Unit 8

Transaction and Concurrency control

Transaction Concepts

- ✓ Collections of operations that form a single logical unit of work are called transactions.
- ✓ It is a unit of program execution that accesses and possibly updates various data items
- ✓ A database system must ensure proper execution of transactions despite failures, either the entire transaction executes, or none of it does.
- ✓ During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully (is committed), the database must be consistent.
- ✓ After the transaction commits the changes it has made to the database persist, even if there are system failures.
- ✓ Multiple transaction can also execute in parallel. Example: Airlines reservation system

Example: transaction to transfer 50 from account A to account B:

```
1.    read(A)
2     A := A – 50
3.    write(A)
4.    read(B)
5.    B := B + 50
6.    write(B)
```

Two main issues to deal with:

- ✓ Failures of various kinds, such as hardware failures and system crashes
- ✓ Concurrent execution of multiple transactions

ACID Properties of Transaction

Atomicity

- ✓ This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none.
- ✓ There must be no state in a database where a transaction is left partially completed.
- ✓ Transaction is indivisible, it completes entirely or not at all, despite failures.

Consistency

- ✓ This means that integrity constraints must be maintained so that the database is consistent before and after the transaction.
- ✓ It refers to the correctness of a database.

Isolation

- ✓ Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.
- ✓ In simple words, if a transaction T1 is being executed and using the data item X, then data item cannot be accessed by another transaction until the transaction T1 ends.
- ✓ This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state.

Durability

- ✓ The durability property of transaction indicates that the changes made to the database by a committed transaction must persist in the database.
- ✓ These changes must not be lost by any kind of failure.

Example:

Transaction to transfer 50 from account A to account B

```
1.    read(A)
2     A := A – 50
3.    write(A)
4.    read(B)
5.    B := B + 50
6.    write(B)
```

Atomicity requirement: If the transaction fails after step 3 and before step 6, after write(A) but before write(B), then the amount has been deducted from A but not added to B. In this case the system should ensure that its update are not reflected in database. Otherwise this results an inconsistent database state.

Consistency requirement: The sum of A and B is unchanged by the execution of the transaction.

Isolation requirement: If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

	T1	T2
1.	read(A)	
2.	$A := A - 50$	
3.	write(A)	
4.	read(B)	
5.	$B := B + 50$	read(A), read(B), print(A+B)
6.	write(B)	

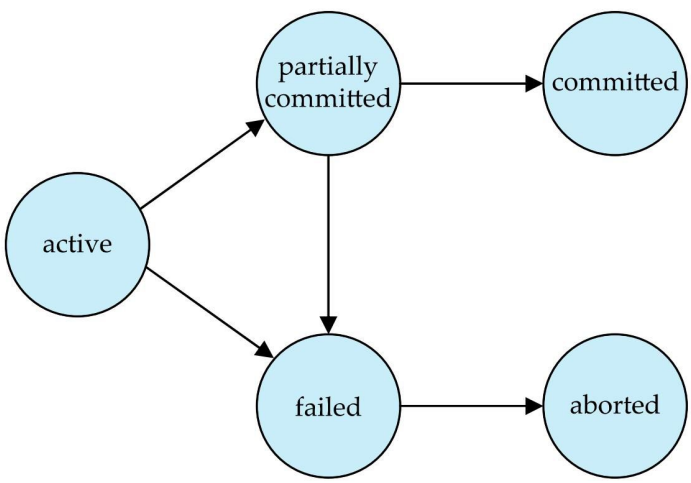
- ✓ Here, isolation can be ensured by running transactions serially, that is one after the another.
- ✓ However, executing multiple transactions concurrently has significant benefits.

Durability requirement: Once the user has been notified that the transaction has completed (i.e., the transfer of the 50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction Model and state diagram

A transaction is seen by the DBMS as a series, or list of actions. To ensure the reliability and consistency of database operations, even in the presence of system failures or errors, a transaction model is necessary.

We therefore established a simple transaction model named **transaction states.** These are the states which tell about the current state of the transaction and also tell how we will further do the processing in the transactions.



Active – This is the first state of transaction and in this state the instructions of the transaction are executing. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

Partially committed - After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Database, then the state will change to "committed state" and in case of failure it will go to the "failed state".

Failed -When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Database.

Aborted -After having any type of failure the transaction goes from "failed state" to "aborted state" and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

Committed – It is the state the transaction is complete and the changes are made permanent on the database

Schedules

When several transactions are executing concurrently then order of execution of various instructions is known as schedules.

Types of schedules:

1. Serial schedule
2. Non-serial schedule

1. Serial schedule

In serial schedules

- ✓ All the transactions execute serially one after the other.
- ✓ It is a type of schedule where commit or abort of one transaction initiates the execution of next transaction.
- ✓ When the first transaction completes its cycle, then the next transaction is executed and so on.
- ✓ No interleaving occurs in serial schedule

2. **Non serial schedule**

In non-serial schedules

- ✓ Multiple transactions execute concurrently.
- ✓ Operations of all the transactions are inter leaved or mixed with each other.
- ✓ It contains many possible orders in which the system can execute the individual operations of the transactions.

Schedule 1

Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

In this schedule

- ✓ There are two transactions T1 and T2 executing serially one after the other.
- ✓ Transaction T1 executes first.
- ✓ After T1 completes its execution, transaction T2 executes.
- ✓ So, this schedule is an example of a Serial Schedule.

Schedule 2

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

In this schedule

- ✓ There are two transactions T1 and T2 executing serially one after the other.
- ✓ Transaction T2 executes first.
- ✓ After T2 completes its execution, transaction T1 executes.
- ✓ So, this schedule is also an example of a Serial Schedule.

Schedule 3

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In this schedule,

- ✓ There are two transactions T1 and T2 executing concurrently.
- ✓ The operations of T1 and T2 are interleaved.
- ✓ So, this schedule is an example of a Non-Serial Schedule.

Note:

Two schedules are said to be equivalent schedules if the execution of first schedule is identical to the execution of second schedule. Here schedule 3 is equivalent to schedule 1. In schedule 1, 2 and 3, the sum A + B is preserved.

schedule 4

The following concurrent schedule does not preserve the value of *(A + B)*.

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- ✓ A schedule of S of n transaction is serializable if it is equivalent to some serial schedule of same n transactions.
- ✓ Two schedule are called result equivalent if they produce the same final state of the database.
- ✓ A schedule is **serializable** if it is equivalent to the serial schedule.

Things to be understand:

- ☞ *Serializability of schedules ensures that a non-serial schedule is equivalent to a serial schedule.*
- ☞ *It helps in maintaining the transactions to execute simultaneously without interleaving one another.*
- ☞ *In simple words, serializability is a way to check if the execution of two or more transactions are maintaining the database consistency or not.*
- ☞ *Basic assumption in each transaction preserve data consistency. Thus serial execution of a set of transaction preserves data consistency.*
- ☞ *Main objective of serializability is to find a non-serial schedule that allow transactions to execute concurrently without interference and produce a same effect on the database state that could be produced by a serial execution.*

Different forms of schedule equivalence give rise to notations of:

- ➢ Conflict serializability
- ➢ View serializability

Conflict serializability

- ✓ A non-serial schedule is a conflict serializable if, after performing some swapping on the non-conflicting operations, it can be transforms into a serial schedule.
- ✓ Actions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some data item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
- ❖ I_i=read(Q) ,I_j=read(Q) I_i and I_j don't conflict
- ❖ I_i=read(Q) ,I_j=write(Q) They conflict
- ❖ I_i=write(Q) ,I_j=read(Q) They conflict
- ❖ I_i=write(Q) ,I_j=write(Q) They conflict

- ✓ If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Things to be understand

Two operations inside a schedule are called conflicting if they meet these three conditions:

- ✓ They belong to two different transactions.
- ✓ They are working on the same data item.
- ✓ One of them is performing the WRITE operation.

Example:

Schedule 1:

T1	T2
Read(A)	
	Read(A)

Here, in **Schedule 1** two operations are non -conflicting so they can be swapped.

After swapping schedule becomes

T1	T2
	Read(A)
Read(A)	

Schedule 2:

T1	T2
Read(A)	
	Write(A)

Here in schedule 2 two operations are conflicting, so they cannot be swapped.

Schedule 3:

T1	T2
Read(A)	
	Write(B)

Here in schedule 3 two operations are non-conflicting, because in spite of having one write operation, but operation is performing in different data item.
So they can be swapped after swapping schedule becomes

T1	T2
	Write(B)
Read(A)	

Conflict Equivalent and Conflict serializable

- ✓ Two schedules are **conflict equivalent** if they can be turned one into another by a sequence of non-conflicting swaps of adjacent actions.
- ✓ We say that a schedule S is **conflict serializable if** it is conflict equivalent to a serial schedule.

Here, **Schedule 1** can be transformed into **Schedule 2,** a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 1 is conflict serializable.

Schedule1

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 2

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Testing for conflict serializability

- ✓ Precedence graph or serialization graph is commonly used to test conflict serializability of a schedule.
- ✓ It is set of directed graph (V,E) consisting of set of nodes V={T1,T2,T3,T4,..Tn} and a set of directed edges E={e1,e2,e3….en}
- ✓ The graph contains one node for each transaction Ti.
- ✓ An edge ei is of the form Ti→Tj where Ti is starting node of ei and Tj is ending node of ei.
- ✓ For each transaction Ti participating in schedule S ,create node labeled Ti in the precedence graph.

For each case in **schedule S**

- ➢ Where Tj executes a read_item(X) after Ti executes a write_item(X) ,create an edge (Ti→Tj)
- ➢ Where Tj executes a write_item(X) after Ti executes a read_item(X) ,create an edge (Ti→Tj)
- ➢ Where Tj executes a write_item(X) after Ti executes a write_item(X) ,create an edge (Ti→Tj)

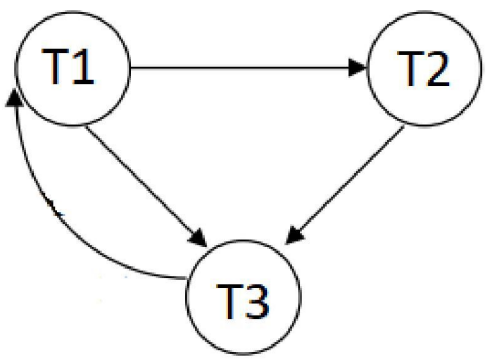
The schedule S is serializable if and only if the precedence graph has no cycles.

Example 1

Test serializability of a given schedule

T1	T2	T3
Read(X)		
		Read(X)
Write(X)		
	Read(X)	
		Write(X)

The precedence graph is as below



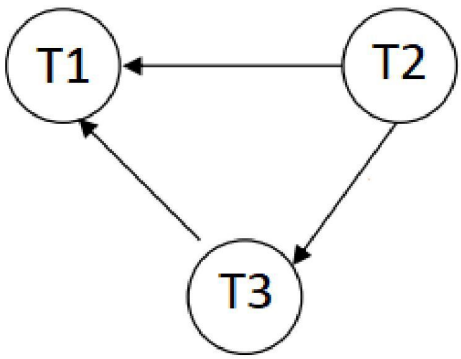
Since the graph contains a cycle, hence it is not conflict serializable.

Example 2

Test serializability of a given schedule

T1	T2	T3
		Read(X)
	Read(X)	
		Write(X)
Read(X)		
Write(X)		

The precedence graph is as below.



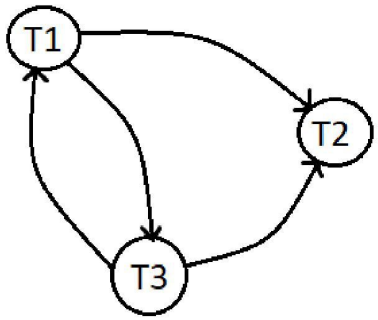
Here graph does not contain a cycle, hence it is conflict serializable.

Example 3:

Test serializability of a given schedule

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

The precedence graph is as below.



Since the graph contains a cycle, hence it is not conflict serializable.

Example 4:

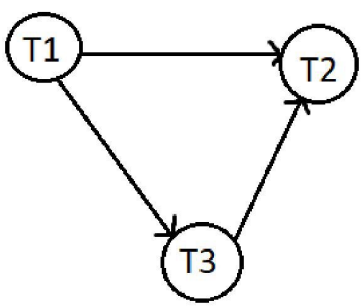
Consider the schedule S1.

S1: r1(x),r3(y) w1(x),w2(y),r3(x),w2(x)

Creating tabular representation of above schedule

T1	T2	T3
r(x)		
		r(y)
w(x)		
	w(y)	
		r(x)
	w(x)	

The precedence graph is as below.



Since the graph is acyclic ,the schedule is conflict serializable.

Performing Topological sort on this graph would give a possible serial schedule that is conflict equivalent to schedule S1.

In topological Sort, we first select the node with in-degree 0,which is T1. This would be followed by T3 and T2 so,S1 is conflict serializable since it is conflict equivalent to serial schedule T1→T3→T2.

View serializability

- ✓ A schedule is view serializable if it is view equivalent to serial schedule.
- ✓ Every conflict serializable schedule is also view serializable but reverse may not always true.
- ✓ Every view serializable schedule that is not conflict serializable has blind writes.

Let S and S' be two schedules with the same set of transactions. S and S' are view equivalent if the following three conditions are met, for each data item Q.

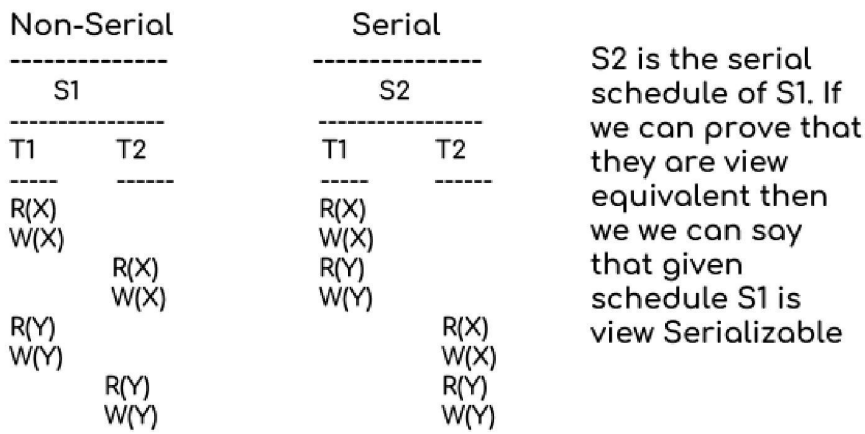
1. If in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.

2. If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .

3. The transaction (if any) that performs the final write(Q) operation in schedule S must also perform the final write(Q) operation in schedule S'.

Let us discuss above conditions with examples.

Example 1:



Initial read

- ✓ In schedule S1, transaction T1 first reads the initial value of X and in also schedule S2 transaction T1 reads the initial value of X. This conditions also satisfies for Y also.
- ✓ We checked for both data items X & Y and the initial read condition is satisfied in S1 & S2.

Updated read

- ✓ In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.
- ✓ In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.
- ✓ The update read condition is also satisfied for both the schedules.

Final write

- ✓ In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.
- ✓ Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.
- ✓ We checked for both data items X & Y and the final write condition is satisfied in S1 & S2.

Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Example 2:

Schedule 1

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

Schedule 2

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

Initial Read

In schedule 1, transaction T1 first reads the initial value of A and in also schedule 2 transaction T1 reads the initial value of A.

Updated read

In both schedule 1 and Schedule 2, there is no read except the initial read that's why we don't need to check that condition.

Final write

In schedule 1, the final write operation on A is done by transaction T3. In Schedule 2 also transaction T3 performs the final write on A.

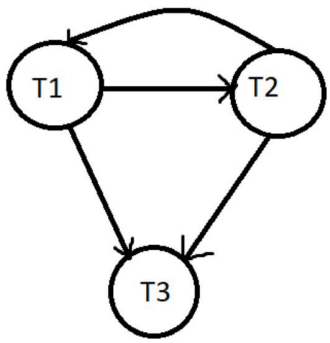
Here all the conditions are satisfied. So, schedule 1 is view equivalent to the schedule 2 which is serial schedule. As we know if any schedule is view equivalent to the serial schedule we can say that schedule is view serializable. So schedule 1 is view serializable.

Note:

Schedule that is view serializable may not be conflict serializable. In above Example 2

Schedule 1 is view serializable but not conflict serializable and also it must be noted that Every view serializable schedule that is not conflict serializable has blind writes(write without reading)

To test conflict serializability of schedule 1 in above Example 2 lets draw precedence graph



Here, precedence graph contains cycle so schedule is not conflict serializable but view serializable.

Concurrent Executions

A multiple transactions are allowed to run concurrently in the system.

Advantages of **concurrent executions** are:

- ✓ Increased processor and disk utilization: leading to better transaction throughput
  E.g. one transaction can be using the CPU while another is reading from or writing to the disk
- ✓ Reduced average response time for transactions: short transactions need not wait behind long ones.

Problems with concurrency

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner then it might lead to several problems. Such problems are called concurrency problems.

Concurrency problems in transactions:

1) Dirty read problem (Temporary update problem)
2) Unrepeatable read problem
3) Lost update problem
4) Phantom read problem

1) Dirty read problem

Reading the data written by an uncommitted transaction is called as dirty read.

The dirty read problem occurs when one transaction update an data item of database, and somehow the transaction fails. But before the item gets rollback, the updated database item is accessed by another transaction.

Example:

T1	T2
Read(A) A=A+100 Write(A) . . . . . . Failure	 Read(A) A=A-200 Write(A) commit

Here, in above transaction

- ✓ T2 reads the value of A written by the uncommitted transaction T1.
- ✓ T2 writes the updated value of A.
- ✓ T2 commit
- ✓ T1 fails in later stages and rollbacks.
- ✓ Thus the value of T2 read now stands to be incorrect.
- ✓ Therefore database becomes inconsistent.

2) Unrepeatable read problem

This problem occurs when two or more read operation of the same transaction read different values of the same database item.

T1	T2
Read(A) Read(A)	 Read(A) A=A+100 Write(A)

Here,

- ☞ T1 reads the value of A(=500 say)
- ☞ T2 reads the value of A(=500)
- ☞ T2 updates the value of A from 500 to 600
- ☞ T1 again reads value of A (but=600)

Here, T1 gets reads different value of A in second reading, that was updated by T2.

3) Lost update problem

In the Lost update problem, the update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

T1	T2
Read(A) A=A-50 Write(A) Commit	 Write(A) commit

Here,

- T1 reads the value of A(=500 say)
- T1 updates the value of A(=450)
- T2 does blind write , say A=25(write without read)
- T2 commit
- When T1 commits, it writes A=25 in the database

4) Phantom read problem

T1	T2
Read(A) delete(A)	 Read(A) Read(A)

The phantom read problem occurs when a transaction reads a data item once but when it tries to read the same data item again, an error occurs saying that the items does not exist.

Here,

- T1 reads A
- T2 reads A
- T1 deletes A
- T2 tries to reading A but does not find it

Concurrency control

- ✓ Concurrency control protocols are the set of rules which are maintained in order to solve the concurrency control problems in the database.
- ✓ It ensures that the concurrent transactions can execute properly while maintaining the database consistency.

Some of the currency control protocols are as follows:

1. **Lock- based protocol**
2. **Graph based protocol**

1. Lock –based protocol

a) Simple locking protocol

In this protocol each transaction cannot read or write the data until it acquires an appropriate lock on it.

Data items can be locked in two modes

1. **Shared (S) mode**

If a transaction T_i has a shared mode lock on data item Q then

- ✓ T_i can be read them but not update Q
- ✓ Any other transactions can also obtain shared lock on the same data item Q but no exclusive lock

S-lock is requested using **lock-S** instruction.

2. Exclusive(X) mode

If a transaction T_i has exclusive mode lock on data time Q then

- ✓ T_i can both read and update Q
- ✓ No other transaction can obtain either of shared lock or Exclusive lock on the same data item Q

X-lock is requested using **lock-X** instruction

Based on these locking modes we can define Lock-compatibility matrix

	S	X
S	True	False
X	False	False

- ✓ A transaction may granted a lock on data item if the requested lock is compatible with locks already held on data items by other transactions.
- ✓ If a lock cannot granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- ➢ In above compatibility matrix, shared mode is compatible with only shared mode.
- ➢ If any transactions holds an exclusive lock on 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 data item as long as it access item.

Problems associated with simple locking protocol

1) The serializability cannot be always assured

Let us consider a banking example. Let A and B be the two bank accounts that are accessed by transaction T1 and T2. Transaction T1 transfer transfers 50 from account B to account A. Transaction T2 displays the total amount of money in accounts A and B – that is , the sum (A+B)

```
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);
```

Transaction T1

```
T2: lock-S(A)
read(A)
unlock(A)
lock-S(B);
read(B);
unlock(B);
display(A+B);
```

Transaction T2

Suppose that values of account A and B are 100 and 200 respectively. If these two transactions are executed serially, either in the order T1 ,T2 or the order T2,T1 then transaction T2 will be display the value 300.If these transaction executed concurrently, then schedule 1 as shown in figure below is possible. In this case transaction T2 displays 250 which is incorrect. **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.**

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

2) 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 T3 nor 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 T3 or T4 must be rolled back and its locks released.

3) Starvation

Starvation is also possible if concurrency control manager is badly designed.

For example

- ✓ Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock.
- ✓ Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish.
- ✓ But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it.

In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved.

We can avoid starvation of transactions by granting locks in the following manner:

When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

1. There is no other transaction holding a lock on Q in a mode that conflicts with M.

2. There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti .Thus, a lock request will never get blocked by a lock request that is made later.

b) Two phase locking protocol

- ✓ Simple Lock-based protocol does not guarantee Serializability. Schedules may follow the preceding rules but a non-serializable schedule may result.
- ✓ This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL **ensures serializability**.

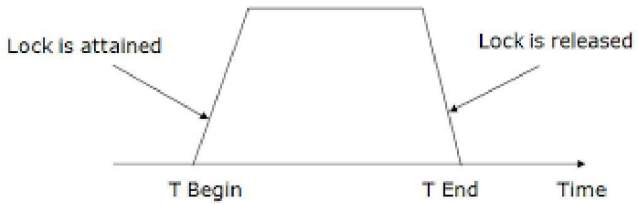
Here transaction issues lock and unlock request in two phases.

Phase 1: Growing phase

In growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Phase 2: Shrinking phase

In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.



Note : If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

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).

Here, growing and shrinking phases of 2PL can be illustrated by using the following diagram

Time	T1	T2
1	Lock-S(A)	
2		Lock-S(A)
3	Lock-X(B)	
4		
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)

Transaction T1

Growing Phase: From Time 1 to 3.
Shrinking Phase: From Time 5 to 7
Lock Point: At time 3

Transaction T2

Growing Phase: From Time 2 to 6
Shrinking Phase: From Time 8 to 9
Lock Point: At Time 6

As we say, 2PL will generate the serial schedule then what will be the sequence of transaction execution?

Answer: If locking point of Transaction T1 comes earlier than transaction T2 then the T1 will execute first than T2.

***Note**: Locking point means the point where a transaction acquired its final lock*

Although,2-Phase Locking ensures serializability, but there are still some drawbacks of 2-PL.They are as follows:

1) **The schedule that is produced through 2PL may be irrecoverable**

   Example:

T1	T2
Lock-X(A)	
R(A)	
Write(A)	
Unlock(A)	
.	Lock-S(A)
.	Read(A)
.	Unlock(A)
.	Commit
Failure	

If a transaction does a dirty read operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called an irrecoverable schedule.

The above schedule is a irrecoverable because of the reasons mentioned below

- ✓ The transaction T2 is also committed before the completion of transaction T1.
- ✓ The transaction T1 fails later and there are rollbacks.
- ✓ The transaction T2 reads an incorrect value.
- ✓ The transaction T2 which is performing a dirty read operation on A.

Finally, the transaction T2 cannot recover because it is already committed.

2) **The schedule that is produced through 2PL locking may still contains a deadlock problem**

T1	T2
Lock-X(A)	
	Lock-X(B)
Lock-X(B)	
	Lock-X(A)

Here,

- ✓ In T1 Exclusive lock on data A is granted.
- ✓ In T2 Exclusive lock on data B is granted
- ✓ In T1 Exclusive Lock on data B waits until T2 unlock Exclusive Lock on B
- ✓ Exclusive Lock in Data A is wait until T1 unlock the Exclusive lock on A

3) Cascading rollback problem

Time	T1	T2	T3
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Unlock(A)		
5		Lock-S(A)	
6		Read(A)	
7			Lock-S(A)
8			Read(A)

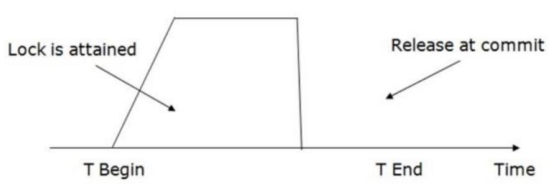
If the rollback of one transaction cause the rollback of other dependent transactions called cascading rollback.

- ✓ Here, T2 is dependent on T1,and T3 is dependent on T2
- ✓ If T1 failed,T1 must be rolled back ,similarly as T2 is dependent on T1,T2 also must be rolled back.
- ✓ Again as T3 is dependent on T2,T3 must also be rolled back.

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 rollbacks.
- ✓ The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- ✓ Strict-2PL protocol does not have shrinking phase of lock release.
- ✓ Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.



Rigorous two-phase locking:

- ✓ A transaction must **hold all locks** till commit/abort.
- ✓ Transactions can be serialized in the order in which they commit.

Note: *The difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.*

2. Graph based protocols

Graph Based Protocols are yet another way of implementing Lock Based Protocols.

As we know the prime problems with Lock Based Protocol have been avoiding Deadlocks and ensuring a Strict Schedule. We've seen that Strict Schedules are possible with following Strict or Rigorous 2-PL. We've even seen that Deadlocks can be avoided if we follow Conservative 2-PL but the problem with this protocol is it cannot be used practically. Graph Based Protocols are used as an alternative to 2-PL.

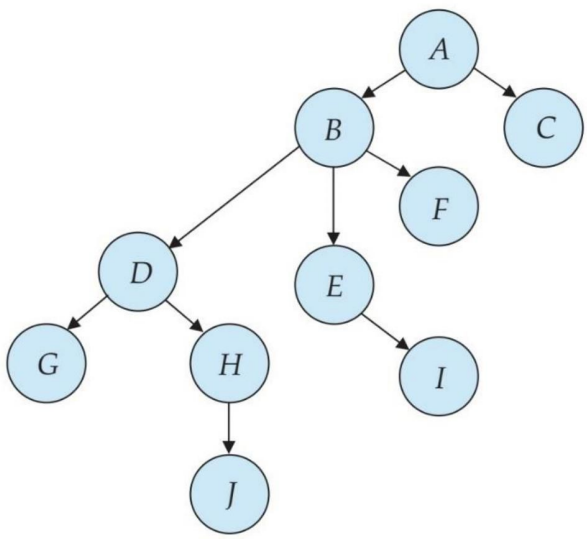
Implementation

Tree Based Protocols is a simple implementation of Graph Based Protocol. A prerequisite of this protocol is that we know the order to access a Database Item. For this we implement a Partial Ordering on a set of the Database Items (D) {d1, d2, d3, ….., dn} . The protocol following the implementation of Partial Ordering is stated as-

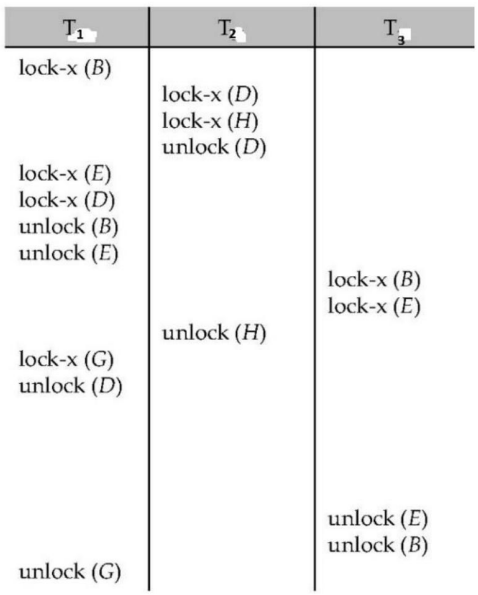
- ✓ If di –> dj then any transaction accessing both di and dj must access di before accessing dj.
- ✓ Implies that the set D may now be viewed as a directed acyclic graph (DAG), called a database graph.
- ✓ Only Exclusive Locks are allowed.
- ✓ The first lock by Ti may be on any data item. Subsequently, a data Q can be locked by Ti only if the parent of Q is currently locked by Ti.
- ✓ Data items can be unlocked at any time.
- ✓ A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

Following the Tree based Protocol ensures Conflict Serializability and Deadlock Free schedule. We need not wait for unlocking a data item as we did in 2-PL protocol, thus increasing the concurrency.

Now, let us take an example, following is a Database Graph which will be used as a reference for locking the items subsequently.



Taking an example based on the above Database Graph, we have three Transactions in this schedule and we will only see how Locking and Unlocking works.



T_1	T_2	T_3
lock-x (*B*)		
	lock-x (*D*)	
	lock-x (*H*)	
	unlock (*D*)	
lock-x (*E*)		
lock-x (*D*)		
unlock (*B*)		
unlock (*E*)		
		lock-x (*B*)
		lock-x (*E*)
	unlock (*H*)	
lock-x (*G*)		
unlock (*D*)		
		unlock (*E*)
		unlock (*B*)
unlock (*G*)		

Data items Locked and Unlocked are following the same rule as given above and follow the Database Graph.

Advantages

- ✓ Ensures Conflict Serializable Schedule.
- ✓ Ensures Deadlock Free Schedule
- ✓ Unlocking can be done anytime.

Disadvantages

- ✓ Unnecessary locking overheads may happen sometimes, like if we want both D and E, then at least we have to lock B to follow the protocol.
- ✓ Cascading Rollbacks is still a problem.
- ✓ We don't follow a rule of when Unlock operation may occur so this problem persists for this protocol.

SQL Standard Isolation Levels

Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.

A **transaction isolation level** is defined by the following phenomena:

- ✓ Dirty Read
- ✓ Non-Repeatable read
- ✓ Phantom Read

Based on these phenomena, **The SQL standard defines four isolation levels:**

Read Uncommitted: Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.

Read Committed: This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.

Repeatable Read: This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.

Serializable: This is the highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

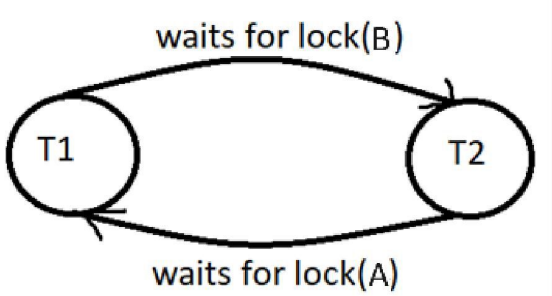
Deadlock

- ✓ A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ✓ None of the transactions can make progress in such a situation.
  Simple Example of deadlock is given below
  - ✓ Transaction T1 request resource A and received it.
  - ✓ Transaction T2 request resource B and received it.
  - ✓ Transaction T2 request resource A and is queued up waiting the release of A by transaction T1.
  - ✓ Transaction T1 request resource B and is queued up waiting the release of B by transaction T2.

Consider a schedule

T1	**T2**
Lock-X(A)	
Read(A)	
A:=A-50	
Write(A)	
	Lock-S(B)
	Read(B)
	Lock-S(A)
Lock-X(B)	

- ✓ Neither T_1 nor T_2 can make progress
- ✓ Executing **lock-S***(A)* causes T_2 to wait for T_1 to release its lock on *A*
- ✓ while executing lock-**X***(B)* causes T_1 to wait for T_2 to release its lock on *B*.
  Such a situation is called a **deadlock**.



Conditions for deadlock

We can say a deadlock is occurred if these four conditions occur simultaneously.

1. Mutual Exclusion: Only one transaction at a time can use a resource.
2. Hold and wait: A transaction holding at least one resource is waiting to acquire additional resources held by other transactions.
3. **No preemption**: A resource can be released only voluntarily by the transaction holding it after the transaction completed its task.
4. **Circular wait:** There exist a set {T1, T2,…….TN} of waiting process such that T1 is waiting for a resource that is held by T2, T2 is waiting for a resource held by T3 and finally TN is waiting for resource held by T1.

- ✓ There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention protocol** to ensure that the system will never enter a deadlock state.
- ✓ Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection and deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback.

Deadlock Prevention

Deadlock Prevention Protocols ensure that the system will never enter into a deadlock state.

Two approach to deadlock prevention

- ✓ First approach ensures that no cyclic waits can occur by ordering the requests for locks, requiring all locks to be acquired together.
- ✓ Second approach performs transaction rolled back instead of waiting for a lock, whenever the wait could be potentially result in a deadlock.

The first approach requires that each transaction locks all its data items before execution. Moreover either all are locked in one step or none are locked. There are two main disadvantages.

- ✓ It is often hard to predict, before the transaction begins, what data items need to be locked.
- ✓ Data items utilization may be very low, since many of the data items may be locked but unused for it a long time.

The second approach for preventing deadlock is use preemption and transaction rollback. Two different deadlock prevention schemes using timestamps have been proposed.

a) Wait –die scheme (non-preemptive)

- ✓ When transaction T_i requests a data item currently head by T_j . T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e T_i older than T_j) .Otherwise T_i rolled back(dies).
- ✓ For example, suppose that transactions T1,T2 and T3 have timestamp 20,30 and 40 respectively. If T_1 request data item held by T_2, then T_1 will be wait. If T3 requests a data item held by T2, then T3 will be rolled back.

b) Wound –wait scheme(preemptive)

- ✓ When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j(i.e T_i is younger than T_j) . Otherwise T_j rolled back.(T_j is wounded by T_i)
- ✓ For example, suppose that transactions T1 ,T2 and T3 have timestamp 20,30 and 40 respectively. If T1 request a data item held by T2, then data item will be preempted from T2 and T2 will be rolled back. If T3 requests a data item held by T2,then T3 will wait.

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

c) 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. Thus deadlock is not possible.
- ✓ Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

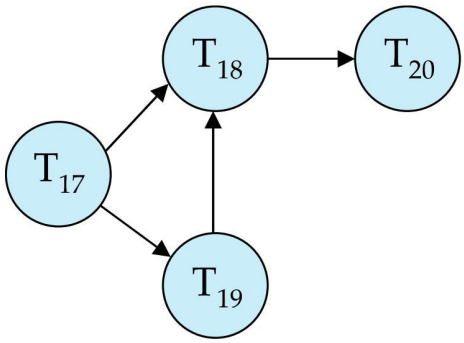
Deadlock can be described as a wait for graph, which consist of a pair G=(V,E)

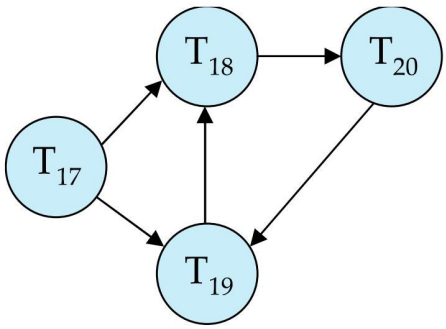
- V is a set of vertices (all the transactions in the system)
- E is a set of edges;each element is ordered pair $T_i \rightarrow T_j$.

If $T_i \rightarrow T_j$ is in E, then there is directed edge from T_i to T_j ,implying that T_i is waiting for T_j to release a data item.

When a T_i requests a data item concurrently being held by T_j , then 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





Wait for graph without a cycle **Wait for graph with a cycle**

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must recover from deadlock. The most common solution is to rollback one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of victim:** Given a set of deadlocked transactions ,we must determine which transaction to rollback to break the deadlock. We should roll back the transactions that will incur minimum cost. Many factors may determine the cost of rollback, including:
   a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b) How many data items the transaction has used.
   c) How many more data items the transactions needs for it to complete.
   d) How many transactions will involved in the rollback.

2. **Rollback:** Once we have decided that a particular solution transaction must be rolled back, we must determine how far this transaction should be rolled back.
   - ✓ Simplest solution is a total rollback. Abort the transaction and then restart it.
   - ✓ It is more effective solution is partial rollback. Rollback the transaction only as far as necessary to break the deadlock.
3. **Starvation:** In a system where the selections of victims is based primarily on cost factors, It may happen that the same transaction is always picked as a victim. As a result this transaction never completes its designated task, thus there is a starvation. We must ensure that a transaction can be picked as a victim only a small (finite number of times). The most common solution is to include the number of rollbacks in cost factor.