CONCURRENCY CONTROL

When 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 to Be granted	Locks already existing: Shared	Locks already existing: 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)
Read A;
A = A – 100;
Write A;
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)
Read B;
B = B + 100;
Write B;
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)
Read A;
Temp = A * 0.1;
Unlock (A); (Unlocking A)
Lock-X (C); (Exclusive Lock, we want to both read C's value and modify it)
Read C;
C = C + Temp;
Write C;
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;	
	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).