Transactions And Concurrency Management

Unit 2

Contents

- Transactions : properties, states
- Concurrent transactions
- Concurrency control : schedules
- Optimistic concurrency control / optimistic scheduling

Transactions

- A transaction is defined as unit of work in a database system.
- Database systems that deal with large number of transactions is called transaction processing systems.
- A transaction is a unit of data processing.
- Examples of transactions at a bank: withdrawal or deposit of money, transfer of money from one account to another account.
- A transaction would involve manipulation of one or more values in database. Some transactions write data value without reading a data value.

Pseudo Code For A Transaction

- Example 1 : Transaction to withdraw money from an account 'X'.

```
TRANSACTION WITHDRAWAL(withdrawal_amount)
Begin transaction
    IF X exist then
        READ X.balance
        If X.balance > withdrawal_amount
        THEN SUBTRACT withdrawal_amount
         WRITE X.balance
        COMMIT
```

```
            ELSE
            DISPLAY "TRANSACTION CANNOT BE PROCESSED"
        ELSE DISPLAY "ACCOUNT X DOES NOT EXIST"
End transaction
```

- Example 2 : Pseudo code for transaction to transfer amount from one account to another account

```
TRANSACTION (x, y, transfer_amount)
Begin transaction
        If X AND Y exist then
If X AND Y exist then
              READ x.balance
              If x.balance > transfer_amount THEN
 x.balance=x.balance-transfer_amount
              READ y.balance
                  y.balance=y.balance+transfer_amount
```

```
            COMMIT
             ELSE DISPLAY ("BALANCE IN X NOT OK")
            ROLLBACK
        ELSE DISPLAY ("ACCOUNT X OR Y DOES NOT EXIST")
End transaction
```

- COMMIT makes sure that all the changes made by transactions are made permanent.
- ROLLBACK terminates the transactions and rejects any changes made by the transaction.

Properties Of Transaction

- A transaction has four properties :
  - Atomicity
  - Consistency
  - Isolation or Independence
  - Durability or Permanence

Atomicity : It defines transaction to be a single unit of processing and it has to be done completely or not at all.

For example, consider a transaction to increase salary of employee by 20%.

Transaction T, this consists of following operations:

Read (Salary)

Salary= Salary + 0.2 * Salary

Write(Salary)

According to atomicity property, all the operations of the transaction must be done or not at all i.e., the transaction can't be aborted after Salary= Salary + 0.2 * Salary.

Consistency : This property ensures that a complete transaction execution takes a database from one consistent state to another consistent. If a transaction fails, even the databases should come back to a consistent state. Looking again at the account transfer system, the system is consistent if the total of all accounts is constant.

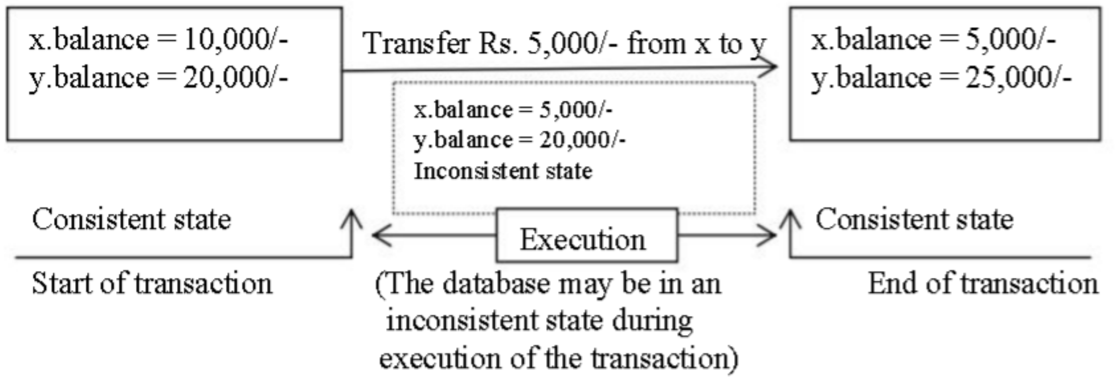


Figure 1: A Transaction execution

Isolation or Independence : This property states that the updates of a transaction should not be visible till they are committed. It guarantees that the progress of other transactions do not affect the outcome of this transaction.

For example, if another transaction that is a withdrawal transaction which withdraws an amount of Rs. 5000 from X account is in progress, whether fails or commits, should not affect the outcome of this transaction.

Durability : This property necessitates that once a transaction has committed the changes made by it be never lost because of subsequent failure. Therefore, transaction is also a basic unit of recovery.

States Of A Transaction

- A transaction has many states of execution. These states are :
- Start state
- Execute state
- Abort/ Rollback state
- Commit state

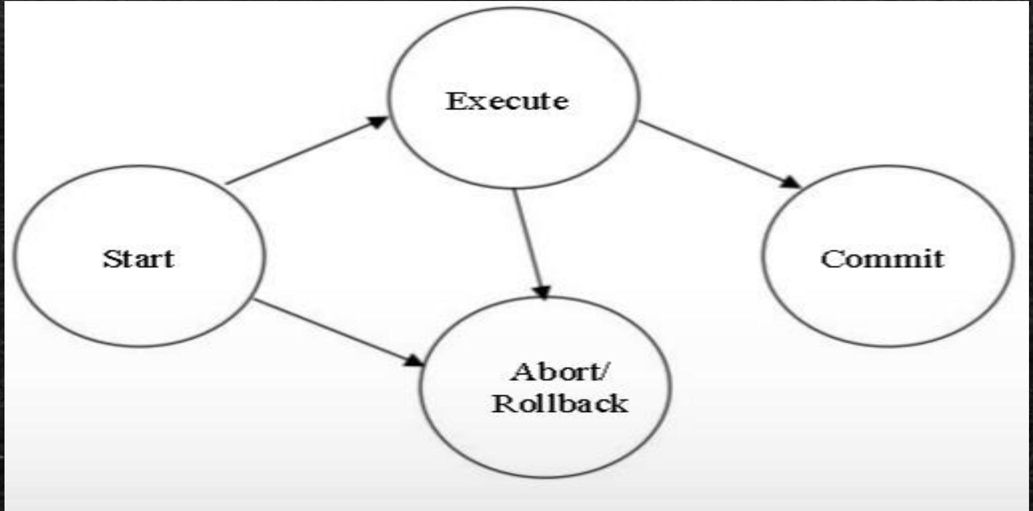


Figure 2: States of transaction execution

- A transaction is started as a program. From the start state as the transaction is scheduled by the CPU it moves to the Execute state, however, in case of any system error at that point it may also be moved into the Abort state.

- During the execution transaction changes the data values and database moves to an inconsistent state. On successful completion of transaction it moves to the commit state where the durability feature of transaction ensures that the changes will not be lost. In case of any error the transaction goes to Rollback state where all the changes made by the transaction are undone. Thus, after commit or rollback database is back into consistent state. In case a transaction has been rolled back, it is started as a new transaction.

Start state - This is the initial state of every transaction. In this state the transaction is being executed. For example, updating or inserting or deleting a record is done here. But it is still not saved to the database.

Execute state - Once the transaction is scheduled by the CPU it moves from the start state to the Execute state.

Abort state – When a system error occurs at the time of execution its moved to the abort state. During the execution, transaction changes the data values and database moves to an inconsistent state.

Committed – On successful completion of transaction, the transaction moves to the committed state, unless it fails. In this state, all the transactions are permanently saved to the database. This step is the last step of a transaction.

Rollback state -In case of any error the transaction goes to Rollback state where all the changes made by the transaction are undone. After commit or rollback state, database is back into consistent state. In case a transaction has been rolled back, it is started as a new transaction.

The Concurrent Transactions

- All the commercial DBMS support multi-user environment.
- Execution of multiple transactions simultaneously is called concurrent transactions.
- DBMS must ensure that transaction of one user doesn’t effect the transaction of other or even the transactions issued by the same user should not get into the way of each other.
- Concurrency related problems arise if two transactions are contending for the same data item and at least one of the concurrent transactions wishes to update a data value in the database. When concurrent transactions only read same data item and no updates are performed on these values, then it does not cause any concurrency related problems.

Why do we need concurrency control?

- Let us explain this by taking a banking application dealing with checking and savings accounts.
- Let this contain two transactions T1 and T2 were T1 is transaction to move Rs. 100 from Sharma's checking account balance X to savings account balance Y.

```
Transaction T1

A : Read X
    Subtract 100
    Write X
B : Read Y
    Add 100
    Write Y
```

Let Transaction T2 be transaction to get the total assets of Mr. Sharma.

Transaction T2

Read X

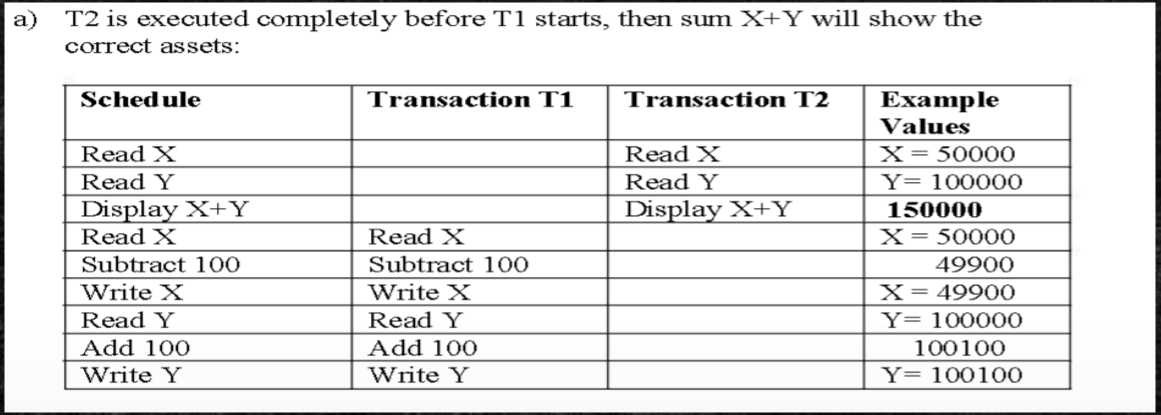
Read Y

Display X+Y

- Suppose these transactions are executed simultaneously, the execution of these instructions can be mixed in many ways. This is known as schedule.
- A schedule S is defined as the sequential ordering of the operations of the 'n' interleaved transactions.

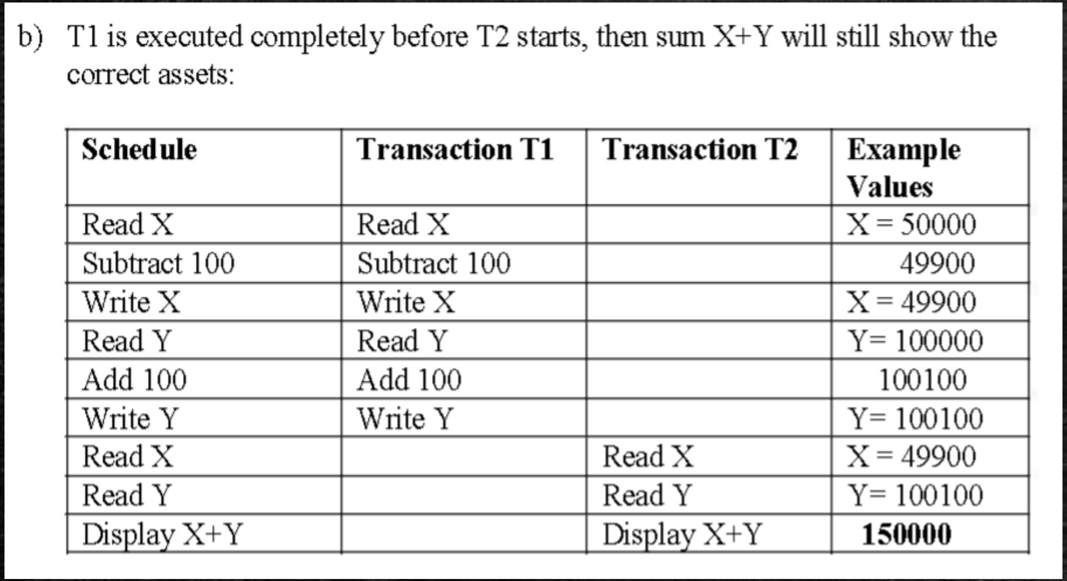
Conflicting Operation in Schedule:

- Two operations of different transactions conflict if they access the same data item and one of them is a write operation.
- Let us see three simple ways of interleaved instruction execution of transactions T1 and T2.



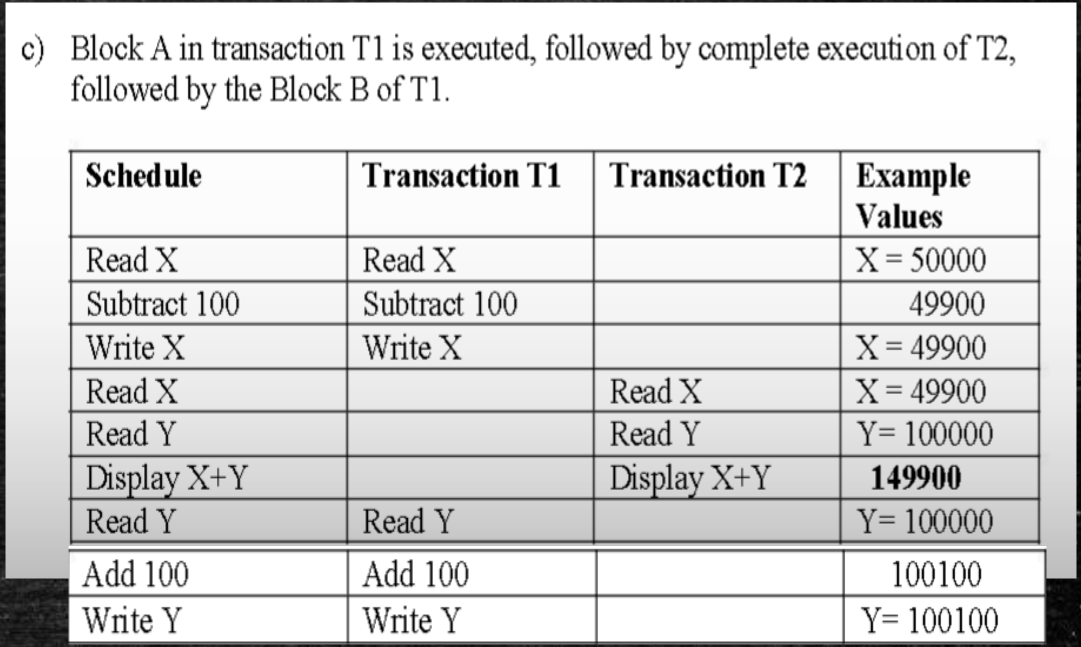
a) T2 is executed completely before T1 starts, then sum X+Y will show the correct assets:

Schedule	Transaction T1	Transaction T2	Example Values
Read X		Read X	X = 50000
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**150000**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100



b) T1 is executed completely before T2 starts, then sum X+Y will still show the correct assets:

Schedule	Transaction T1	Transaction T2	Example Values
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100100
Display X+Y		Display X+Y	**150000**



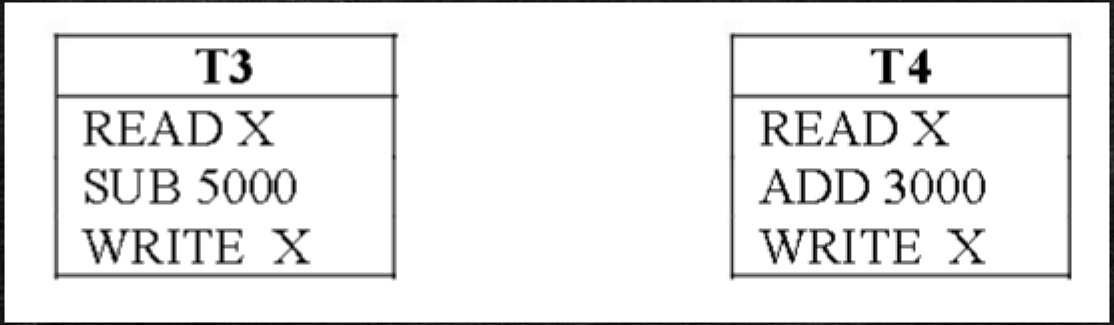
c) Block A in transaction T1 is executed, followed by complete execution of T2, followed by the Block B of T1.

Schedule	Transaction T1	Transaction T2	Example Values
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**149900**
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100

In this execution (c) an incorrect value is being displayed. This is because Rs. 100 although removed from X is not added to Y and is therefore missing.

Problems Of Concurrent Transaction

- Let us consider Transactions T3 and T4. T3 reads the balance of account X and subtracts a withdrawal amount of Rs. 5000. T4 reads the balance of account X and adds an amount of Rs.3000.

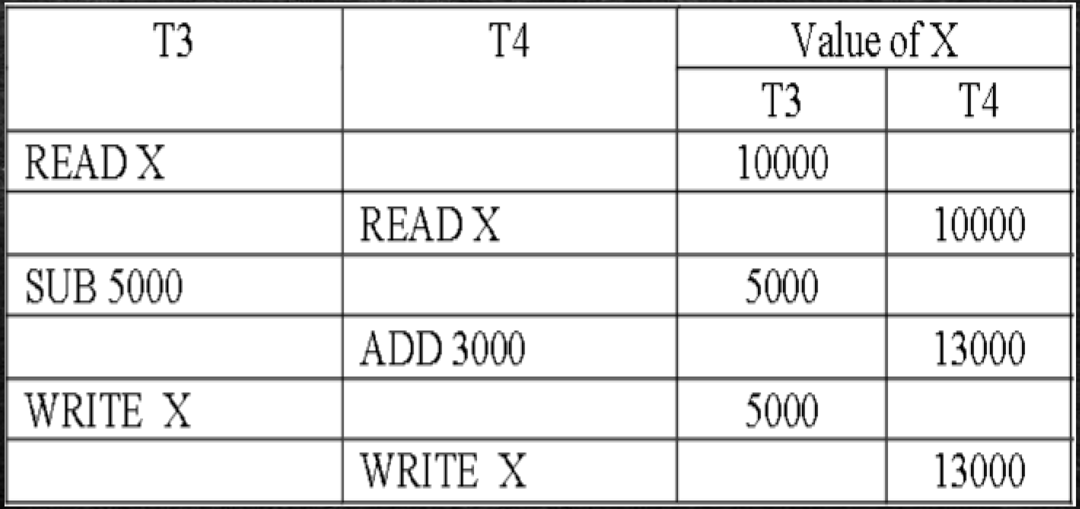


T3
READ X
SUB 5000
WRITE X

T4
READ X
ADD 3000
WRITE X

- Problems of Concurrent transactions :

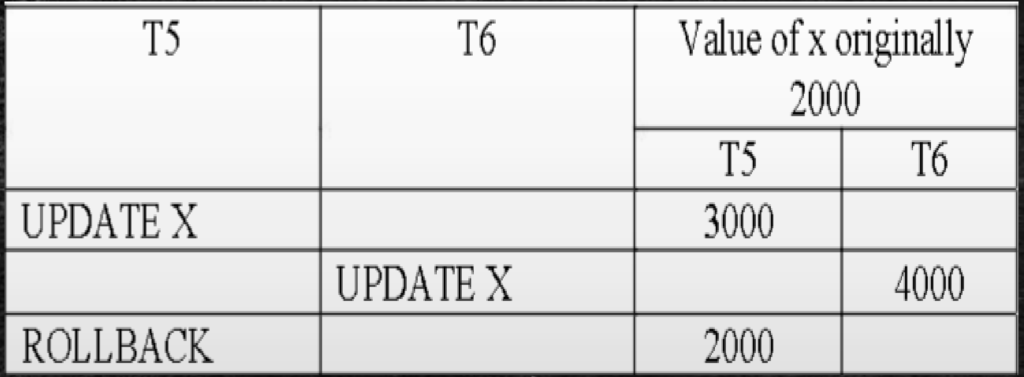
1. Lost Updates: Update made by one transaction is overridden by another transaction. Suppose the two transactions T3 and T4 run concurrently and they happen to be interleaved in the following way.



T3	T4	Value of X	
		T3	T4
READ X		10000	
	READ X		10000
SUB 5000		5000	
	ADD 3000		13000
WRITE X		5000	
	WRITE X		13000

Assume the initial vale of X to be 10000.
After the execution of both the transactions the value X is 13000 while the semantically correct value should be 8000. The problem occurred as the update made by T3 has been overwritten by T4. The root cause of the problem is that one of the updates has been lost and we say that lost update has occurred.

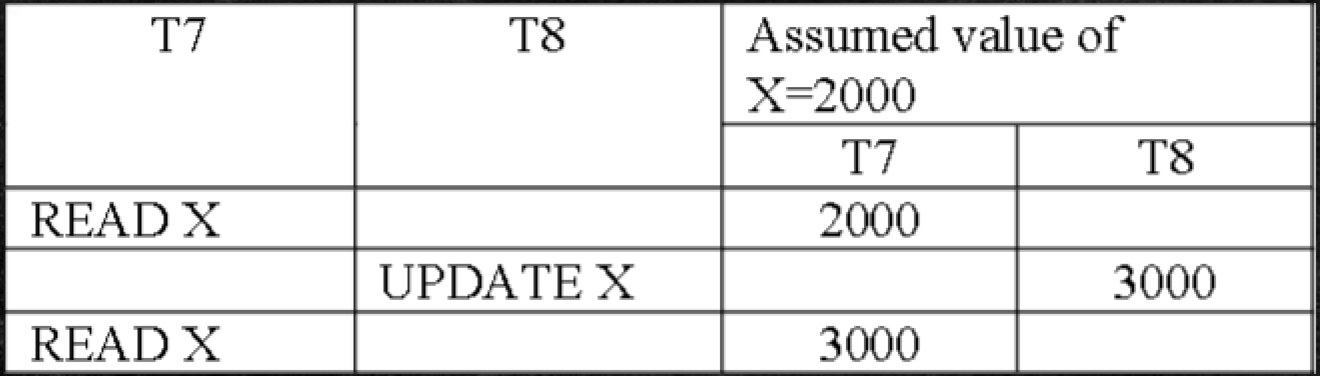
Another way in which the lost updates occur are:



T5	T6	Value of x originally 2000	
		T5	T6
UPDATE X		3000	
	UPDATE X		4000
ROLLBACK		2000	

Here T5 and T6 updates the same item X. Thereafter T5 decides to undo its action and rolls back causing the value of X to 2000. In this case the update performed by T6 has got lost and a lost update is said to have occurred.

2. Unrepeatable reads : If transaction T1 reads an item twice and the item is changed by an another transaction T2 between the two reads hence T1, finds two different values on its two reads. Suppose T7 reads X twice during its execution. If it did not update X itself it could be very disturbing to see a different value of X in its next read. But this occurs if, there is an update operation between the two read operations.

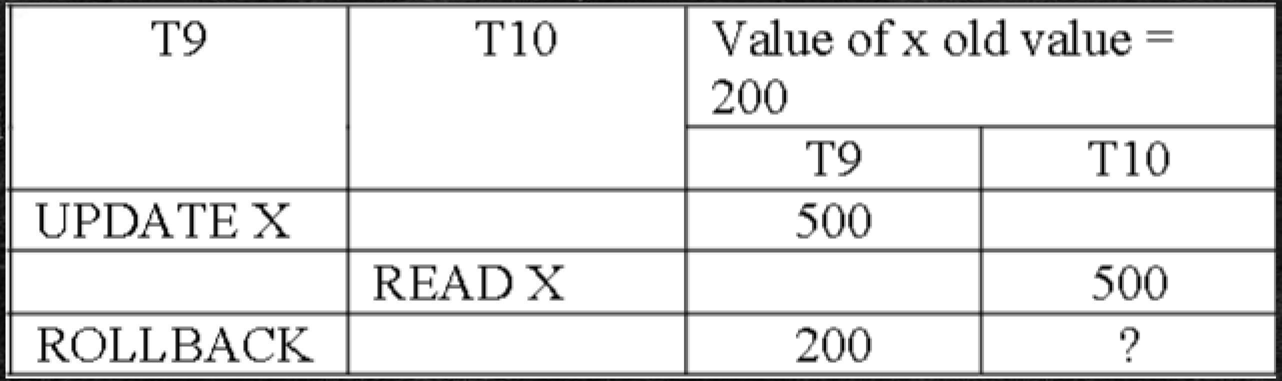


T7	T8	Assumed value of X=2000	
		T7	T8
READ X		2000	
	UPDATE X		3000
READ X		3000	

Thus, the inconsistent values are read and results of the transaction may be in error.

3. Dirty Reads : A transaction reads a value which has been updated by another transaction. This update has not been committed and the later transaction aborts.

For example, T10 reads a value which has been updated by T9. This update has not been committed and T9 aborts.



T9	T10	Value of x old value = 200	
		T9	T10
UPDATE X		500	
	READ X		500
ROLLBACK		200	?

Here T10 reads a value that has been updated by transaction T9 that has been aborted. Thus T10 has read a value that would never exist in the database and hence the problem. Here the problem is isolation of transaction.

4. Inconsistent Analysis : The problem as shown with transactions T1 and T2 where two transactions interleave to produce incorrect result during an analysis by Audit is an example of such a problem. This problem occurs when more than one data items are being used for analysis, while another transaction has modified some of those values and some are yet to be modified. Thus an analysis transaction reads values from the inconsistent state of the database that results in inconsistent analysis.

- From the above problems, we can conclude that the prime reason for the problems of concurrent transaction is that a transaction reads an inconsistent state of the database that has been created by other transactions.

- These problems cannot occur if the transaction do not read the same data values. The conflict occurs only if 1 transaction updates a data value while another is reading or writing the data value.

- Common technique to overcome this is to restrict access to data items that are being read or written by one transaction and is being written by another transaction. This technique is known as locking.

The Locking Protocol

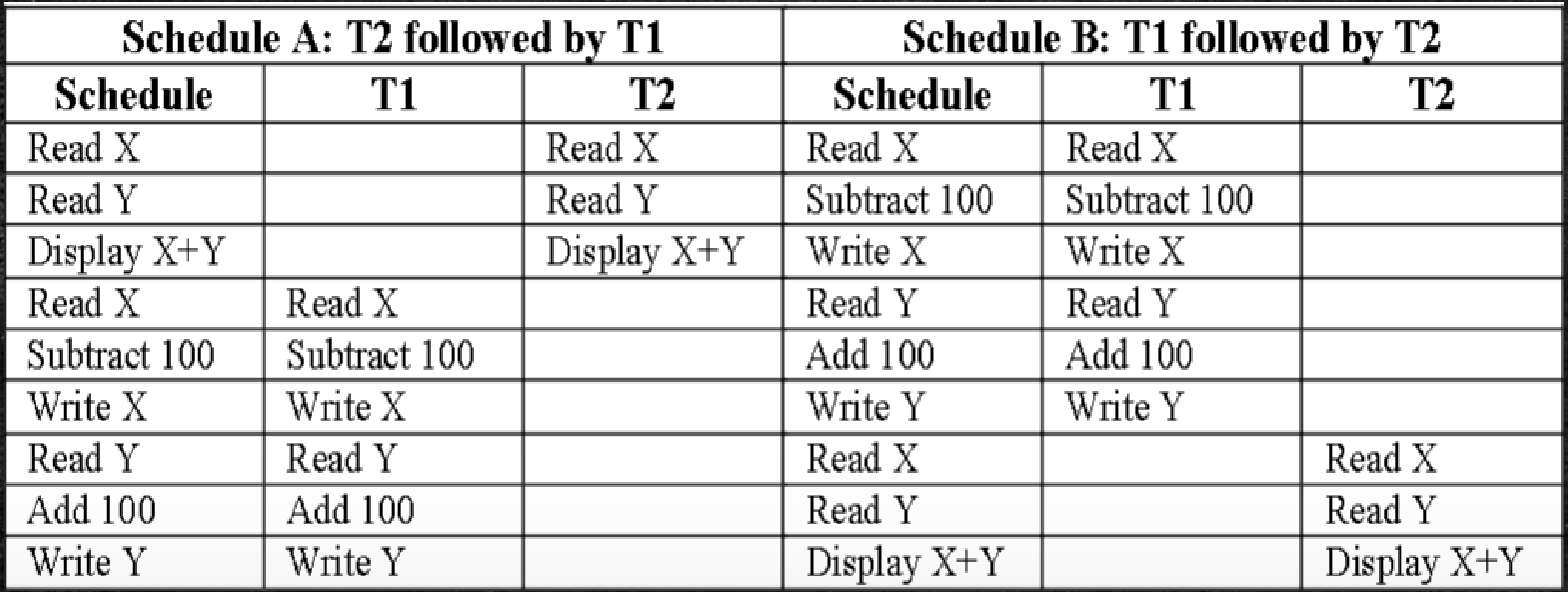
- To control concurrency related problems we use locking.
- A lock is a variable that is associated with a data item in the database.
- A lock can be placed by a transaction on a shared resource that it desires to use.
- When this is done, the data item is available for the exclusive use for that transaction i.e., other transactions are locked out of that data item.
- When a transaction that has locked a data item does not desire to use it any more, it should unlock the data item so that other transactions can use it.
- If a transaction tries to lock a data item that is already locked by some other transaction, it cannot do so and waits for the data item to be unlocked.
- The component of DBMS that controls and stores lock information is called the Lock Manager.
- The locking mechanism helps us to convert a schedule into a serializable schedule.

Serialisable Schedules :

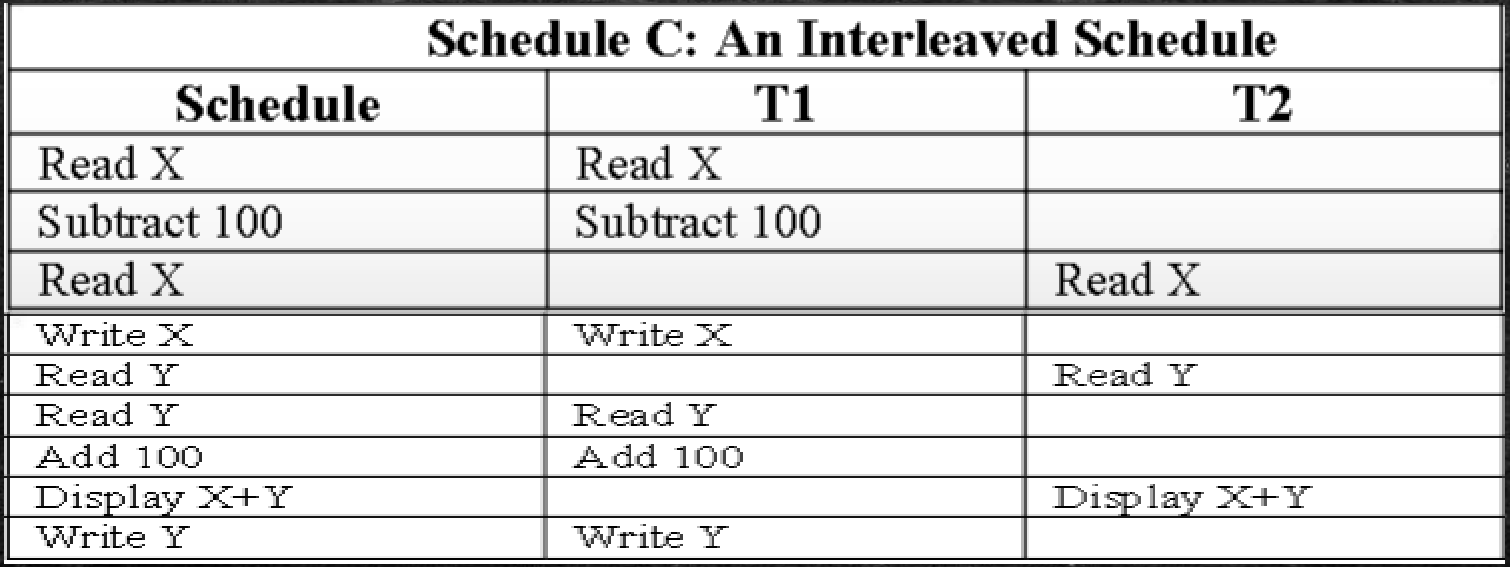
- Serialisability theory attempts to determine the correctness of the schedules.
- The rule of this theory is :
  - A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same ‘n’ transactions.

A serial schedule is a schedule in which either transaction T1 is completely done before T2 or transaction T2 is completely done before T1.

The following figure shows the two possible serial schedules of transactions T1 and T2.



Schedule A: T2 followed by T1			Schedule B: T1 followed by T2		
Schedule	T1	T2	Schedule	T1	T2
Read X		Read X	Read X	Read X	
Read Y		Read Y	Subtract 100	Subtract 100	
Display X+Y		Display X+Y	Write X	Write X	
Read X	Read X		Read Y	Read Y	
Subtract 100	Subtract 100		Add 100	Add 100	
Write X	Write X		Write Y	Write Y	
Read Y	Read Y		Read X		Read X
Add 100	Add 100		Read Y		Read Y
Write Y	Write Y		Display X+Y		Display X+Y



Schedule C: An Interleaved Schedule		
Schedule	**T1**	**T2**
Read X	Read X	
Subtract 100	Subtract 100	
Read X		Read X
Write X	Write X	
Read Y		Read Y
Read Y	Read Y	
Add 100	Add 100	
Display X+Y		Display X+Y
Write Y	Write Y	

- Now we find out whether this interleaved schedule is performed in the same order as that of a serial schedule. If it does, then it's a serial schedule, otherwise not. In case its not equivalent to a serial schedule, it may result in problems due to concurrent transactions.

Serialisability :

- Any schedule that produces the same results as a serial schedule is called a serializable schedule.
- How to find if a schedule is serializable or not?
- By using the notion of precedence graph, an algorithm is devised to determine whether an interleaved schedule is serializable or not.
- In the precedence graph, the transactions of the schedule are represented as nodes. It also contains directed edges. An edge from the node representing transactions Ti to node Tj means that there exists a conflicting operation between Ti and Tj and Ti precedes Tj

in some conflicting operations.

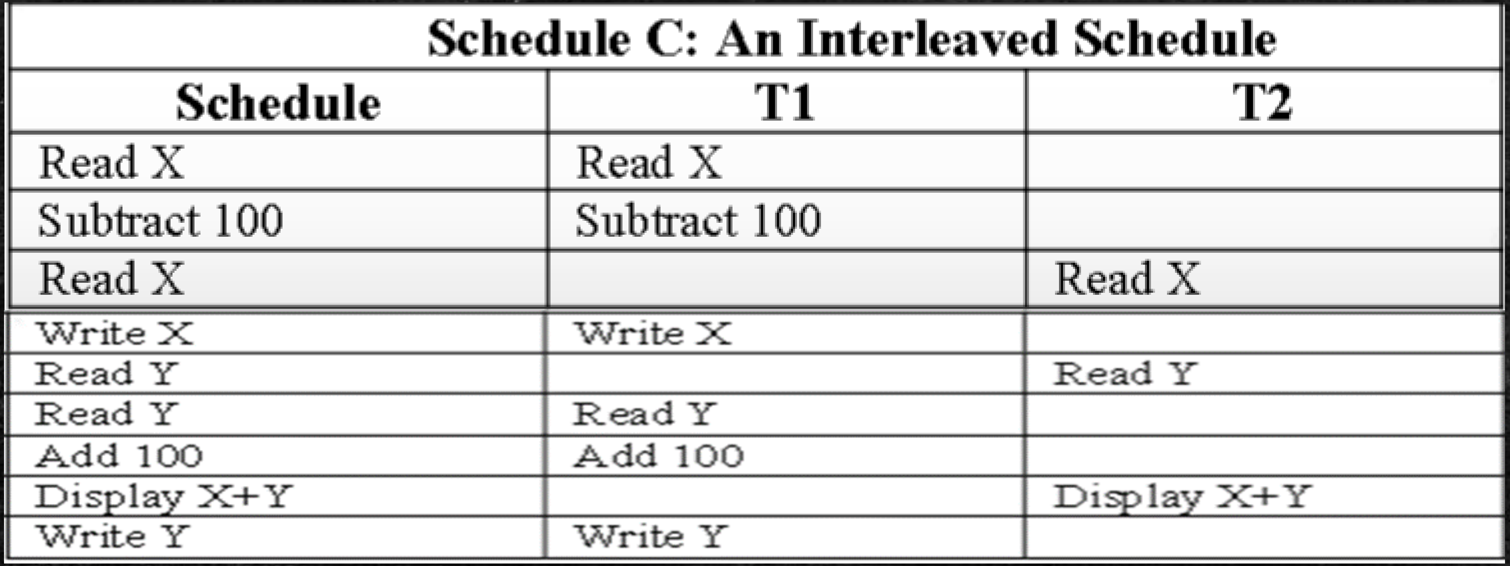
- A serializable schedule is the one that contains no cycle in the graph.

Steps to construct a precedence graph :

1. Create a node for every transaction in the schedule.
2. Find the precedence relationships in conflicting operations. Conflicting operations can be : (read-write) or (write-read) or (write-write) on the same data item in two different transactions.
    1. For a transaction Ti which reads an item A, find a transaction Tj that writes A later in the schedule. If such a transaction is found, draw an edge from Ti to Tj.
    2. For a transaction Ti which has written an item A, find a transaction Tj later in the schedule that reads A. If such a transaction is found, draw an edge from Ti to Tj.
    3. For a transaction Ti which has written an item A, find a transaction Tj that writes A later than Ti. If such a transaction is found, draw an edge from Ti to Tj.

3. If there is any cycle in the graph, the schedule is not serializable, otherwise, find the equivalent serial schedule of the transaction by traversing the transaction nodes starting from the node that has no input edge.

Example to construct precedence graph :



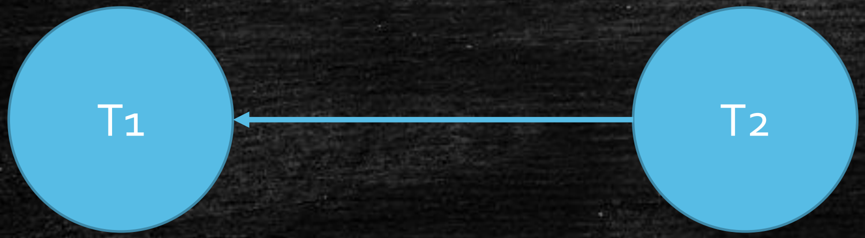
Schedule C: An Interleaved Schedule

Schedule	T1	T2
Read X	Read X	
Subtract 100	Subtract 100	
Read X		Read X
Write X	Write X	
Read Y		Read Y
Read Y	Read Y	
Add 100	Add 100	
Display X+Y		Display X+Y
Write Y	Write Y	

Step 1 : We draw the nodes for transactions T1 and T2. Number of nodes = Number of transactions.

Step 2 : In the above schedule, transaction T2 reads data item X, which is subsequently written by T1, so there is an edge from T2 to T1.

$$\begin{tikzcd} T_1 & T_2 \arrow[l] \end{tikzcd}$$



Step 3 : Also, T2 reads a data item Y, which is subsequently written by T1, thus there is an edge from T2 to T1. This edge exists, so there is no need to redo it.

Step 4 : There is no cycle in the graph, thus this schedule is serializable.

The equivalent serial schedule would be T2 followed by T1.

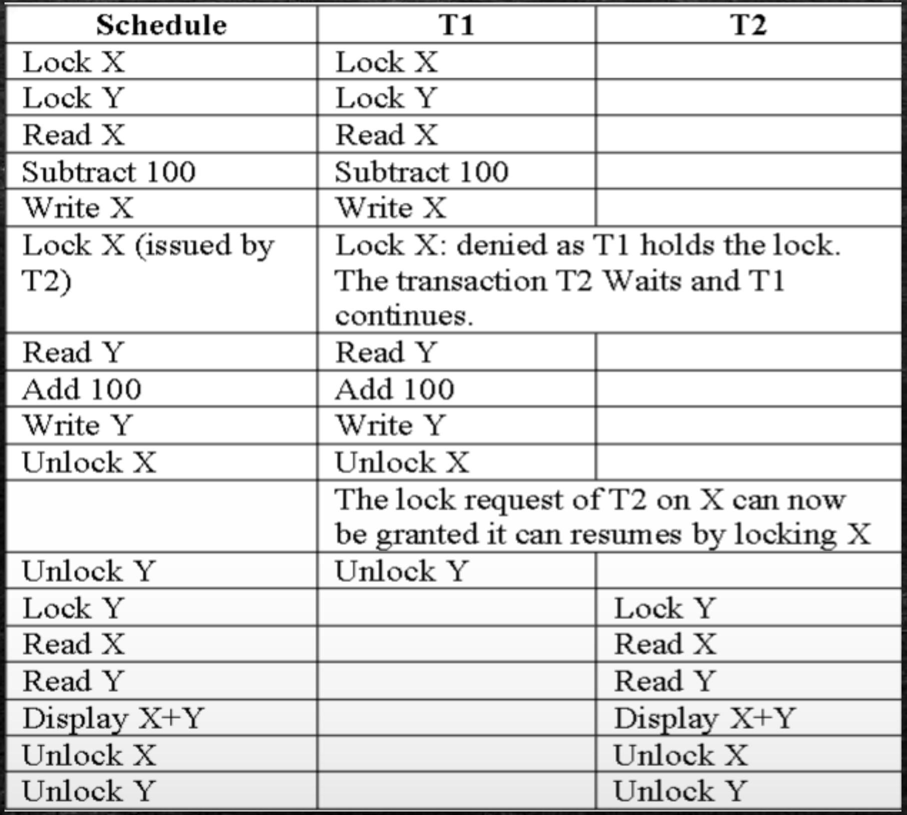
Locks

- Serialisability is a test whether a given interleaved schedule is ok or has a concurrency related problem.
- It does not ensure that the interleaved concurrent transactions do not have any concurrency related problem.
- Locks ensure that the interleaved concurrent transactions do not have any concurrency related problem.
- Types of locks :

There are two basic types of locks :

- Binary lock : This locking mechanism has two states for a data item : locked or unlocked.
- Multiple-mode locks : In this locking type each data item can live in three states read locked or shared locked, write locked or exclusive locked or unlocked.

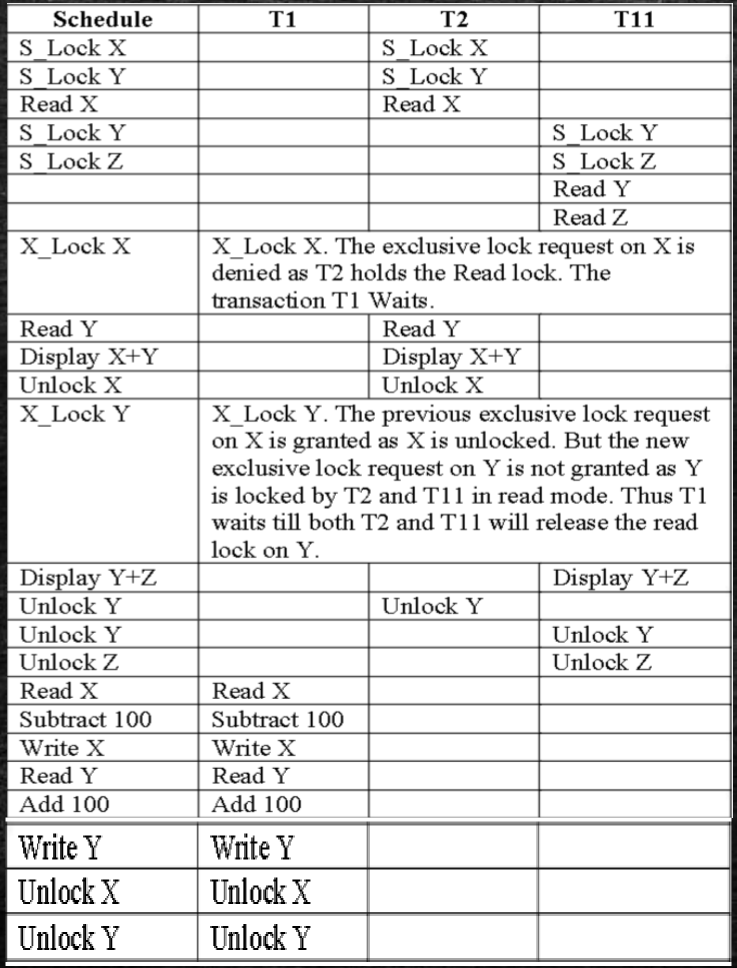
Binary locks :



Schedule	T1	T2
Lock X	Lock X	
Lock Y	Lock Y	
Read X	Read X	
Subtract 100	Subtract 100	
Write X	Write X	
Lock X (issued by T2)	Lock X: denied as T1 holds the lock. The transaction T2 Waits and T1 continues.	
Read Y	Read Y	
Add 100	Add 100	
Write Y	Write Y	
Unlock X	Unlock X	
	The lock request of T2 on X can now be granted it can resumes by locking X	
Unlock Y	Unlock Y	
Lock Y		Lock Y
Read X		Read X
Read Y		Read Y
Display X+Y		Display X+Y
Unlock X		Unlock X
Unlock Y		Unlock Y

Multiple mode locks :

- It offers two locks: shared locks and exclusive locks.
- Shared lock :
- It is requested by a transaction that wants to just read the value of data item.
- A shared lock on a data item does not allow an exclusive lock to be placed but permits any number of shared locks to be placed on that item.
- Exclusive lock :
- It is requested by a transaction on a data item that it needs to update.
- No other transaction can place either a shared lock or an exclusive lock on a data item that has been locked in an exclusive mode.



Schedule	T1	T2	T11
S_Lock X		S_Lock X	
S_Lock Y		S_Lock Y	
Read X		Read X	
S_Lock Y			S_Lock Y
S_Lock Z			S_Lock Z
			Read Y
			Read Z
X_Lock X	X_Lock X. The exclusive lock request on X is denied as T2 holds the Read lock. The transaction T1 Waits.		
Read Y		Read Y	
Display X+Y		Display X+Y	
Unlock X		Unlock X	
X_Lock Y	X_Lock Y. The previous exclusive lock request on X is granted as X is unlocked. But the new exclusive lock request on Y is not granted as Y is locked by T2 and T11 in read mode. Thus T1 waits till both T2 and T11 will release the read lock on Y.		
Display Y+Z			Display Y+Z
Unlock Y		Unlock Y	
Unlock Y			Unlock Y
Unlock Z			Unlock Z
Read X	Read X		
Subtract 100	Subtract 100		
Write X	Write X		
Read Y	Read Y		
Add 100	Add 100		
Write Y	Write Y		
Unlock X	Unlock X		
Unlock Y	Unlock Y		

Two Phase Locking (2PL)

- It’s a concurrency control method in which locking and unlocking is done in two phases.
- The two-phase locking protocol consists of two phases :

Phase 1 : The lock acquisition phase or growing phase : If a transaction T wants to read an object, it needs to obtain the S (shared) lock. If T wants to modify an object, it needs to obtain X (exclusive) lock. No conflicting locks are granted to a transaction. New locks on items can be acquired but no lock can be released till all the locks required by the transaction are obtained.

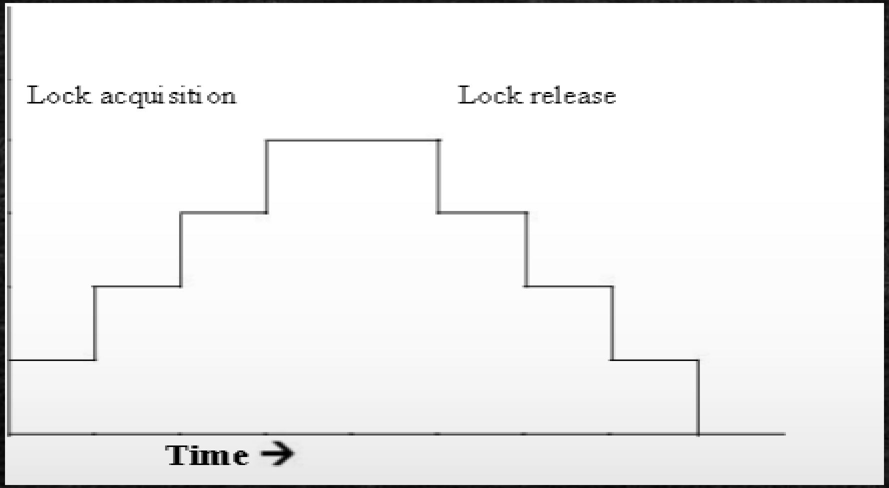
Phase 2 : Lock Release Phase or shrinking phase: The existing locks can be released in any order but no new lock can be acquired after a lock has been released. The locks are held only till they are required.

- Normally the locks are obtained by the DBMS. Any legal schedule of transactions that follow 2PL protocol is guaranteed to be serializable.
- The 2PL protocol has been approved for it correctness.

- There are two types of 2PL :

1. The Basic 2PL
2. Strict 2PL

The basic 2PL allows release of lock at any time after all the locks have been acquired.



- The basic 2PL suffers from the problem that it can result into loss of atomic / isolation property of transaction as theoretically speaking once a lock is released on a data item it can be modified by anther transaction before the first transaction aborts or commits.
- To avoid the above problem we use strict 2PL. The strict 2PL is graphically depicted below.

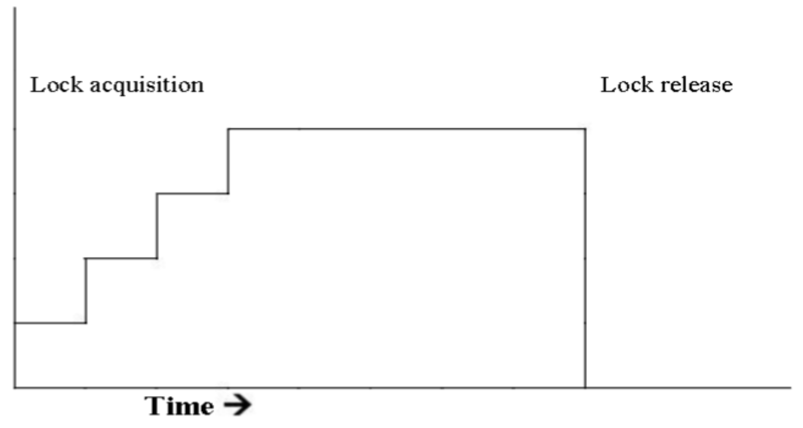
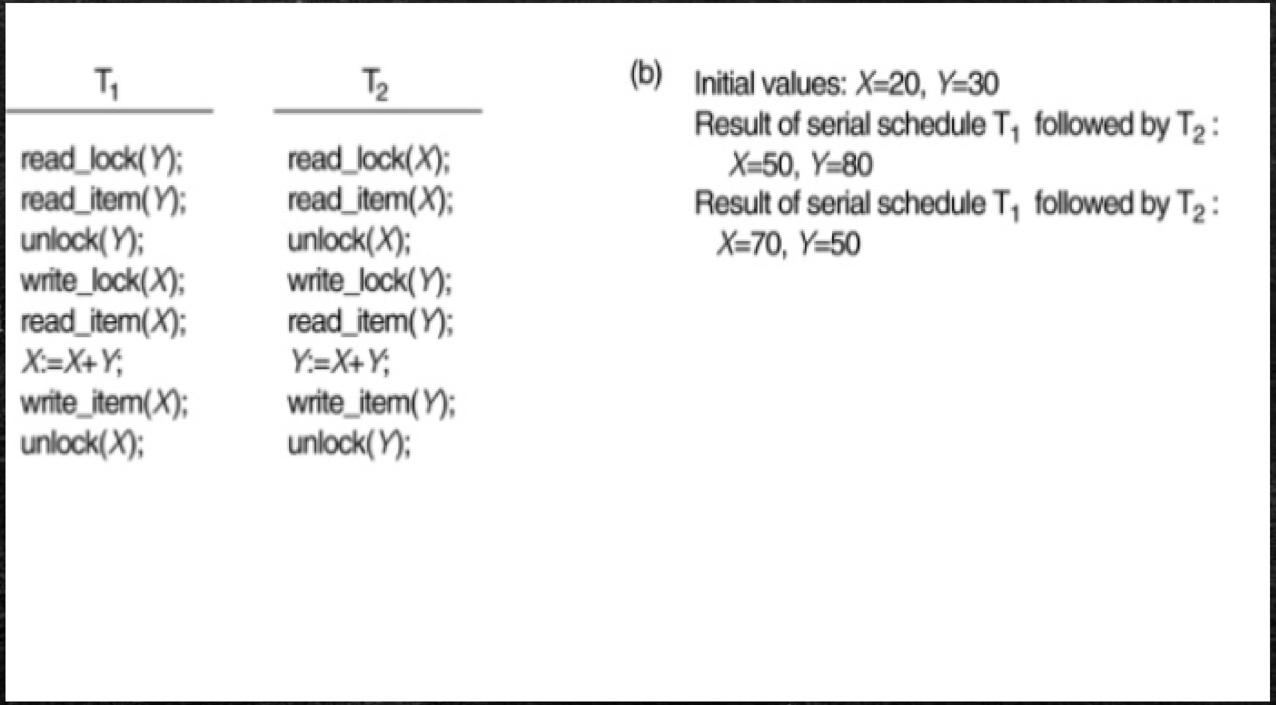


Figure 11: Strict Two Phase Locking

- Basic disadvantage of strict 2PL is that it restricts concurrency as it locks the item beyond the time it is needed by a transaction.
- 2PL solves the problem of concurrency and atomicity, but introduces another problem known as deadlock.
- Transactions T1 and T2 that follow two-phase locking protocol (next slide):

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Transactions T1 and T2 that do not obey 2PL.



T_1	T_2
read_lock(*Y*);	read_lock(*X*);
read_item(*Y*);	read_item(*X*);
unlock(*Y*);	unlock(*X*);
write_lock(*X*);	write_lock(*Y*);
read_item(*X*);	read_item(*Y*);
X:=*X*+*Y*;	*Y*:=*X*+*Y*;
write_item(*X*);	write_item(*Y*);
unlock(*X*);	unlock(*Y*);

(b) Initial values: *X*=20, *Y*=30
Result of serial schedule T_1 followed by T_2 :
X=50, *Y*=80
Result of serial schedule T_1 followed by T_2 :
X=70, *Y*=50

Deadlock And Its Prevention

- A deadlock is a condition that occurs when two or more different database tasks are waiting for each other and none of the task is willing to give up the resources that other task needs.
- Consider two transactions and a schedule involving these transactions :

T1	T2	Schedule
X_lock A	X_lock A	T1: X_lock A
X_lock B	X_lock B	T2: X_lock B
:	:	T1: X_lock B
:	:	T2: X_lock A
Unlock A	Unlock A	
Unlock B	Unlock B	

- After T1 has locked A, T2 locks B and then T1 tries to lock B, but unable to do so and waits for T2 to unlock B. Similarly, T2 tries to lock A but finds that it is held by T1 which has not yet unlocked it and thus waits for T1 to unlock A. At this stage, neither T1 nor T2 can proceed since both of these transactions are waiting for the other to unlock the locked resource. In such a situation, we say that a deadlock has occurred, since two transactions are waiting for a condition that will never occur.

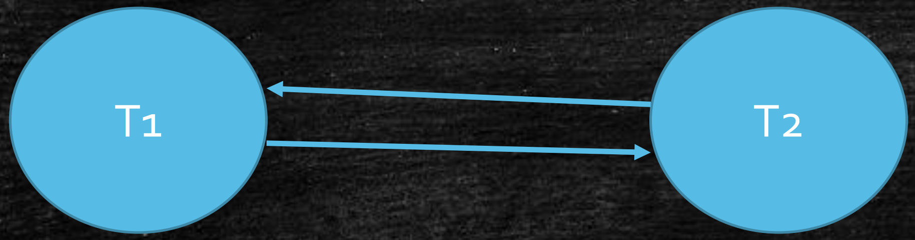
Deadlock detection :

- Simple way to detect a state of deadlock is to draw a directed graph called a "wait for" graph. Wait for graph is maintained by the lock manager of the DBMS.

- This graph G is defined by the pair (V,E) where Vis a set of vertices/nodes and E is set of edges/arcs.

- Each transaction is represented by a node and an arc from Ti->Tj, Ti is waiting for a data item/resource that is held by Tj.

- When transaction Ti requests for a data item that is held by Tj then the edge Ti->Tj is inserted in the "wait for" graph. This edge is removed when transaction Tj is no longer holding the data item needed by transaction T1.
- A deadlock in the system of transaction occurs, if and only if the wait-for graph contains a cycle.
- Deadlock can be detected by doing a periodic check for cycles in graph.



Wait for graph of T1 and T2

- In the above figure, T1 and T2 are the two transactions. T1 and T2 are waiting for each other to unlock a resource held by the other, forming a cycle, causing a deadlock problem.

Deadlock conditions :

- A deadlock occurs because of the following conditions :

- Mutual exclusion : states that at least one resource cannot be used by more than one process at a time. The resources cannot be shared between processes. A resource can be locked in exclusive mode by only one transaction at a time.
- Non-preemption : A data item can only be unlocked by the transaction that locked it. No other transaction can unlock it.
- Partial allocation : A transaction can acquire locks on database in a piecemeal fashion.
- Circular waiting : transactions lock part of data resources needed and then wait indefinitely to lock the resource currently locked by other transactions. states that one process is waiting for a resource which is being held by second process and the second process is waiting for the third process and so on and the last process is waiting for the first process. It makes a circular chain of waiting.

- To prevent deadlock, one has to ensure that at least one of these transactions does not occur.

Deadlock Prevention :

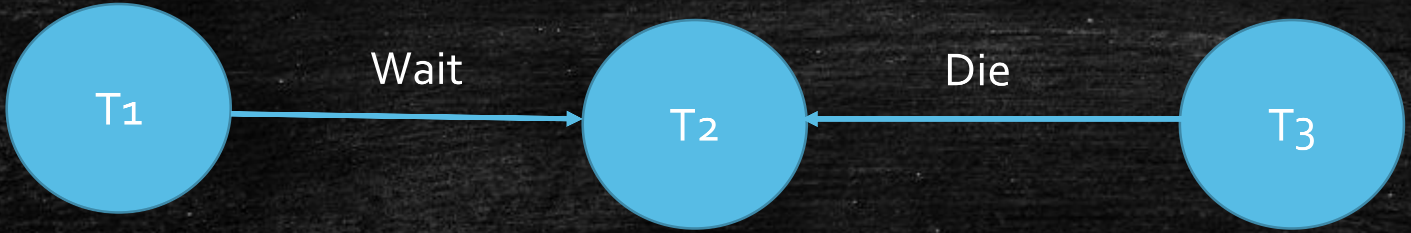
- Deadlock can be prevented by having the basic logic : not to allow circular wait to occur. In this approach, some of the transactions are rolled back instead of letting them wait.

- There exists two such schemes . These are :

- "Wait-die" scheme : Its based on non- preemptive technique. Its based on simple rule :

```
If Ti requests a database resource that is held by Tj
    then if Ti has a smaller timestamp than that of Tj
         it is allowed to wait;
     else Ti aborts
```

- A timestamp may be defined as a sequence number that is unique for each transaction. Therefore, a smaller timestamp means an older transaction.
- For example, assume that three transactions T1, T2 and T3 where generated in that sequence, then if T1 requests for a data item which is currently held by transactions T2, it is allowed to wait as it has a smaller time stamping than that of T1.
- However, if T3 requests for a data item which is currently held by transaction T2, then T3 is rolled back (die).



- "Wound wait" scheme : Its based on a preemptive technique. Its based on a simple rule :

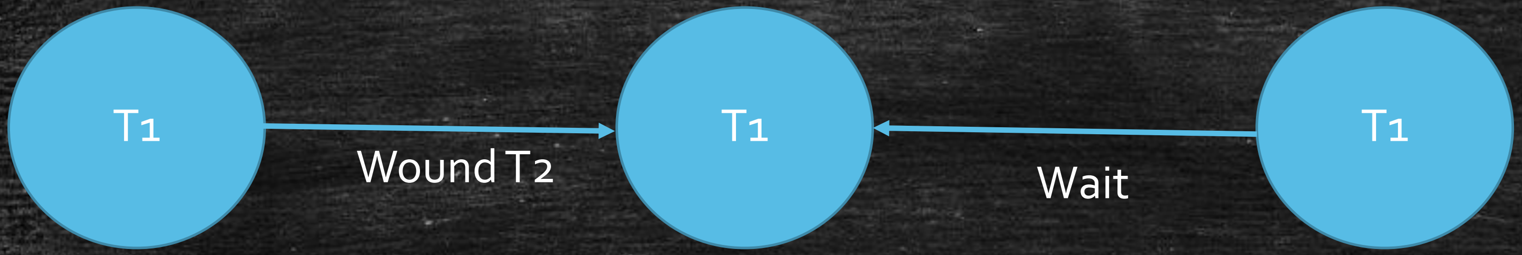
If Ti requests a database resource that is held by Tj

    then if Ti has a larger timestamp(Ti is younger) than that of Tj

        it is allowed to wait;

    else Tj is wounded up by Ti.

For example, assume that there are three transactions T1, T2 and T3 were generated in that sequence, then if T1 requests for a data item which is held by transaction T2, then T2 is rolled back and data item is allotted to T1 as T1 has a smaller time stamping than that of T2. However, if T3 requests for a data item which is currently held by transaction T2, then T3 is allowed to wait.



- Whenever any transaction is rolled back, it would not make a starvation condition. Both "wait-die" and "wound-wait" scheme avoid starvation.
- The number of aborts and rollbacks will be higher in wait-die scheme than in the wound-wait scheme.
- Problem with these is that they may result in unnecessary rollbacks.

Optimistic Concurrency Control

- Optimistic concurrency control is a method used to prevent concurrency related problems.
- Basic logic behind this is to allow the concurrent transactions to update the data items assuming that the concurrency related problem will not occur. However, we need to reconfirm our view in the validation phase.
- Optimistic concurrency control algorithm has the following phases :

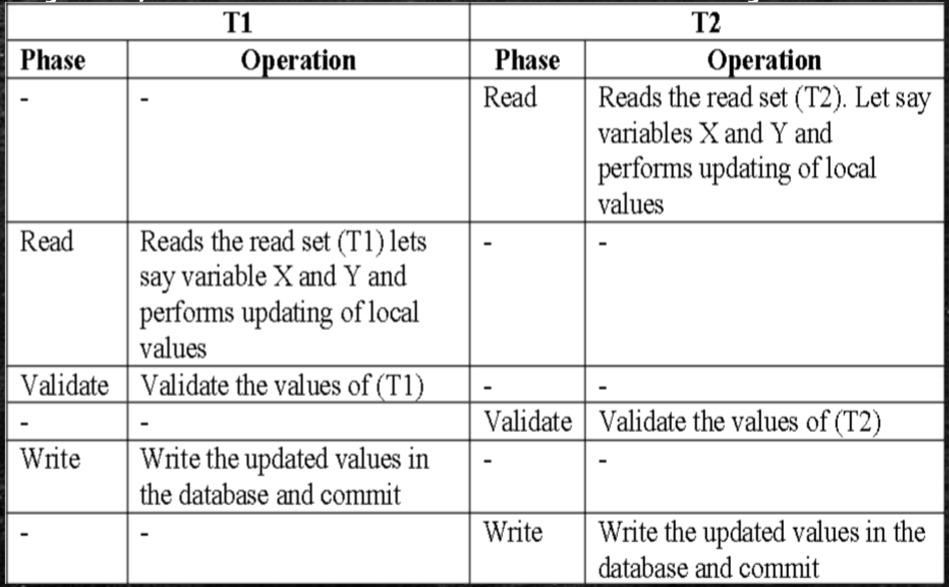
1. Read phase : A transaction T reads the data items from the database into its private workspace. All the updates of the transaction can only change the local copies of the data in the private workspace.
2. Validate phase : Checking is done to confirm whether the read values have changed during the time transaction was updating the local values. This is performed by comparing the current database values to the values that were read in the private workspace. In case, the values have changed the local copies are thrown away and the transaction aborts.

3. Write phase : If validation phase is successful the transaction is committed and updates are applied to the database, otherwise the transaction is rolled back.

- Some of the terms defined to explain optimistic concurrency control :

- Write-set(T) : all data items that are written by a transaction T.
- Read-set(T) : all data items are read by a transaction T
- Timestamps : for each transaction T, the start time and the end time are kept for all the three phases.

Example : In this case both T1 and T2 get committed. Read set of T1 and Read Set of T2 are both disjoint, also the Write sets are also disjoint and thus no concurrency occurs.



T1		T2	
Phase	**Operation**	**Phase**	**Operation**
-	-	Read	Reads the read set (T2). Let say variables X and Y and performs updating of local values
Read	Reads the read set (T1) lets say variable X and Y and performs updating of local values	-	-
Validate	Validate the values of (T1)	-	-
-	-	Validate	Validate the values of (T2)
Write	Write the updated values in the database and commit	-	-
-	-	Write	Write the updated values in the database and commit