Transaction – An action or series of actions that are being performed by a single user or application program which reads or updates the contents of the database.

ex1

Let consider transaction of only one SQL statement. Even one SQL statement consist of one or more operations.

For example, the operations are –

(i) R(A); – read value of A and store in buffer.

(ii) A := A - 1000;

(iii) W(A) – write the value from buffer to database.

ex2

Example of two SQL statement.

Transaction Ti	
Read (A); A := A-100; Write(A);	→ 1st SQL statement
Read (B); B := B+100; Write (B);	→ 2nd SQL statement

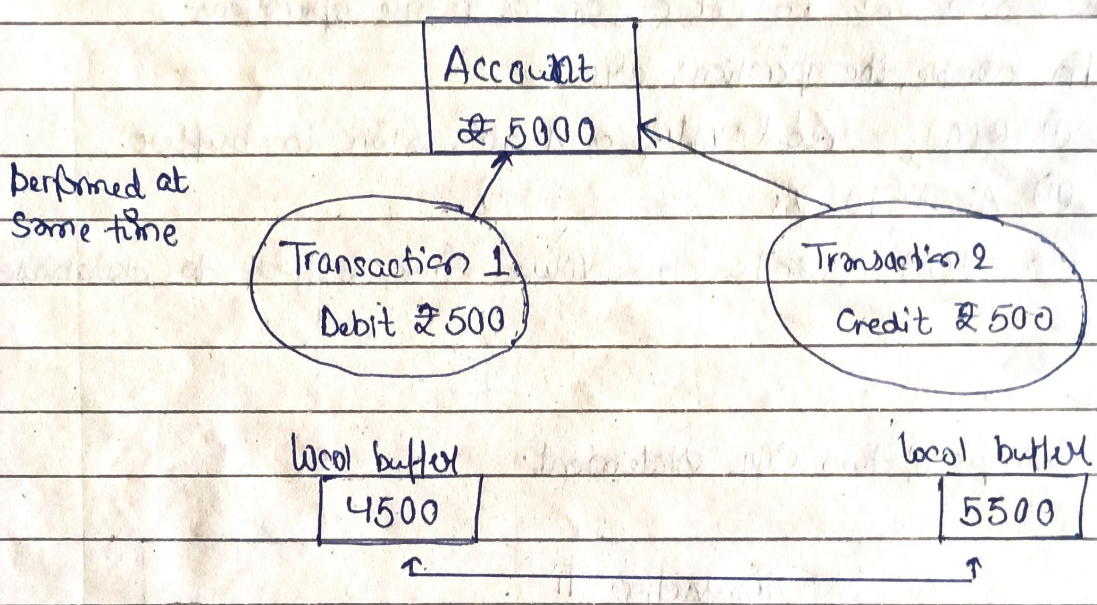
Properties of Transaction (ACID Property) [need to follow otherwise our DB system fails.]

1. **Atomicity** – The "all or nothing" property.
   A transaction is an indivisible entity that is either performed entirely or will not get performed at all.
   (Handled by Transaction Management Component)

2. Consistency – Transaction must transform the database from one consistent state to another consistent state.

(Managed by application programmer who codes the transaction)

ex



Out of these who ever gets updated last will become the final value which is wrong because actual value should be ₹5000.

So this is inconsistant state.

3. Isolation

While the execution of one transaction, it should not be interfered by another transaction. If multiple transactions are running simultaneously, they must not be affected by one another.

(Managed by concurrency control component)

4. Durability

The effects of an accomplished transaction are permanently recorded in the database and must not get lost or vanished due to subsequent failure.

★ i.e. changes made by committed transaction (→ commit) is permanent in the system and can not be undone.

(Managed by Recovery Management Component)

★ Once you commit you can't rollback, before that you can.

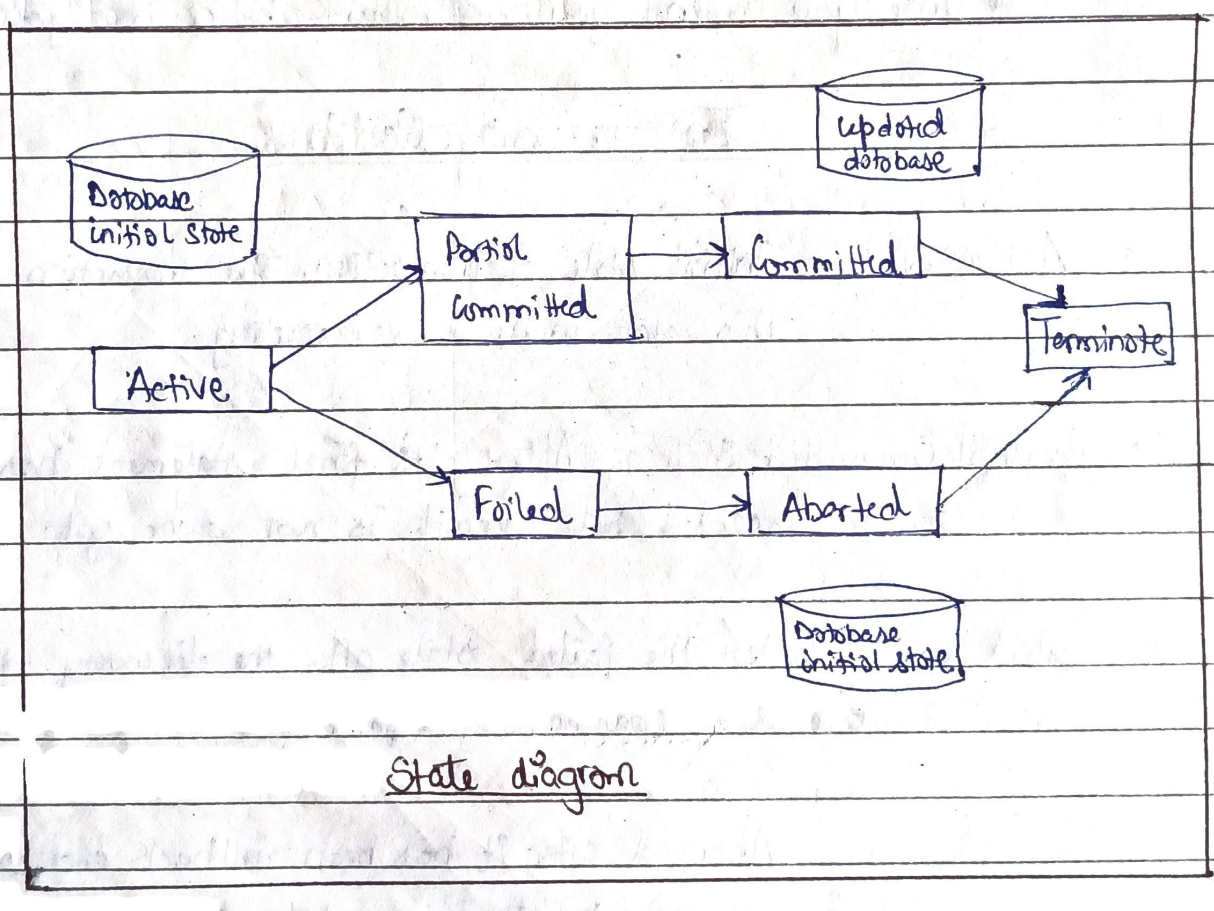
Transaction States

- **Active State** – Initial state of transaction. The transaction stays in this state while it is executing.
- **Partial Committed State** – After it's final statement has been executed but commit is not done yet.
- **Failed state** – Enter the failed state after the discovery that normal execution can no longer proceed.
- **Aborted state** – Aborted after it has been rollback and the database is restored to its prior state before the transaction.
- **Committed state** – Occur after successful completion of the transaction (when you perform commit operation)
- **Terminate state** – Transaction is either committed or aborted.

☆ When transaction enters the aborted state, the system has 2 options –

(I) Restart the transaction – If aborted because of hardware failure or some software error.

(II) Kill the transaction – If the application program that initiated the transaction has some logical error.



State diagram

☆ Scheduling in Transaction

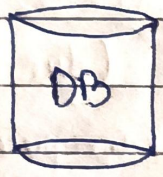
Process of arranging the transactions in such a way that each transaction will get executed without violating ACID property.

For ex

Let T1 and T2 both are trying to access DB at the same time.

T1 ⇒

T2 ⇒



T1
Read (A);
A := A+5;
Write(A);
Read (B)
B := B+100;
write (B);

T2
Read (A)
Temp := A/2
A := A+Temp;
write (A);

Types of Scheduling

1. Serial Scheduling – Transaction are going to be executed one after one.

★ If at a time 'n' process are trying to access DB n! type of serial scheduling possible.

Way of representing

	T1	T2	T3	.... => T1 then T2 then T3
1st	---			...
2nd		---		
⋮			---	

Advantages

(i) Easy to Implement. (just put them in a queue)

(ii) Always ACID property follows.

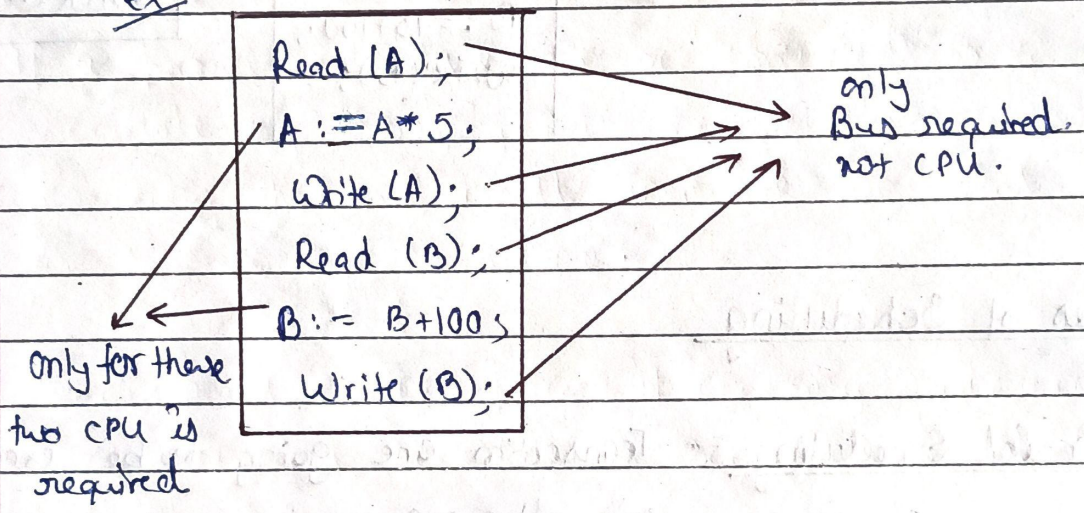
Disadvantage

(i) Through put will be less. (due to reason 2)

→ Throughput – Amount of work completed in a unit of time.

(ii) No optimum use of CPU.

ex



2. **Parallel or Concurrent Scheduling** – Transactions are allowed to execute simultaneously without violating ACID property.

☆ Total no. of parallel scheduling possible

= Total no. of scheduling – Serial Scheduling

$$= \frac{a_1! + a_2! + a_3! + a_4! \cdots + a_n!}{a_1! \times a_2! \times a_3! \cdots a_n!} - n!$$

Where $a_1, a_2 \cdots a_n$ are the no. of operations present in transaction $t_1, t_2 \cdots t_n$.

We need to represent in this way:

Ex

T1	T2
Read (A);	
A := A * 5;	
Write (A);	
	Read (A);
	Temp := A/2;
Read (B);	
B := B + 100;	
write (B);	
	A := A + Temp;
	write (A);

Advantages

1) Maximum CPU utilization
2) Max^m Through Put.

Disadvantages

1) Need high programming skill and proper algorithm so that the result of one transaction doesn't affect another transaction's result.

☆ Serializability

- There is an algorithm, which basically check if a parallel Scheduling can be converting into Serial schedule without hampering the end result of any transaction.
- Whether a system remains in a consistent state after converting a concurrent schedule to a serial schedule.
- If a parallel schedule can be converted to a serial schedule by swapping bet^n the operations without hampering the end result then the concurrent schedule is said to be Serializable.
- Serializability – The process of checking whether a concurrent schedule is serializable or not using an algorithm.

Ways to check the Serializability

1. Conflict Serializability
2. View Serializability.

(I) **Conflict Serializability** – If it can be transformed to a serial schedule by swapping non-conflicting operations.

Conflicting operations – When they follow following conditions –:

1. They belongs to diff^t transactions.
2. They operate on same data item.

3. At least one of them is a write operation.

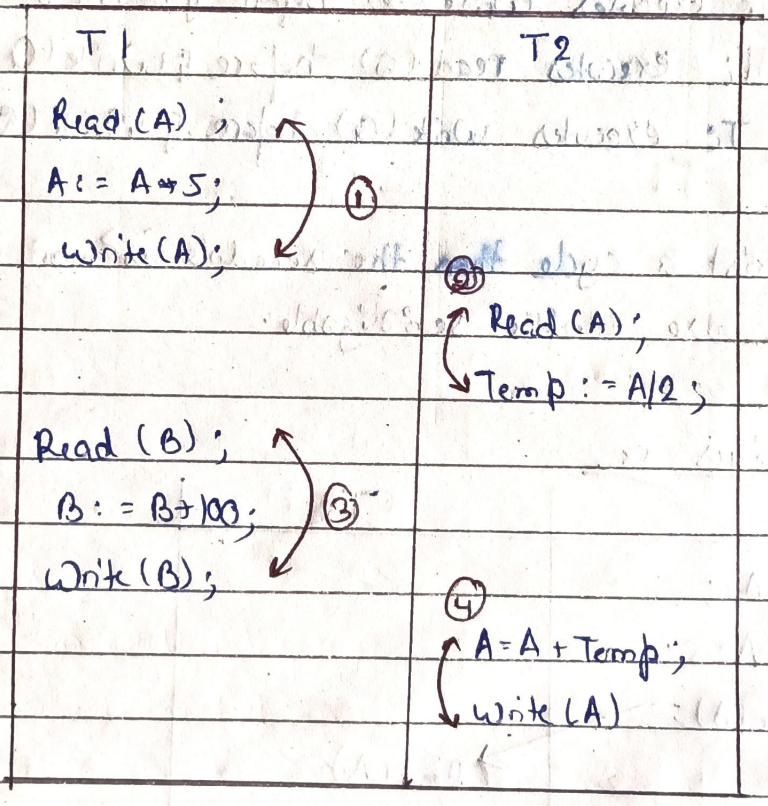
ex

R1(A), W2(A) ✓ Conflicting

R1(A), R2(A) ✗ Non-conflicting.

R1(A), W2(B) ✗ Non-conflicting.

ex



① and ② = Conflicting

② and ③ = Non-conflicting.

So by swaping section ② and ③ we can able to complete T1 first then T2, so we achieve serial transaction

∴ hence serializable.

Algorithm

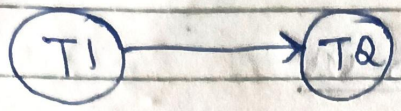
Create a directed graph from S.
Consist of pair $G = (V, E)$ i.e. (Vertices, Edges)
Vertices will represent - transactions
Edge only for conflicting operation pairs
i.e. $T_i \longrightarrow T_j$ exist if

- T_i executes write(Q) before T_j read(Q) or
- T_i executes read(Q) before T_j write(Q) or
- T_i executes write(Q) before T_j write(Q)

If there exist a cycle then the schedule is non conflict Serializable else conflict serializable.

Taking previous ex.

T1	T2
Read(A);	
A := A+5;	
Write(A); →	
	Read(A);
	Temp := A/2;
Read(B);	
B := B+100;	
Write(B);	
	A := A + Temp;
	Write(A);



no cycle

∴ conflict serializable.

(2) View Serialization

(S - Concurrent schedule and S1 is serial schedule)

Two schedules S and S1 are said to be view equivalent if the following three conditions :-

- If T1 Read (Q) in schedule S, then it must also read (Q) in S1. (initial read)
- The transaction that performs the final Write (Q) in S must also perform the final Write (Q) in S1. (final Write)
- Let Ti use a data produced by Tj, so in serial scheduling Ti must use the data produced by Tj only.

ex

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

if we exchange this then serial scheduling will be T1, T2 then T3. And in that the initial read is done by T1, final write is done by T3.

But it is not conflict serializable because of Blind write (writing without reading).

Concurrency problems in DBMS transactions

1. Temporary Update Problem

or Dirty read Problem.

→ Occur when one transaction updates an item and fails. But that updated item is used by another transaction before the item is changed or reverted back to its last value.

ex

T1	T2
read (X)	
X = X - N	
write (X)	
	read (X)
	X = X + M
	write (X)

→ When transaction read un-committed value, this problem may arise.

let T1 fails after this but as the value of X rollback but T2 has used the previous value of X.

2. Incorrect Summary Problem

Consider a situation where one transaction is applying the aggregate fn on some records while another transaction is updating there records. The aggregate fn may calculate value before updation and for some after updation.

ex

let T1 & T2

T1 is calculating avg marks and T2 is updating marks of student.

3. Lost Update problem

Update done on one data item by a transaction is lost as it is overwritten by the update done by another transaction.

ex

T1	T2
Read (A);	
A := A - 100;	
	Read (A);
	A := A + 200;
	Write (A);
Write (A);	

This value lost.

4. Unrepeatable Read problem

Happen when two or more read operations of the same transaction read diff values of the same variable.

ex

X = 5

T1	T2
Read (X);	
	Read (X); → 5
X := X + 100;	
write (X);	
	Read (X); → 105

5. Phantom read problem

Occur when a transaction reads a variable once but when it tries to read that same variable again, an error occurr saying that the variable does not exist.

ex

T1	T2
Read (X)	
	Read (X)
Delete (X)	
	Read (X) // error

Lock Based Concurrency Control Protocol

Allow concurrent schedules but ensure that the schedules are conflict/ view serializable and are recoverable and maybe even cascadeless.

Diff^n categories of protocol –

- **Lock Based Protocol**

  A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it. They synchronize the access by concurrent transactions to the database items.

  → Before performing operation on any data item we need to lock that data item.

Types of lock

1) **Shared lock** – Read only lock. (S-lock)

   In this lock transaction do not have permission for updating the value. It is requested using lock-S instruction.

2) **Exclusive Lock (X-lock)** – Data item can be both read and write. This is exclusive and can't be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Lock Compatibility Matrix

	T1(S)	T2(X)
T2(S)	✓	X
T2(X)	X	X

→ Each transaction need to request for lock and if that lock is compatible then it will be given.

→ Any no. of transaction can hold shared locks on an item.

(1) → If any transaction holds an exclusive (X) on the item no other transaction may hold any lock on the item.

→ If any lock can't be granted then it made to wait till all incompatible locks held by other transactions have been released.

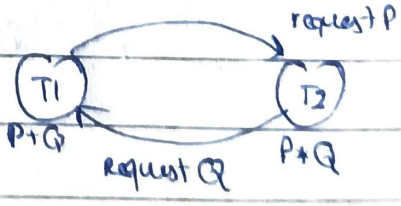
Changing lock

Upgrade locks – S(A) is upgraded to X(A) if T1 is the only transaction holding the S-lock on element A.

downgrade locks – X(A) is downgraded to S(A) anytime, you don't need to check any conditions because of reason (1).

- Disadvantage of locks – Starvation

Deadlock ex



Deadlock ex -

T1	T2
lock-X(B);	
read(B);	
B := B - 50;	
write(B);	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A);	
...	...

Starvation ex

If T1 is reading X and T2 is waiting for T1 to do it's job so that it can have write lock but in bet= new new read(x) transaction come, so T2 is still waiting and hence face starvation.

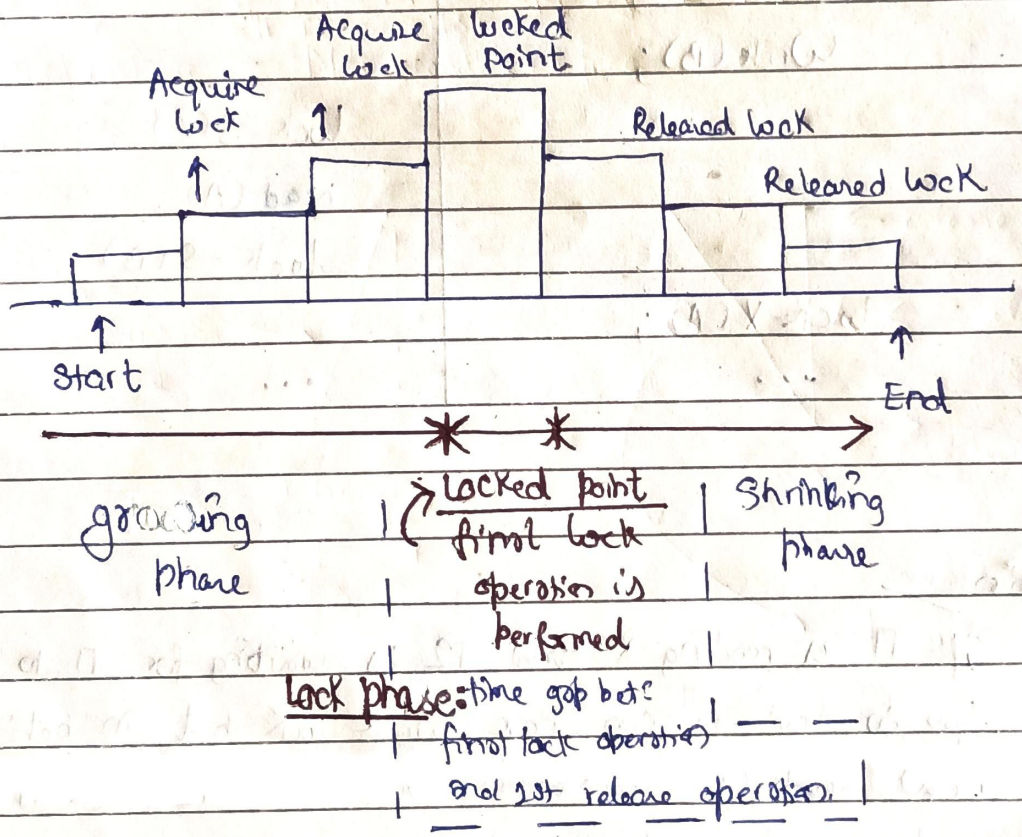
So there are diff= ways to use lock so that the above problem should be removed -

1) 2 Phase Locking Protocol

A transaction is said to follow Two phase locking protocol if locking & unlocking can be done in two phases -

i) Growing phase – New locks on data item may be acquired but none can be released.

ii) Shrinking phase – Existing lock may be released but no new locks can be acquired.



Ex

T1	T2
lock S(A)	
lock X(B)	lock S(A)
...	...
Unlock(A)	lock S(C)
Unlock(B)	...
	Unlock(A), Unlock(C);

growing phase for T1
lock Point
shrinking phase for T1
growing phase for T2
lock point
shrinking phase for T2.

★ Types of 2-Phase locking protocol

1) Basic 2PL

If we put lock in this phase style we are going to face two problems -

a) Deadlock & starvation is still possible.

b) Cascading Rollback is possible

→ If rollback of one transaction leads to series of rollback including operations of some other transaction then it is called Cascading rollback.

ex

T1	T2
Lock X(A)	
Read (A)	
Write (A)	
Read (B)	
Unlock (A)	
Unlock (B)	
	Lock X(A)
	Read (A)
if now T1 fails it	Write (A)
should be roll back	Unlock(A) T2 too!

Q Why 2 phase locking system is considered as a bargain bet : concurrency and maintaining the ACID properties?

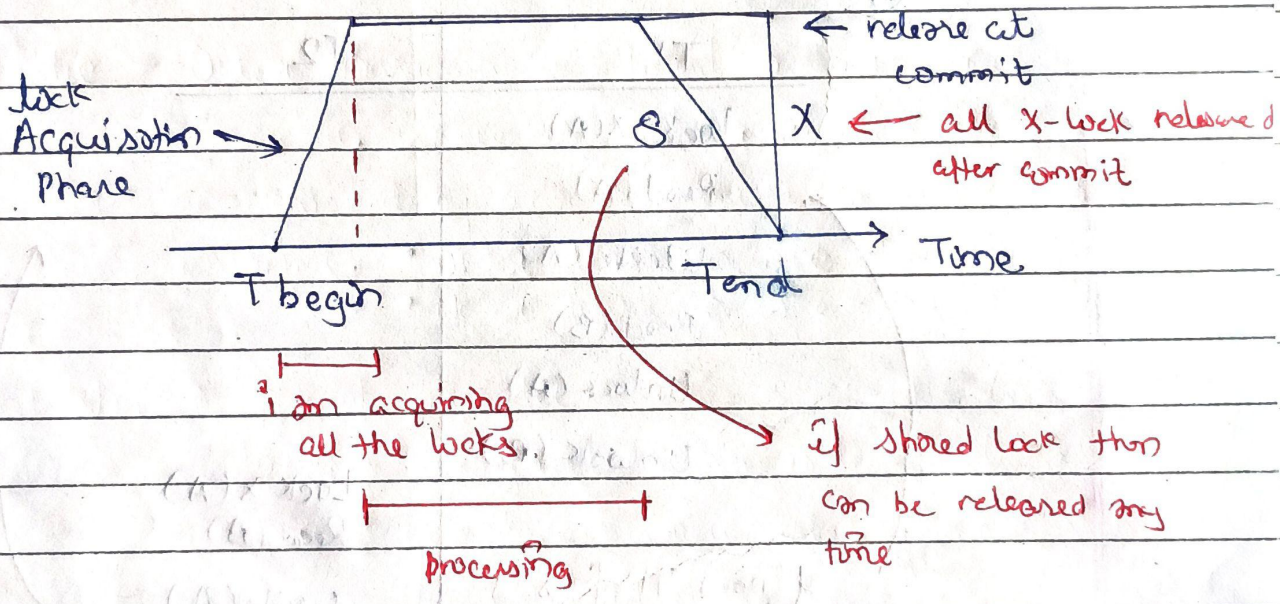
It limit the amount of concurrency that occur in a schedule because a transaction may not able to release a data item after it has used it. This may be because of other restrictions we have put on it to ensure serializability, deadlock freedom etc.

ii) Strict 2PL

Advantage – Avoid cascaded rollbacks
– It also says that all exclusive-locks should be held until the transaction commits or aborts.
– If item is modify by T1 then no other can read it till T1 commits.

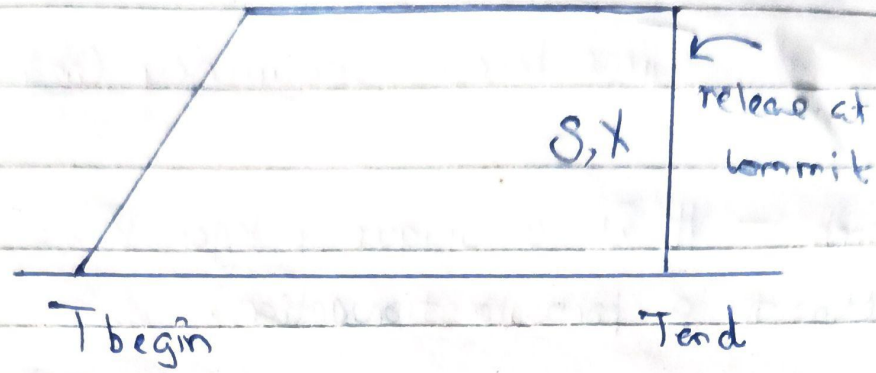
Disadvantage – Not deadlock free

avoid dirty read.



iii) Rigorous 2PL

Advantage – Avoid cascading rollback.
– Release all types of lock until the transaction commits
– There is no shrinking phase.



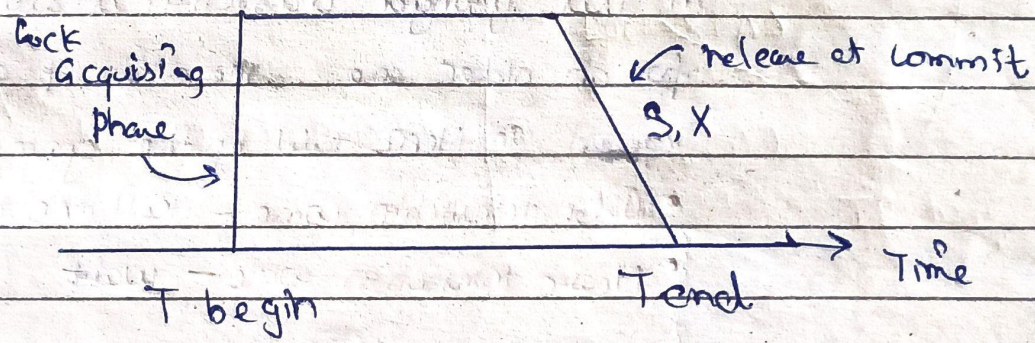
IV) Conservative 2PL

Also called Static - two PL.

Advantage - Almost free from deadlock as all required data items are listed in advanced.

- We don't have any growing phase.

→ It required locking all data items to access before the transaction starts.



Time based Concurrency Control

TimeStamp – It is a unique integer value given by a system to a particular transaction.

- Older transaction get smaller value.
- For giving purpose we may use – Counter, data & time value etc.

Wait die – Older transaction is allowed to wait for a younger transaction where as younger transaction requesting an item held by older transaction is aborted & restarted.
  - Older transaction come – wait
  - Newer transaction come – aborted (die)

Wound Wait – opp. of wait die.
  – In this younger transaction is allowed to wait for an older one, where as if older transaction request an item held by the younger transaction
    - Older transaction come – Kill the newer one & start itself
    - Newer transaction come – wait

→ In both cases we are killing the younger transaction so it may lead to deadlock