UNIT -5
TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the**Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the**ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

```
ReadA;
A=A–100;
WriteA;
ReaB;
B=B+100;
Write B;
```

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction, (i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which two transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.
Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.
Durability: It states that once a transaction has been complete the changes it has made should be permanent.
As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.
There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.
If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

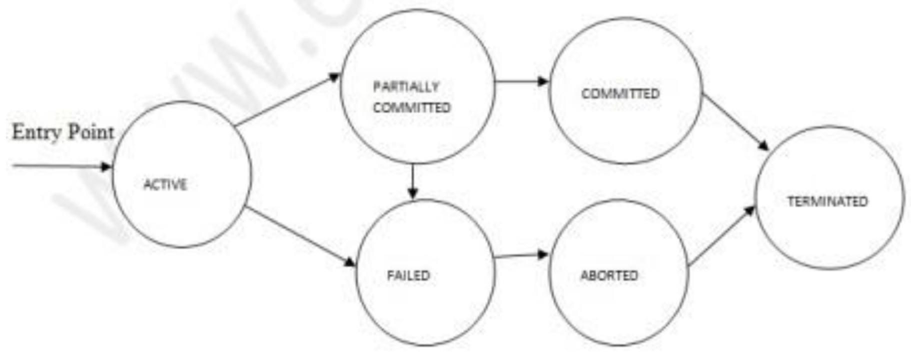
Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

Partially Committed: At any given point of time if the transaction is executing properly, then it

is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.
Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.
Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.
Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.
Terminated: Either committed or aborted, the transaction finally reaches this state.
The whole process can be described using the following diagram:



Concurrent Execution
A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.
In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.
Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

T1
ReadA;
A=A–100;

```
WriteA;
ReadB;
B=B100;
Write B;
                T2
ReadA;
Temp=A*0.1;
ReadC;
C=C+Temp;
WriteC;
```

T2 is a new transaction which deposits to account C 10% of the amount in account A.
If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

```
    T1                    T2

Read A;
A = A - 100;
Write A;
                          Read A;
                          Temp = A * 0.1;
                          Read C;
                          C = C + Temp;
                          Write C;
Read B;
B = B + 100;
Write B;
```

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.
It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

```
    T1                    T2

Read A;
A = A - 100;
                          Read A;
```

```
                        Temp = A * 0.1;
                        Read C;
                        C = C + Temp;
                        Write C;
Write A;
Read B;
B = B + 100;
Write B;
```

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

In the above example, we detected the error simple by examining the schedule and applying common sense. But there must be some well formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules. This brings us to our next topic, the concept of Serializability.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

It may happen that we may want to execute the same set of transaction in a different schedule on another day. Keeping in mind these rules, we may sometimes alter parts of one schedule (S1) to create another schedule (S2) by swapping only the non-conflicting parts of the first schedule. The conflicting parts cannot be swapped in this way because the ordering of the conflicting instructions is important and cannot be changed in any other schedule that is derived from the first. If these two schedules are made of the same set of transactions, then both S1 and S2 would yield the same result if the conflict resolution rules are maintained while creating the new schedule. In that case the schedule S1 and S2 would be called **Conflict Equivalent**.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Except in these three cases, any alteration can be possible while creating S2 by modifying S1.

CONCURRENTCY CONTROL:

hen multiple transactions are trying to access the same sharable resource, there could arise many problems if the access control is not done properly. There are some important mechanisms to which access control can be maintained. Earlier we talked about theoretical concepts like serializability, but the practical concept of this can be implemented by using **Locks** and **Timestamps**. Here we shall discuss some protocols where Locks and Timestamps can be used to provide an environment in which concurrent transactions can preserve their Consistency and Isolation properties.

Lock Based Protocol

A lock is nothing but a mechanism that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose. Since there are two types of operations, i.e. read and write, whose basic nature are different, the locks for read and write operation may behave differently.

Read operation performed by different transactions on the same data item poses less of a challenge. The value of the data item, if constant, can be read by any number of transactions at any given time.

Write operation is something different. When a transaction writes some value into a data item, the content of that data item remains in an inconsistent state, starting from the moment when the writing operation begins up to the moment the writing operation is over. If we allow any other

transaction to read/write the value of the data item during the write operation, those transaction will read an inconsistent value or overwrite the value being written by the first transaction. In both the cases anomalies will creep into the database.
The simple rule for locking can be derived from here. If a transaction is reading the content of a sharable data item, then any number of other processes can be allowed to read the content of the same data item. But if any transaction is writing into a sharable data item, then no other transaction will be allowed to read or write that same data item.
Depending upon the rules we have found, we can classify the locks into two types.
Shared Lock: A transaction may acquire shared lock on a data item in order to read its content. The lock is shared in the sense that any other transaction can acquire the shared lock on that same data item for reading purpose.
Exclusive Lock: A transaction may acquire exclusive lock on a data item in order to both read/write into it. The lock is excusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item.
The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

Locks already existing

	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

How Should Lock be Used?
In a transaction, a data item which we want to read/write should first be locked before the read/write is done. After the operation is over, the transaction should then unlock the data item so that other transaction can lock that same data item for their respective usage. In the earlier chapter we had seen a transaction to deposit Rs 100/- from account A to account B. The transaction should now be written as the following:
Lock-X (A); (Exclusive Lock, we want to both read A's value and modify it)
ReadA;
A=A–100;
WriteA;
Unlock (A); (Unlocking A after the modification is done)
Lock-X (B); (Exclusive Lock, we want to both read B's value and modify it)
ReadB;
B=B+100;
WriteB;
Unlock (B); (Unlocking B after the modification is done)
And the transaction that deposits 10% amount of account A to account C should now be written as:
Lock-S (A); (Shared Lock, we only want to read A's value)
ReadA;
Temp=A*0.1;
Unlock(A);(UnlockingA)
Lock-X (C); (Exclusive Lock, we want to both read C's value and modify it)
ReaDC;

C=C+Temp;
WriteC;
Unlock (C); (Unlocking C after the modification is done)
Let us see how these locking mechanisms help us to create error free schedules. You should remember that in the previous chapter we discussed an example of an erroneous schedule:

T1	T2
Read A;	
A = A - 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

We detected the error based on common sense only, that the Context Switching is being performed before the new value has been updated in A. T2 reads the old value of A, and thus deposits a wrong amount in C. Had we used the locking mechanism, this error could never have occurred. Let us rewrite the schedule using the locks.

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
	Lock-S (A)
	Read A;
	Temp = A * 0.1;
	Unlock (A)
	Lock-X(C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Write A;	
Unlock (A)	
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	

Unlock (B)

We cannot prepare a schedule like the above even if we like, provided that we use the locks in the transactions. See the first statement in T2 that attempts to acquire a lock on A. This would be impossible because T1 has not released the excusive lock on A, and T2 just cannot get the shared lock it wants on A. It must wait until the exclusive lock on A is released by T1, and can begin its execution only after that. So the proper schedule would look like the following:

```
  T1                    T2

Lock-X (A)
Read A;
A = A - 100;
Write A;
Unlock (A)
                      Lock-S (A)
                      Read A;
                      Temp = A * 0.1;
                      Unlock (A)
                      Lock-X(C)
                      Read C;
                      C = C + Temp;
                      Write C;
                      Unlock (C)
Lock-X (B)
Read B;
B = B + 100;
Write B;
Unlock (B)
```

And this automatically becomes a very correct schedule. We need not apply any manual effort to detect or correct the errors that may creep into the schedule if locks are not used in them.

Two Phase Locking Protocol

The use of locks has helped us to create neat and clean concurrent schedule. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks.

The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

Growing Phase: In this phase the transaction can only acquire locks, but cannot release any lock. The transaction enters the growing phase as soon as it acquires the first lock it wants. From now on it has no option but to keep acquiring all the locks it would need. It cannot release any lock at this phase even if it has finished working with a locked data item. Ultimately the transaction reaches a point where all the lock it may need has been acquired. This point is called **Lock Point**.

Shrinking Phase: After Lock Point has been reached, the transaction enters the shrinking phase. In this phase the transaction can only release locks, but cannot acquire any new lock. The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock

Point. From now on it has no option but to keep releasing all the acquired locks. There are two different versions of the Two Phase Locking Protocol. One is called the Strict Two Phase Locking Protocol and the other one is called the Rigorous Two Phase Locking Protocol.

Strict Two Phase Locking Protocol

In this protocol, a transaction may release all the shared locks after the Lock Point has been reached, but it cannot release any of the exclusive locks until the transaction commits. This protocol helps in creating cascade less schedule.

A **Cascading Schedule** is a typical problem faced while creating concurrent schedule. Consider the following schedule once again.

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
Unlock (A)	
	Lock-S (A)
	Read A;
	Temp = A * 0.1;
	Unlock (A)
	Lock-X(C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	
Unlock (B)	

The schedule is theoretically correct, but a very strange kind of problem may arise here. T1 releases the exclusive lock on A, and immediately after that the Context Switch is made. T2 acquires a shared lock on A to read its value, perform a calculation, update the content of account C and then issue COMMIT. However, T1 is not finished yet. What if the remaining portion of T1 encounters a problem (power failure, disc failure etc) and cannot be committed? In that case T1 should be rolled back and the old BFIM value of A should be restored. In such a case T2, which has read the updated (but not committed) value of A and calculated the value of C based on this value, must also have to be rolled back. We have to rollback T2 for no fault of T2 itself, but because we proceeded with T2 depending on a value which has not yet been committed. This phenomenon of rolling back a child transaction if the parent transaction is rolled back is called Cascading Rollback, which causes a tremendous loss of processing power and execution time.

Using Strict Two Phase Locking Protocol, Cascading Rollback can be prevented. In Strict Two Phase Locking Protocol a transaction cannot release any of its acquired exclusive locks until the transaction commits. In such a case, T1 would not release the exclusive lock on A until it finally commits, which makes it impossible for T2 to acquire the shared lock on A at a time when A's value has not been committed. This makes it impossible for a schedule to be cascading.

Rigorous Two Phase Locking Protocol

In Rigorous Two Phase Locking Protocol, a transaction is not allowed to release any lock (either shared or exclusive) until it commits. This means that until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock; but cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock.

Timestamp Ordering Protocol

A **timestamp** is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or data item had been activated in any way. We, who use computers, must all be familiar with the concepts of "Date Created" or "Last Modified" properties of files and folders. Well, timestamps are things like that.

A timestamp can be implemented in two ways. The simplest one is to directly assign the current value of the clock to the transaction or the data item. The other policy is to attach the value of a logical counter that keeps incrementing as new timestamps are required.

The timestamp of a transaction denotes the time when it was first activated. The timestamp of a data item can be of the following two types:

W-timestamp (Q): This means the latest time when the data item Q has been written into.

R-timestamp (Q): This means the latest time when the data item Q has been read from.

These two timestamps are updated each time a successful read/write operation is performed on the data item Q.

How should timestamps be used?

The timestamp ordering protocol ensures that any pair of conflicting read/write operations will be executed in their respective timestamp order. This is an alternative solution to using locks.

For Read operations:

1. If TS (T) < W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has already been overwritten by some other transaction. Hence the value which T wanted to read from Q does not exist there anymore, and T would be rolled back.
2. If TS (T) >= W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has been written and committed by some other transaction earlier. Hence T will be allowed to read the value of Q, and the R-timestamp of Q should be updated to TS (T).

For Write operations:

1. If TS (T) < R-timestamp (Q), then it means that the system has waited too long for transaction T to write its value, and the delay has become so great that it has allowed another transaction to read the old value of data item Q. In such a case T has lost its relevance and will be rolled back.
2. Else if TS (T) < W-timestamp (Q), then transaction T has delayed so much that the system has allowed another transaction to write into the data item Q. in such a case too, T has lost its relevance and will be rolled back.
3. Otherwise the system executes transaction T and updates the W-timestamp of Q to TS (T).