Transaction Processing and Concurrency Control

7.1 ACID Properties

A transaction is a program unit whose execution may change the contents of a database.

Example in Bankdb - 10000

R denote read operation

$$\left.\begin{array}{l}\text{read} <\!- R(a) \longleftarrow \text{Current Money}\\ \text{update} <\!- a\text{-}5000\\ \text{writing} <\!- W(a)\end{array}\right\}\ \text{Transaction}$$

Bankdb -> 5000

$$\begin{array}{c}\text{Database before}\\\text{transaction}\end{array} \xrightarrow[\text{transaction operation}]{\text{After}} \begin{array}{c}\text{Database after}\\\text{transaction(Consistent State)}\end{array}$$

or,

Transaction is used to represent a logical unit of database processing that must be completed entirely to ensure correctness.

ACID Properties of Transaction

1. Atomicity : It ensures that a transaction will run to completion as an indivisible unit at the end of which either no changes have occurred to the database or database has been changed in a consistent manner.

ALL or NONE

Example : Transfer Rs 500 From Account A to B

T : R (A , a)
  a - 500
  W (A , a)
  R (B , b)
  b = b + 500
  W (B , b)

Initially, A = 2000 (Suppose)
B = 3000 (Suppose)
Total = 5000
After Successful Completion
A = 2000 - 500 = 1500
B = 3000 + 500 = 3500
Total = 5000

If power failure occurs at W (A , a), Rs 500 will be deducted from Account A but won't be added to Account B. That is,

A = 1500

B = 3000

Total = 4500 != 5000. (Shows Inconsistency)

2. <u>Consistency</u> : It shows correctness of it ensures that if the database was in a consistent state before the start of a transaction, then on termination the database will also be in a consistent state.

Sum (A , B) = Sum (A , B)

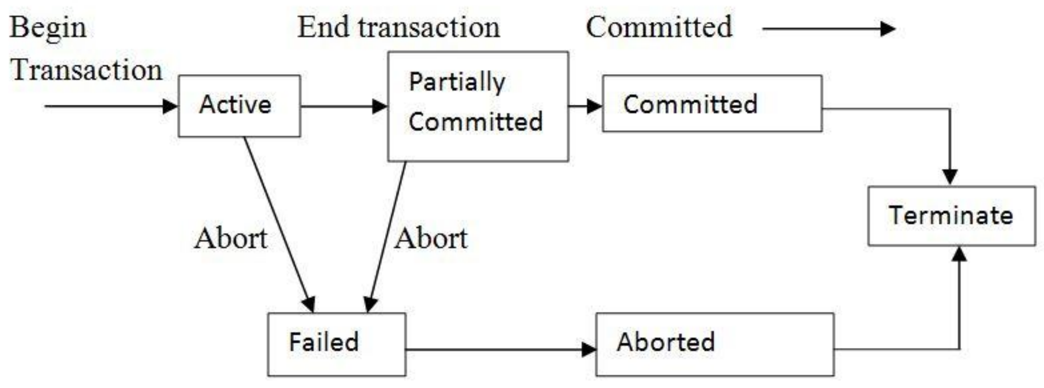
Before	After
Transaction	Transaction

3. <u>Isolation</u> : It indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until it terminates.

<u>T1</u>	<u>T2</u>
R (A)	
a - 500	
W (A)	
	R (A)
	R (B)
	Print (A , B)
R(B)	
B = B + 500	
W (B)	

4. <u>Durability</u> : All updates done by a transaction must become permanent. Or it ensures that the commit actions of a transaction on its termination will be reflected in the database.

<u>Transaction State Diagram</u>



Active : In this state, the transaction is being executed. This is the initial state of every transaction.

Partially Committed : When a transaction executes its final operation, it is said to be in a partially committed state.

Failed : A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further. **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −

- Re-start the transaction
- Kill the transaction

Committed : If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

7.2 Concurrent Execution

It implies interleaving execution of operations of a transaction. Multiple transactions are allowed to run concurrently in the system.

Benefits

- Helps in reducing waiting time
- Improved throughput and resource utilization

Schedule

It represents the order in which instructions of a transactions are executed.

T_1 --> T_2 --> T_3

Schedule

Problems with concurrent execution

1. Lost update problem (W - W conflict)

It occurs when two transactions that accesses the same database items have their operations interleaved in a way that makes the value of the database item incorrect.

Example :Let the value of A is 1000

T_1	T_2
R(A)	
A = A - 50	
	R(A)
	A = A + 100
W(A)	
	W(A)

2. Temporary Update (Dirty Read) Problem [W-R conflict]

It occurs when one transaction updates a database item and then the transaction fails, but its update is read by some other transaction.

Let A = 100

T_1	T_2
R(A)	
A = A + 50	
W(A)	
	R(A)
	A=A+10
	W(A)
R(B)	
Power Failure	Commit

3. Unrepeatable Read (W-R Conflict)

If a transaction "T_i" reads an item value twice and the item is changed by another transaction "T_j" in between the two Read operation. Hence, "T_i" receives different values for its two read operations of the same item. Example : Let A=1000

T_1	T_2
R(A)	
	R(A)
	A=A+2000
	W(A)
R(A)	

4. Incorrect summary problem

If one transaction is calculating an aggregate summary function on a number of records, while other transaction is updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated results in incorrect summary.

T_1	T_2
	Sum=0
	R(A)
	Sum=Sum+A
	R(Y)
	Sum=Sum+Y
R(Y)	
Y=Y+100	
W(Y)	

<u>Schedule</u>

A schedule 'S' of n transactions T_1, T_2, ...T_n is an ordering of operation of the transactions in chronological order.

T_i (X -> Y operation)

In any schedule T_{i_x} -> T_{j_y}

T_1	T_2
R(X)	
	W(X)

T_1 is followed by T_2

When several transactions are executing concurrently then the order of execution of various instruction is known as Schedule.

Types of Schedule

<u>1. Serial Schedule</u>

It does not interleave the actions of any operation of different transactions . It always ensure a consistent state.

Let we have T_1, T_2, T_3

$$\left.\begin{array}{l} T_1 \to T_2 \to T_3 \\ T_1 \to T_3 \to T_2 \end{array}\right\}$$ Serially executed and transaction of one is not interleaved with the operation of another transaction.

Example : Let A = 100

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)

Here T_1 is followed by T_2 (Serial Schedule)

Again, When T_1 is followed by T_2

T_1	T_2
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A+100
	W(A)

In T_1 After writing A, value of A is 150. Then T_2 reads A and operation A+100 is equals 250. That final value of A is 250.

Again When T_2 is followed by T_1

T_1	T_2
	R(A)
	A=A+100
	W(A)
R(A)	
A=A+50	
W(A)	

In this case, for T_2, initially A=100.

Then A=A+100=100+100=200

Then T_1 reads value of A i.e., A=200

Then A=A+50=200+50=250

Hence We get consistency in serial schedule whether T_1 is followed by T_2 or T_2 is followed by T_1.

Now, if schedule is not serial, then

T_1	T_2
R(A)	
A=A+50	
	R(A)
	A=A+100
W(A)->150	
	W(A)->200

Inconsistent state.

2. Complete Schedule

If the last operation of each transaction is either abort or commit.

T_1	T_2
R(A)	R(A)
W(A)	
Commit	
	W(A)
	Abort

Complete Schedule.

3. Recoverable Schedule

It is one where for each pair of transaction (T_i , T_j), such that T_j reads a data item that was previously written by T_i, then the commit operation of T_i should appear before commit operation of T_j.

T_1	T_2
R(A)	R(A)
A+50	
W(A)	
	R(A)
	A+100
	W(A)
R(B)	
W(B)	
Commit	
	Commit

If failure occurs after R(B), then it rollback to initial state of transaction operation.

Here commit operation is after T_j write operation, so it can lead to inconsistent state of database.

4. Cascadeless Schedule

It is one where for each pair of transaction (T_i , T_j) such that T_j reads a data item written by T_i, then the commit operation of T_i should appear before the read operation of T_j.

Roll back

T_1	T_2	T_3
R(A)		
W(A)		
	R(A)	
	W(A)	
Commit		R(A)
	Commit	W(A)
		Commit

This is called cascading rollback.

To overcome this problem (cascading rollback), commit operation should be done after write operation of each transaction. That is,

T_1	T_2	T_3
R(A)		
W(A)		
Commit		
	R(A)	
	W(A)	
	Commit	
		R(A)
		W(A)
		Commit

5. Strict Schedule

If a value written by a transaction can 't be read or overwritten by another transaction until the transaction is either aborted or committed.

[Note : Every strict schedule is both cascadeless and recoverable.]

T_1	T_2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)

Conflict Operation

Operations are said to be conflicting if

- belongs to different transactions
- Access to same db item say "A"
- At least one of them is a write operation[RW, WR, WW]

T_1	T_2
R(A)	
	R(A)
W(A)	

RR - Not conflict

$R_2(A)$ is followed by $W_1(A)$ is conflict operation. Here 1 denotes for transaction one and 2 denotes for transaction two.

Equivalent Schedule

Two schedules S_1 and S_2 are said to be equivalent schedule if they produce the same final database state. Let A = 100,

S_1	S_2
100 <- R(A)	100 <- R(A)
110 <- A+10	110 <- A*1.1
110 <- W(A)	110 <- W(A)

Result equivalent schedule : Produce final db state for some initial values of data.

Conflict Equivalent : Two schedules are said to be conflict equivalent if all conflicting operations in both schedules must be executed in same order.

Example 1 : Check for conflict equivalent

$S_1 : R_1(A), R_2(B), W_1(A), W_2(B)$

$S_2 : R_2(B), R_1(A), W_2(B), W_1(A)$

S_1

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)

S_2

T_1	T_2
	R(B)
R(A)	
	W(B)
W(A)	

S_1 is conflict equivalent to S_2

Example 2 : S_1 : $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$

S_2 : $R_1(A)$, $W_1(A)$, $R_1(B)$, $R_2(B)$, $W_2(B)$

S_1

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	

S_2

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(B)
	W(B)

S_1 is not conflict equivalent to S_2

Serializability

When multiple transactions in any schedule are running concurrently then there is a possibility of database inconsistency. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state. A non serial schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

There are two types of serializability

1. **Conflict Serializability**

   It is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

   If any non-serial schedule is conflict equivalent to serial schedule, then it is called as conflict serializable schedule.

S_1

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

S_2

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

For S_1
$R_1(A)$ is followed by $W_2(A)$
$W_1(A)$ is followed by $R_2(A)$
$W_1(A)$ is followed by $W_2(A)$
$R_1(B)$ is followed by $W_2(B)$
$W_1(B)$ is followed by $R_2(B)$
$W_1(B)$ is followed by $W_2(B)$

For S_2
$R_1(A)$ is followed by $W_2(A)$
$W_1(A)$ is followed by $R_2(A)$
$W_1(A)$ is followed by $W_2(A)$
$R_1(B)$ is followed by $W_2(B)$
$W_1(B)$ is followed by $R_2(B)$
$W_1(B)$ is followed by $W_2(B)$

Hence **S_1** is **conflict serializable schedule**

<u>Test for conflict serializability</u>

To check conflict serializability, precedence graph is used.

Let 'S' be a schedule, construct a directed graph known as precedence graph. Graph consists of a pair of G = (V , E)

Where,

V : a set of vertices

E : a set of edges

Algorithm for creating a precedence graph

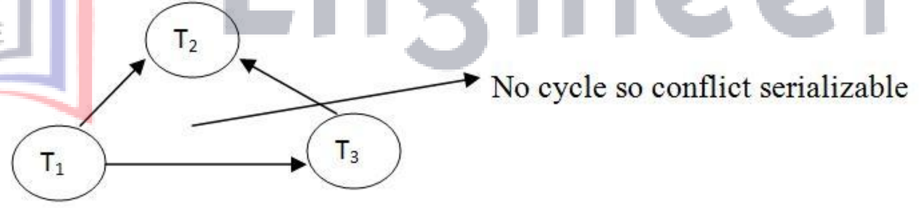
Step 1: Create a node for each transaction

Step 2 : Draw a directed edge from T_i to T_j , if T_j reads a value of an item written by T_i.

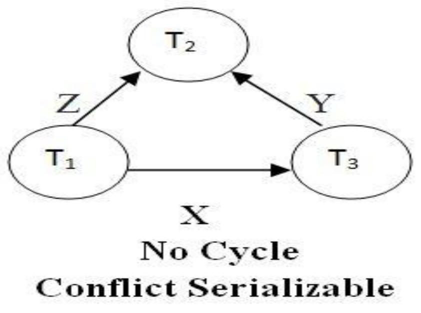
Step 3 : Draw a directed edge from T_i to T_j , if T_j writes a value into an item after it has been read by T_i.

Step 4 : Draw a directed edge from T_i to T_j , if T_j writes after T_i write.

A schedule is conflict serializable iff its precedence graph is acyclic. That is,

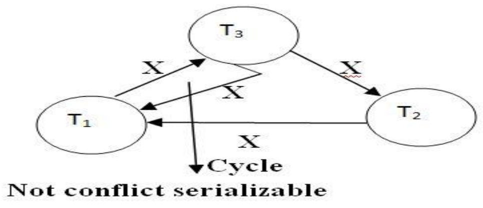


Example 1 : Check for conflict serializability



T_1	T_2	T_3
R(X)		
	R(Z)	
R(Z)		
		R(X)
		R(Y)
		W(X)
	R(Y)	
	W(Z)	
	W(Y)	

Example 2 : Check for conflict serializability using precedence graph



T_1	T_2	T_3
R(X)		
		R(X)
		W(X)
W(X)		
	R(X)	

2. **<u>View Serializability</u>**

View Serializability is a process to find out that a given schedule is view serializable or not. Two schedule S and S' are view equivalent if the following conditions are met :

Algorithm

Step 1 : For each data item Q, if T_i reads an item value of Q in schedule S, then T_i in S' must also reads an initial value of Q.

Step 2 : If T_i executes read Q in S and that value was produced by T_j (if any), then T_i in S' also read the value of Q that was produced by T_j.

Step 3 : For each data item Q, the transaction that performs the final write (Q) operation in schedule S must also perform the final write (Q) in schedule S'.

(A schedule is new serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable but not vice-versa.)

Example : Check for view serializability

S_1

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

S_2

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

- For S_1, T_1 reads data item A whereas for S_2, same T_1 reads data item A
- T_2 reads data item A after T_1 writes A in S_1, and in S_2 also T_2 reads data item A after T_1 writes A.
- T_2 write B finally in S_1, and in S_2 also T writes B finally.

Hence S_1 is view equivalent to S_2.

Concurrency Control Technique(Lock Based Protocol)

It is the process of managing simultaneous execution of transactions in a shared database to ensure the serializability of transaction.

Purpose of concurrency control techniques :

- To enforce isolation
- To preserve database consistency

Technique

Lock Based Protocol : A lock guarantees exclusive use of a data item to current transactions. Lock is used - to access data item (Lock Acquire)
- after completion of transaction (Release Lock)

All data items must be accessed in a mutually exclusive manner.

Types of Lock

1. Shared Lock
    Denoted by Lock S
    Used to read data item value only
2. Exclusive Lock
    Denoted by Lock X
    Used to both read and write

Compatibility between lock models

	S	X
S	True	X_{false}
X	X_{False}	X_{False}

Whenever exclusive lock is involved, it can't be accessible with another transactions.

Example of lock protocols, we have data item B

T_1	T_2
Lock-X(B)	
R(B)	
B-50	
W(B)	
Unlock (B)	
	Lock-S(B)
	R(B)
	Unlock (B)

Exclusive Lock

Shared Lock

[note : Any number of transactions can hold shared lock on an item but exclusive lock can be hold by only one transaction at a time.]

Conversion of Lock

1. Upgrading - Shared Lock to Exclusive Lock (Read Lock to Write Lock)
2. Downgrading - Exclusive Lock to Shared Lock (Write Lock to Read Lock)

Two Phase Locking (2PL)

It requires both Lock and Unlock being done in 2 phases. Whenever we use as for example 4 data items, these 4 items must be locked first at same time and then unlocked in next time.

Phases

- Growing phase
  New Locks on items can be acquired. When transaction acquire final item, phase reaches to point called Lock point from where shrinking phase starts.
- Shrinking phase
  Existing locks are released but no new lock can be acquired. After lock point shrinking starts onwards.

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

Let's see a transaction implementing 2-PL.

	T_1	**T_2**
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)**
10	**…….**	**……**

This is just a skeleton transaction which shows how unlocking and locking works with 2-PL.

Transaction T_1:

- Growing Phase is from steps 1-3.
- Shrinking Phase is from steps 5-7.
- Lock Point at 3

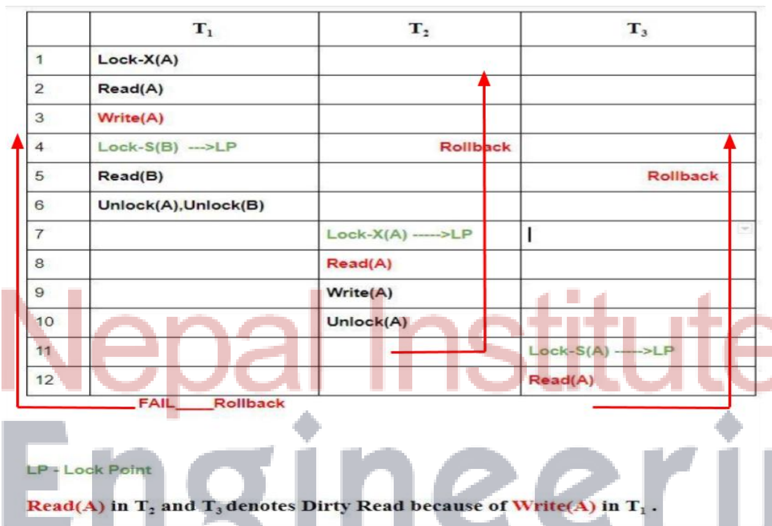
Transaction T_2:

- Growing Phase is from steps 2-6.
- Shrinking Phase is from steps 8-9.
- Lock Point at 6

What is **LOCK POINT ?**The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand.
I have said that 2-PL ensures serializablity, but there are still some drawbacks of 2-PL. Let's glance at the drawbacks:

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation is possible.

Cascading Rollbacks in 2-PL
Let's see the following Schedule:



	T_1	T_2	T_3
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Lock-S(B) --->LP	Rollback	
5	Read(B)		Rollback
6	Unlock(A),Unlock(B)		
7		Lock-X(A) ------>LP	
8		Read(A)	
9		Write(A)	
10		Unlock(A)	
11			Lock-S(A) ------>LP
12			Read(A)

FAIL___Rollback

LP - Lock Point

Read(A) in T_2 and T_3 denotes Dirty Read because of Write(A) in T_1 .

Take a moment to analyze the schedule. Yes, you're correct, because of Dirty Read in T_2 and T_3 in lines 8 and 12 respectively, when T_1 failed we have to rollback others also. Hence **Cascading Rollbacks are possible in 2-PL.** I have taken skeleton schedules as examples because it's easy to understand when it's kept simple. When explained with real time transaction problems with many variables, it becomes very complex.

Deadlock in 2-PL
Consider this simple example, it will be easy to understand. Say we have two transactions T_1 and T_2.

Schedule: Lock-X_1(A) Lock-X_2(B) Lock-X_1(B) Lock-X_2(A)

Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL. Two phase locking may also limit the amount of concurrency that occur in a schedule because a Transaction may not be able to release an item after it has used it. This may be because of the protocols and other restrictions we may put on the schedule to ensure serializablity, deadlock freedom and other factors. This is the price we have to pay to ensure serializablity and other factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

The above mentioned type of 2-PL is called **Basic 2PL**. To sum it up it ensures Conflict Serializability but ***does not*** prevent Cascading Rollback and Deadlock.
2-PL protocol enforces serializability but may reduce concurrency due to following reasons:

- Holding lock un-necessarily
- Locking too early
- Penalty to other transaction

Variations of 2-PL

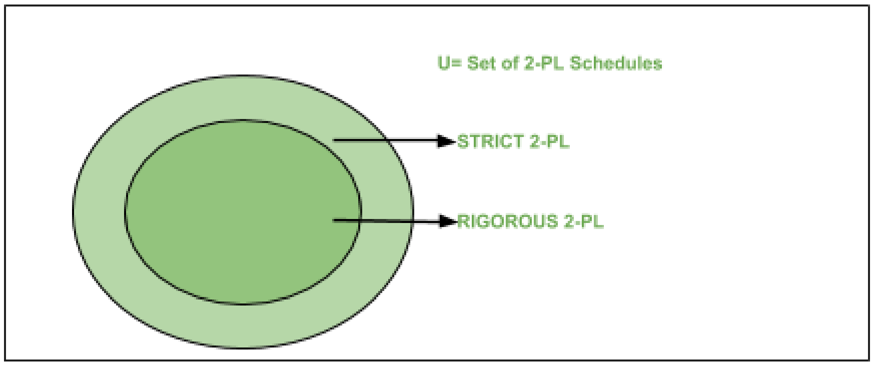
- **Strict - 2PL**
  This requires that in addition to the lock being 2-Phase **all Exclusive(X) Locks** held by the transaction be released until after the transaction commits.
  Following Strict 2-PL ensures that our schedule is:
  - Recoverable
  - Cascadeless

  Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!
- **Rigorous - 2PL**
  This requires that in addition to the lock being 2-Phase **all Exclusive(X) and Shared(S) Locks** held by the transaction be released until after the transaction commits.
  Following Rigorous 2-PL ensures that our schedule is:
  - Recoverable
  - Cascadeless

  Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still Deadlocks are possible!
  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 more easy.

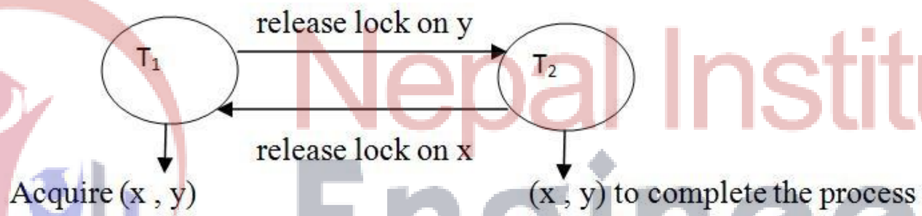
- **Conservative - 2PL**
  Static 2-PL, this protocol requires the transaction to lock all the items it access before the Transaction begins execution by pre-declaring its read-set and write-set. If any of the pre-declared items needed cannot be locked, the transaction does not lock any of the items, instead it waits until all the items are available for locking.
  Conservative 2-PL is *Deadlock free* and but it does not ensure Strict schedule(More about this here!). However, it is difficult to use in practice because of need to pre-declare the read-set and the write-set which is not possible in many situations. In practice, the most popular variation of 2-PL is Strict 2-PL.
  The Venn Diagram below shows the classification of schedules which are rigorous and strict. The universe represents the schedules which can be serialized as 2-PL. Now as the diagram suggests, and it can also be logically concluded, if a schedule is Rigorous then it is Strict. We can also think in another way, say we put a restriction on a schedule which

makes it strict, adding another to the list of restrictions make it Rigorous. Take a moment to again analyze the diagram and you'll definitely get it.



Deadlock Handling and Prevention

A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.



Deadlock detection

Simple way to detect a state of deadlock is to draw wait-for graph.

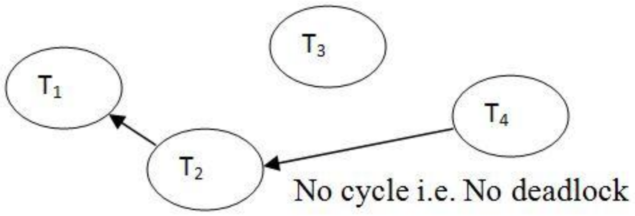
G(V , E)

Where, V is nodes describing transactions and E is directed edges

When T_i is waiting for a data item held by T_j.

Transactions	Data item	Lock Mode
T	Q	Shared
T	P,Q	Exclusive , Exclusive
T	Q	Shared
T	P,Q	Exclusive , Exclusive

Wait for graph is :



Deadlock Prevention Techniques

Deadlock Prevention : This protocol ensures that the system will never enter into a deadlock state. To prevent deadlock, following process should not occur.

1. Mutual Exclusion : Transaction can access same data item at given time which should not happen to prevent deadlock.
2. Hold & Wait
3. No Pre-emption : One transaction is running and another transaction require lock from that running transaction, which should not happen.
4. Circular Wait

A) Use of timestamps

Assuming that T_i requests a data item currently held by T_j.

Wait - Die Scheme : If $T_s(T_i) < T_s(T_j)$ i.e. T_i is smaller than T_j or older than T_j. Then T_i is allowed to wait otherwise if T_i is greater (Younger) than T_j then T_i is aborted (rolled back) & restart it later with same timestamp value.

Wound-Wait Scheme : If $T_s(T_i) < T_s(T_j)$ i.e. T_i is smaller than T_j or older than T_j. Then abort T_j (T_i wounds T_j) and restarts it with same timestamps. And if $T_s(T_i) > T_s(T_j)$, then T_i is allowed to wait.

B) Time out based scheme

Based on lock-timeouts, a transaction that has requested a lock waits for at most a specified amount of time. If the lock is not granted within that time, transaction is said to time out and it rolls itself back and restarts.

Example : If T_1 can wait for data item x for only 10 seconds. And within this time, if T_1 don't get lock on x, then T_1 will rollback and restarts.

Starvation

A transaction is starved if it can't proceed for an indefinite period of time while other transactions in the system continue normally.

Other reasons for starvation - Algorithm dealing with deadlock

Prevention select same transaction as victim repeatedly thus causing it abort and never finish execution.

Example : If we have T_1 ..... T_5, where T_1 is starved that is T_1 is victim, then remove starvation :

- Modify the timestamp of T_1
- Increase the priority of T_1

Deadlock recovery

1. Selection of victim transaction

Let's T_1 and T_2 are deadlocked

Here T_1 is selected as victim then it will roll back or abort and T_2 will finish its operation on data item.

Here victim should be of minimum cost. We choose victim by choosing

- a) Length of transaction
- b) Choose that transaction which uses less number of data item
- c) Choose transaction that need more data items to be locked.
- d) Choose transaction which cause minimum number of roll back of other transactions.

2. Rollback

- a) Full Roll back : Here transaction gets rolled back to its starting point.
- b) Partial Roll back : Here transaction gets rolled back to save point or lock point

3. Starvation

Select the transaction as victim carefully so that victim don't get starved again.