is a set of vertices (transactions) and E is a set of edges; each edge is ordered pair $T_i \rightarrow T_j$.
    - If $T_i \rightarrow T_j$, then there is a directed edge from T_i to T_j . Here, T_i is waiting for T_j to release data item.
    - When transaction T_i request a data item held by T_j then $T_i \rightarrow T_j$ is inserted in wait for graph. This edge is removed only when T_i no longer holding data item needed by T_j.
    - The system is in deadlock state if and only if wait for graph has a cycle.
    - The system invokes a deadlock detection algorithm periodically to look for a cycle.

- **Deadlock Detection:**

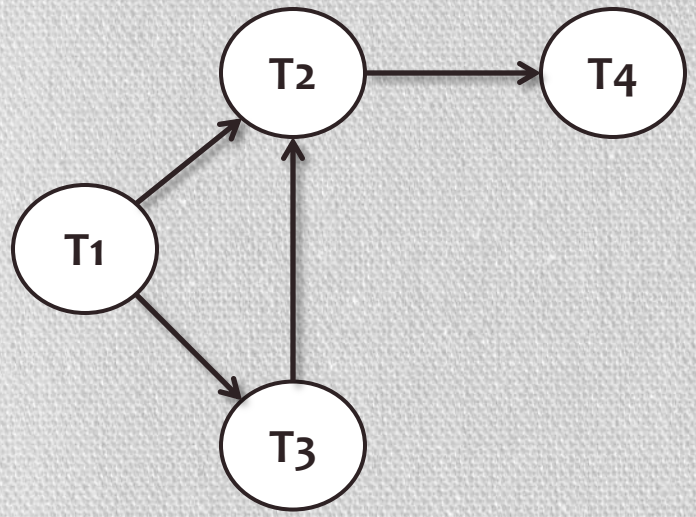


Fig: Wait for graph without cycle

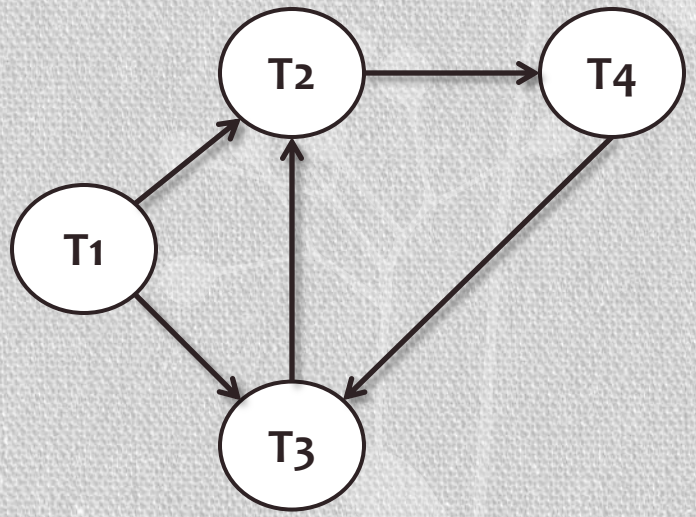


Fig: Wait for graph with cycle

- **Deadlock Recovery:**
  - When deadlock is detected:
    - Some transactions will have to rollback to break deadlock. (select a victim)
    - Rollback – determine how far to rollback transaction.
      - Total rollback : abort transaction and then restart it.
      - Partial rollback : rollback transaction only as far as necessary to break the deadlock, and is more effective.
  - Starvation happens if same transactions is always chosen as victim.
    - Must ensure that transaction can be picked as a victim only a small no. of times.