UNIT - V

Transaction Processing & Concurrency Control.

Transaction:-

A Transaction is logical unit of work which takes the database from one consistent state to another consistent state while moving from one consistent state to another consistent state the database may pass through multiple discrete steps.

At the end of the transaction the database may move back to its original state (in the case of failure) or to the next logical step (in the case of success)

Properties Of a Transaction:-

Atomicity:-

Either all operations of the transactions are reflected properly in the database or None are. This either all or none property is called Atomicity.

Consistency:-

The database must move from one consistent state to another consistent state. so the Execution of the transaction must preserve the Consistency of the database.

Isolation:-

Transaction can run independently without any interference. Isolation property ensures that each transaction is unaware of other transactions executing concurrently in the System.

Even though multiple transactions execute concurrently the system guarantees that for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

Durability :-

After the transaction completes successfully, the changes it has made to the database persist even if there are system failures

The above properties are often called ACID properties of a transaction.

Ex:- Transferring $50 from Account A to Account B

```
Ti :-   read (A);
        A: = A-50;
        write (A);
        read (B);
        B: = B+50;
        write (B);
```

Transaction states:-

In the absence of failures all transactions complete successfully and atomicity is acheived. However a transaction may not successfully complete its execution such transaction are termed as aborted. To ensure atomicity an aborted transaction must have no effect on the state of the database system. A transaction

is said to be rolled back if the database system is restored to the state that the system, was in before the execution of transaction.

If a transaction completes its execution successfully then it is a committed transaction A committe transaction that performs updates transforms the database into a new consistent state.

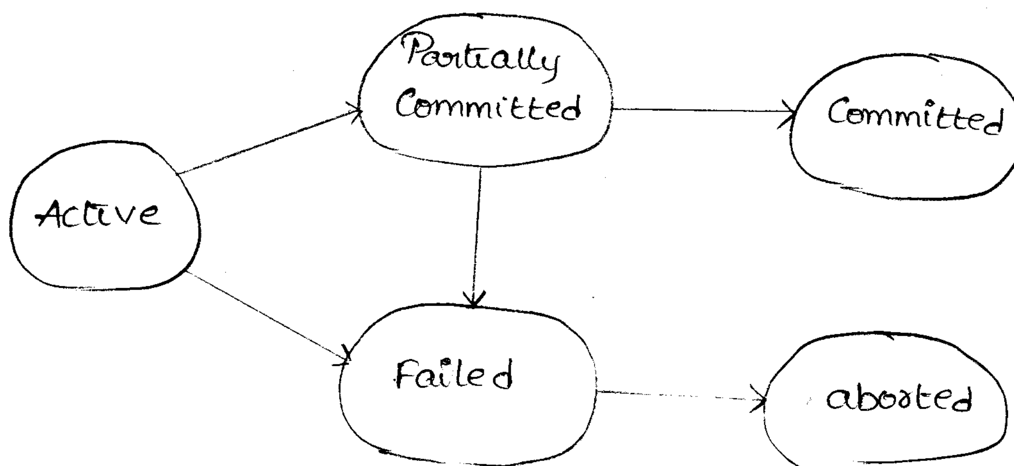
A transaction can be in any of the following states.

Active:-
It is the initial state, the transaction stays in this state while it is executing. partially committed: After the final statement has been executed.

Committed:- After successfull completion

Failed:- After the discovery that normal execution can no longer proceed

Aborted:-
After the transaction has been rolled back and the database restored to its state prior to the start of the transaction.



Implementation of Atomicity and Durability:

The recovery management component of a database system can support atomicity and durability by a variety of schemes. A very simple and an inefficient scheme is <u>shadow copy scheme</u>. This scheme assumes only one transaction is active and the database is simply a file on the disk. A pointer called <u>db-pointer</u> is maintained on disk and it points to the current copy of the database.

Shadow copy scheme:-

In this scheme, a transaction which wants to update the database first creates a complete copy of the database. All updates are done on the new data base copy leaving the original copy (shadow copy) untouched. If at any point a transaction has to be aborted, the system deletes the new copy. The old copy of the database has not been affected.

If the transaction completes, it is committed as follows.

→ The O.S is asked to make sure that all pages of new copy of the database have been written out to disk. After all the pages are written to disk, the database system updates the pointer db-pointer to point to the new copy of the database. The new copy then becomes the current copy of the database and the old copy of database is deleted.

will not be damaged even if there is a system failure.. when the system restarts, it will read db-pointer and will see the contents of all database after all the updates are transformed by the transaction.

Concurrent Executions:-

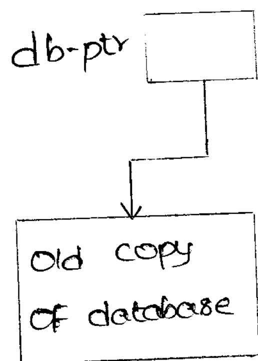
Transaction processing systems allow multiple transactions to run concurrently. There are two (2) good reasons for allowing concurrency.

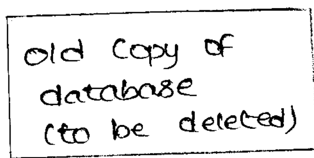
* Improved throughput and resource utilization:-

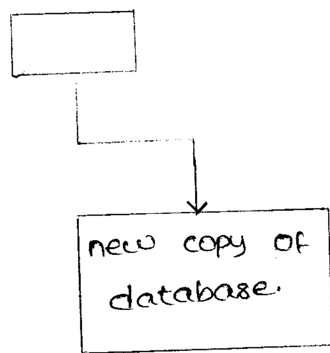
To increase the throughput of the system (the number of transactions that can be executed in a given amount of time) The transaction processing system usually allow multiple transactions to run concurrently It also increases the CPU and disk utilization.

* Reduced Waiting Time:-

There may be mix of transactions running on a system some short and some long. If transactions run serially, a short transaction may have to wait for a long transaction to complete which can lead to unpredictable delays, in running a transaction. If transactions are running on different parts of the database they are allowed to share CPU cycles and disk accesses among them. So concurrent transactions reduces the unpredictable delays in running transactions. It also reduces average response time (the average time for a transaction to be completed after it has been submitted)







a) Before update   b) After update.

If the updated db-pointer written to disk then transaction is said to be committed.

Now we consider how the technique handless transaction and System failures.

Transaction Failure:-

If the transaction fails at any time before the db-pointer is updated we can abort the transaction by deleting the new copy of the database. If the transaction has been committed all the updates performed by that transaction are in the database pointed to by db-pointer

System Failure:-

If the system Fails at any time before the updated db-pointer is written to disk when the system restarts it will read db-pointer and see the original contents of the database and none of effects of the transactions will be visible on the database

If System fails after the db-pointer has been updated on disk. Before the pointer is updated, all the updated pages of the new copy of the database were written to disk. We assume that once a file is written to disk, its contents

When several transactions run concurrently database consistency can be destroyed despite the correctness of each individual transaction. To ensure the consistency of the database the database system must control the interactions among the concurrent transactions.

Consider the following transactions

Let T_1 and T_2 are transactions that transfer funds from one account to another transaction T_1 transfers $50 from account A to account B. It is defined as

```
T1 :-   read(A) ;
        A:= A-50;
        write (A);
        read(B)
        B:= B+50;
        write (B) ;
```

Transaction T_2 transfers 10% of balance from account A to account B;

```
T2 :-   read(A)
        temp:=A * 0.1;
        A:= A - temp;
        write (A);
        read (B);
        B:= B+ temp;
        write(B);
```

Suppose the current values of accounts A and B are $1000 and $ 2000.

Schedules:-

Schedule indicates the way in which we are executing the transactions i.e it represents an actual execution sequence

Serial Schedule:-

If all available transactions are executed from start to finish one by one then we call the schedule as a serial schedule.

T_1	T_2
read(A)	
A:= A-50	
write (A)	
read (B)	
B:= B+50	
write (B)	
	read(A)
	temp:= A*0.1
	A = A-temp
	write (A)
	read (B)
	B:= B+temp
	write (B)

Here we are executing the transactions T_1 followed by T_2. After executing the transactions the values of A and B are 855 and 2145 respectively. so A+B is preserved

If the transactions are executed one at a time in the order T_2 followed by T_1, the final values of A and B are 850 and 2150 and the such A+B is preserved.

T1	T2
	read (A)
	temp := A*0.1
	A := A-temp;
	write (A)
	read (B)
	B := B + temp
	write (B)
read (A)	
A := A-50	
write (A)	
read (B)	
B := B+50	
write (B)	

Thus for a set of n transactions there exist n! different valid serial schedules.

When a database system executes several transactions concurrently, the corresponding schedule need not be serial. If two transactions are running concurrently, the O.S executes one transaction for a little while, then perform context switch, execute the second for some time and then switch back to first transaction for some time and so on. Here CPU time is shared among all the transactions.

Several Execution Sequences are possible. It is not possible to predict exactly how many instructions of a transaction will be completed (executed) before CPU switches to another transaction.

The no. of possible schedules for a set of n transactions is much larger than n!

Non Serial Schedule:

If the operations from available set of transactions are interleaved then it is called a non serial schedule

T1	T2
1000 2000	
read (A)	
A = A-50	
write (A)	
950	
	read (A) 95
	temp = A*0.1
	A = A-temp
	write (A)
	A = 855
2000 read (B)	
B = B+50;	
write (B)	
2050	
	read (B)
	B = B + temp 2145
	write (B)

In this case the value of sum of A+B is preserved

All Concurrent executions may not result in a correct state

Ex:-

T1		T2	
read(A)			
A = A-50			1000
950		read(A)	
		temp = A * 0.1	
		A = A - temp	100
		write (A)	900
			900
		read (B)	
900	write (A)		2000
	read (B)		
	B = B+50		
2050	write (B)		
			2150
		B = B + temp	
		write (B)	

After executing the (program) the above schedule we arrive at an inconsistent state i.e sum A+B is not preserved by the execution of the two transactions

If the control of concurrent execution is left entirely to O's many (possible) schedules are possible (some schedules leaves the database in an inconsistent state). It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The concurrency control component of the database system carries out this task.

Dis Advantages of Serial Schedule:-

1. Since there is no interleaving operation there is no proper utilization of CPU when transactions are waiting for i/p and o/p operations.

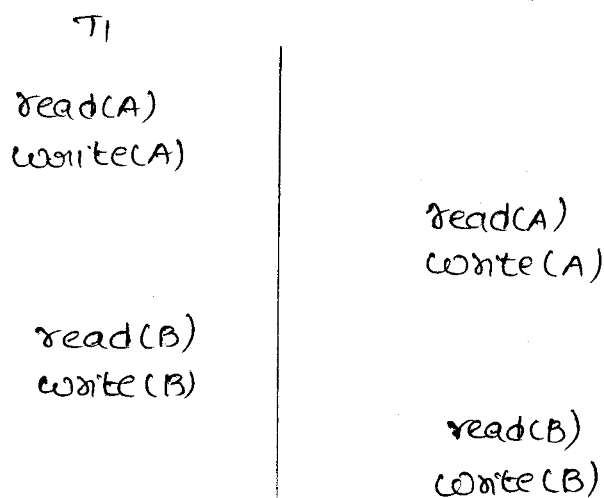
2. Long transaction may block small transaction for a long time.

Serializability :-

Serializability is used to determine which of the non serial schedules always give correct results and which will give error results.

A schedule S of n transactions is said to be serializable if it is equivalent to some serial schedule of the same n transactions i.e. If the interleaved schedule produces same result as that serial schedule then the interleaved schedule is said to be serializable.

Transactions are programs. It is computational difficult to determine (exactly what operations a transaction performs and) how operation of various transactions interact. Here we are considering only 2 operations Read and write



For two schedules to be equivalent, the operations applied to each data item (affected by the schedules should be applied to that item) in both schedules must be in the same order. There are two types of equivalences. They are conflict and view equivalence and they lead to conflict and view serializiablity.

Conflict Serializability:-

Let us consider a schedule in which there are two consective instructions I_i and I_j of transactions T_i and T_j respectively ($i \neq j$). If I_i and I_j refers to different data items then we can swap I_i and I_j without affecting the results of any instruction in the schedule.

But, If I_i and I_j refer to same dataitem then the order of 2 steps may matter. Since we are dealing with read and write operations there are four possible cases:-

1. I_i = read(Q), I_j = read(Q) The order of I_i and I_j doesnot matter since the same value of Q is read by I_i and I_j regardless of the order.
2. I_i = read(Q), I_j = write(Q) If I_i comes before I_j, then T_i doesnot read the value of Q that is written by T_j in instruction I_j. If I_j comes before I_i, then T_i reads the value of Q that is written by T_j. Thus the order of I_i and I_j matters.

   (Don't change their order in any non serial schedule)

3. I_i = write(Q) I_j = read(Q), the order of I_i and I_j matters for reasons similar to those of the previous case.

4. I_i = write(Q) I_j = write(Q). Since both instructions are write operations, the order of these instructions doesnot affect either Ti and Tj. However the value obtained by next read (Q) operation of S is affected, since the result of only the latter of the two write operations is preserved in the database. If there is no other write (Q) operation after I_i and I_j in S, then the order of I_i and I_j directly affects the final value of Q in the database state that results from S.

Two operations are said to be conflict if they satisfy all three conditions.

1) They must belong to 2 different transactions
2) They must access the same data item
3) Atleast one of the operation is write operation

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

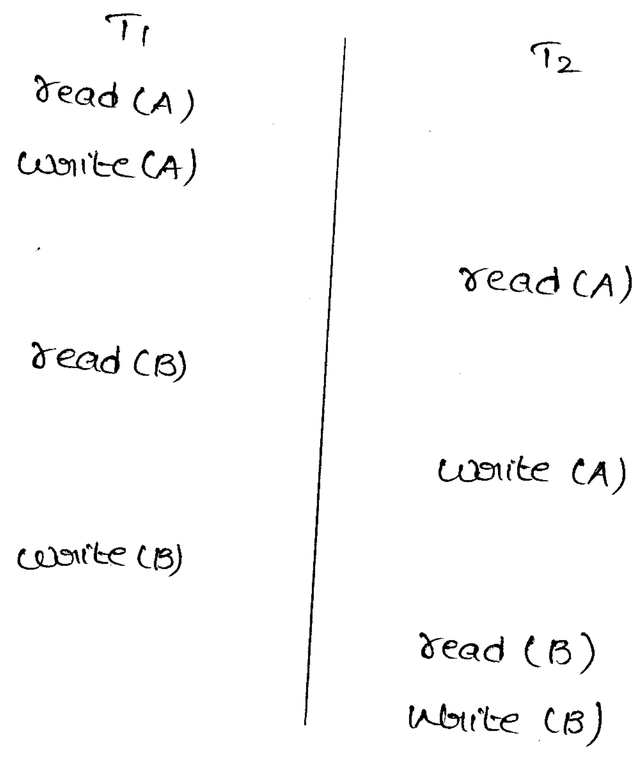
→ The write (A) Instruction of T_1 conflicts with read(A) Instruction of T_2.

→ But write (A) Instruction of T_2 doesnot conflict with read(B) instruction because the two instruction access different data items.

Let I_i and I_j be consective instructions of a Schedule S. If I_i and I_j are instructions of different transactions and I_i and I_j doesnot conflict, then we can Swap I_i and I_j to produce a new schedule S'.

Ex:-

Since the write (A) Instruction of T_2 doesnot conflict with read (B) instruction of T_1 we can ~~supply~~ Swap these instructions.



T_1	T_2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

We Continue to Swap Non Conflicting instructions

→ Swap read(B) of T_1 with read (A) of T_2.

→ Swap write (B) of T_1 with write (A) of T_2

→ Swap write (B) of T_1 with read (A) of T_2.

Then we get

T_1	
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

If a schedule S can be transformed into a schedule S' by a series of swaps of non conflicting instructions, we say that S and S' are conflict equivalent.

View Serializability :-

Consider two schedules S and S' they are said to be view equivalent if the following conditions are met

1) For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must in schedule S' also read the intial value of Q.

2) For each data item Q if transaction Ti executes read(Q) in schedule S' that value was produced by a write(Q) operation executed by transaction Tj. then read(Q) operation of transaction Ti in schedule S' also read the value of Q that was produced by the same write(Q) operation of transaction Tj

3) For each data item Q, the transaction that performs the final write (Q) operation in schedule S must perform the final write (Q) operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and performs same computation. condition 3 coupled with condition 1 and 2 ensures that both schedule result in the same final system state

Ex:-

Schedule 1 :-

T1	T2
read(A)	
A = A-50 ;	
write (A)	
B = B+50	
write(B)	
	read(A)
	temp = A*0.1
	A = A-temp;
	write (A)
	read (B)
	B = B+temp
	write (B)

Schedule 2:

T1	T2
	read(A)
	temp = A*0.1
	A = A-temp
	write(A)
	read(B)
	B = B+temp
	write(B)
read(A)	
A = A-50;	
write (A)	
B = B+50;	
write (B)	

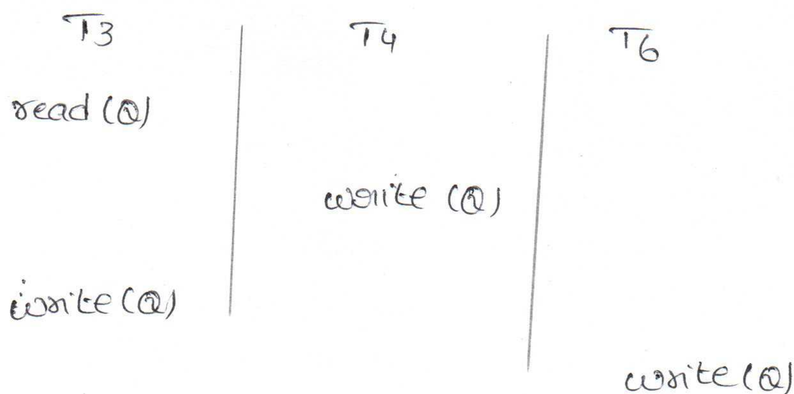
Schedule 3:-

T_1	T_2
read (A)	
A = A-50;	
write (A)	
	read(A)
	temp = A*0.1
	A = A-temp
	write (A)
read (B)	
B = B+50;	
write(B)	
	read(B)
	B = B+temp
	write (B)

Here schedule 1 is not view equivalent to schedule 2 since the value account A read by T_2 was produced by T_1 in schedule 1, where as this case doesnot hold in schedule 2 However schedule1 is view equivalent to schedule 3. why because the values of A and B read by T_2 was produced by T_1 in both cases.

Every conflict serializable schedule is also view serializable but these are some view serializable schedules that are not conflict serializable

Consider the following schedule



T3	T4	T6
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

The above schedule is not conflict serializable since every pair of consective instructions conflict and no swapping instructions are possible.

In the above schedule T4 & T6 performs write(Q) operation without performing a read (Q) operation writes of this type is called <u>Blind writes</u>.

Blind writes appear in any view serializable schedule that is not conflict serializable.

Recoverability :

Assume if a transaction Ti fails, we need to undo the effect this transaction to ensure atomicity property. In a System that Allows Concurrent execution, it is necessary to ensure that any transaction Tj that is also dependent on Ti is also aborted. To acheive this surety, we need to place restrictions on the type of schedules in the system.

Recoverable schedule:-

A recoverable schedule is one where for each pair of transaction Ti and Tj that Tj reads the data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj

Consider the following schedule

T8	T9
read(A)	
write(A)	
	read(A)
read(B)	

Here T9 performs only one operation read(A). Suppose the system allows T9 to commit immediately after executing read(A) i.e T9 commits before T8. If suppose T8 fails before it commits. Since T9 has read the value of data item A written by T8 we must abort T9 to ensure transaction atomicity. However T9 is already committed and cannot be aborted. So the above is example for non recoverable schedule.

Cascadeless Schedules:-

Even if a schedule is recoverable to recover correctly from the failure of Transaction Ti we may have to rollback several transactions

Consider the partial schedule.

T10	T11	T12
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)

Transaction T_{10} writes a value of A that is read by T_{11}, T_{11} writes a value of A that is read by T_{12}. Suppose that at this point T_{10} fails. T_{10} must be rolled back. Since T_{11} is dependent on T_{10}, T_{11} must be rolled back, since T_{12} is dependent on T_{11}, T_{12} must be rolled back.

If a single transaction failure leads to a series of transaction rollbacks then it is called cascading roll back.

Cascading rollback is undesirable we have to restrict the schedules such that cascading rollbacks cannot occur. Such schedules are called cascadeless schedules.

Cascadeless Schedule:-

It is one where for each pair of transaction T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Every cascadeless schedule is recoverable.

Testing for Serializability:-

While designing concurrency control schemes we must ensure that schedules generated by schemes are serializable. Most concurrency control schemes donot actually test for serializability rather protocols are developed to guarantee that the schedule is serializable. Hence we must test schedules for serializability. We have efficient methods for testing conflict serializability and no efficient method for testing view serializability

Consider a schedule S. construct a directed graph called precedence graph from S. This graph consists of a pair G=(V,E) where V is set of vertices and E is set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \to T_j$ for which one of three condition holds.

1. T_i executes write(Q) before T_j executes read(Q)
2. T_i executes read(Q) before T_j executes write(Q)
3. T_i executes write(Q) before T_j executes write(Q)

If an edge $T_i \to T_j$ exists in the preedence graph, then in schedules T_i must appear before T_j

Schedule 1

T1	T2
read(A)	
A= A-50	
write(A)	
read(B)	
B= B+50	
write(B)	
	read(A)
	temp= A*0.1
	A= A-temp
	write(A)
	read(B)
	B= B+temp
	write(B)

Schedule 2

T1	T2
	read(A)
	temp= A*0.1
	A= A-temp
	write(A)
	read(B)
	B= B+temp
	write(B)
read(A)	
A= A-50	
write(A)	
read(B)	
B=B+50	
write(B)	

Schedule 4

T_1	T_2
read(A)	
A = A-50	
	read(A)
	temp = A*0.1
	A = A-temp
	write(A)
	read(B)
write(A)	
read(B)	
B = B+50	
write(B)	
	B = B+temp
	write(B)

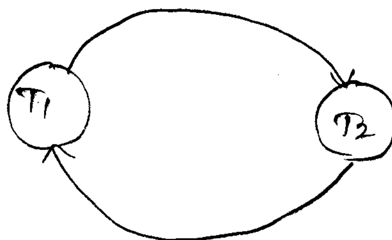
Precedence graph for schedule 1

For schedule

Since all instructions in T_1 are executed before the first instruction of T_2 is executed

The precedence graph for schedule 4

It contains the edge $T_1 \to T_2$, because T_1 executes read(A) before T_2 executes write(A) and the edge $T_2 \to T_1$ because T_2 executes read(B) before T_1 executes write(B)



If the precedence graphs contains cycle then the schedule is not conflict serializable. If the graph contains no cycles then the schedule is serializable. Schedule S_4 contains a cycle so it is not conflict serializable where as S_1 and S_2 are conflict serializable.

Concurrency control:-

When several transactions execute concurrently the isolation property may no longer preserved. To ensure the system must control the interactions among the transactions. This is achieved through concurrency control schemes. The concurrency control schemes are based on serializable property.

Lock Based protocols:-

One way to ensure serializability is to require that data items may be accessed in a mutually exclusive manner i.e., while one transaction is accessing a data item, no other transaction can modify that data item. The most widely used method to implement is to lock, the data item which are required for that transaction.

Lock:-

A lock is a variable that is associated with each data item in the database and describes the status of that data item with respect to possible operations that can be applied to that data item.

There are various modes in which a data item may be locked.

Shared Lock(S):-

If a transaction T_i has obtained a lock in shared mode on item Q, then T_i can read but cannot write Q.

Exclusive Mode/update lock:-

If a transaction T_i has obtained an exclusive mode lock (denoted by X) on item Q, then T_i can both read and Write Q.

Lock Compatibility matrix:-

	S	X
S	True	False
X	False	False

Every transaction request a lock in an appropriate mo on any data item Q, depending upon the type of operation. The transaction makes a request to the concurrency control manager. If the manager grants the lock to the transaction, then transaction proceed with the operation.

Assume A and B are arbitary lock modes. The compatability function is defined as follows. Transaction T_i requests a lock of mode A on item Q on which transaction T_j is currently holding a lock of mode B. If transaction T_i is granted a lock then we can say mode A is compatible with mode B. An element comp(A,B) of the matrix has the value true if and only if mode A is compatible with node B.

A transaction requests a shared lock on data item Q by executing lock-S(Q), exclusive lock by executing lock-X(Q) and unlock a data item by executing Unlock(Q) instruction. To access a data item transaction T_i must first lock that item. If the data item is locked by another transaction in an incompatible locks held by other transactions have been released.

Eg:- T_1 : lock-x(B)
read(B)
B=B-50;
write(B)
unlock(B)
lock-x(A)
read(A)
A=A+50;
write(A)
unlock(A)

T_2 : lock-S(A)
read(A)
unlock(A)
lock-S(B)
read(B)
unlock(B)
display(A+B);

Dead Lock:- consider a partial schedule shown below.

T_3	T_4
lock-x(B)	
read(B)	
B=B-50;	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-x(A)	

Since T_3 is holding exclusive lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly T_4 is holding a shared-mode lock on A and T_3 is requesting Exclusive lock on A, T_3 is waiting for T_4 to unlock A. So, we have arrived at a point where neither of these transactions can never proceed with its normal execution. This situation is called Deadlock. When a dead lock occurs the system must rollback one of the two transactions. Once a transaction has been rolled back the data items that are locked by that transaction unlocked. These data items are then available to the other transactions which can continue with its execution

Locking protocol:-

It is a set of rules, which indicates when a transaction may lock and unlock each of the data items. Locking protocols restricts the no of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedu

Granting of locks:-

When a transaction requests a lock, on a data item in a particular mode and no other transaction has a lock on the same data item in a conflicting mode the lock can be granted Care must be taken to avoid the following scenario.

Transaction T_2 has a shared mode lock on a data item and T_1 requests an Exclusive mode lock on the same data item. T_1 has to wait for T_2 to release a shared mode lock. Meanwhile T_3 may request a shared mode lock on the same data item. The lock request is compatible with lock granted to T_2 so lock may be granted to T_3. But again T_4 also requires a shared lock on same data item and it is granted for T_4. Here T_1 never gets the exclusive mode lock on same the data item. The transaction T_1 may never make progress and it is said to "Starved".

The transaction is left in wait state indefinitely and unable to get new lock even though the system is not in Dead lock.

We can avoid starvation of transaction by granting locks in the following manner:

When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency control M, the concurrency control Manager grants the lock provided that

* There is no other transaction holding a lock on Q in a mode that conflicts with M.
* There is no other transaction that is waiting for a lock on Q and that made its lock requests before T_i.

Two phase Locking protocol:-

It is used to ensure serializability. This protocol requires that each transaction issue lock and unlock requests in two phases.

Growing phase:-

A transaction may obtain locks but may not release any lock.

Shrinking phase:-

A transaction may release locks but may not obtain new locks.

Initially the transaction is in growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase and it can issue no more lock requests.

Ex:-

T_1	T_2
Lock(B)	Lock(B)
Read(B)	read(B)
B = B-50;	B = B-50;
write(B)	write(B)
Unlock(B)	lock(A)
lock(A)	read(A)
Read(A)	A = A+50;
A = A+50;	write(A)
write(A)	unlock(B)
Unlock(A)	unlock(A)

T_1 is not in 2PL but T_2 is in 2PL

Advantages:-

→ 2PL ensures conflict serializability.

Disadvantages:-

→ Does not ensure freedom from deadlock.

Lockpoint:-

The point in the schedule where the transaction has obtained its final lock is called the lockpoint of the transaction (The end of the growing phase). Cascading roll back may occurs in 2PL.

Eg:-

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

The above transactions are in 2PL but the failure of T_5 after read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

Strict 2PL:-

Cascading roll backs can be avoided by a modification of 2PL called the Strict 2PL. In this a transaction cannot release any exclusive locks until the transaction

commits. So no other transaction can read or write an item that is written by the transaction unless it has committed.

Another variant of 2PL is rigorous 2PL which requires that all locks be held until the transaction commits. Most database system implement either 2PL or rigorous 2PL. Transaction is in its expanding phase until it ends.

Lock Conversions:-

We can provide a mechanism for upgrading a shared lock to exclusive lock and downgrading an exclusive lock to shared lock. We denote Conversion from shared to exclusive modes by upgrade and from exclusive to shared by down grade. Upgrading can takes place in growing phase where as downgrading can takes place only in shrinking phase.

Time stamp based protocols:-

The use of locks combined with the two phase locking protocols allows us to guarantee the serializability of schedules. A different approach that gurantees serializability involves using transaction timestamps to order transaction execution for a equivalent serial schedule.

Time stamps:-

A timestamp is a unique identifier created by the DBMS to identify a transaction. Time stamp values are assigned in the order in which transactions are submitted to the system. So a time stamp is considered as the transaction start time. For each transaction T_i in the system a unique time stamp is assigned and it is

denoted by $TS(T_i)$. When a new transaction T_j enters the system then $TS(T_i) < TS(T_j)$. This is known as Time stamp ordering scheme. To implement this scheme each data item(Q) is associated with two time stamp values.

W-timestamp(Q):- It denotes the largest time stamp of any transaction that executed Write(Q) successfully.

R-timestamp(Q):- It denotes the largest time stamp of any transaction that executed the read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

Time Stamp Ordering protocol:-

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in time stamp order. This protocol operates as follows.

1. Suppose that transaction T_i issues read(Q)

a) If $TS(T_i) <$ W-timestamp(Q) then T_i needs to read a value of Q that was already overwritten. Hence the read operation is rejected and T_i is rolled back.

b) If $TS(T_i) \geq$ W-timestamp(Q) then read operation is executed and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and $TS(T_i)$.

2. Suppose that transaction T_i issues write(Q).

a) If $TS(T_i) <$ R-timestamp(Q), this means that a younger transaction is already using the current value of the item and it would be an error to update it now. So the system

rejects the write operation and rolls T_i back.

b) If $TS(T_i) < w\text{-timestamp}(Q)$ then T_i is attempting to write an absolute value of Q. Hence the system rejects this operation and rolls T_i back.

c) Otherwise the system executes write operation and sets w-timestamp(Q) to $TS(T_i)$.

This scheme is called basic timestamp ordering and guarantees that transactions are conflict serializable and the results are equivalent to some serial schedule.

Advantages:-

1. Conflicting operations are processed in timestamp order and therefore it ensures conflict serializability.
2. Since time stamps do not use locks, deadlocks cannot occur.

Disadvantages:-

1. Starvation may occur if a transaction is continually aborted and restorted.
2. It does not ensure recoverable schedules

Multiple Granularity:-

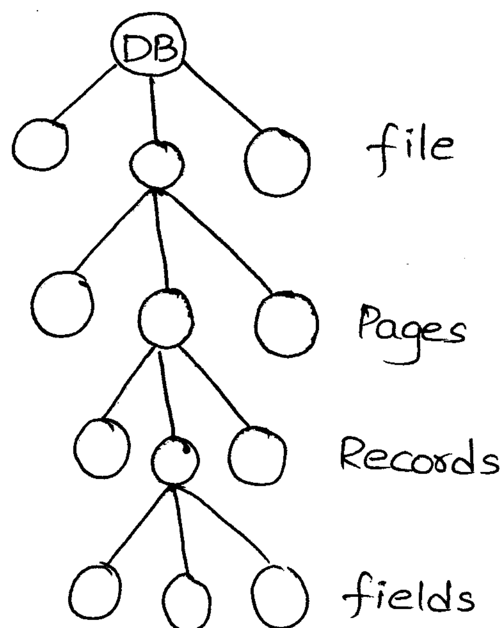
In all the concurrency control schemes, we have used each individual data item as the unit on which synchronization is performed. However it would be advantageous to group several data items and to treat them as one individual unit.

Eg:- If a transaction T_i needs to access the entire data base it uses a locking protocol. Here T_i must lock each data item in the database so it is time consuming process. So it would be better if T_i would issue a single lock request to lock the entire database. On the otherhand if transaction T_j needs to access only a few data items it should not be required to lock the entire database.

A data item can be one of the following

- database record
- Field value of a database record
- A disk block
- a whole file
- whole database.

The size of the datatitem is called the data item granularity.



Depending upon the application we are going for highest level of locking i.e., the entire database or lowest level of locking(locking data items).

We can use Shared and Exclusive locks to lock a node. When a transaction locks a node in shared or exclusive mode all the descendants of the node are also locked in the same mode. To make multiple granularity level locking practical, additional types of locks called intension Locks are needed. The idea behind intension lock is for a transaction to indicate along the path from root to the desired node, what type of lock it will require from one of node's descendants. There are three types of intension locks.

1. Intension-shared (IS) indicate that a shared lock will be requested on some descendant node.
2. Intention-exclusive (IX) indicate that an exclusive lock(x) will be requested on some descendant node(s).
3. Shared-intension-exclusive (SIX) indicates that the current node is locked in shared mode but an exclusive lock will be requested on some descendant node(s).

Compatability matrix:-

	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Each transaction T_i can lock the data item using the following rule.

1) It must observe the lock compatibility function in the above table.
2) It must lock the root of the tree first and can lock it in any mode.
3) It can lock a node Q in S or IS node only if it currentl has the parent of Q locked in either IS or IX mode.
4) It can lock a node Q in X, SIX or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5) It can lock a node only if it has not previously unlocke any node (T_i is 2 phase).
6) It can unlock a node Q only if it currently has none of the children of Q locked.

Disadvantages:-

* Dead lock may occur.