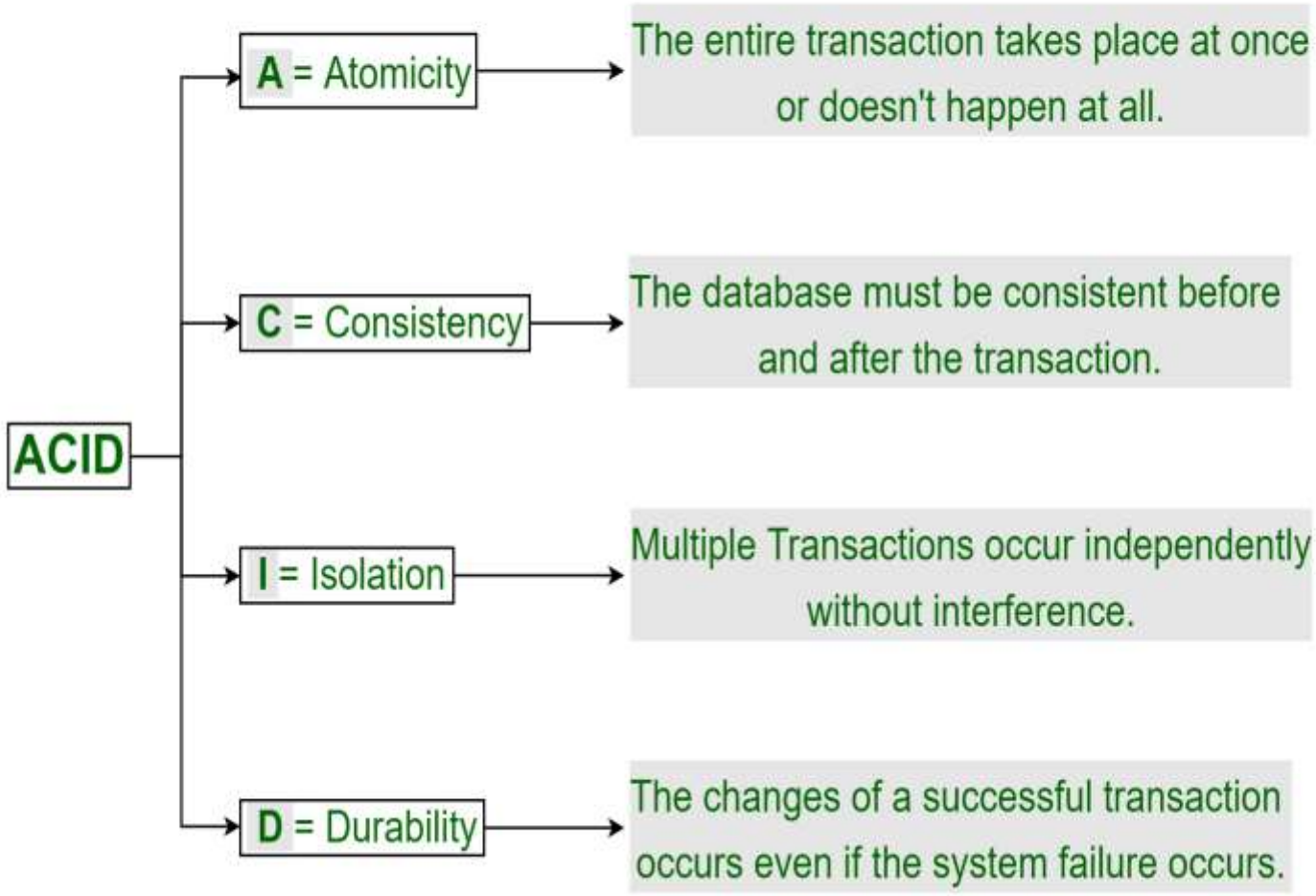
Unit No. 5

Transaction Processing

Transaction Concept:-

- A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.
- A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

ACID Properties in DBMS



- ACID Properties:-
- **Atomicity** – This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction. It involves the following two operations.
  —**Abort**: If a transaction aborts, changes made to database are not visible.
  —**Commit**: If a transaction commits, changes made are visible.
- Atomicity is also known as the 'All or nothing rule'.

- Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2.**(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

Referring to the example above,

The total amount before and after the transaction must be maintained.

Total **before T** occurs = **500 + 200 = 700**.

Total **after T occurs** = **400 + 300 = 700**.

Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.
- Let **X**= 500, **Y** = 500. Consider two transactions **T** and **T".**

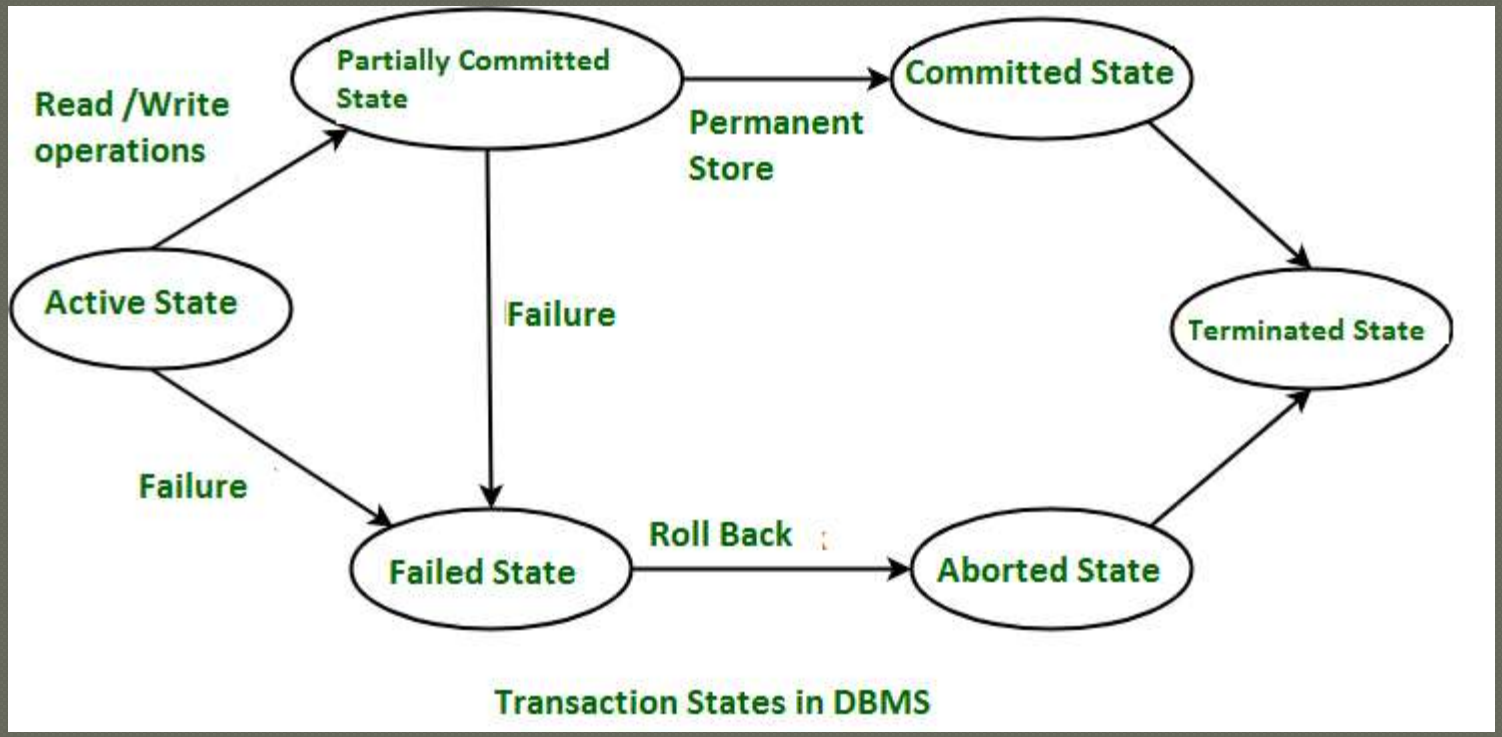
T	T"
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T"** starts. As a result , interleaving of operations takes place due to which **T"** reads correct value of **X** but incorrect value of **Y** and sum computed by
T": (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

Transaction States in DBMS:-

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the Transaction and also tell how we will further do processing we will do on the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.



Transaction States in DBMS

There are different types of Transaction States :-

1] Active State –

When the instructions of the transaction is running then the transaction is in active state. If all the read and write operations are performed without any error then it goes to "partially committed state", if any instruction fails it goes to "failed state".

2] Partially Committed –

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

3] Failed State-

When any instruction of the transaction fails it goes to "failed state" or if failure occurs in making permanent change of data on Data Base.

4] Aborted State –

After having any type of failure the transaction goes from "failed state" to "aborted state" and in before states the changes are only made to local buffer or main memory and hence these changes are deleted or rollback.

5] Committed State –

It is the stage when the changes are made permanent on the Data Base and transaction is complete and therefore terminated in "terminated state".

6] Terminated State –

If there is any roll back or the transaction come from "committed state" then the system is consistent and ready for new transaction and the old transaction is terminated.

- **Concurrent Execution of Transaction:-**

In the transaction process, a system usually allows executing more than one transaction simultaneously. This process is called a concurrent execution.

- **Advantages of concurrent execution of a transaction:-**

1.Decrease waiting time or turnaround time.
2.Improve response time
3.Increased throughput or resource utilization.

- **Concurrency problems:-**

-Several problems can occur when concurrent transactions are run in an uncontrolled manner, such type of problems is known as concurrency problems.

-There are following different types of problems or conflicts which occur due to concurrent execution of transaction:

1.Lost update problem (Write – Write conflict)

- This type of problem occurs when two transactions in database access the same data item and have their operations in an interleaved manner that makes the value of some database item incorrect.
- If there are two transactions T1 and T2 accessing the same data item value and then update it, then the second record overwrites the first record.
- **Example:** Let's take the value of A is 100

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2	A=A-50	
t3		Read(A)
t4		A=A+50
t5	Write(A)	
t6		Write(A)

-Here,

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction deducts the value of A by 50.
- At t3 time, T2 transactions read the value of A i.e., 100.
- At t4 time, T2 transaction adds the value of A by 150.
- At t5 time, T1 transaction writes the value of A data item on the basis of value seen at time t2 i.e., 50.
- At t6 time, T2 transaction writes the value of A based on value seen at time t4 i.e., 150.
- So at time T6, the update of Transaction T1 is lost because Transaction T2 overwrites the value of A without looking at its current value.
- Such type of problem is known as the Lost Update Problem.

2.Dirty read problem (W-R conflict)-

This type of problem occurs when one transaction T1 updates a data item of the database, and then that transaction fails due to some reason, but its updates are accessed by some other transaction.

Example: Let's take the value of A is 100

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2	A=A+20	
t3	Write(A)	
t4		Read(A)
t5		A=A+30
t6		Write(A)
t7	Write(B)	

- **Here,**
- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction adds the value of A by 20.
- At t3 time, T1transaction writes the value of A (120) in the database.
- At t4 time, T2 transactions read the value of A data item i.e., 120.
- At t5 time, T2 transaction adds the value of A data item by 30.
- At t6 time, T2transaction writes the value of A (150) in the database.
- At t7 time, a T1 transaction fails due to power failure then it is rollback according to atomicity property of transaction (either all or none).
- So, transaction T2 at t4 time contains a value which has not been committed in the database. The value read by the transaction T2 is known as a dirty read.

3.Unrepeatable read (R-W Conflict)-

- It is also known as an inconsistent retrieval problem. If a transaction T_1 reads a value of data item twice and the data item is changed by another transaction T_2 in between the two read operation. Hence T_1 access two different values for its two read operation of the same data item.
- **Example:** Let's take the value of A is 100

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2		Read(A)
t3		A=A+30
t4		Write(A)
t5	Read(A)	

- **Here,**
- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T2transaction reads the value of A i.e., 100.
- At t3 time, T2 transaction adds the value of A data item by 30.
- At t4 time, T2 transaction writes the value of A (130) in the database.
- Transaction T2 updates the value of A. Thus, when another read statement is performed by transaction T1, it accesses the new value of A, which was updated by T2. Such type of conflict is known as R-W conflict.

- **Schedule:-**
- A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.
- **1.Serial Schedules:**
  Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.
- **Example:** Consider the following schedule involving two transactions T_1 and T_2.

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

where R(A) denotes that a read operation is performed on some data item ‘A’
This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

- **2.Non-Serial Schedule:**
  This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.

a.Serializable:

This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput.

a1.Conflict Serializable:

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

a2.View Serializable:

A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

b.Non-Serializable:

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

- **b1.Recoverable Schedule:**
  Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.
- **Example –** Consider the following schedule involving two transactions T_1 and T_2.

- **Example –** Consider the following schedule involving two transactions T_1 and T_2.

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

- This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

- There can be three types of recoverable schedule:
- **a.Cascading Schedule:** Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort.
- Example:

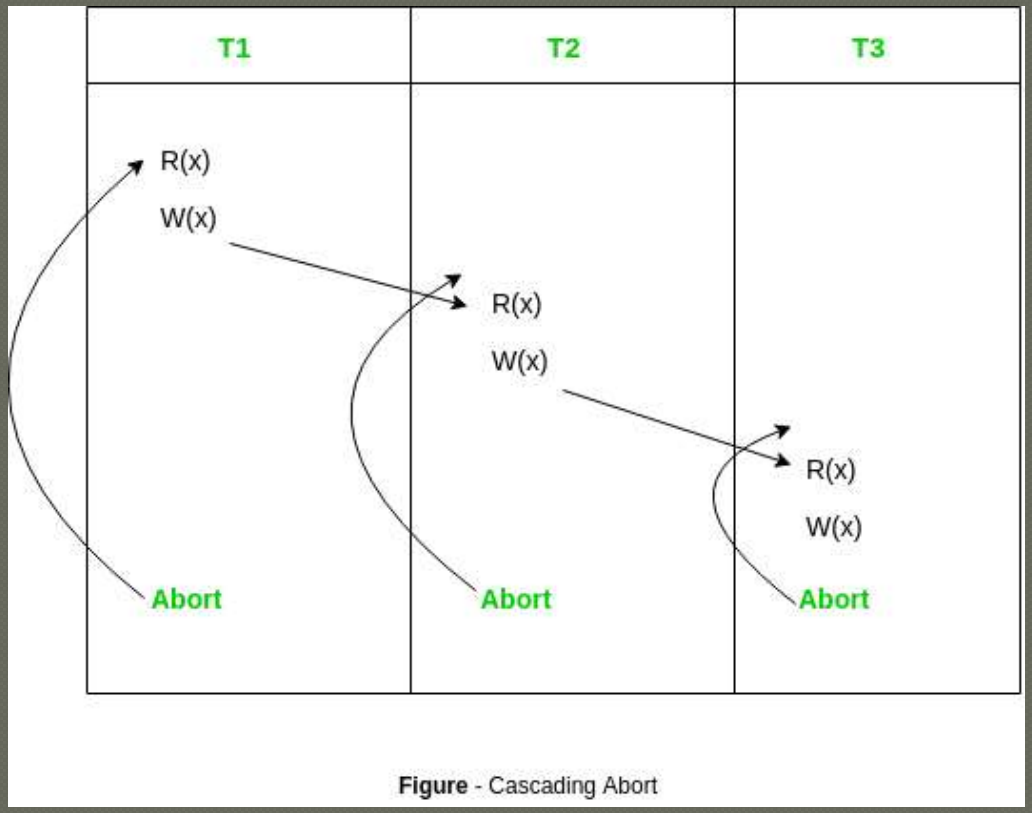


Figure - Cascading Abort

- **b.Cascadeless Schedule:**
  Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.

- **Example:** Consider the following schedule involving two transactions T_1 and T_2.

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

- This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

- **Example:** Consider the following schedule involving two transactions T_1 and T_2.

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

- It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as T_2 has already read the uncommitted value written by T_1.

- **c.Strict Schedule:**
  A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.
  In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.
- **Example:** Consider the following schedule involving two transactions T_1 and T_2.

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

- This is a strict schedule [illegible]d writes A which is written by T_1 only after the commit of T_1.

- **2.Non-Recoverable Schedule:**

  Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

- T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

It can be seen that:

- Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
- Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
- Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

Lock based protocols:-

- **LOCK:-**
- A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.
- All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.
- A lock may deny access to other transactions to prevent incorrect results.

- **1.Shared Lock (S):** also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
- **2.Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock -X instruction.
- **Lock Compatibility Matrix**

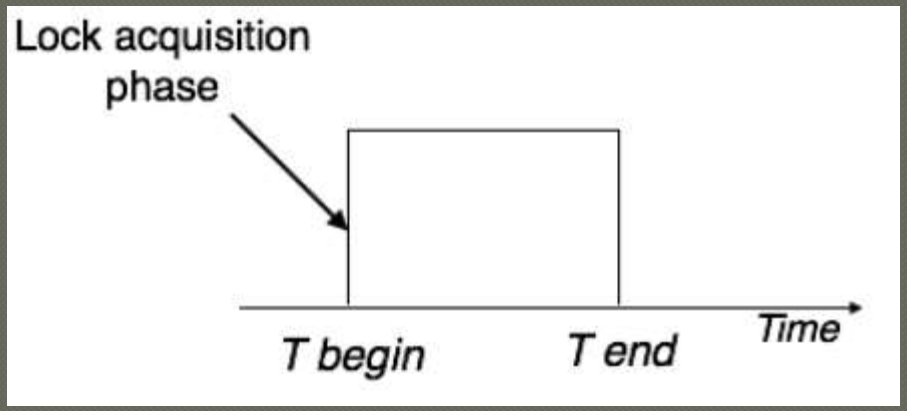
	S	X
S	✔	✗
X	✗	✗

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

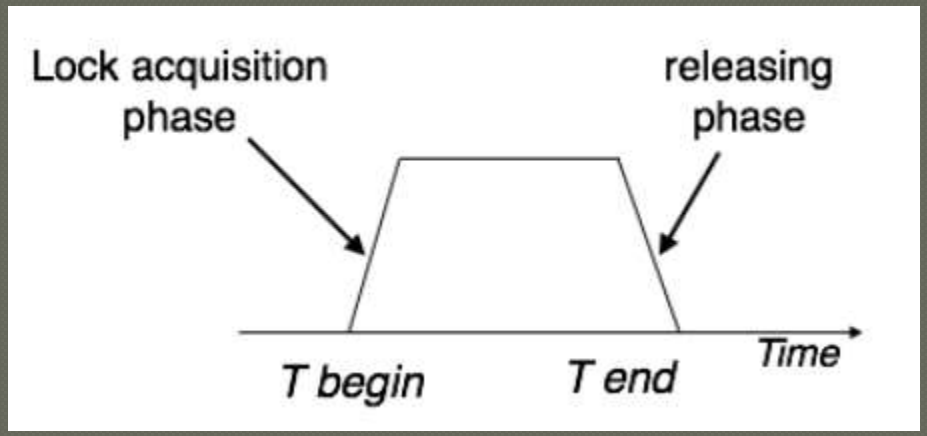
- **Upgrade / Downgrade locks :** A transaction that holds a lock on an item **A** is allowed under certain condition to change the lock state from one state to another.
  Upgrade: A S(A) can be upgraded to X(A) if T_i is the only transaction holding the S-lock on element A.
  Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

- There are four types of lock protocols available –
- **Simplistic Lock Protocol**
- Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.
- **Pre-claiming Lock Protocol**
- Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand.

- If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.

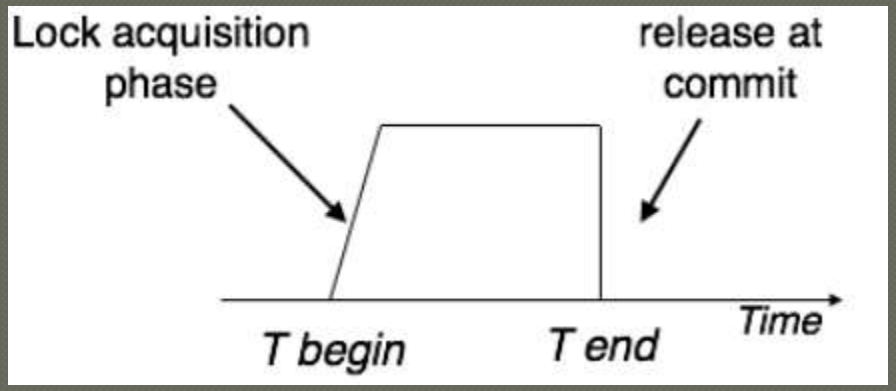


- **Two-Phase Locking 2PL**
- This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.



- Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.
- To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

- **Strict Two-Phase Locking**
- The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



- Strict-2PL does not have cascading abort as 2PL does.