

Concurrency Control Protocol

Need of Concurrency Control Protocol

To maintain the consistency of shared data access during concurrent execution.
Attempting to have maximum possible concurrency without inconsistency.

Mechanisms To Control the Concurrency:

There are two mechanisms by which we can control concurrency according to our needs. The first mechanism is that in which the user is responsible to write the consistent concurrent transactions and they are called **Lock Based Protocols**.

And the second mechanism is that in which system itself tries to detect possible inconsistency during concurrent execution and either the inconsistency recovered or avoided. They are called **Time Stamping Protocols**.

Classification of concurrency control protocol

Lock based protocols

Binary locks

Shared / exclusive locks or read / write locks

2 phase locking protocol

Basic 2pl

Conservative 2pl or static 2pl

Strict 2pl

Rigorous 2 PL

Graph Based Protocol

Timestamp based protocol

Timestamp ordering protocol

Thomas write rule

Multiple granularity protocol

Multi version protocols

What is locking?

The concurrency problems can be solved by means of concurrency control technique called locking.

A LOCK variable (equivalent to semaphores used in OS) is associated with each data item which is used to identify the status of the data item (whether the data is in use or not).

When a transaction T intends to access the data item, it must first examines the associated LOCK.

If no other transaction holds the LOCK, the scheduler lock the data item for T.

If another transaction T1 wants to access same data item then the transaction T1 has to wait until T releases the lock.

Thus, at a time only one transaction can access the data item.

Using locks, we ensure Serializability and Recoverability but sometimes they may lead to Deadlock.

Deadlock occurs when each transaction T in set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.

Transaction Responsibility : Legal Transaction

To request lock before use of data item

To release lock after use of data item.

Example :

T1

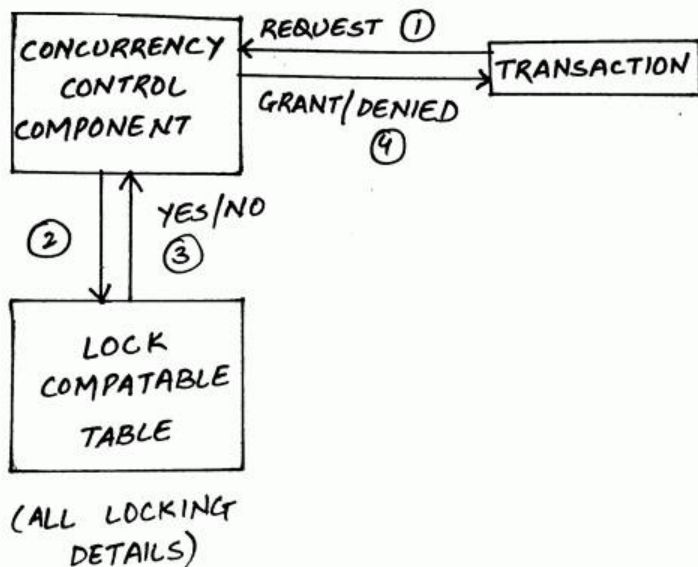
L(A) // Locks the data item A

R(A) // Read data item A

W(A) // Write data item A

U(A) // Unlock data item A

L(B) // Locks the data item B



Timestamp ordering

A different approach that guarantees serializability involves using transaction timestamps in order transaction execution for an equivalent serial schedule. Timestamp ordering is an alternative to locking in which the problem of concurrency control can be handled by assigning ordered timestamps to transactions.

In this Scheme :

Transaction timestamp TS (T) is a unique identifier which tells the order in which the transaction are started. Every transaction is assigned a timestamp by Database Administrator (DBA) such that Timestamp is unique.

Time stamps are in increasing order.

Hence, if transaction T_1 starts before transaction T_2 , then $TS(T_1) < TS(T_2)$. A timestamp based scheduler orders conflicting operations according to their timestamp values. Thus if $P_i(X)$ and $Q_i(X)$ are two conflicting operations on data item X requested by transaction T_i and T_j , $P_i(X)$ will be scheduled before $Q_i(X)$ if and only if $TS(T_i) < TS(T_j)$.

Shared and Exclusive Locks or Read/Write Locks

Need of Shared/Read and Exclusive/Write Lock

The binary lock is too restrictive for data items because at most one transaction can hold on a given item whether the transaction is reading or writing. To improve it we have shared and exclusive locks in which more than one transaction can access the same item for reading purposes. i.e. the read operations on the same item by different transactions are not conflicting.

What are Shared and Exclusive Locks

In this types of lock, system supports two kinds of lock :

- Exclusive(or Write) Locks and
- Shared(or Read) Locks.

Shared Locks

If a transaction T_i has locked the data item A in shared mode, then a request from another transaction T_j on A for :

- **Write operation on A** : Denied. Transaction T_j has to wait until transaction T_i unlock A .
- **Read operation on A** : Allowed.

Example :

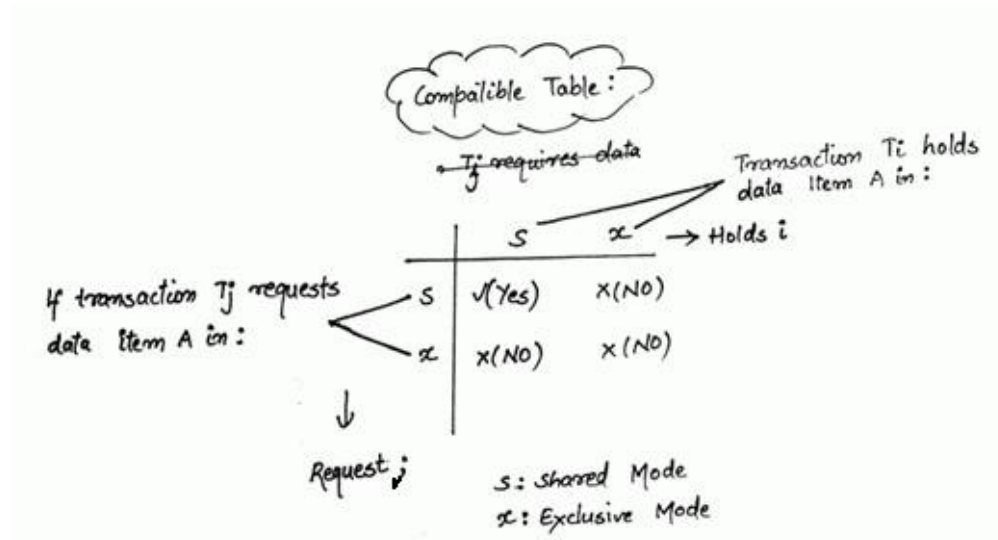
<u>T</u>	
$S(A)$	
$R(A)$	✓
$W(A)$	×

Exclusive Locks

If a transaction T_i has locked a data item a in exclusive mode then request from some another transaction T_j for-

- **Read operation on A** : Denied
- **Write operation on A** : Denied

COMPATIBLE Table for Shared and Exclusive Locks :



What does the Compatibility Table means ?

- If a transaction has lock a data item in shared mode, then another transaction can lock the same data item in shared mode.
- If a transaction has lock a data item in shared mode, then another transaction cannot lock the same data item in exclusive mode.
- If a transaction has lock a data item in exclusive mode, then another transaction cannot lock the same data item in shared mode as well as exclusive mode.

Operations Used with Shared and Exclusive Locks

1. Read_lock(A) or s(A)
2. Write_lock(A) or X(A)
3. Unlock(X) or U(A)

Implementation of Shared and Exclusive Locks

Shared and exclusive locks are implemented using 4 fields :

1. Data_item_name
2. LOCK
3. Number of Records and
4. Locking_transaction(s)

Again to save space, items that are not in the lock table are considered to be unlocked. The system maintains only those records for the items that are currently locked in the lock table.

Value of LOCK(A) : Read Locked or Write Locked

- **If LOCK(A) = write-locked** - The value of locking transaction is a single transaction that holds the exclusive(write) Lock on A.
- **If LOCK(A) = read-locked** - The value of locking transaction is a list of one or more transactions that hold the Shared(read) on A.

Transaction Rules for Shared and Exclusive Locks

Every transaction must obey the following rules :

1. A transaction T must issue the operation **s(A)** or **read_lock(A)** or **x(A)** or **write_lock(A)** before any read(A) operation is performed in T.
2. A transaction T must issue the operation **x(A)** or **write_lock(A)** before any write(A) operation is performed in T.
3. After completion of all read(A) and write(A) operations in T, a transaction T must issue an unlock(A) operation.
4. If a transaction already holds a read (shared) lock or a write (exclusive) lock on item A, then T will not issue an unlock(A) operation.
5. A transaction that already holds a lock on item A, is allowed to convert the lock from one locked state to another under certain conditions.
 - **Upgrading the Lock by Issuing a write_lock(A) Operation or Conversion of read_lock() to write_lock()** :
 - **Case 1 - When Conversion Not Possible** : A transaction T will not issue a write_lock(A) operation if it already holds a read (shared) lock or write (exclusive) lock on item A.
 - **Case 2 - When Conversion Possible** : If T is the only transaction holding a read lock on A at the time it issues the write_lock(A) operation, the lock can be upgraded;
 - **Downgrading the Lock by Issuing a read_lock(A) or Conversion of write_lock() to read_lock()** : A transaction T downgrade from the write

lock to a read lock by acquiring the write_lock(A) or x(A), then the read_lock(A) or s(A) and then releasing the write_lock(A) or x(A).

Points About Shared and Exclusive Locking :

- Shared and Exclusive Lock results more concurrency than Binary Locking.
- Shared and Exclusive Lock may not ensure serializability i.e. Non Serializable Schedule is possible to execute using shared and exclusive locking.

2 Phase Locking

Need of 2 Phase Locking

Binary locks or shared and exclusive locks in transaction does not guarantee serializability of schedules on its own. For example, consider a banking transaction where the read write locking rules are used, but we get a non serializable schedule which give incorrect result.

T₁	T₂
write_lock(X);	read_lock(X);
read_item(X);	read_item(X);
X := X-100;	read_lock(Y);
write_item(X)	read_item(Y);
unlock(X);	Z := X+Y;
write_lock(Y)	display(Z);
read_item(Y)	unlock(Y);
Y := Y+100;	unlock(X);
write_item(Y);	
unlock(Y);	

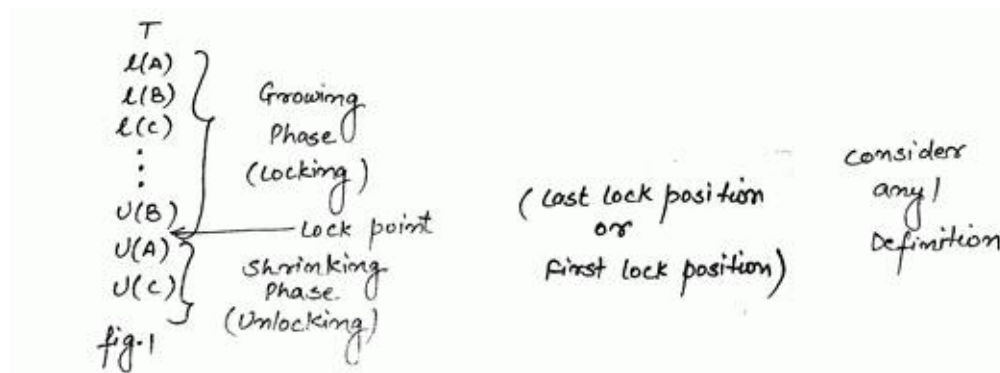
To ensure the serializability, we use two phase locking.

Two Phase Locking (2PL)

In this scheme, each transaction makes lock and unlock request in 2 phases :

A Growing Phase(or An Expanding Phase or First Phase) : In this phase, new locks on the desired data item can be acquired but none can be released.

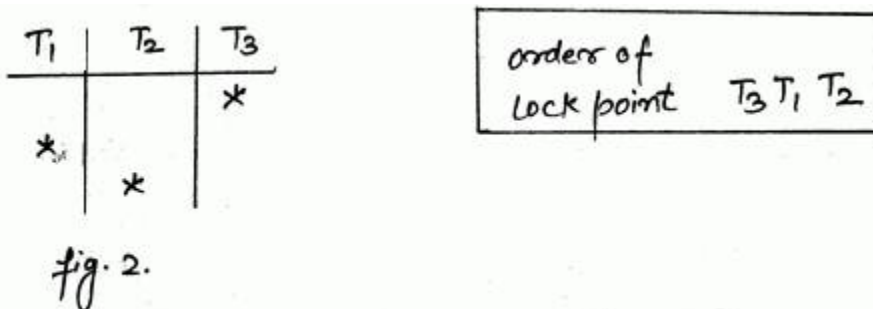
A Shrinking Phase (or Second Phase) : In this phase, existing locks can be released but no new locks can be acquired.



A 2 phase locking always results serializable schedule but it does not permit all possible serializable schedules i.e. some serializable schedules will be prohibited by the protocol.

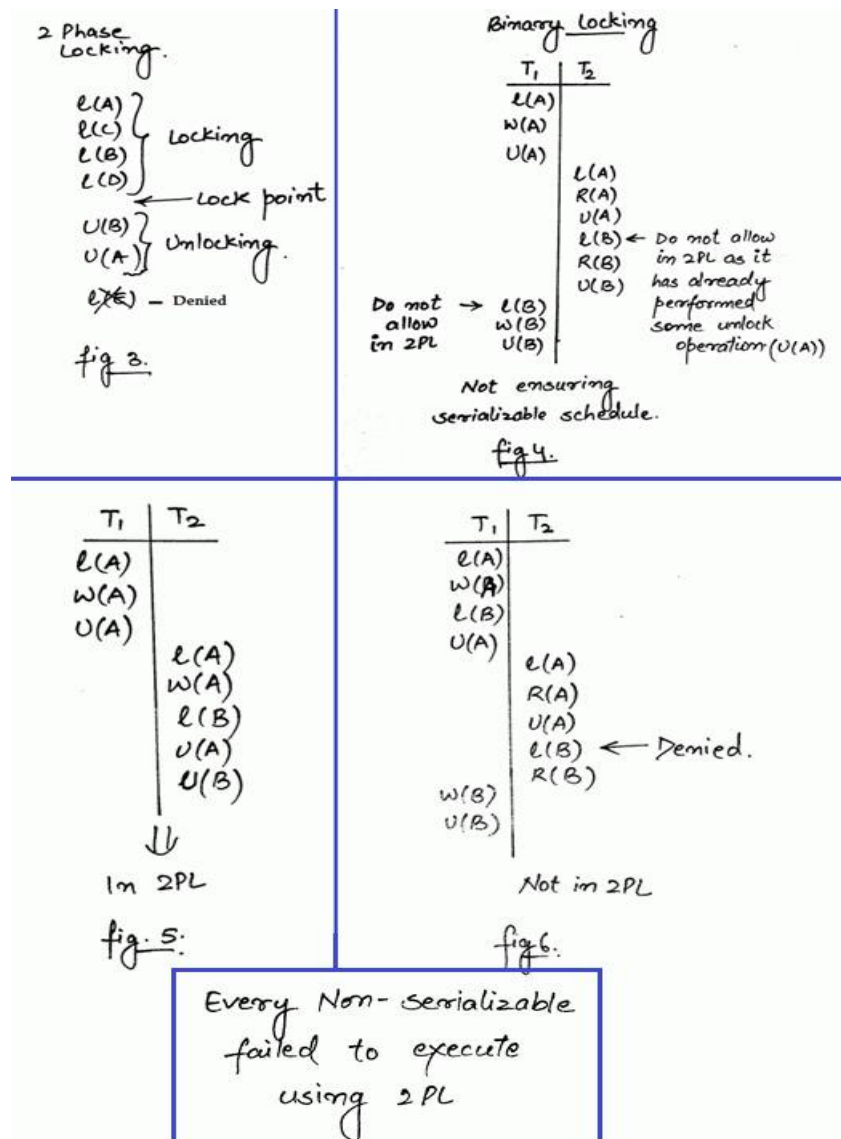
Strategy :

A transaction T does not allowed to request any lock if T has performed already some unlock operation and every equivalent serial schedule is based on the order of the LOCK point.



Question : Consider the scenario and find out weather it is in 2 phase locking or not?

Solution :



This technique is also called as Basic 2PL.

Points About 2 Phase Locking

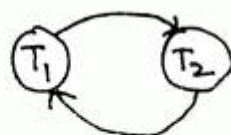
Every non serializable schedule failed to execute using two phase locking.

2 phase locking always results serializable schedule.

Equivalent serial schedule is based on the order of the lock point.

If schedule is allowed to execute by 2PL then schedule is conflict serializable but not vice versa.

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



Not conflict serializable
& also not allowed by 2PL.

versa.

If schedule is not conflict serializable, then schedule is not allowed to execute by 2PL.

A schedule is allowed to execute by 2 PL, then it is always serializable.

Serial \neq serializable

concurrent execution
is not allowed.

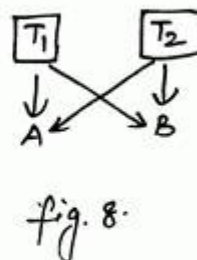
concurrent execution
is allowed

Combining Two Phase Locking and Binary Locking

2PL + Binary lock.

T_1	T_2
$L(A)$	
$R(A)$	
	$L(B)$
	$R(B)$
Demied $\rightarrow L(B)$	$L(A) \rightarrow$ Demied.
$R(B)$	
	$R(A)$

fig 7.



The first problem in binary locking + 2 phase locking is that it leads to deadlock. In the given example, T_1 is waiting for T_2 to get B and T_2 is waiting for T_1 to get A. Hence deadlock occurred.

The second problem in binary locking + 2 phase locking is that it provides less concurrency because in binary locking, 2 or more transactions simultaneously not allowed to perform read operation. This can easily be shown in the below figure 9.

T ₁	T ₂
L(A)	
R(A)	
	L(A) → Denied.
	R(A)
R(A)	

fig. 9.

T ₁	T ₂
L(A)	
W(A)	
U(A)	
	L(A)
	R(A)
	U(A)

fig 10.

The third problem (shown in fig 10) in binary locking + 2 phase locking is that the same lock is used for read as well as write, so we are unable to differentiate whether the transaction locks a data item for reading purpose or for writing purpose.

Combining 2 Phase Locking + Shared and Exclusive Locking

Only shared and exclusive locks may not ensure serializability i.e. non serializable schedule is possible to execute using shared and exclusive locking. For example,

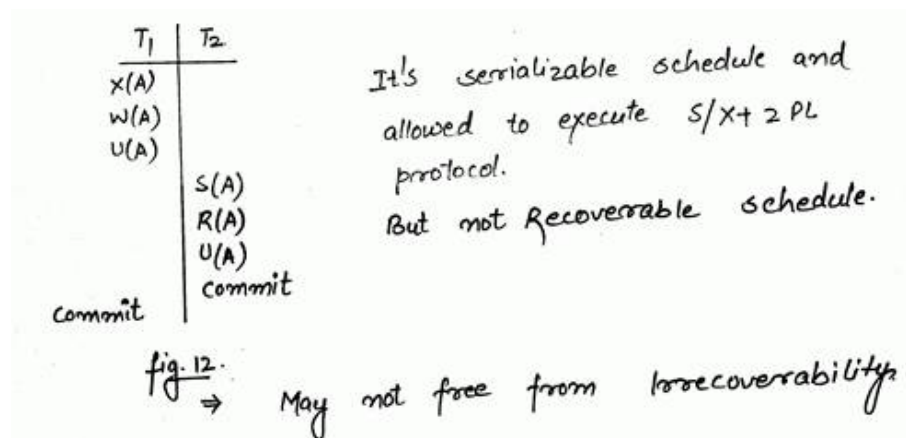
T ₁	T ₂
X(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
	L(B)
	R(B)
	U(B)
X(B)	
R(B)	
U(B)	

fig 11

only s/x may not ensure serializability.

(i.e. Non-serializable schedule possible to execute using s/x locking).

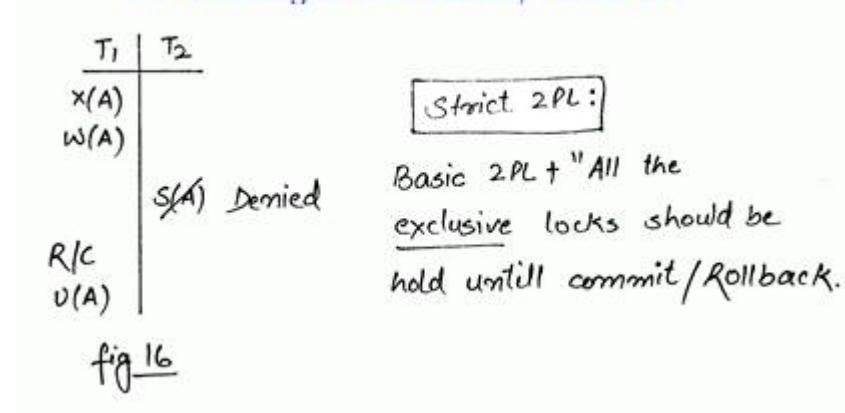
But when we combine both 2 phase locking and shared and exclusive locking, then it always results serializable schedule and the equivalent serial schedule is based on the order of LOCKING.



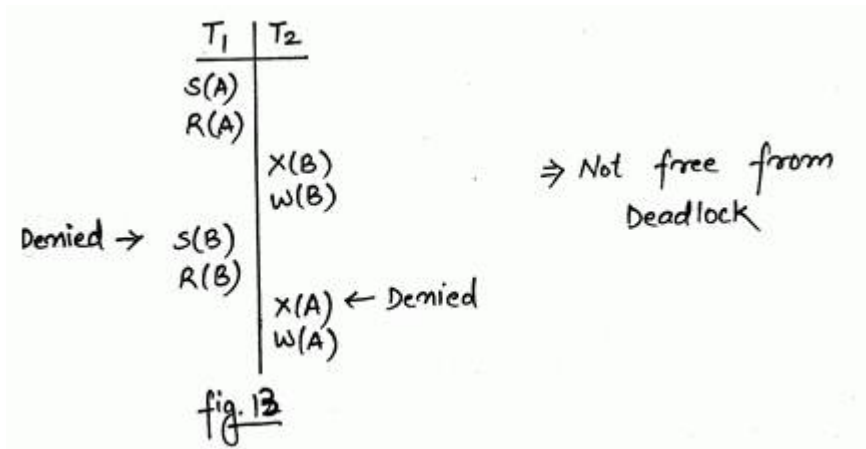
Problems occurred in (2 Phase Locking + Shared and Exclusive Locking)

The **first problem** in 2PL + S/X Locking is that however, we get a serializable schedule but it may not free from irrecoverability. **Example** : The schedule in figure 11 is not recoverable schedule. **To avoid irrecoverability**, the Exclusive lock is called until commit/rollback. Hence other transaction is not allowed either to read or write. (no dependency so no problem).

For Avoiding Irrecoverability : Strict 2PL

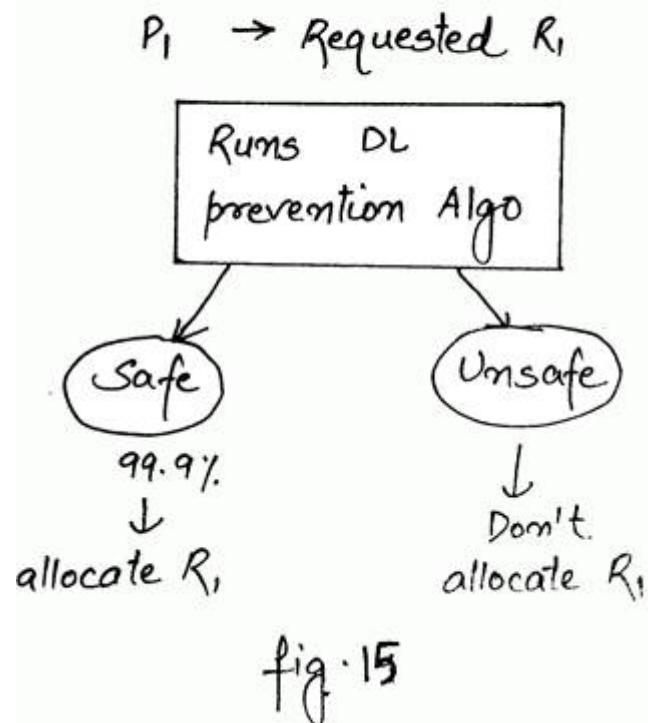


The **second problem** in 2PL + S/X Locking is that it is not free from deadlock. For example

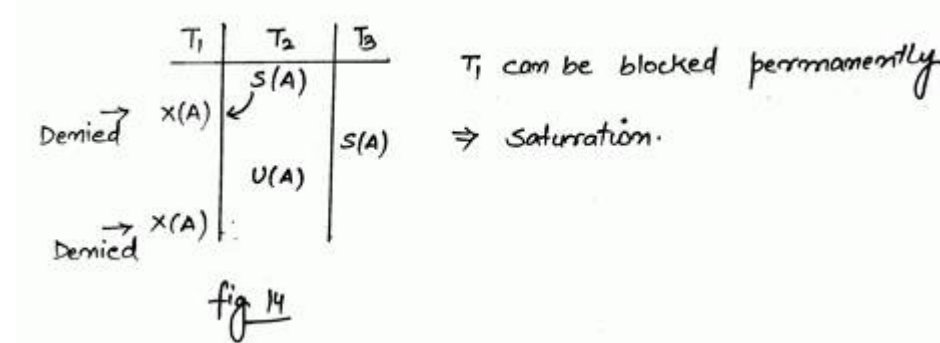


To recover from deadlock, deadlock prevention algo is used. But the deadlock prevention algorithm takes more CPU time than the actual work and 99.9 % times the algorithm returns safe. Therefore it is a wastage of precious CPU time. So Unix/Windows/Solaris don't use deadlock prevention algorithm.

Deadlock Prevention



The **third problem** in 2PL + S/X Locking is that the schedule is not free from **starvation**.



Variations of 2 Phase Locking :

There are number of variations of 2 Phase Locking :

Basic 2PL

Conservative 2pl (or static 2PL)

Strict 2PL

Rigorous 2PL

Basic 2PL

The technique of two phase locking describe above is known as Basic 2PL.

Conservative 2PL(or Static 2PL)

In Conservative 2PL protocol, a transaction has to lock all the items it access before the transaction begins execution. To avoid deadlocks, we can use Conservative 2PL.

Basic 2pl + all locks required to execute the transaction should be hold before starting its execution

Advantages of Conservative 2PL :

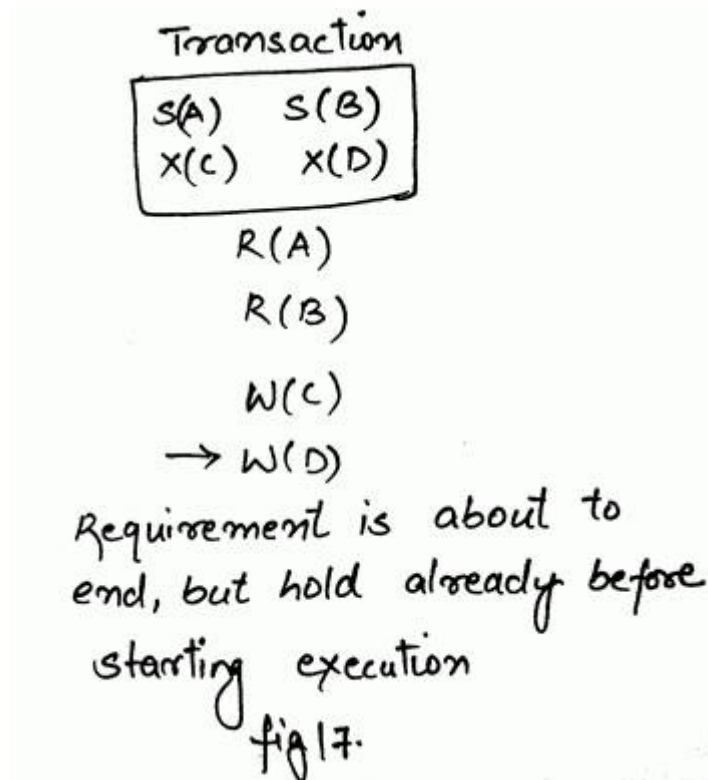
No possibility of deadlock.

Ensure serializability.

Drawbacks of Conservative 2PL :

Less throughput.

Less resource utilisation because it holds the resources before the transaction begins execution.



Less concurrency

Prediction of all the required resources before execution is also too complex.

Irrecoverability possible since no restriction on unlock operation.

However conservative 2pl is a deadlock free protocol but it is difficult to use in practice.

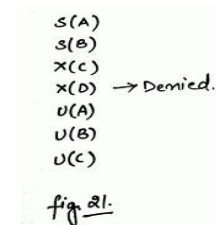
Starvation is possible.

How Starvation is possible ??

Question : What if some required resource is not available at the time of holding i.e. before the transaction begins execution?

Solution : It will unlock all other resources hold by it because they were free.

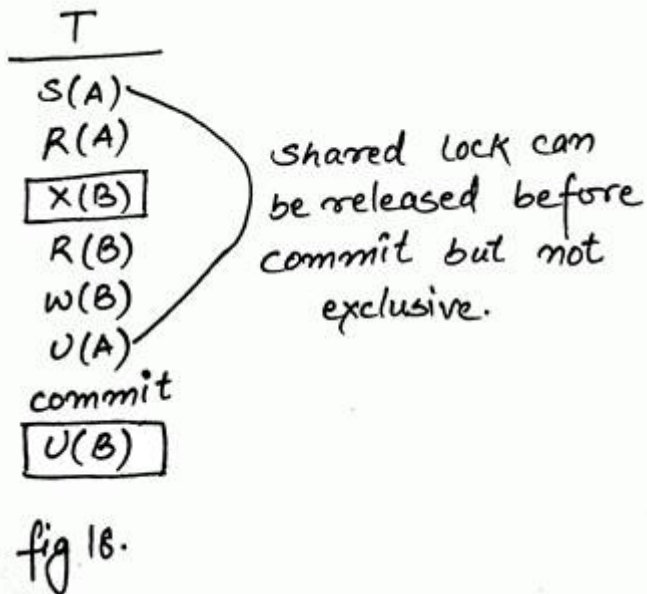
It may cause starvation. Because if at a time A is not available, another point of time B is not available and in another point of time C is not available.



Strict 2PL

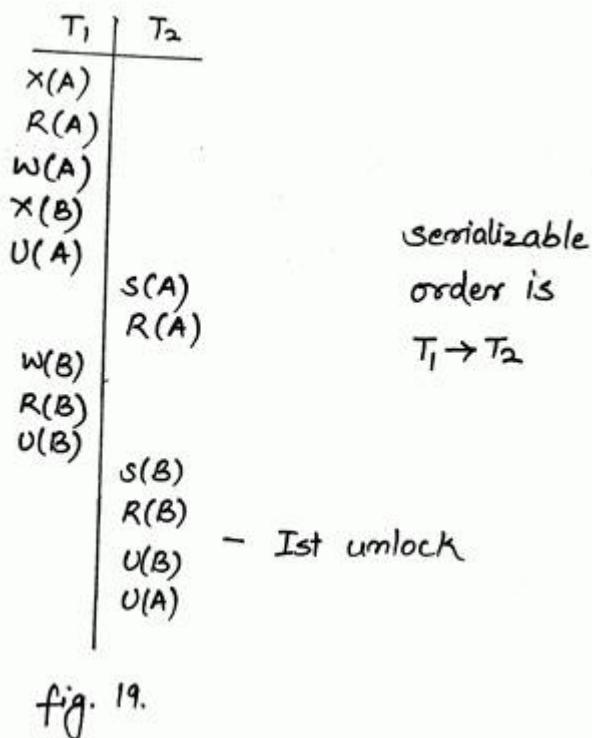
A transaction T does not release any of its exclusive(write) locks until after it commits or aborts.

Basic 2pl + all the exclusive locks should we hold until commit / rollback.



Points about Strict 2PL

Strict 2PL ensures serializability and the equivalent serial schedule is based on the lock point.



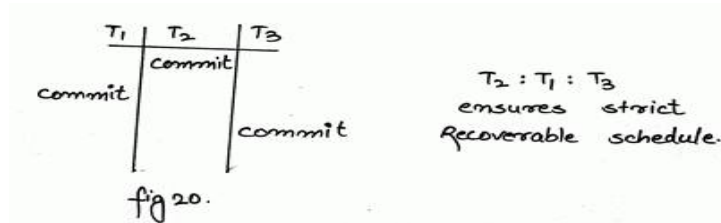
Strict 2PL also ensures strict recoverable schedule.
 Strict 2PL may not free from deadlock and starvation.

Rigorous 2PL Protocol

Transaction does not release any of its write locks and read locks until after it commits or aborts.

Basic 2PL + All S/X locks should be hold until commit / rollback.

Rigorous 2pl protocol ensures serializability and the equivalent serial schedule is based on the order of COMMIT.

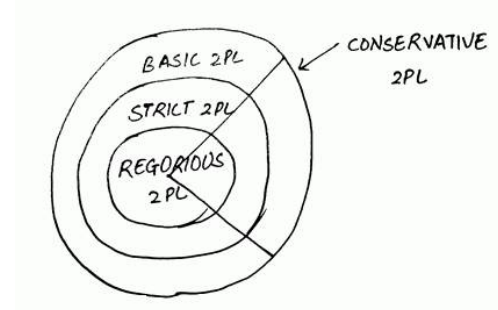


Example

Difference Between 2PL and Rigorous 2PL Protocol

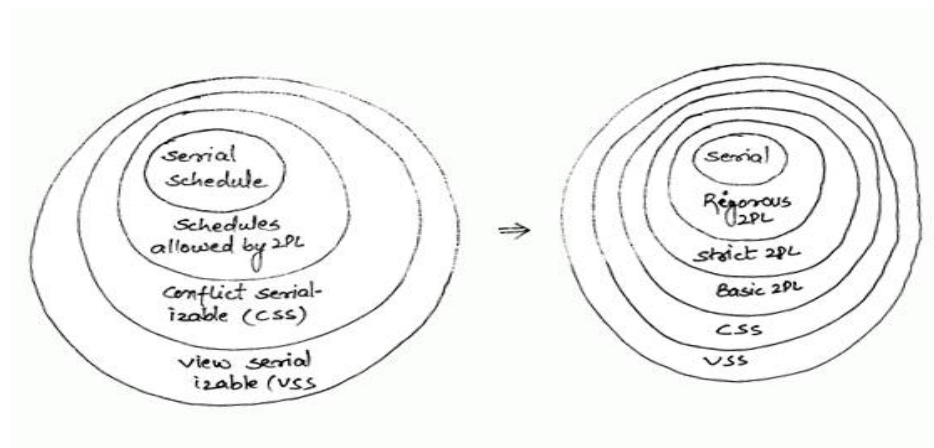
Strict 2PL	Rigorous 2PL
Equivalent serial schedule is based on the lock point.	Equivalent serial schedule is based on the order of commit.
Only exclusive locks should hold up to commit / rollback.	Both S / X locks are to be hold.

Expressive Power of Variations of 2 Phase Locking



There is no comparison between conservative 2PL and (Strict 2PL, Basic 2PL and Rigorous 2PL)

Relation Between Conflict Serializable, View Serializable and 2 Phase Locking Schedules



Binary Locks

A binary lock can have 2 States or values

- Locked (or 1) and
- Unlocked (or 0)

We represent the current state(or value) of the lock associated with data item X as LOCK(X).

Operations used with Binary Locking

1. **lock_item** : A transaction request access to an item by first issuing a lock_item(X) operation.
 - If **LOCK(X) = 1 or L(X)** : the transaction is forced to wait.
 - If **LOCK(X) = 0 or U(x)** : it is set to 1(the transaction **locks** the item) and the transaction is a load to access item X.
2. **unlock_item** : After using the data item the transaction issues an operation **unlock(X)**, which sets the operation **LOCK(X) to 0** i.e. unlocks the data item so that X may be accessed by another transactions.

Transaction Rules for Binary Locks

Every transaction must obey the following rules :

- A transaction T must issue the lock(X) operation before any read(X) or write(X) operations in T.
- A transaction T must issue the unlock(X) operation after all read(X) and write(X) operations in T.
- If a transaction T already holds the lock on item X, then T will not issue a lock(X) operation.

- If a transaction does not holds the lock on item X, then T will not issue an unlock(X) operation.

Example :

T_1	T_2
$L(A)$	
$W(A)$	
$U(A)$	
	$L(A)$
	$R(A)$
	$U(A)$
	$L(B)$
	$R(B)$
	$U(B)$
$L(B)$	
$R(B)$	
$W(B)$	
$U(B)$	

Points About Binary Locking :

- A binary lock enforces **mutual exclusion** on the data item.
- Serializability is not ensure i.e. may not eliminate non serializable schedule.
- The binary locks are simple but restrictive and so are not used in practice.

Implementation of Binary Locks :

Binary lock is implemented using **3 fields plus a queue** for transactions data waiting to access item. **The 3 fields are :**

1. Data_item_name
2. LOCK
3. Locking_transaction

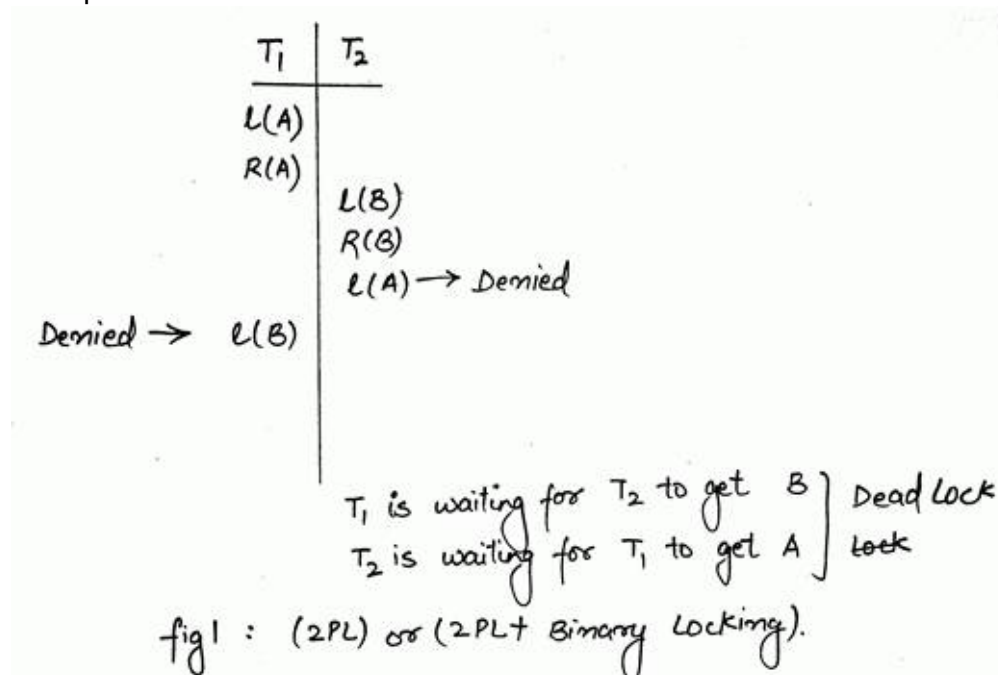
To keep track of and control access to locks, DBMS has a **lock manager subsystem**. Items that are not in the lock table are considered to be unlocked. The system maintains only those records for the items that are currently lock in the lock table.

Deadlock in Transaction

What is Deadlock and How the Deadlock is occurred?

Deadlock occurs when each transaction **T** in a set of 2 or more transactions is waiting for some item that is locked by some other transaction **T'** in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transaction in the set to release the lock on an item.

Example :



Where the Deadlock Occurs ?

The concurrency control technique called locking may lead to deadlock.

Locking

2 Phase Locking (2PL)

2PL + Binary Locking

2PL + Shared and Exclusive Locking

However, 2 phase locking ensures serializability but it may lead to a deadlock state. The combination of (binary lock + 2 PL) and the combination of (shared and exclusive lock + 2 PL) also may lead to a deadlock state.

Example :

T_1	T_2
$S(A)$	
$R(A)$	$X(B)$
	$W(B)$
Denied $\rightarrow S(B)$	$X(A) \rightarrow$ Denied

fig2: (2PL + shared & exclusive locking)

T_1 is waiting for T_2 to get B.] Deadlock
 T_2 is waiting for T_1 to get A.]

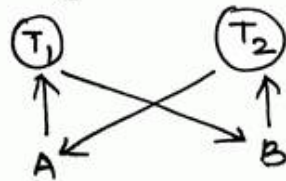


fig.3

What is Deadlock Detection and Deadlock Prevention ?

Deadlock Detection deals with deadlocks in which the system check if state of deadlock actually exists or not whereas Deadlock Prevention Protocols are used to prevent from deadlock.

When to Use Deadlock Detection and When to Use Deadlock Prevention?

If the transaction load is light or if the transactions are short and each transaction lock only a few items, then **we should use a deadlock detection scheme**.

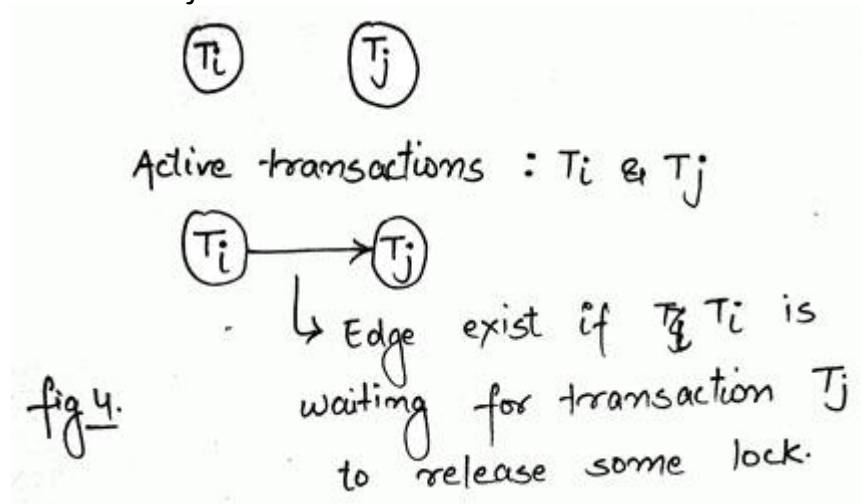
However, if the transaction load is quite heavy or if the transaction are long and each transaction uses many data items, then **we should use a deadlock prevention scheme**.

Deadlock Detection Schemes

Wait-For-Graph (WFG)

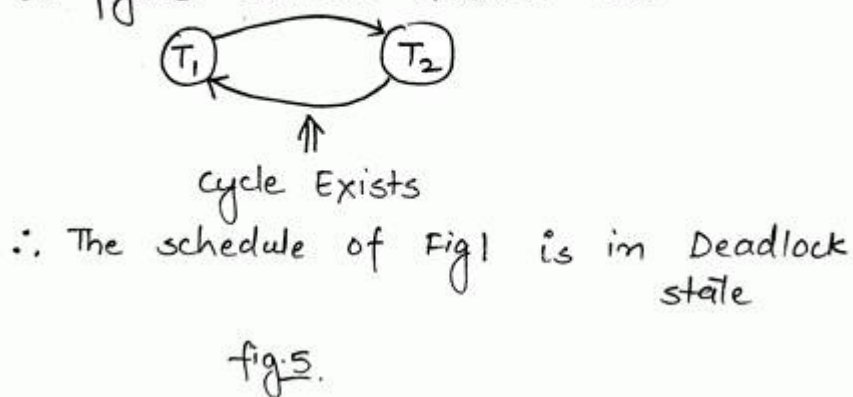
To detect the state of deadlock, directed graph is to be constructed which is called wait-for-graph(WFG). The nodes of WFG are labelled with active transaction names. and

an edge exists from $T_i \rightarrow T_j$ from node T_i to node T_j iff transaction T_i is waiting for transaction T_j to release some lock.



For a deadlock to occur, the WFG must have a cycle and the scheduler can detect deadlocks by checking for cycles in the WFG.

The WFG (Wait for Graph) for the schedule in figure can be drawn as:



After the detection of the deadlock, it can be prevented by aborting a transaction.

The scheduler should select the transaction among the transactions involved in the cycle of WFG whose absorption involves minimum cost i.e. it tries to select younger transactions (which do not made many changes).

The chosen transaction for abort is called **victim transaction**.

Point :

The algorithm for victim selection generally avoid those transactions that have been for a long time and that have performed many updates.

Timeout Method

Timeout Method is more practical scheme for deadlock. In timeout method if a transaction waits for a period longer than a system defined timeout period, the system assumes that the transaction may be in deadlock state and aborts it regardless of whether a deadlock actually exist or not.

Deadlock Prevention Protocols

Deadlock prevention protocols are used to prevent from deadlock. There are various deadlock prevention protocols which are :

Conservative Two Phase Locking (Conservative 2PL)

Since each transaction lock all the items it needs in advance, therefore no deadlock occurs. If any of the items cannot be obtained then none of the items are locked.

The conservative 2pl is not a practical method as it provides less concurrency and it can leads to starvation also.

Ordering All the Items Protocol

This protocol also limits concurrency because it involves ordering all the items in the database and making sure that the transaction that need several items will lock them according to that order.

It is also not a practical method as the programmer order system is aware of the chosen order of the items.

No Waiting Algorithm

In No Waiting Algorithm scheme if a transaction is unable to obtain a lock it is immediately abort it and then restarted after a certain time delay without seeing whether a deadlock will actually occur or not.

This Algorithm can cause transactions to abort and restart needlessly.

Cautious Waiting Algorithm

To reduce the number of needless aborts and restarts, the cautious algorithm is used. Suppose there are two transaction T_i and T_j . T_i is waiting for a data item X which is locked by T_j .

If T_j : Blocked(waiting for some other locked data item) - Abort T_i .

If T_j : Not Blocked(not waiting for some other locked data item) - T_i is blocked and allowed to wait.

The Cautious Waiting is Deadlock Free because no transaction will never wait for another blocked transaction.

Deadlock Prevention Using the Concept of Timestamp Ordering [TS(T)]

Transaction timestamp $TS(T)$ is a unique identifier assigned to each transaction based on the order in which transaction are started.

Transaction timestamp $TS(T)$

Timestamp $TS(T)$ is a unique identifier but not any kind of time. $TS(T)$ can be understand as a priority identifier in which the lesser number identifies the older transaction and the greater number identifies the younger transaction i.e.

if T_1 starts before transaction T_2 ,
then, $TS(T_1) < TS(T_2)$.
(Lesser priority) (Higher priority)
(Older Transaction) (Younger Transaction)

There are two methods for preventing deadlock using the concept of timestamp ordering :

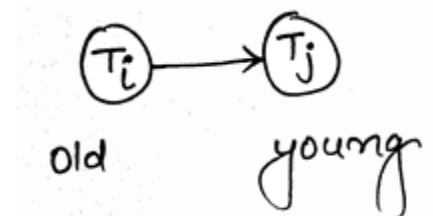
Wait Die

Wound Wait

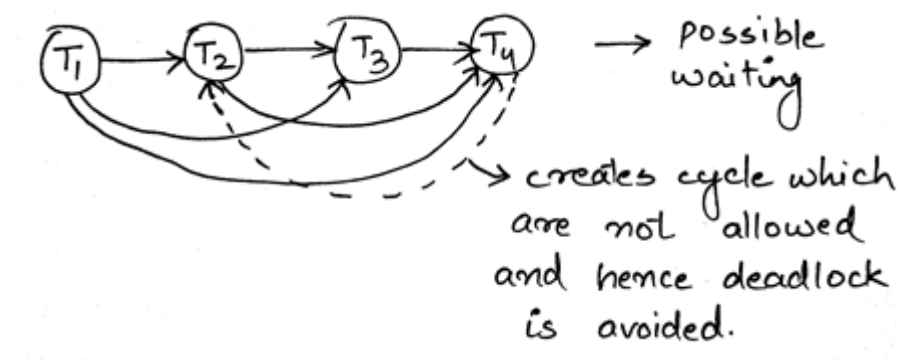
Suppose there are two transaction T_i and T_j where T_i is older than T_j i.e. $TS(T_i) < TS(T_j)$,
The following rules are followed by these schemes :

Wait die :

If transaction T_i is waiting for a data item X which is locked by T_j , then T_i is allowed to wait.

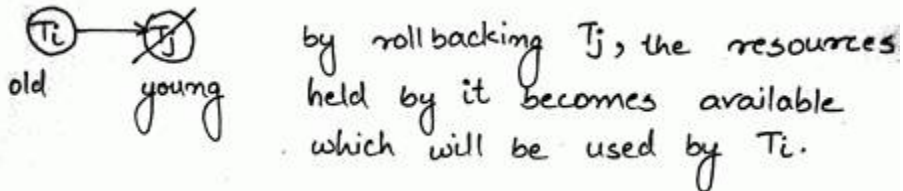


If transaction T_j is waiting for a data item X which is locked by T_i abort T_j or rollback T_j and restart it later with the same timestamp.

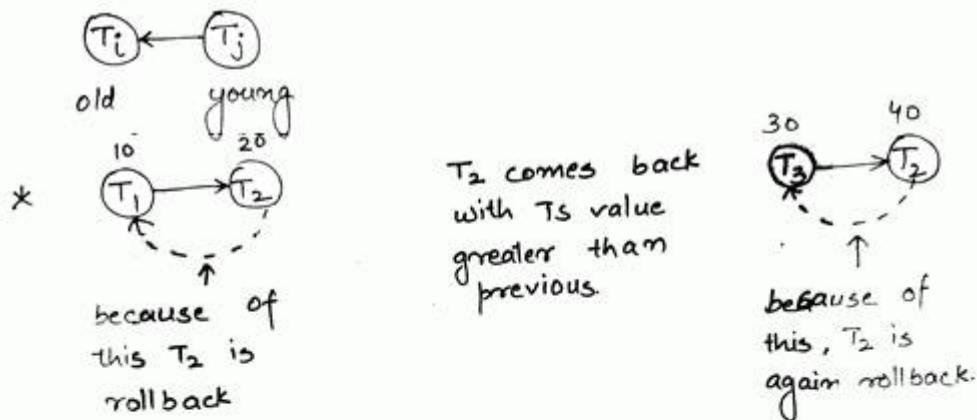


Wound Wait :

If transaction T_i is waiting for a data item X which is locked by T_j then abort T_j or rollback T_j and restart it later with the same timestamp.

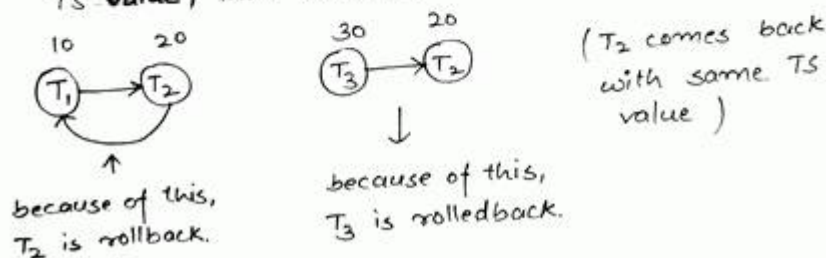


If transaction T_j is waiting for a data item X which is locked by T_i , then T_j is allowed to



∴ hence starvation still possible.

But if we are able to start T_2 with same TS value, then starvation is not possible.



so restart T_j with same Timestamp (TS) value.

wait.

Starvation in Deadlock (Livelock)

What is Starvation?

The indefinite waiting of a transaction is called as **starvation**.

When does Starvation Occurs?

Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.

Example :

- Suppose 2 transactions T1 and T2 are trying to acquire a lock on a data item X, the scheduler grants the lock to T1.
- Before the execution of T1 is over, suppose another transaction T3 also request unlock on data item X and when T1 unlock data item X, both T2 and T3 are trying to acquire a lock on a data item X.
- If the scheduler now grants this lock to T3, then T2 has to wait again.
- If another 4th transaction T4 may ask for data item X and the scheduler may deprive T2 again when T3 unlocks data item X, then T2 may have to wait again.
- Thus T2 may have to wait indefinitely although there is no deadlock. Such a situation is called **Livelock or Starvation**.

Starvation generally happens when we use a Priority Queue in which a lower priority transaction will starve due to frequently coming of higher priority transaction. It can also occurred because of victim selection if the algorithm select the same transaction as victim repeatedly, thus causing it to abort and never finish execution.

Solutions of Starvation

Solution 1 : Using a FCFS Queue

For avoiding the Livelock is to follow a fair scheduling policy such as First Come First Serve (FCFS) queue in which transactions are unable to lock an item in the order in which they originally requested the lock.

Solution 2 : Increasing Priority

Allow some transaction to have priority over others. But increasing the priority of a transaction, it has to wait longer until a highest priority transaction comes and proceeds.

Solutions 3 : Modifying the Victim Selection Algorithm

The algorithm can use higher priority for transactions that have been aborted multiple time to avoid this problem.

Solution 4 : Using wait-die and wound-wait schemes

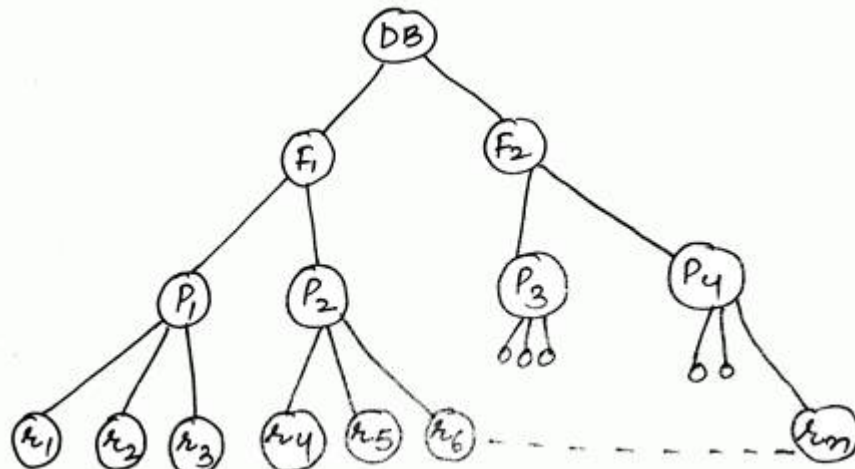
The wait-die and wound-wait schemes can also be used to avoid the problem of starvation because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

Multiple Granularity Locking Protocol

Before defining Multiple Granularity - let us define what is Granularity ? Granularity is the size of data item allowed to lock. **Multiple Granularity** is the hierarchically breaking up the database into portions which are lockable and maintaining the track of what to be lock and how much to be lock so that it can be decided very quickly either to lock a data item or to unlock a data item.

Example of Multiple Granularity

Suppose a database is divided into files; files are divided into pages; pages are divided



into records.

If there is a need to lock a record, then a transaction can easily lock it. But if there is a need to lock a file, the transaction has to lock firstly all the records one after another, then pages in that file and finally the file. So, there is a need to provide a mechanism for locking the files also which is provided by multiple granularity.

Why there is a Need to provide a Mechanism for Locking Files as well as Records ?

- If we allow a mechanism for locking records only, then to lock a file, the transaction will have to lock all records in that file (say 10000 at one time) one after another, which is a wastage of time.
- If we allow a mechanism for locking file only, then for a transaction to lock only five records, it will have to lock the whole file and therefore no other transaction will be able to use that file.

- So, there is a need to provide the locks for files as well as records (provided by multiple granularity).

Now the next question which comes in mind is that :

How do we Know that Some Transaction has Lock the Record or File or Database ? Because :

- If some transaction has lock a record then it should not be allowed to lock the file or database.
- Or if some transaction has lock a record of a file and there is a need to lock a record of another file, then it should be allowed to lock that record.

Keeping these parameters in mind, the system should be provided with maximum concurrency without inconsistent locking because it will lead to deadlock situation. And these parameters or protocols are called **multiple granularity protocol**.

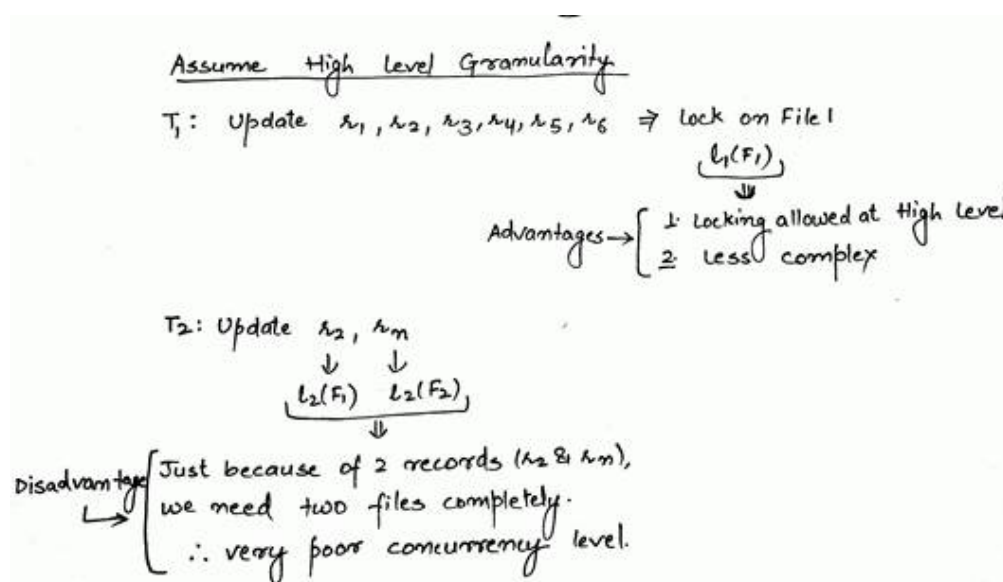
Granularity at High Level : Locking at High Level

Disadvantage of Granularity at High Level:

1. Less concurrency level

Advantage of Granularity at High Level :

1. Locking is allowed at high level
2. Less complex
3. Easy to implement



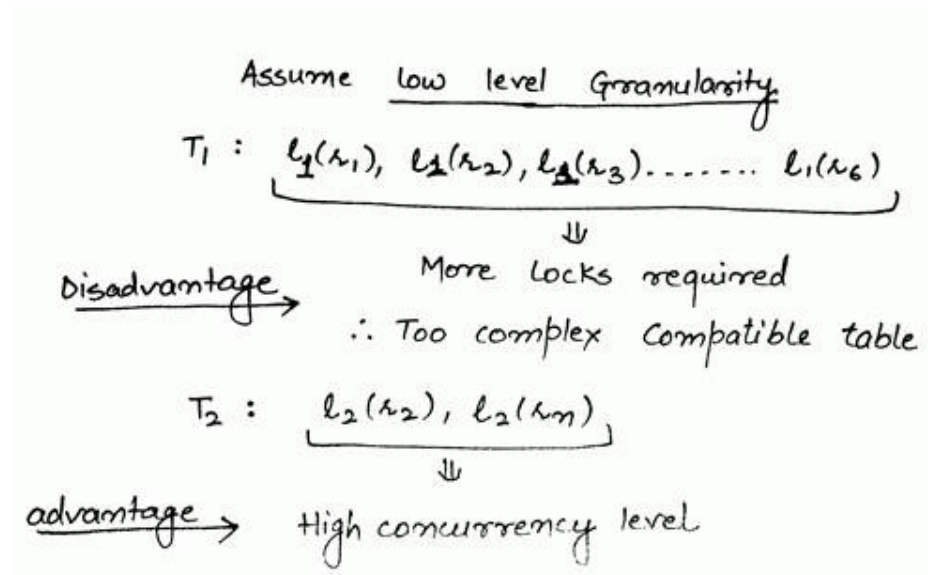
Granularity at Low Level : Locking at Low Level

Disadvantage of Granularity at Low Level:

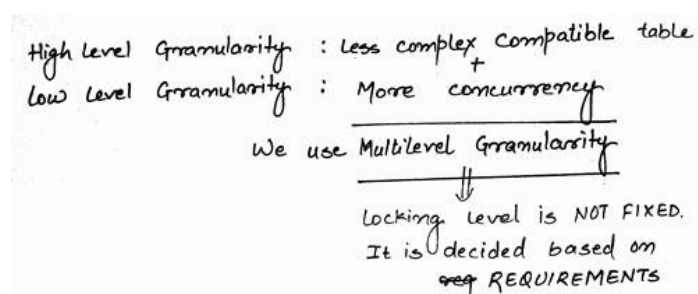
1. More locks required : therefore too complex compatible table.

Advantage of Granularity at Low Level:

1. High concurrency level



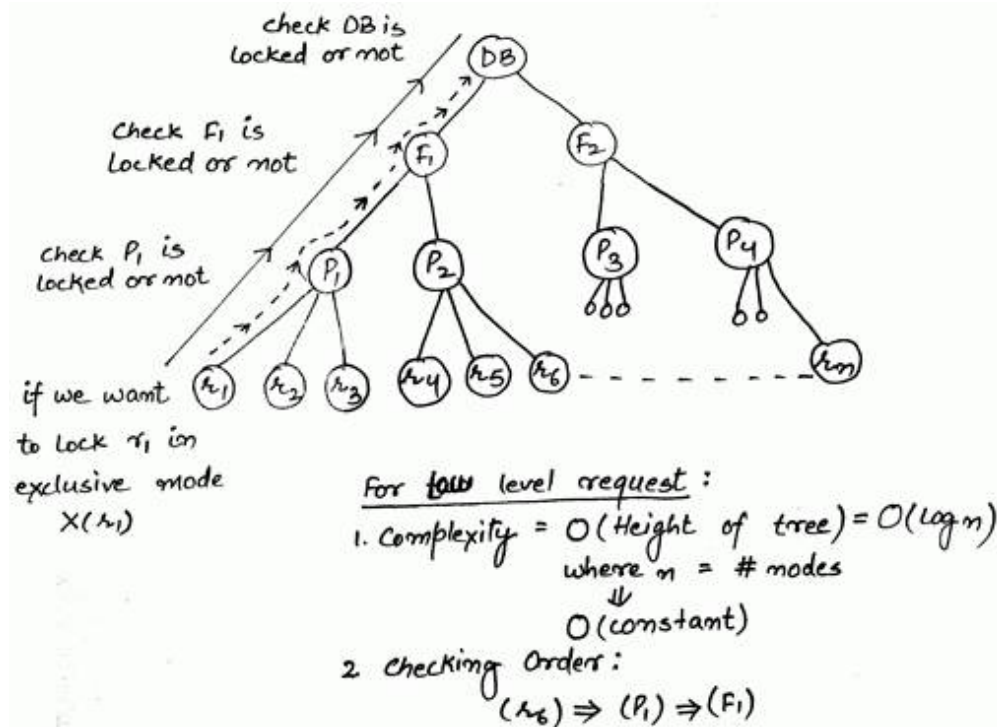
In order to get advantages of both (high level granularity + low level granularity), we use Multiple



Granularity

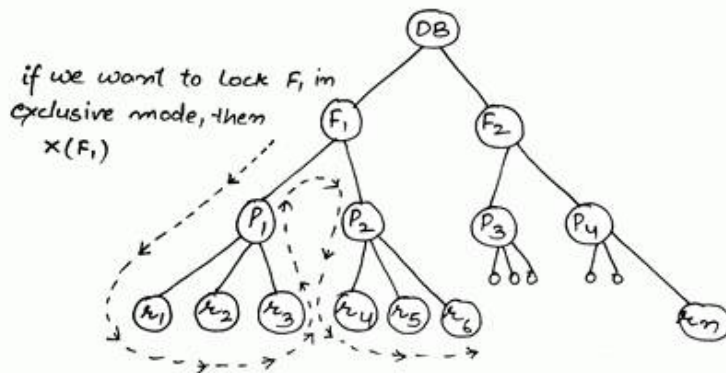
Now consider the figure 1 : **Points**

1. If F1 has been locked by transaction T1 and T2 wants to access the record r6, then it should not be allowed to do so, to preserve the consistency.
2. **For a low-level request** (say if r1 has to be locked in exclusive mode) then we have to check the locks only upto the height of the tree (i.e. whether P1, F1, DB is locked or not).



3. **For a high-level request** (say if F1 has to be locked in exclusive mode), we have to thoroughly search the tree (all descendants of that particular node) if any node is locked or not. Concurrency control will grant the permission to lock F1 only if

none of the descendant of F1 should be locked by other transactions. **Example :**



For high level request :

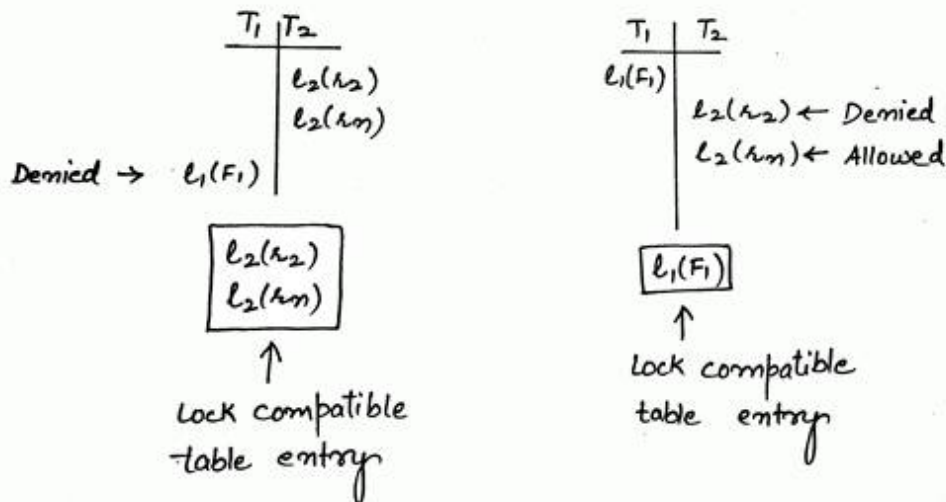
1. Complexity: Higher complexity
 $\Rightarrow O(\text{Exponential})$ { since the tree is not surely arbitrary tree }

2. Checking Orders :

$(F_1) \Rightarrow (P_1) \Rightarrow (R_1) \Rightarrow (R_2) \Rightarrow (R_3) \Rightarrow (P_2) \Rightarrow (R_4) \Rightarrow (R_5) \Rightarrow (R_6)$

Can we Decrease the Complexity of Searching Process ?

To Decrease the search a little bit, we can maintain information about the file whose any record is locked by a transaction, is maintained in lock compatible table or say some traces must be left along the path (i.e. height of the tree). So that whenever a request of Lock is arrived, we can check the table directly.



Concept of Intention Locks

S : Shared Locks/Mode

X : Exclusive Locks/Mode

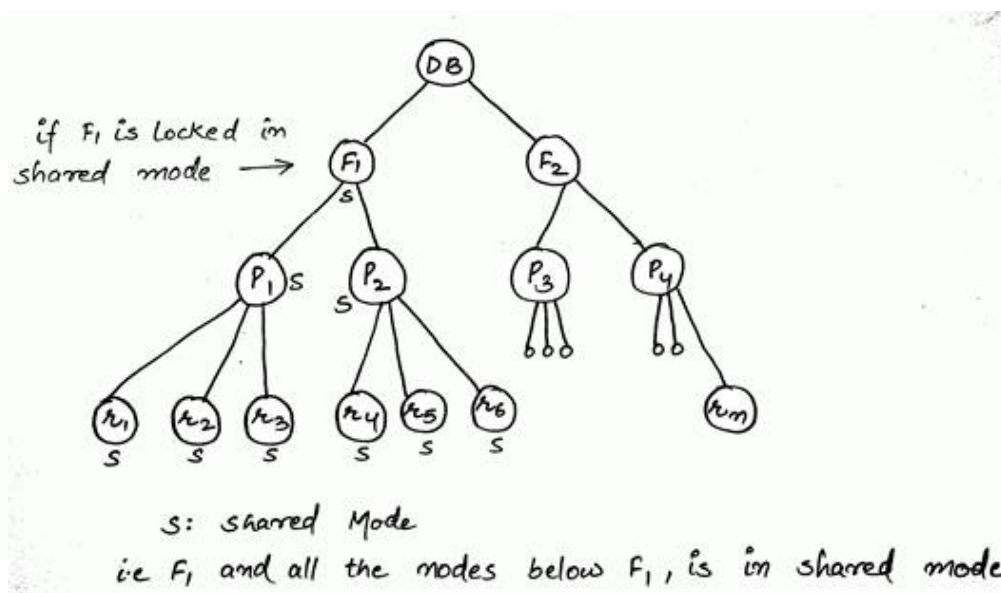
IS : Intention Shared Locks/Mode

IX : Intention Exclusive Locks/Mode

SIX : Shared and Intention Locks/Mode

Shared Mode (S):

If a node is locked in **shared mode**, then that node and all the nodes below it are locked in shared mode.



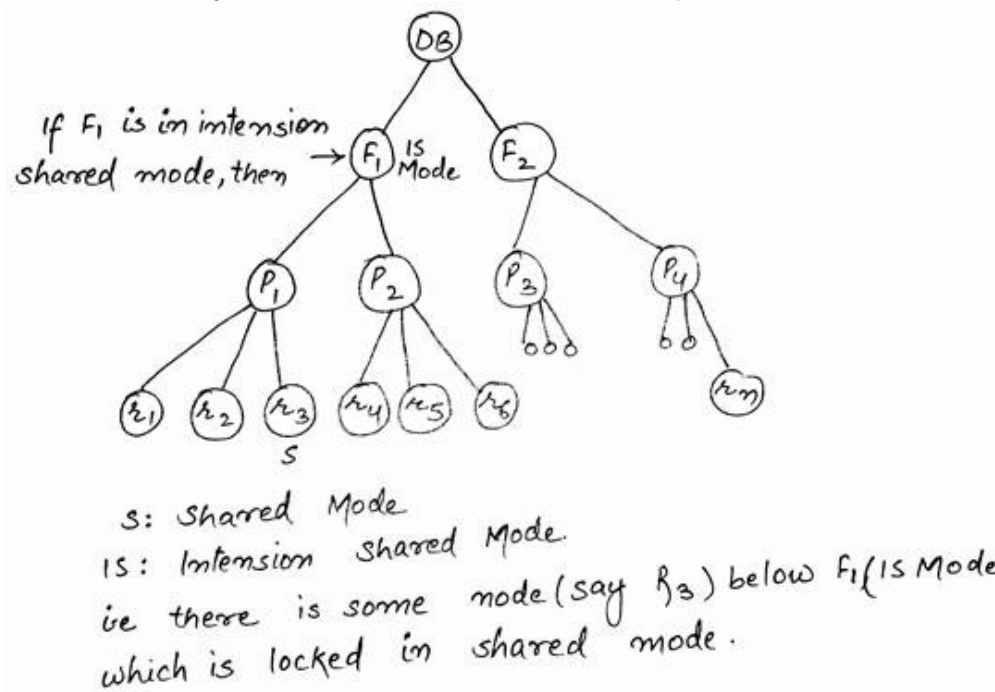
Exclusive mode (X):

If a node is locked in **exclusive mode**, then that node and all the nodes below it are locked in exclusive mode.

Intention Shared Mode (IS):

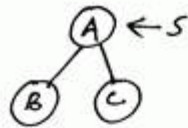
If a node is locked in **intention shared mode**, then there is some node below it which is locked in shared mode. **In other words**, a node n locked by transaction T in IS mode

means that any descendants of N can be request shared (Si) lock by transaction T.



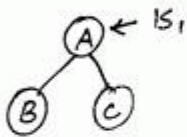
Shared mode (S) and Intention Shared Mode (IS) :

Shared & Intension Shared :



$\frac{T_1}{S(A)}$
 $R(B) \leftarrow \text{Granted}$
 $R(C) \leftarrow \text{Granted}$

If we have a direct lock as say shared lock, then we can directly access/read the data item under it.



$\frac{T_2}{IS(A)}$
 $R(B) \leftarrow \text{Denied}$

$\frac{T_3}{IS(A)}$
 $S(B)$
 $R(B) \leftarrow \text{Now read is granted}$

If we applied intension lock, then before reading a data item under it, we have to apply shared lock firstly.

ie 2 locks are required

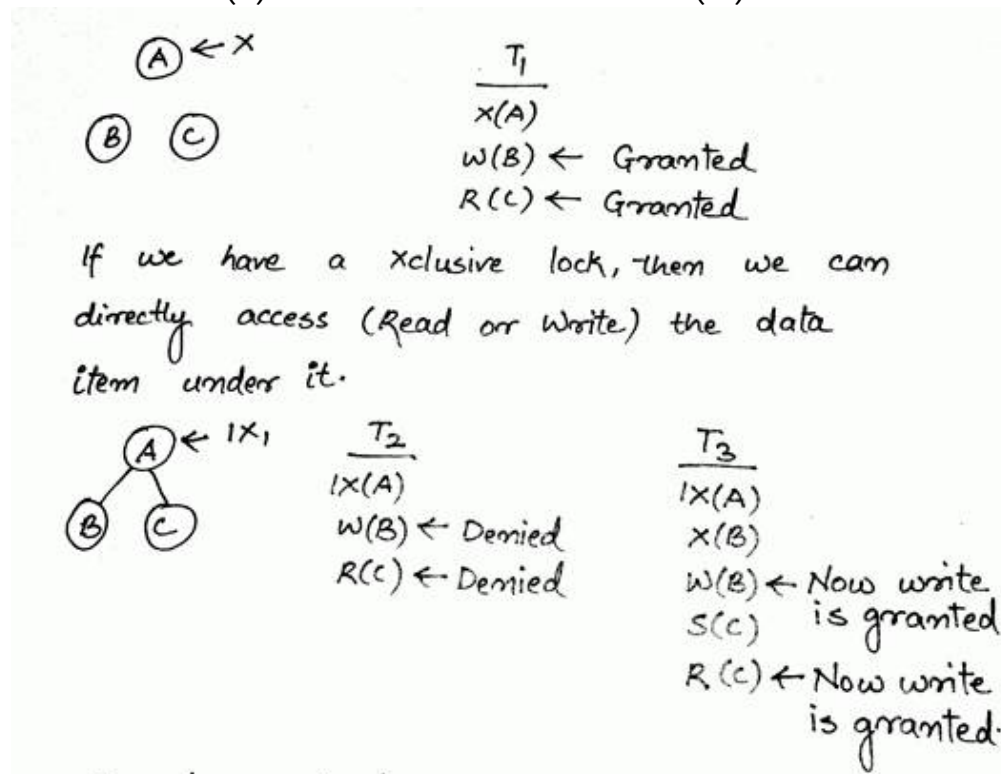
1. IS()

2. S()

Intention Exclusive Mode (IX):

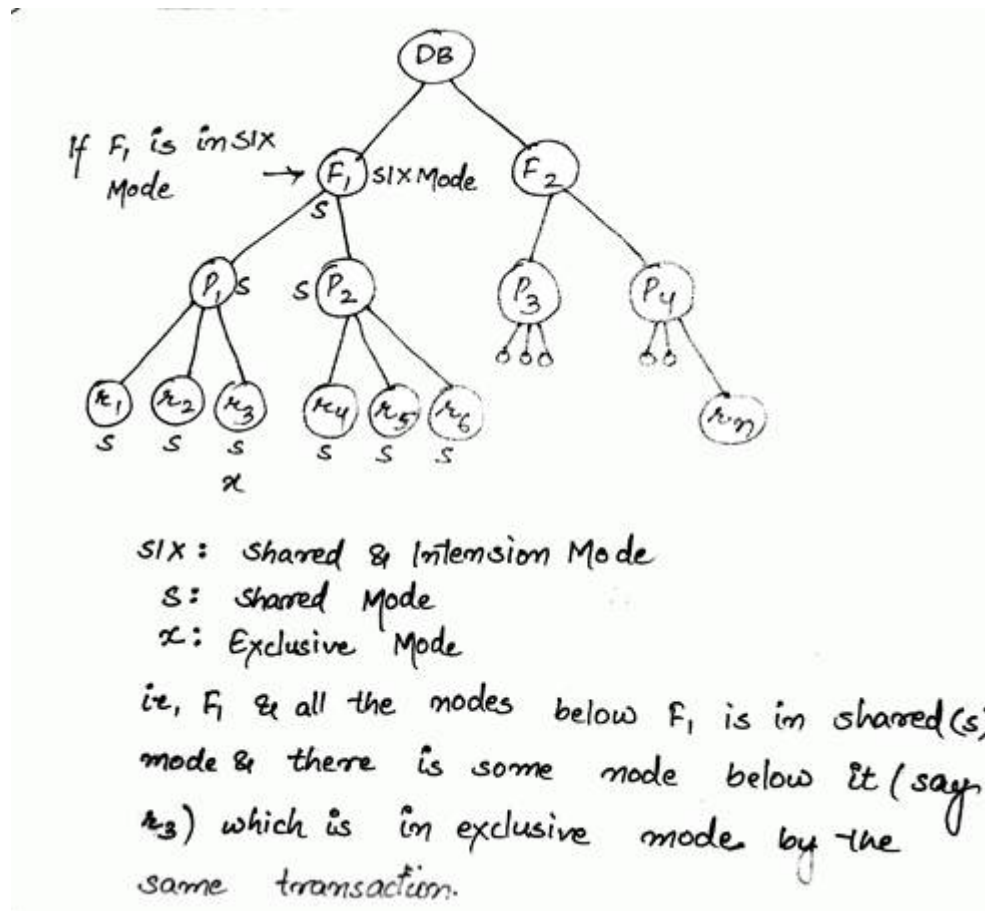
If a node is locked in **intention exclusive mode**, then there is some node in the tree below this node which is locked in exclusive mode. **In other words**, a node N locked by transaction T in IX mode means that any descendants of N can be request shared lock / exclusive lock by transaction T.

Exclusive Mode(X) and Intention Exclusive Mode(IX) :



Shared and Intention Exclusive Mode (SIX):

If a node is locked in shared and intention exclusive mode (SIX) then this node is locked in shared mode and there is some node below it which is locked in exclusive mode by same transaction.



Points in SIX mode :

Consider F_1 is in SIX mode buy transaction T_1 :

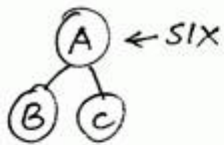
- If another transaction (say T_2) wants to lock (r_2 : shared) in shared mode, - then T_2 is allowed to lock it.
- If another transaction (say T_2) wants to lock (r_2 : shared) in exclusive mode, - then T_2 is not allowed to lock it.

Shared & Intension Exclusive :

⇓

Read Directly, { No shared lock required }
for reading

for write, { First lock exclusively, &
then we can write on data
item



T₁
R(B)
R(C) } - Direct Read - Granted
W(B) ← Denied

T₂
R(B)
R(C) } - Direct Read
X(B)
W(B) ← Now write is granted

Lock Compatibility matrix

The lock compatibility matrix grants to transaction to access a data item say N only if the locks required by them are compatible.

The Lock Compatibility Matrix :

Transaction T₁ holds data item N in:

If transaction T_j requests data item N in:

Request j

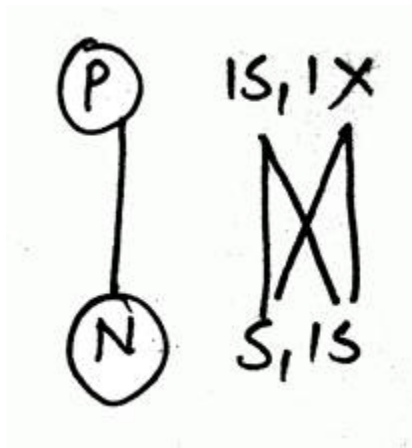
	IS	IX	S	SIX	X	→ Holds i
IS	T	T	T	T	F	
IX	T	T	F	F	F	
S	T	F	T	F	F	
SIX	T	F	F	F	F	
X	F	F	F	F	F	

What Does the Lock Compatibility Matrix Says ?

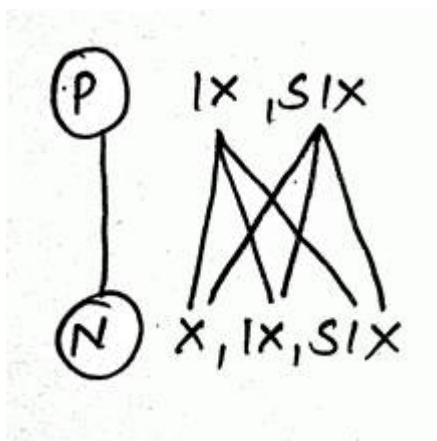
If a transaction T_i holds a data item N in IS' mode and transaction T_j requests the data item N in IS mode then T_j is allowed to lock it in IS mode. i.e. If T_i holds a data item N in IS mode means that there is some node below N in the tree which is in shared mode. And T_j requests the data item N in IS mode means that T_j is requesting some node below N in shared mode, then the request for T_j is allowed.

Guidelines or Rules for Locking

1. A node N can be locked by transaction T only if parent of N is already locked.
2. A node N can be locked by transaction T in S,IS mode only if parent of N is already locked by IX or IS mode.



3. A node N can be locked by transaction T in X, IX, SIX mode only if parent is already locked by IX or SIX mode by transaction T .



4. Transaction T requests for lock node N only if it has not locked any node.
5. A node N can be locked by two different transactions only if both locks are compatible.

6. A node N can be unlocked by transaction T only if none of the child is locked by transaction T.

Strict Multilevel Granularity Protocol

The **rule 7** (defined below) + **rules 1 to 6** (mentioned above for locking) are included in strict multilevel granularity. **Rule 7** : Hold exclusives locks until commit.

Timestamp Ordering Protocols

In timestamp based protocols, the system itself tries to detect possible inconsistency during concurrent execution and recovers from it or avoids it. In it, for every transaction the system executes, the system gives the timestamp to that transaction i.e. it provides a set of timestamps to every transaction T_i by unique timestamp (any integer value) which is denoted by $TS(T_i)$ where TS is timestamp of T_i .

What is $TS(T_i)$??

Whenever a transaction begins to execute that is just prior to its execution it is provided a timestamp. This timestamp may be a system related actual real time stamp which is based on the system time or it may just be a counter.

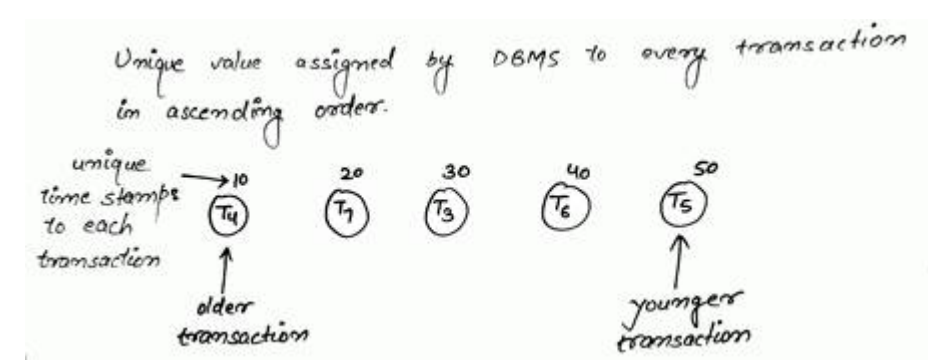
Example :

For example, Say a particular transaction T4 starts - counter = 1

When a transaction T7 starts - counter = 2

When a transaction T3 starts - counter = 3.

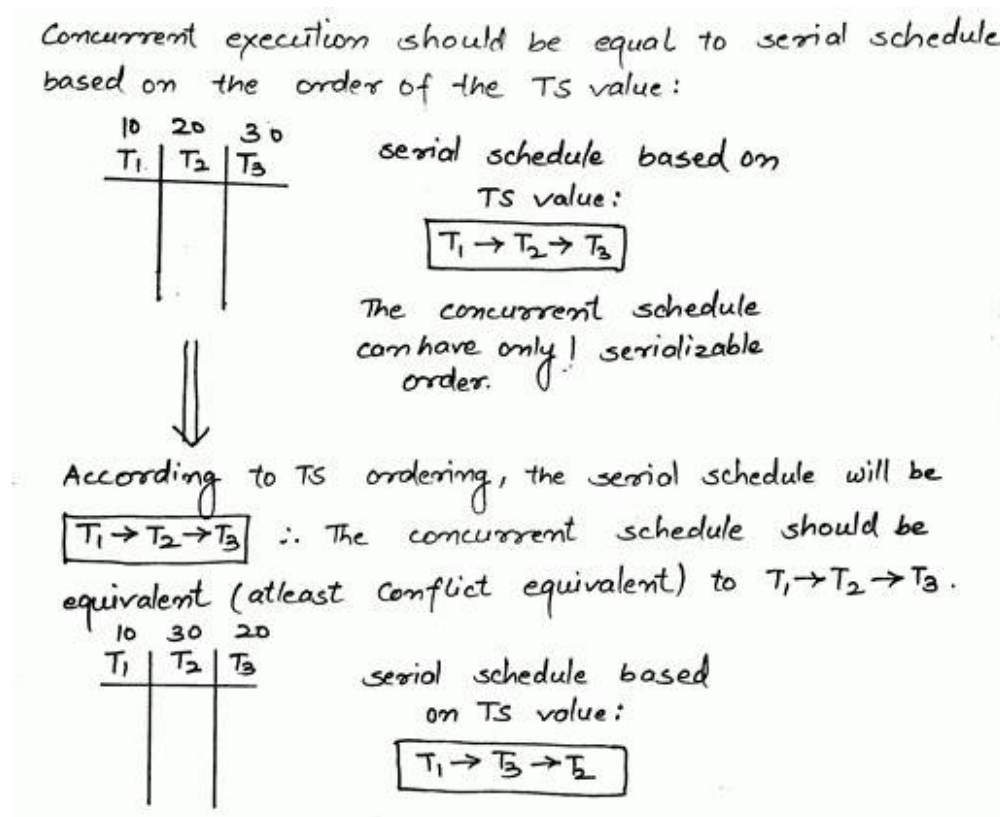
Whenever we want a transaction to execute the counter is incremented by 1. It is just a logical timing value which indicates that this transaction started before or another transaction.



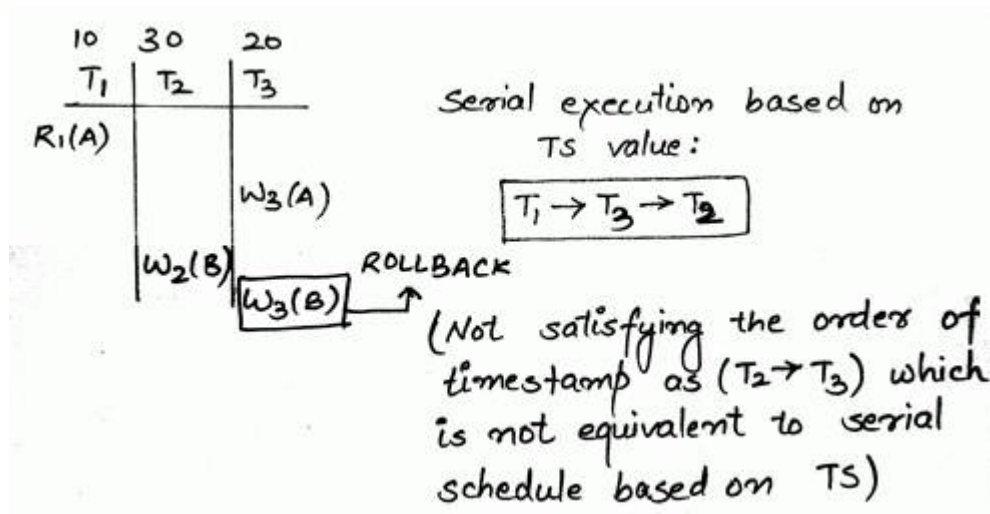
Therefore if we have,

$TS(T_i) < TS(T_j)$ then it means that transaction T_i starts its execution before T_j in the system.

- So a timestamp indicates when this transaction will start.
- The time stamp based protocols will try to check whether the concurrent execution may be consistent or not, or the schedule that occurs must be atleast conflict equivalent to T_i followed by T_j to the serial schedule T_i followed by T_j .



- Conflict Operations are allowed in the Timestamp ordering schedule, but they must have the order given by the timestamp. If they do not satisfy the order, then Rollback of transaction may or may not be performed.



What are R-Timestamp(RTS) and W-Timestamp(WTS)?

For Every data item which will be shared, we have got two timestamps :

- W-Timestamp(Q) or WTS(Q) : Write-timestamp of data item Q
- R-Timestamp(Q) or RTS(Q) : Read-timestamp of data item Q

W-Timestamp(Q)

W-Timestamp(Q) denotes the largest TS or TimeStamp value of any transaction that successfully executes Write(Q).

R-Timestamp(Q)

R-Timestamp(Q) denotes the largest TS or TimeStamp value of any transaction that successfully executes Read(Q).

Meaning of R-Timestamp and W-Timestamp :

Suppose there is a data item Q which is shared by transactions T1,T2,T3 and T4 with the below TimeStamps executing some read and write operations.

T1 : TS(1) = 10
 T2 : TS(2) = 20
 T3 : TS(3) = 30
 T4 : TS(4) = 40

10 T_1	20 T_2	30 T_3	40 T_4
$R(Q)$ $W(Q)$			
	$R(Q)$	$R(Q)$ $W(Q)$	

Let initially Q have no value i.e. 0. i.e.

- R-Timestamp = 0 or $RTS(Q) = 0$ and
- W-Timestamp = 0 or $WTS(Q) = 0$

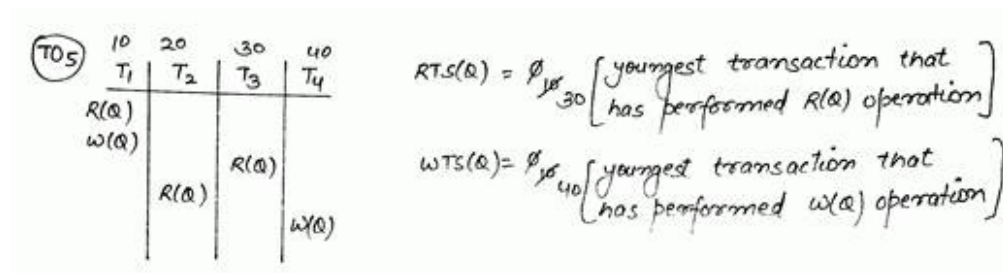
$RTS(Q) = 0$ and $WTS(Q) = 0$ means that no transaction has read or write the data item Q.

When Transaction issues Read(Q) :

- Suppose T_1 executes Read(Q) whose $TS(T_1) = 10$, then we will have $RTS(Q) = 10$.
- Then, T_3 executes Read(Q) whose $TS(T_3) = 30$, then the value of $RTS(Q)$ will remain 10.
- Now, if T_2 executes Read(Q) whose $TS(T_2) = 20$, then the value of $RTS(Q)$ will remain 30.

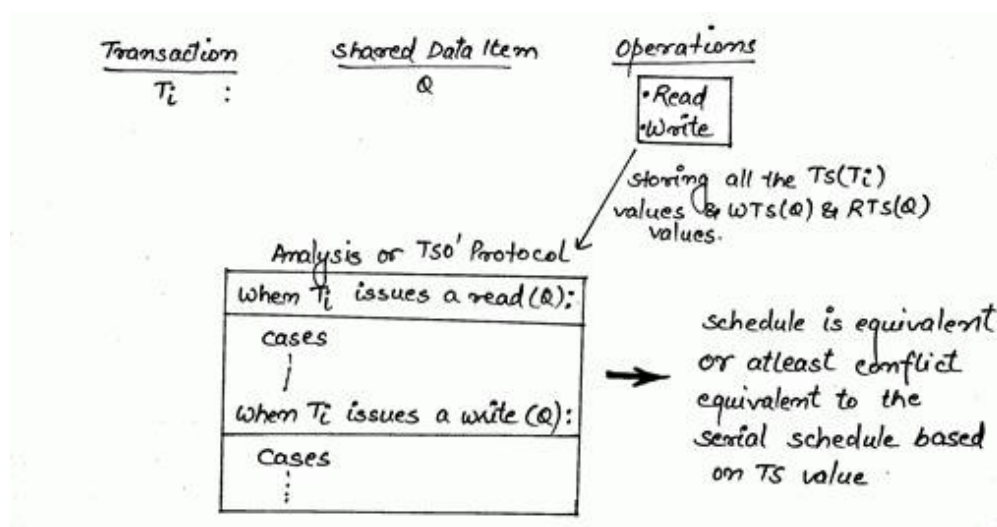
When Transaction issues Write(Q) :

- Suppose T_1 executes Write(Q) whose $TS(T_1) = 10$, then we will have $WTS(Q) = 10$.
- Then, T_4 executes Write(Q) whose $TS(T_4) = 40$, then the value of $WTS(Q)$ will remain 40.



How will we Utilize the Idea of R-Timestamp and W-Timestamp ?

- Suppose transaction T_i is executed which is using shared data. Whenever a read or write is executed by transaction T_i , the concurrency control mechanisms immediately check certain things.
 - "Whenever transaction T_i starts, the concurrency control mechanism sets the new $TS(T_i)$ value and it stores all the $TS(T_i)$ values and all the W-Timestamp values and the R-Timestamp values of the required data."
- Based on this, it does some analysis (called the Timestamping Protocol) and based on this analysis it tries to check whether there can be any inconsistency in the behavior or whether the schedule which it executing is equivalent to the serial schedule to the required serial schedule or not.



- And if it is not, then some corrective actions will takes place by the system itself.

Analysis or TSO Protocol :

When T_i issues a Read(Q) :

Suppose a transaction T_i issues Read(Q) where Q is a shared data item. Now, the TS(T_i) value will be compared with W-Timestamp(WTS) and R-Timestamp(RTS) of Q. Let W-Timestamp(Q) and R-Timestamp(Q) be the start timestamp of a transaction T_j i.e.

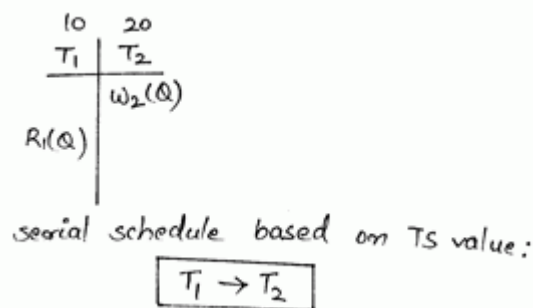
$$TS(T_j) = WTS(Q)$$

and

$$TS(T_j) = RTS(Q)$$

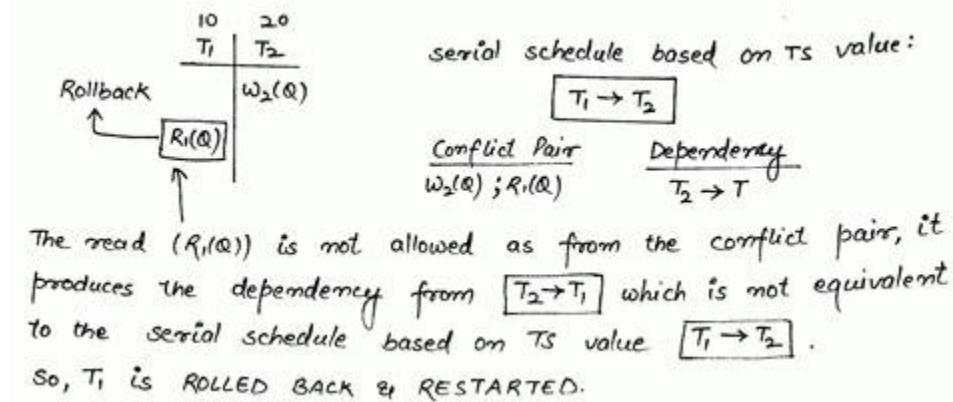
Case 1 : If $TS(T_i) < WTS(Q)$?

- **What does the statement means ?** It means that there is a transaction T_j which has written on data item Q before T_i reads it and the transaction T_j started after transaction T_i .
 - - According to the condition, transaction T_2 is started after transaction T_1 and T_2 is going to write on the data item Q which is read by T_1 .
 - According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
 - But there is a
 - **Conflict Pair** : $W_2(Q); R_1(Q)$
 - **Dependency** : $T_2 \rightarrow T_1$
 - Therefore, the schedule will **never ever be equivalent(at least conflict equivalent)** to the serial schedule $T_1 \rightarrow T_2$. The possible situation is described as with the help of an example.



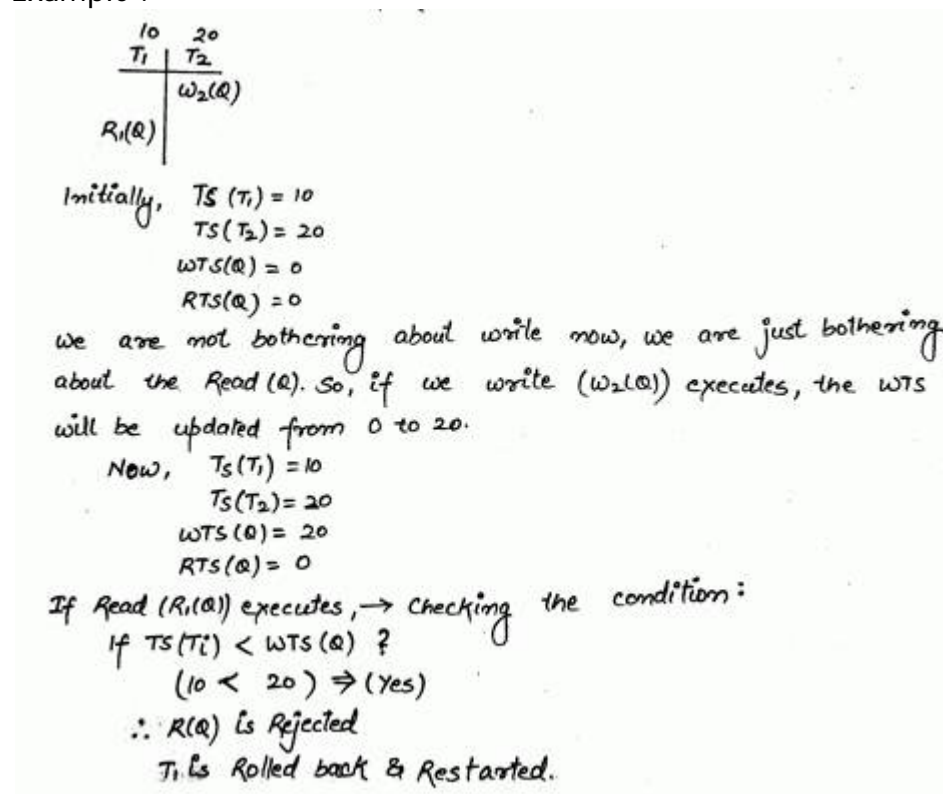
What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will reject the read. This read cannot be allowed because T_1 is trying to read a data item which has been written by transaction T_2 and T_2 is started after T_1 .
- The schedule maybe consistent but it will not be conflict serializable at all to T_1 followed by T_2 ($T_1 \rightarrow T_2$). So in general, T_i will be Rolled Back and so this read will be rejected.



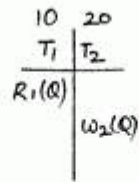
How do we proceed with the transaction T_i or T_1 ?
 ROLLBACK T_i or Say T_1 . RESTART. i.e.
 T_i or T_1 is rolled back and restarted.

Example :



Case 2 : If $TS(T_i) \geq WTS(Q)$?

- **What does the statement means ?** It means that there is a transaction T_j which has written on data item Q after T_i reads it and the transaction T_j started before transaction T_i .
- The possible situation is described as with the help of an example.



serial schedule based on TS value:

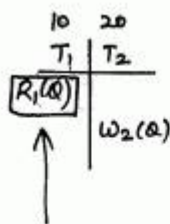
$T_1 \rightarrow T_2$

- According to the condition, transaction T_2 is started before transaction T_1 and T_2 is going to write on the data item Q which is already read by T_1 .
 - According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
 - The Conflict Pair will be :
 - Conflict Pair** : $R_1(Q); W_2(Q)$
 - Dependency** : $T_1 \rightarrow T_2$ as shown.
 - Therefore, the schedule is **conflict equivalent** to the serial schedule $T_1 \rightarrow T_2$.

What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will allow the read by transaction T_1 or say T_1 . and
- It sets the R -Timestamp(Q) as :

$$RTS(Q) = \max(RTS(Q), TS(T_1))$$



serial schedule based on TS value:

$T_1 \rightarrow T_2$

Conflict Pair
 $R_1(Q); W_2(Q)$

Dependency
 $T_1 \rightarrow T_2$

This Read ($R_1(Q)$) is allowed as from the conflict pair, it produces the dependency from $T_1 \rightarrow T_2$ which is equivalent to the serial schedule based on TS value:

$T_1 \rightarrow T_2$

So, Read is allowed by transaction T_1 &

$$RTS(Q) = \max\left(\frac{RTS(Q)}{0}, \frac{TS(T_1)}{10}\right)$$

$$RTS(Q) = 10$$

Why there is an equal to condition in $TS(T_i) \geq WTS(Q)$?

The reason of the equal to condition is that T_i or T_1 itself may have written.

Example :

10 T_1	20 T_2
$W_1(Q)$	
$R_1(Q)$	$W_2(Q)$

Initially, $TS(T_1) = 10$ $RTS(Q) = 0$
 $TS(T_2) = 20$ $WTS(Q) = 0$

Again, we are not bothering about write statements.
 If write ($W_1(Q)$) executes, then the value will be:

$TS(T_1) = 10$ $RTS = 0$
 $TS(T_2) = 20$ $WTS = \emptyset, 10$

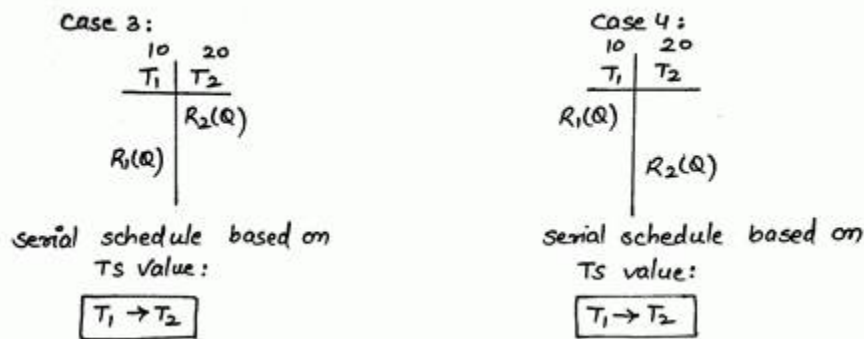
If Read ($R_1(Q)$) executes, \rightarrow checking the condition
 If $[TS(T_1) \geq WTS(Q)]$?
 $[10 \geq 10] \Rightarrow \text{Yes}$

\therefore Read ($R_1(Q)$) is allowed &
 set $RTS(Q) = \max\left(\frac{RTS(Q)}{10}, \frac{TS(T_1)}{10}\right)$

$RTS(Q) = 10$

Case 3 & 4 : If $TS(T_i) < RTS(Q)$? or If $TS(T_i) < RTS(Q)$?

- **Case 3 : What does the statement $TS(T_i) < RTS(Q)$ means?** It means that there is a transaction T_j which reads a data item Q before T_i reads it and the transaction T_j started after transaction T_i .
- **Case 4 : What does the statement $TS(T_i) > RTS(Q)$ means?** It means that there is a transaction T_j which reads a data item Q after T_i reads it and the transaction T_j started before transaction T_i .
- In both cases 3 & 4, transactions T_i and T_j - both are reading. So, there will be no conflict pair exists.
- The possible situation is described as with the help of an example.

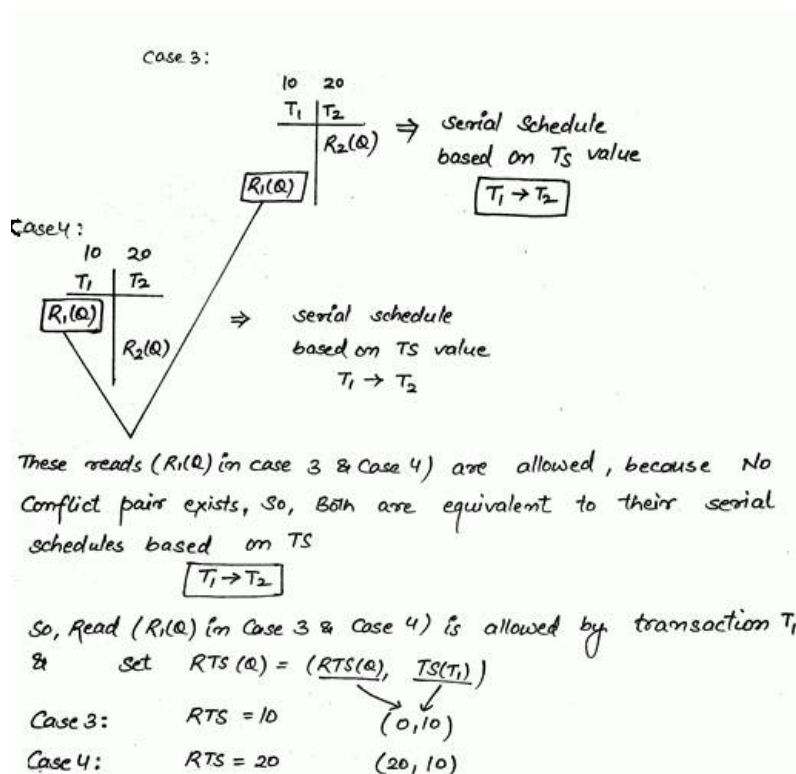


- According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
- Since no Conflict Pair will exist as both transactions (T_1 and T_2) are reading in both the cases 3 & 4.
- Therefore, the schedule is **conflict equivalent** to the serial schedule $T_1 \rightarrow T_2$.

What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will allow the read by transaction T_i or say T_1 . and
- It sets the R-Timestamp(Q) as :

$$RTS(Q) = \max (RTS(Q), TS(T_1))$$



Example for Case 4 :

Example for Case 3: when $TS(T_i) > RTS(Q)$

10	20
T_1	T_2
	$R_2(Q)$
$R_1(Q)$	

Initially, $TS(T_1) = 10$ $RTS(Q) = 0$
 $TS(T_2) = 20$ $WTS(Q) = 0$

Executing $R_2(Q) \Rightarrow$

We are checking with $RTS(Q)$ only, \therefore there is no right performed in the schedule.

So, $[TS(T_2) > RTS(Q)]$ or $[TS(T_2) < RTS(Q)]$

In both the cases, Read is allowed $\therefore (20 > 0) \Rightarrow \text{True}$

\therefore Read ($R_2(Q)$) is allowed

& Set $RTS(Q) = \max(RTS(Q), TS(T_2)) = (0, 20) = 20$

Let's check $R_1(Q)$, Executing $R_1(Q) \downarrow$

$TS(T_1) = 10$ $RTS(Q) = 20$
 $TS(T_2) = 20$ $WTS(Q) = 0$

Now,

$[TS(T_1) > RTS(Q)]$ or $[TS(T_1) < RTS(Q)]$

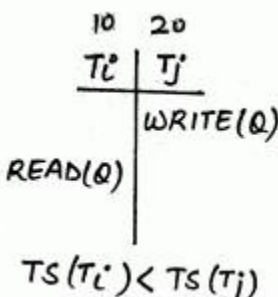
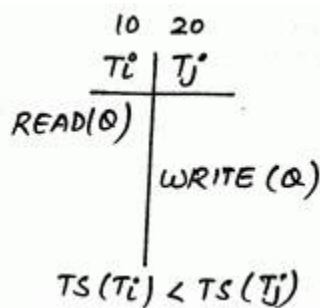
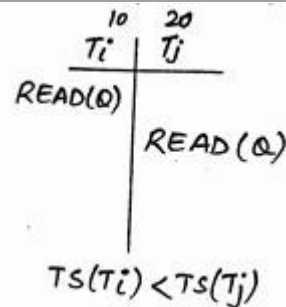
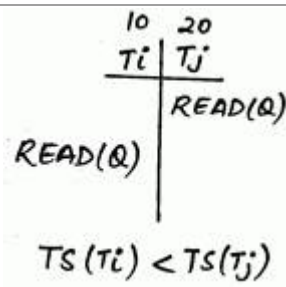
Again, in both the cases, Read is allowed.

$\therefore (10 < 20) \Rightarrow \text{True}$, \therefore Read ($R_1(Q)$) is also allowed.

& Set $RTS(Q) = \max[RTS(Q), TS(T_1)]$
 $= \max(20, 10)$

$RTS(Q) = 20$

The above Cases are summarized into the following table :

When Ti issues a READ(Q) :			
Cases	Meaning of the Cases	Situation Diagram	Solution
Case 1 : $TS(T_i) < WTS(Q)$	Tj has started after Ti and Tj has written before Ti reads it		Reject the READ(Q). RollBack Ti. RESTART.
Case 2 : $TS(T_i) \geq WTS(Q)$	Tj has started before Ti and Tj has written after Ti reads it		
Case 3 : $TS(T_i) < RTS(Q)$	Tj has started after Ti and Tj has read before Ti reads it		READ is Allowed By Transaction Ti. & Set (RTS(Q)) = max(RTS(Q), TS(Ti))
Case 4 : $TS(T_i) > RTS(Q)$	Tj has started before Ti and Tj has read after Ti reads it		

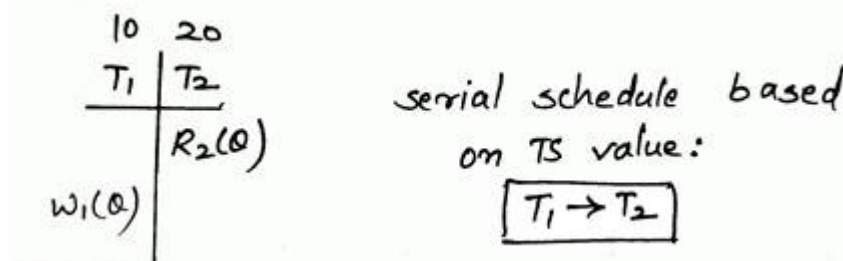
When T_i issues a Write(Q) :

Suppose a transaction T_i issues Write(Q) where Q is a shared data item. Now, the TS(T_i) value will be compared with W-Timestamp(WTS) and R-Timestamp(RTS) of Q. Let W-Timestamp(Q) and R-Timestamp(Q) be the start timestamp of a transaction T_j i.e.

$$\begin{aligned} TS(T_j) &= WTS(Q) \\ \text{and} \\ TS(T_j) &= RTS(Q) \end{aligned}$$

Case 1 : If $TS(T_i) < RTS(Q)$?

- **What does the statement means ?** It means that there is a transaction T_j which reads a data item Q before T_i writes it and the transaction T_j started after transaction T_i .
- The possible situation is described as with the help of an example.



- According to the condition, transaction T_2 is started after transaction T_1 and T_2 is going to read the data item Q which is write by T_1 .
- According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
- But there is a
 - **Conflict Pair** : $R_2(Q); W_1(Q)$
 - **Dependency** : $T_2 \rightarrow T_1$ as shown.
- Therefore, the schedule will **never ever be equivalent(atleast conflict equivalent)** to the serial schedule $T_1 \rightarrow T_2$.

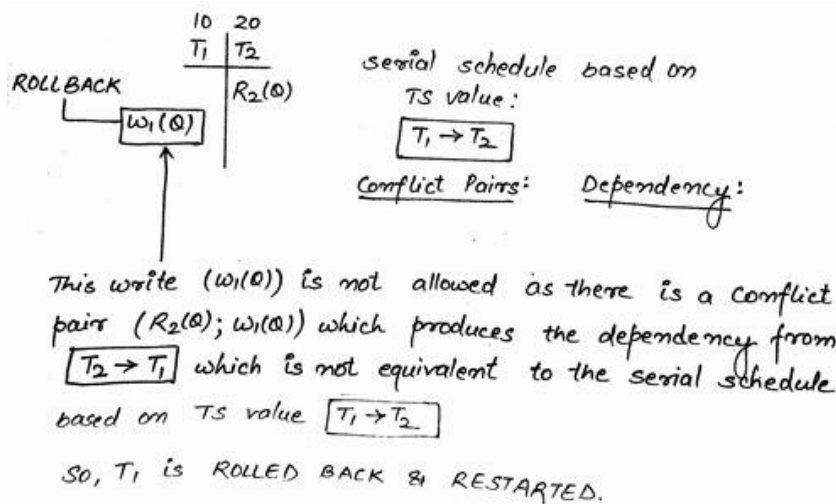
What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will reject the write.
- The schedule maybe consistent but it will **not be conflict serializable** at all to T_1 followed by T_2 ($T_1 \rightarrow T_2$). So in general, T_i will be Rolled Back and so this

write will

be

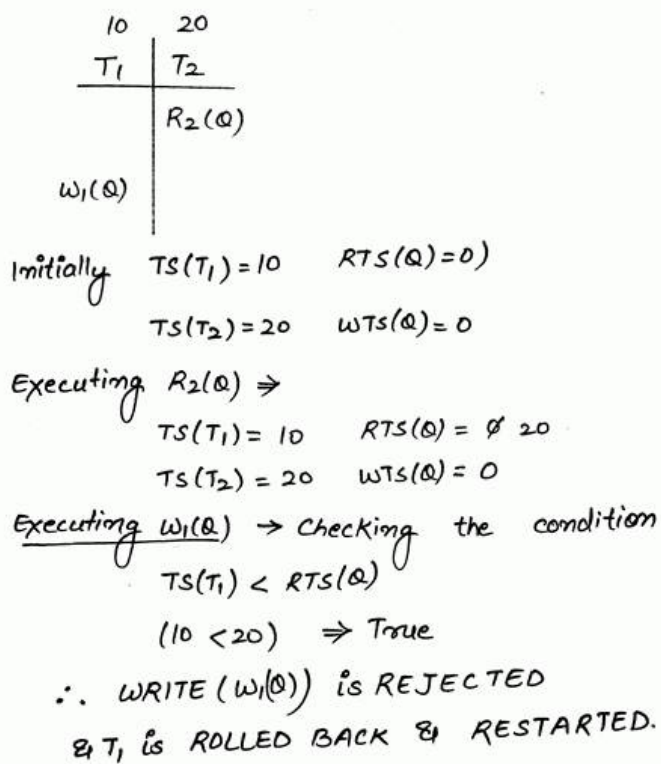
rejected.



How do we proceed with the transaction T_i or T_1 ?

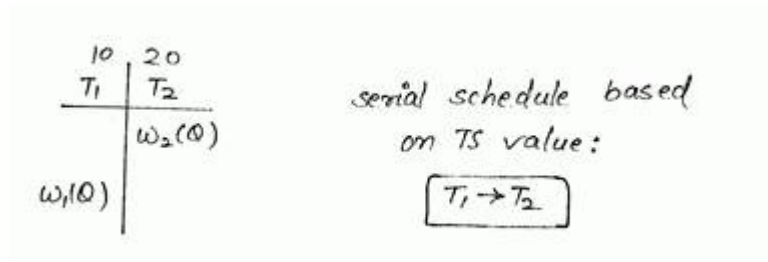
ROLLBACK T_i or Say T_1 . RESTART. i.e.
 T_i or T_1 is rolled back and restarted.

Example :



Case 2 : If $TS(T_i) < WTS(Q)$?

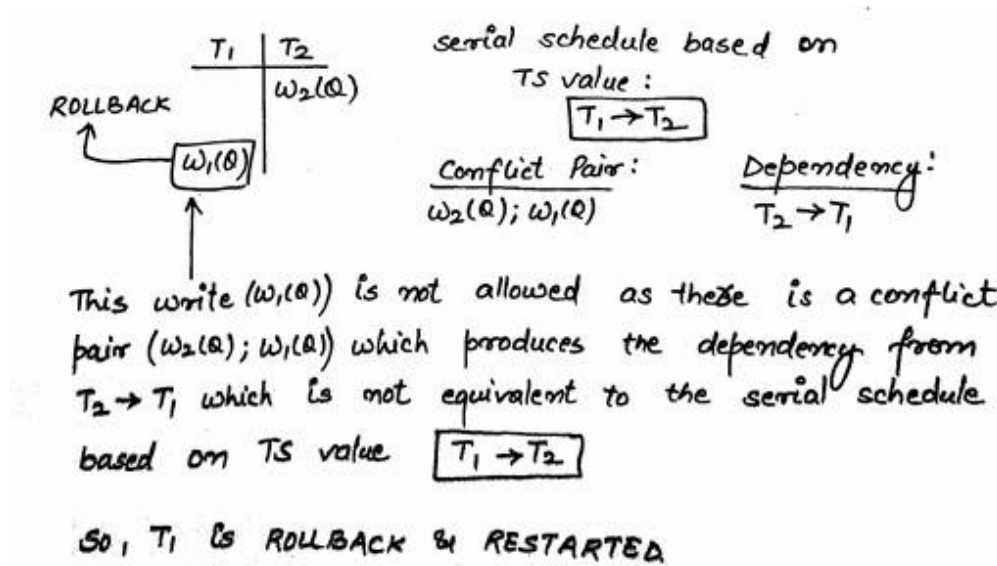
- **What does the statement means ?** It means that there is a transaction T_j which has written on data item Q before T_i writes it and the transaction T_j started after transaction T_i .
- The possible situation is described as with the help of an example.



- According to the condition, transaction T_2 is started after transaction T_1 and T_2 writes on the data item Q which is again write by T_1 .
- According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
- The Conflict Pair will be :
 - **Conflict Pair** : $w_2(Q); w_1(Q)$
 - **Dependency** : $T_2 \rightarrow T_1$ as shown.
- Therefore, the schedule will **never ever be equivalent(atleast conflict equivalent)** to the serial schedule $T_1 \rightarrow T_2$.

What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will reject the write.
- The schedule maybe consistent but it will not be conflict serializable at all to T_1 followed by T_2 ($T_1 \rightarrow T_2$). So in general, T_i will be Rolled Back and so this write will be rejected.



Example :

10	20
T_1	T_2
$w_1(Q)$	$w_2(Q)$

Initially $TS(T_1) = 10$ $RTS(Q) = 0$
 $TS(T_2) = 20$ $RTS(Q) = 0$

Executing $w_2(Q) \Rightarrow$

Comparing value with $WTS(Q)$ & $RTS(Q)$:

$TS(T_2)$ with $RTS(Q) = (0 < 20)$ }
 $TS(T_2)$ with $WTS(Q) = (0 < 20)$ }

write (Q) is allowed.

Now values are updated as:

$TS(T_1) = 10$ $RTS(Q) = 0$
 $TS(T_2) = 20$ $WTS(Q) = 0$

Executing $w_1(Q) \rightarrow$ is ~~is~~ checking the condition
 $TS(T_1) < WTS(Q)?$

$(10 < 20) \neq \text{True}$

$\therefore \text{WRITE } (w_1(Q)) \text{ is REJECTED}$

& T_1 is ROLLED BACK & RESTARTED

Case 3 : If $TS(T_i) > RTS(Q)$?

- **Case 3 : What does the statement $TS(T_i) > RTS(Q)$ means?** It means that there is a transaction T_j which reads a data item Q after T_i writes it and the transaction T_j started before transaction T_i .
- The possible situation is described as with the help of an example.

10	20
T_1	T_2
$w_1(Q)$	$R_2(Q)$

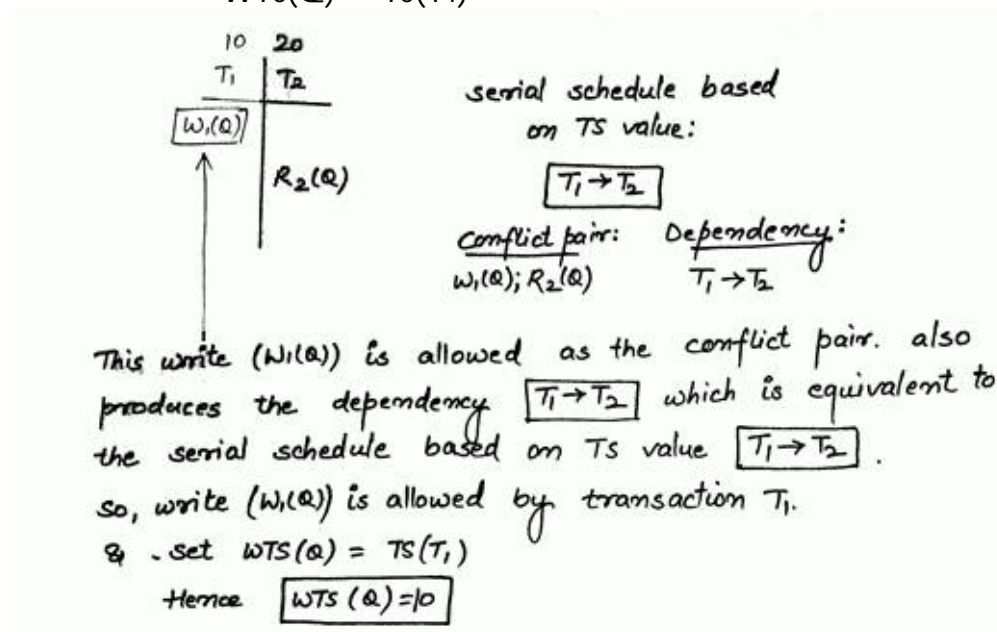
serial schedule based
on TS value :

$T_1 \rightarrow T_2$

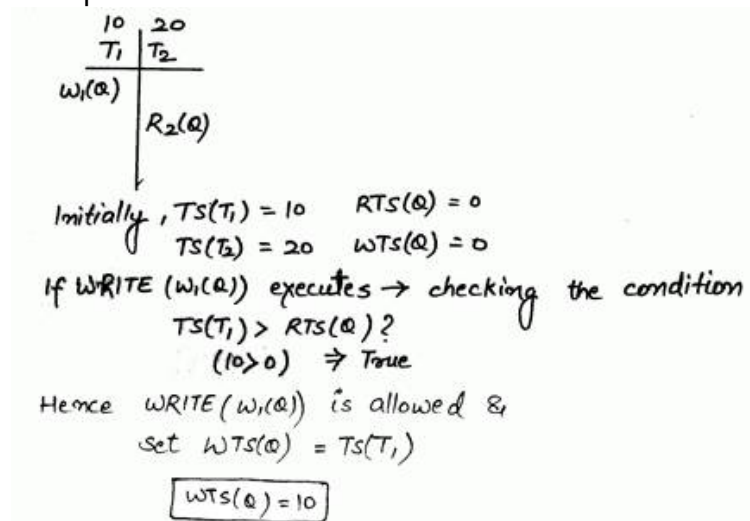
- According to the condition, transaction T2 is started before transaction T1 and T2 reads the data item Q written by T1.
- According to Timestamps defined - the serial schedule will be $T1 \rightarrow T2$.
- The Conflict Pair will be :
 - **Conflict Pair** : $W1(Q); R2(Q)$
 - **Dependency** : $T1 \rightarrow T2$ as shown.
- Therefore, the schedule is **conflict equivalent** to the serial schedule $T1 \rightarrow T2$.

What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will allow the write by transaction T1 or say T1. and
- It sets the W-Timestamp(Q) as :
 $WTS(Q) = TS(T1)$

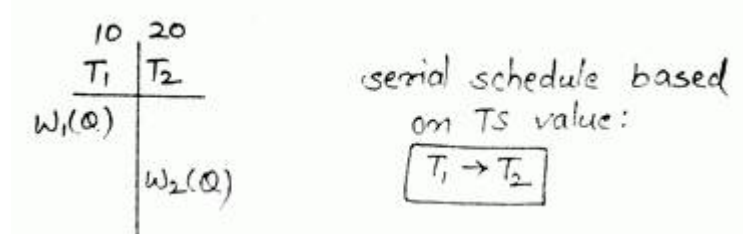


Example :



Case 4 : If $TS(T_i) > WTS(Q)$?

- **Case 4 : What does the statement $TS(T_i) > WTS(Q)$ means?** It means that there is a transaction T_j which writes a data item Q after T_i writes it and the transaction T_j started before transaction T_i .
- The possible situation is described as with the help of an example.



- According to the condition, transaction T_2 is started before transaction T_1 and T_2 writes the data item Q which is already written by T_1 .
- According to Timestamps defined - the serial schedule will be $T_1 \rightarrow T_2$.
- The Conflict Pair will be :
 - **Conflict Pair** : $W_1(Q); W_2(Q)$
 - **Dependency** : $T_1 \rightarrow T_2$ as shown.
- Therefore, the schedule is **conflict equivalent** to the serial schedule $T_1 \rightarrow T_2$.

What happens when concurrency control protocol mechanism detects the situation?

- The concurrency control protocol mechanism will detect the situation and it will allow the write by transaction T_i or say T_1 . and
- It sets the W-Timestamp(Q) as :
- $WTS(Q) = TS(T_1)$

10	20
T_1	T_2
$w_1(Q)$	
	$w_2(Q)$

serial schedule based
on TS value

$$\boxed{T_1 \rightarrow T_2}$$

Conflict pairs:
 $w_1(Q); w_2(Q)$

Dependency:
 $T_1 \rightarrow T_2$

The WRITE ($w_1(Q)$) is allowed as the conflict pair also produces the dependency $\boxed{T_1 \rightarrow T_2}$ which is equivalent to the serial schedule based on TS value $\boxed{T_1 \rightarrow T_2}$

so WRITE ($w_1(Q)$) is allowed by transaction T_1 &
set $WTS(Q) = TS(T_1)$

Hence $\boxed{WTS(Q) = 10}$

Example :

10	20
T_1	T_2
$w_1(Q)$	
	$w_2(Q)$

Initially $TS(T_1) = 10$ $RTS(Q) = 0$
 $TS(T_2) = 20$ $WTS(Q) = 0$

If WRITE ($w_1(Q)$) executes \rightarrow check the condition
 $TS(T_1) > WTS(Q)?$

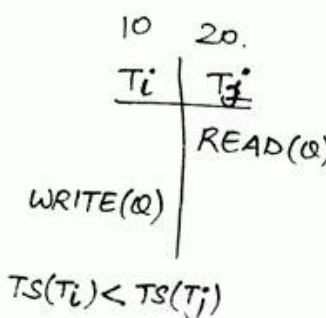
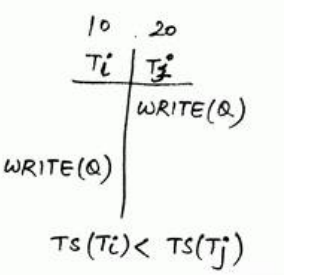
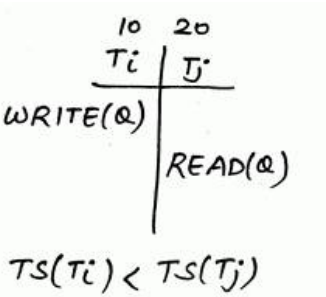
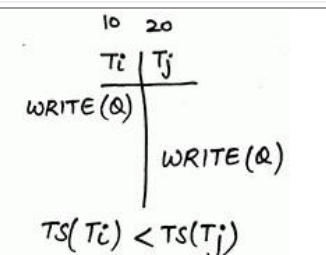
$(10 > 0) \Rightarrow \text{True.}$

Hence WRITE ($w_1(Q)$) is allowed

& set $WTS(Q) = TS(T_1)$

Hence $\boxed{WTS(Q) = 10}$

The above Cases are summarized into the following table :

When T_i issues a $WRITE(Q)$:			
Cases :	Meaning of the Cases	Situation Diagram	Solution
Case 1 : T_j has started after T_i $TS(T_i) < RTS(Q)$ and T_j has read before T_i writes it		 <p>$TS(T_i) < TS(T_j)$</p>	Reject the $Write(Q)$ by Transaction T_i . Rollback T_i Restart T_i.
Case 2 : T_j has started after T_i $TS(T_i) < WTS(Q)$ and T_j has written before T_i writes it		 <p>$TS(T_i) < TS(T_j)$</p>	
Case 3 : T_j has started before T_i $TS(T_i) > RTS(Q)$ and T_j has read after T_i writes it		 <p>$TS(T_i) < TS(T_j)$</p>	$Write(Q)$ is Allowed By Transaction T_i .
Case 4 : T_j has started before T_i $TS(T_i) > WTS(Q)$ and T_j has write after T_i writes it		 <p>$TS(T_i) < TS(T_j)$</p>	& Set $(WTS(Q)) = TS(T_i)$

Points About Timestamp Ordering :

Timestamp Based Ordering Protocol is :

- Conflict serializable i.e. it is conflict equivalent to some serial schedule.
- It is deadlock free because timestamp ordering protocol allows rollback and restart it after detecting a mismatched situation.

Additional Overheads as Compared to Lock Based Protocol

- Overheads of maintaining the timestamps - RTS and WTS
- Checking each transaction whenever a read or write operation is executed.
- Updating each transaction after execution of read and write operation.

Advantages over overheads :

- However, there are a lot of overhead but the advantage is the user need not to be bother at all about what is going on.
- But in Lock Based Protocols - the overheads are of waiting and the responsibility is of the user to write the consistent concurrent transaction.
- Most widely used protocol is timestamp ordering protocol. Timestamp ordering protocol is a system automated protocol for concurrency control.

Thomas write rule

Thomas write rule modify or **improves** the **Basic Timestamp Ordering Algorithm (BTSO Algorithm)**.

According to Basic Timestamp Ordering Algorithm (BTSO Algorithm) :

1. **When Transaction T_i issues READ Operation :**
 - a. **If $TS(T_i) < WTS(Q)$, then**
 - ROLLBACK T_i .
 - RESTART T_i
 - b. **Otherwise,**
 - allowed to execute READ operation by transaction T_i and
 - Set $RTS(Q) = \max(RTS(Q), TS(T_i))$
2. **When Transaction T_i issues WRITE Operation :**
 - a. **If $TS(T_i) < RTS(Q)$, then**
 - ROLLBACK T_i .
 - RESTART T_i .
 - b. **If $TS(T_i) < WTS(Q)$, then**
 - ROLLBACK T_i .
 - RESTART T_i .

c. **Otherwise,**

- Allowed to execute the WRITE operation by transaction T_i and
- Set $WTS(Q) = TS(T_i)$

Criteria For Modification :

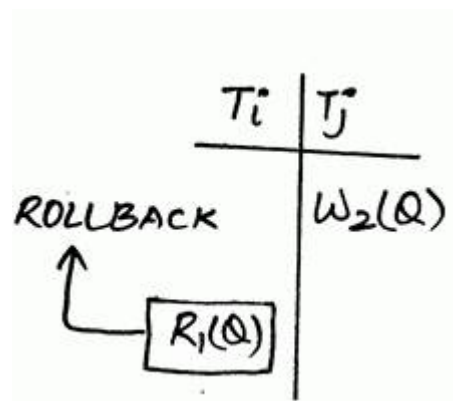
Before improving the BTSO protocol, it must follow some criteria :

- The **First Criteria** used to maintain the consistency of the system.
- After maintaining the consistency of the system, the **next criteria** is to try to see whether we can enhance the concurrency of the mechanism.

Modification in BTSO Protocol

Let us discuss the cases one by one :

1. **When T_i issues a READ ($R(Q)$) :**



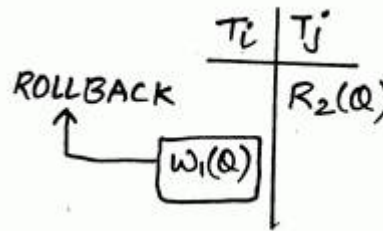
a. **Case (a) :** If $TS(T_i) < WTS(Q)$?

- Transaction T_i must have rolled back as transaction T_i is trying to read a data item Q which is written by transaction T_j which started after T_i .
- If we allow this, then we have thousands of examples, in which this case will lead to problem (As we are not analyzing what the code exactly is. The code may or may not have the problem.)
- So no modification can be done in this case.

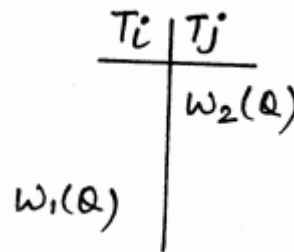
b. **Case (b) :** If $TS(T_i) \geq WTS(Q)$?

- The read is allowed in the otherwise case.
- So no need of modification.

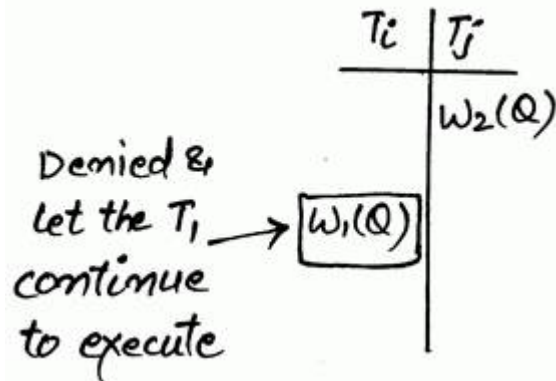
2. When T_i issues a WRITE ($W(Q)$) :



- a. **Case (a) :** If $TS(T_i) < RTS(Q)$?
 - In this case also transaction T_i must have rolled back as transaction T_i is trying to write data item Q which is already read by a transaction T_j .
 - If we allow this, then the schedule will not be conflict serializable at all.
- b. **Case (B) :** $TS(T_i) < WTS(Q)$



- This case can be improved as :
- In this case T_j writes a data item Q which is again updated by T_i and T_j starts after T_i .
- We cannot allow the WRITE ($W_1(Q)$) statement because it will never be conflict serializable.
- **But the question is do we have to rollback T_i ?**
- In BTSSO protocol we do not allow to execute the write statement and rolled back T_i .
- The modification is done as : We do not allow to execute the write statement by T_i but we do not rollback T_i . We just continue to execute with T_i .
- This is because any updates made by transaction T_j (timestamp greater than T_i) has already written the value of Q and any updates made by T_j will be lost.
- Therefore we must ignore the write operation of T_i because it is already outdated and obsolete.
- This way, it prevents a rollback in this situation as late as possible and therefore tries to reduce the amount of rollback.



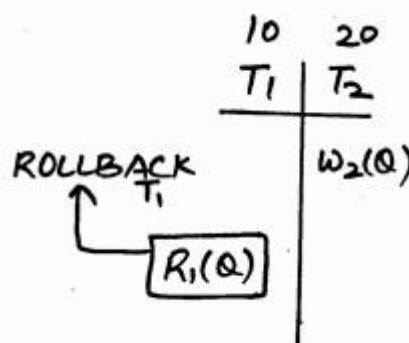
- c. **Case (c)** : The otherwise case $\{TS(T_i) > RTS(Q) \text{ and } TS(T_i) > WTS(Q)\}$
 - The write is allowed in the otherwise case. So no need of modification.

Thomas's Write Rule Timestamp Ordering (TWRTSO)

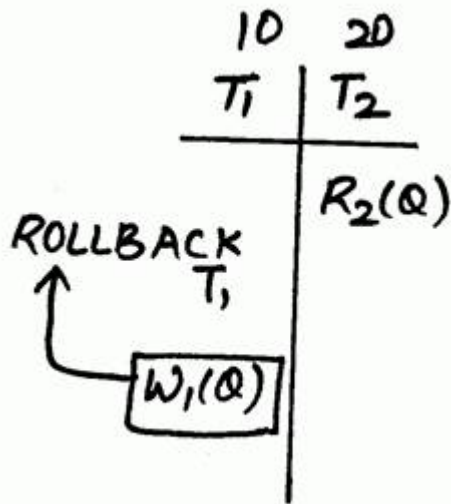
Modification in case (B) when transaction T_i issues a write operation is called **Thomas write rule**. It rejects fewer write operations, by modifying the checks for write operation.

Thomas's Write Rule Timestamp Ordering Protocol (TWRTSO Protocol)

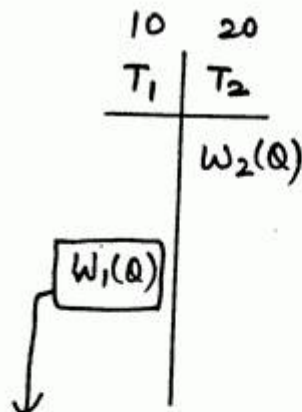
1. **When Transaction T_i issues READ Operation :**
 - a. If $TS(T_i) < WTS(Q)$, then



- ROLLBACK T_i .
- b. **Otherwise,**
 - allowed to execute READ operation by transaction T_i and
 - Set $RTS(Q) = \max(RTS(Q), TS(T_i))$
- 2. **When Transaction T_i issues WRITE Operation :**
 - a. If $TS(T_i) < RTS(Q)$, then



- ROLLBACK T_i .
- b. If $TS(T_i) < WTS(Q)$, then
 - Ignore WRITE operation by T_i and
 - Continue the execution of T_i .



Denied $W_1(Q)$ operation & continue the execution of T_1 .
 (Just ignore the request of older transaction).

Always the younger transaction (T_j) should make final updation

c. Otherwise,

- Allowed to execute the WRITE operation by transaction T_i and
- Set $WTS(Q) = TS(T_i)$

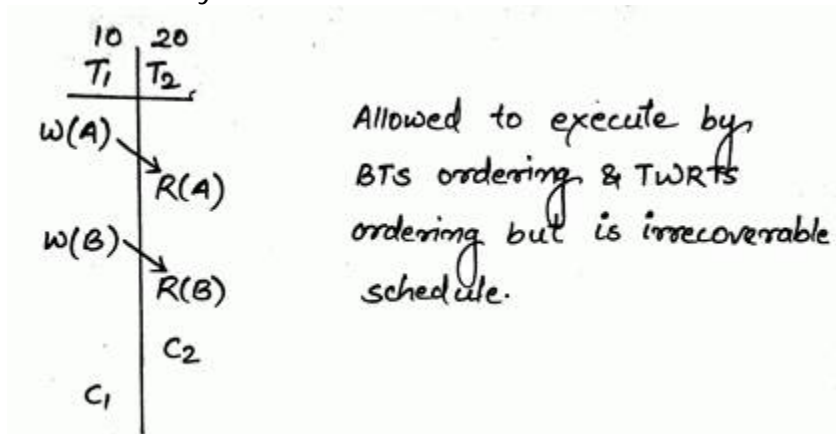
Thomas Write Rule Timestamp Ordering is :

- Deadlock Free
- Ensures Serializability (equivalent serial schedule based of the order of TS value)

Problems in Thomas Write Rule Timestamp Ordering

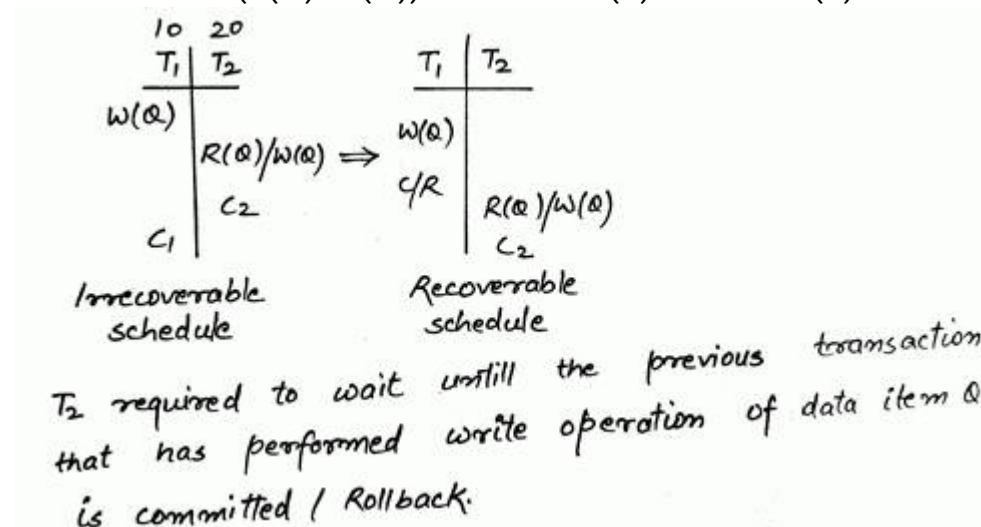
- Starvation may be possible.
- Irrecoverability

Possible.



Strict TSO Protocol : To Avoid Irrecoverability

Concurrent schedule should be equivalent to serial schedule based on TS ordering and If transaction T_i updates data item Q , other transaction T_j is not allowed to read or write on data item Q ($R(Q)/W(Q)$) until commit(C) or rollback(R) of T_i .

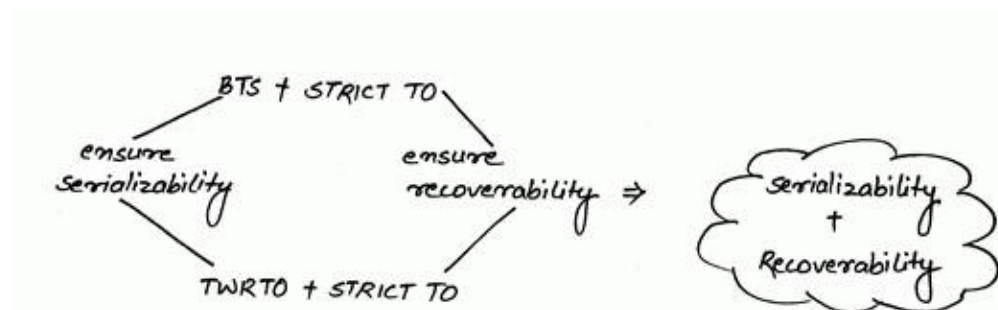


Strict Timestamp Ordering Schedule is :

- Strict Recoverable
- Deadlock Free
- Starvation still Possible

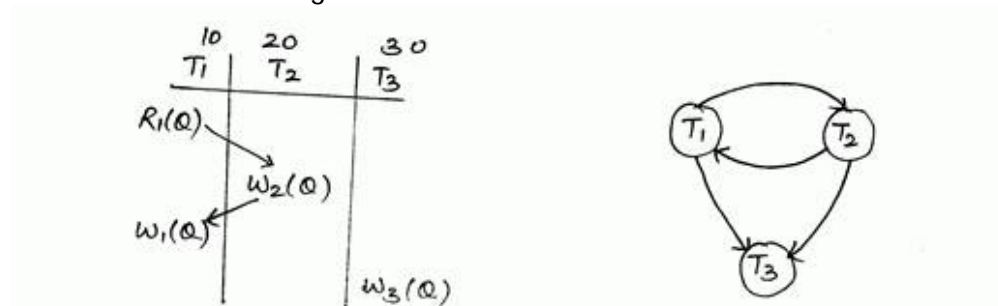
Combining :

Basic Timestamp Ordering + Strict TSO Protocol or Thomas Write Rule Timestamp Ordering + Strict TSO Protocol



View/Conflict Serializability and Timestamp Ordering :

Consider the following Scenario :

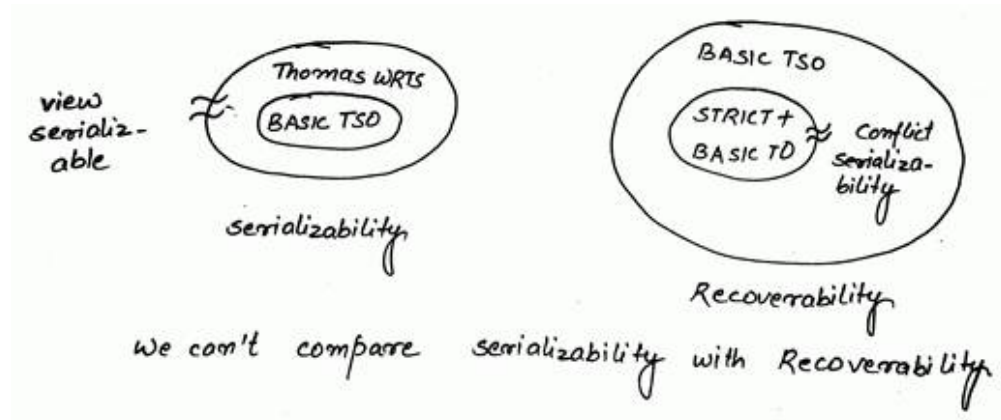


The schedule is Not Conflict Serializable but it is view serializable schedule.

Using BTS : Rollback T_1 because of $w_1(Q)$
ordering

using TWRTO : No ROLLBACKS, Denied $w_1(Q)$ but allowed to execute transaction T_1 .
ordering
Equivalent serial schedule:
 $T_1 \rightarrow T_2 \rightarrow T_3$

If schedule is view serializable schedule and view equivalent serial schedule is based on timestamp value, then Thomas Write Rule Timestamp Ordering Protocol allow to execute the schedule.



Relation between Thomas Write Rule Timestamp Ordering, Basic Timestamp Ordering, and Strict TSO Protocol :

