CONCURRENCY CONTROL

Concurrency control is the process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control.

- Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.

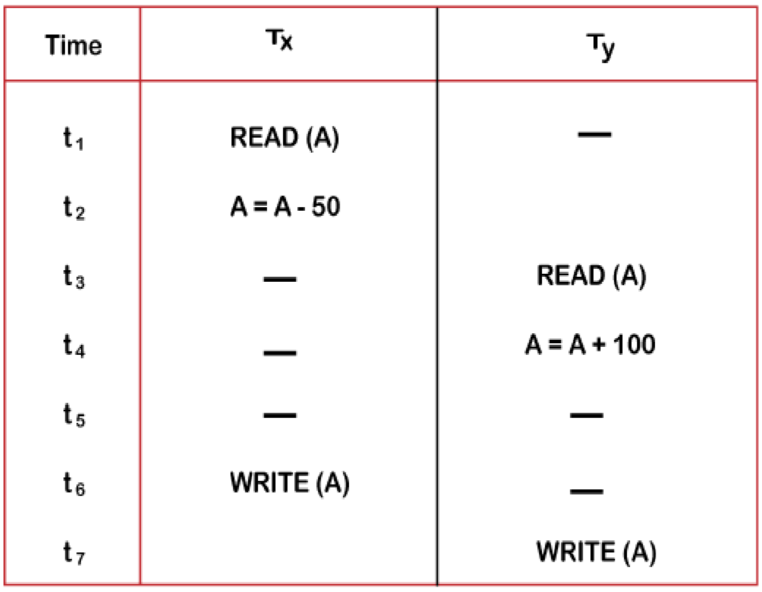
- lost updated problem
- Temporary updated problem
- Incorrect summary problem

Problem 1: Lost Update Problems

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

For example:

Consider the below diagram where two transactions TX and TY, are performed on the same account A where the balance of account A is $300.



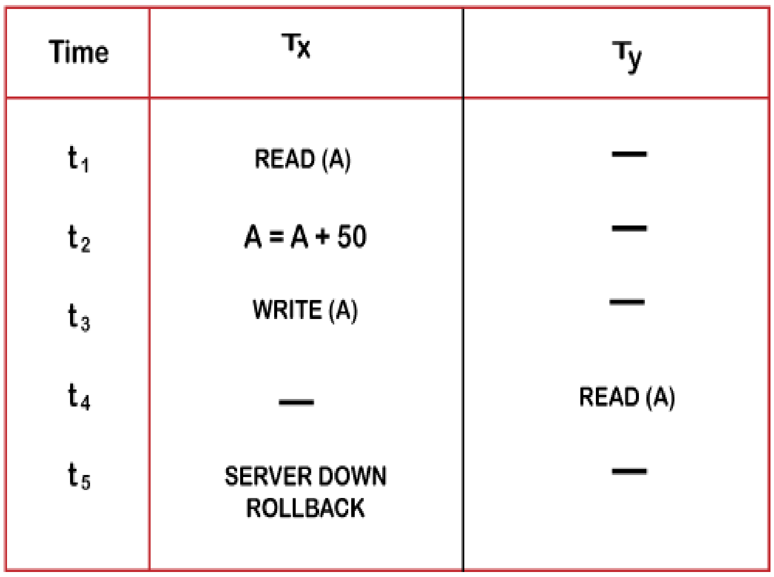
Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY didn't update the value yet.
- Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.
- Hence data becomes incorrect, and database sets to inconsistent.

2. Temporary updated problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.
- Occurs when one transaction can see intermediate results of another transaction

  before it has committed.

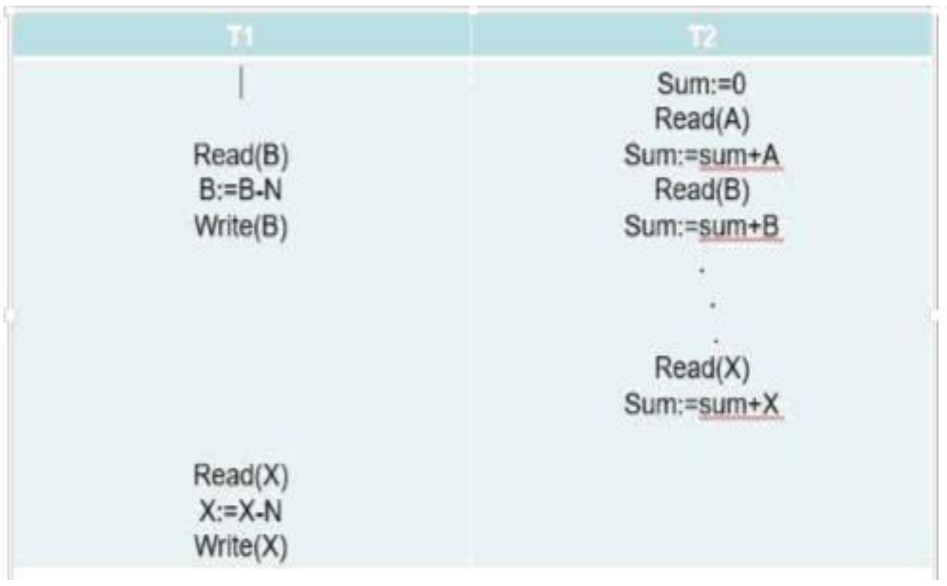


Time	Tx	Ty
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

- At time t1, transaction TX reads the value of account A, i.e., $300.
- At time t2, transaction TX adds $50 to account A that becomes $350.
- At time t3, transaction TX writes the updated value in account A, i.e., $350.
- Then at time t4, transaction TY reads account A that will be read as $350.
- Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Incorrect summary problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.
- Occurs when transaction reads several values but second transaction updates some of them during execution of first.



T1	T2
|	Sum:=0
	Read(A)
Read(B)	Sum:=sum+A
B:=B-N	Read(B)
Write(B)	Sum:=sum+B
	.
	.
	.
	Read(X)
	Sum:=sum+X
Read(X)	
X:=X-N	
Write(X)	

Example:

- T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, T5 has transferred £10 from bal(x) to bal(z), so T6 now has wrong result (£10 too high).
- Problem avoided by preventing T6 from reading bal(x) and bal(z) until after T5 completed updates.

DEADLOCK

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Example : – let us understand the concept of Deadlock with an example

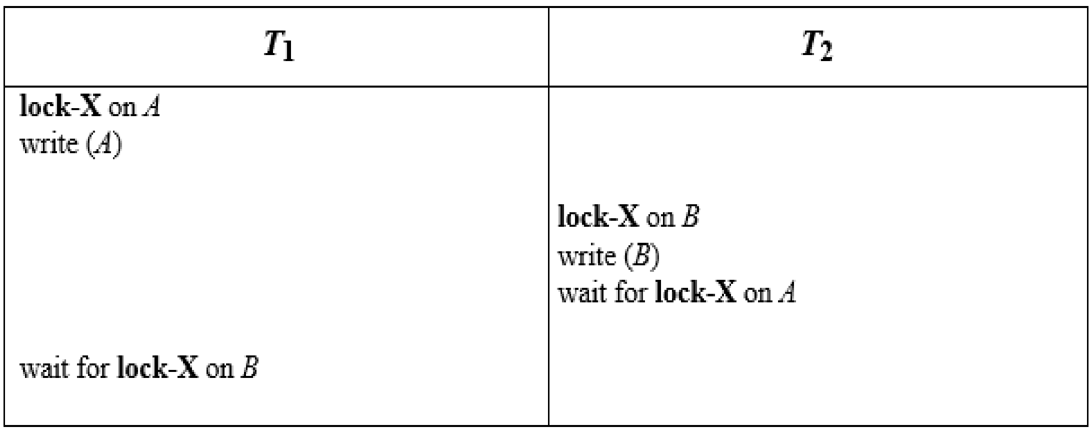
Suppose, Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table. Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.

Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

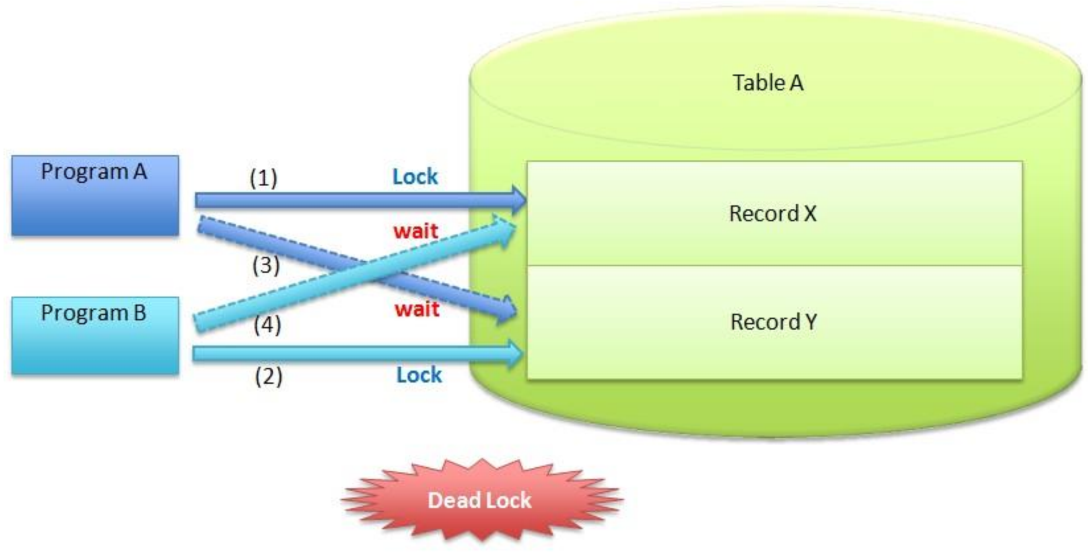
System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set. Consider the following two transactions:

```
T1:          T2
write (A)    write(A)
write (B)    write(B)
```

Schedule with deadlock



T_1	T_2
lock-X on *A* write (*A*)	
	lock-X on *B* write (*B*) wait for **lock-X** on *A*
wait for **lock-X** on *B*	



Deadlock Handling

Deadlock prevention protocol Ensure that the system will never enter into a deadlock

state.

Some prevention strategies :

Approach1

- Require that each transaction locks all its data items before it begins execution either all are locked in one step or none are locked.

Disadvantages

- Hard to predict, before transaction begins, what data item need to be locked.
- Data item utilization may be very low.

Approach 2 – Assign a unique timestamp to each transaction. – These timestamps only to decide whether a transaction should wait or rollback.
Deadlock prevention Schemes:

- wait-die scheme
- wound-wait scheme

(i) wait-die scheme

- Non preemptive technique
- When transaction Ti request a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj. otherwise ,Ti rolled back(dies)
- Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
- A transaction may die several times before acquiring needed data item

Example.

Transaction T1,T2,T3 have time stamps 5,10,15,respectively.

- if T 1 requests a data item held by T2,then T1 will wait.
- If T3 request a data item held by T2,then T3 will be rolled back.

(ii) Wound-wait scheme

- Preemptive technique
- When transaction Ti requests a data item currently held by Tj,Ti is allowed to wait only if it has a timestamp larger than that of Tj. Otherwise Tj is rolled back
- Older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

Example

Transaction T1,T2,T3 have time stamps 5,10,15,respectively.

- if T1 requests a data item held by T2,then the data item will be preempted from T2,and T2 will be rolled back.
- If T3 requests a data item held by T2,then T3 will wait.

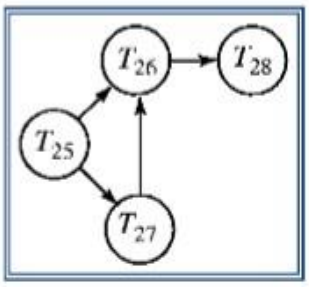
DeadLock Detection

Deadlocks can be described as a wait-for graph, which consists of a pair G = (V,E)

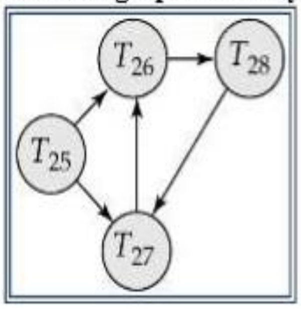
- V is a set of vertices
- E is a set of edges

- If Ti ->Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for graph without a cycle



Wait-for graph with a cycle



Recovery from deadlock

- The common solution is to roll back one or more transactions to break the deadlock.
- Two action need to betaken

- Selection of victim
- Rollback

Selection of victim

- Set of deadlocked transactions, must determine which transaction to roll back to break the deadlock.
- Consider the factor minimum cost

Rollback

- once we decided that a particular transaction must be rolled back, must determine how far this transaction should be rolled back
- Total rollback
- Partial rollback

Starvation :

Starvation or Livelock is the situation when a transaction has to wait for a indefinite period of time to acquire a lock.

Reasons of Starvation –

- If waiting scheme for locked items is unfair. (priority queue)
- Victim selection. (same transaction is selected as a victim repeatedly)
- Resource leak.
- Via denial-of-service attack.

Starvation can be best explained with the help of an example

- Suppose there are 3 transactions namely T1, T2, and T3 in a database that are trying to acquire a lock on data item ' I '.
- Now, suppose the scheduler grants the lock to T1(maybe due to some priority), and the other two transactions are waiting for the lock.
- As soon as the execution of T1 is over, another transaction T4 also come over and request unlock on data item I. Now, this time the scheduler grants lock to T4, and T2, T3 has to wait again.
- In this way if new transactions keep on requesting the lock, T2 and T3 may have to wait for an indefinite period of time, which leads to **Starvation.**

What are the solutions to starvation –

(i) Increasing Priority –

Starvation occurs when a transaction has to wait for an indefinite time, In this situation, we can increase the priority of that particular transaction/s. But the drawback with this solution is that it may happen that the other transaction may have to wait longer until the highest priority transaction comes and proceeds.

(ii) Modification in Victim Selection algorithm –

If a transaction has been a victim of repeated selections, then the algorithm can be modified by lowering its priority over other transactions.

(iii) First Come First Serve approach –

A fair scheduling approach i.e FCFS can be adopted, In which the transaction can acquire a lock on an item in the order, in which the requested the lock.

(iv) Wait die and wound wait scheme –

These are the schemes that use the timestamp ordering mechanism of transaction.

Intent locking

- Intent locks are put on all the ancestors of a node before that node is locked explicitly.
- If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree.

Types of Intent Locking

- Intent shared lock(IS)
- Intent exclusive lock(IX)
- Shared lock (S)
- Shared Intent exclusive lock (SIX)
- Exclusive lock (X)

Intent shared lock(IS)

- If a node is locked in indent shared mode, explicit locking is being done at a lower level of the tree, but with only shared-mode lock
- Suppose the transaction T1 reads record ra2 in file Fa. Then,T1 needs to lock the database, area A1,and Fa in IS mode, and finally lock ra2 in S mode.

Intent exclusive lock(IX)

- If a node is locked in intent locking is being done at a lower level of the tree, but with exclusive mode or shared-mode locks.
- Suppose the transaction T2 modifies record ra9 in file Fa. Then,T2 needs to lock the database, area A1,and Fa in IX mode, and finally to lock ra9 in X mode.

Shared Intent exclusive lock (SIX)

If the node is locked in Shared Intent exclusive mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at lower level with exclusive mode

Shared lock (S)

-T can tolerate concurrent readers but not concurrent updaters in R.

Exclusive lock (X)

-T cannot tolerate any concurrent access to R at all. Lock compatibility

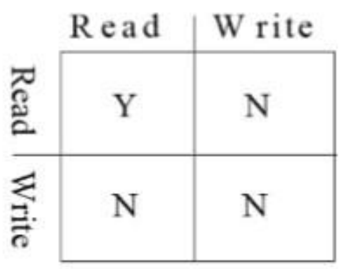
LOCKING PROTOCOLS

A lock is a variable associated with a data item that describe the statues of the item with respect to possible operations that can be applied to it. Locking is an operation which secures
(a) Permission to Read

(b) Permission to Write a data item for a transaction.

Example:

Lock (X). Data item X is locked in behalf of the requesting transaction. Unlocking is an operation which removes these permissions from the data item. Example:
Unlock (X): Data item X is made available to all other transactions. Lock and Unlock are Atomic operations.



	Read	Write
Read	Y	N
Write	N	N

Lock Manager:

- Managing locks on data items.

Lock table:

- Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list

Types of lock

- Binary lock
- Read/write(shared / Exclusive) lock

Binary lock –

It can have two states (or) values 0 and 1.
0 – unlocked
1 - locked

- Lock value is 0 then the data item can accessed when requested.
- When the lock value is 1, the item cannot be accessed when requested.

Binary Lock

Lock_item(x)

```
    B : if lock(x) = 0 ( * item is
    unlocked * ) then lock(x)
                      //1
    else begin
           wait ( until lock(x)
           = 0 ) goto B;
    end;
```

Unlock_item(x)

```
    B : if lock(x)=1 ( * item is locked * )
    then unlock(x)                ¥¥ 0
    else
           printf (‗ already is
           unlocked ‗) goto B;
    end;
```

Read / write(shared/exclusive) lock

Read_lock

- Its also called shared-mode lock
- If a transaction Ti has obtain a shared-mode lock on item X, then Ti can read, but cannot write , X.
- Outer transactions are also allowed to read the data item but cannot write.

Read_lock(x)

```
B : if lock(x) = "unlocked" then                 (1)
begin
       else if
```

```
l      )
o                        // "read_locke
c      d"  ) read(x)      //1
k
(      lock(x) =  "read_locked"   then                   (2)
x      read(x)              //no_of_read(x) +1
   else begin
       wait (until lock(x) =
        "unlocked" ) goto  B;
       end;
```

Write_lock(x)

```
B : if lock(x) = “unlocked” then                               (1)
begin
        lock(x)          //”write_locked”
else
if
        lock(x) = “write_locked”                               (2)
        wait ( until lock(x) = “unlocked”)

else begin
        lock(x)=“read_locked” then                        (3)
        wait ( until lock(x) = ―unlocked‖ )

end;
```

Unlock(x)

```
If lock(x) = “write_locked” then
begin
        unlock(x)                \\“unlocked”
else
if
        lock(x) = “read_locked” then

begin
        read(x)                  \\no_of_read(x) - 1
        if ( no_of_read(x) = 0 ) then

begin
        unlock(x)                \\“unlocked”
end
```

TWO PHASE LOCKING PROTOCOL

This protocol requires that each transaction issue lock and unlock request in two phases

- Growing phase
- Shrinking phase

Growing phase

During this phase new locks can be occurred but none can be released

Shrinking phase

During which existing locks can be released and no new locks can be occurred

Let's see a transaction implementing 2-PL.

	T_1	T_2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	…….	……
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	

This is just a skeleton transaction which shows how unlocking and locking works with 2-PL.
Note for:
Transaction T1:

- Growing Phase is from steps 1-3.
- Shrinking Phase is from steps 5-7.
- Lock Point at 3

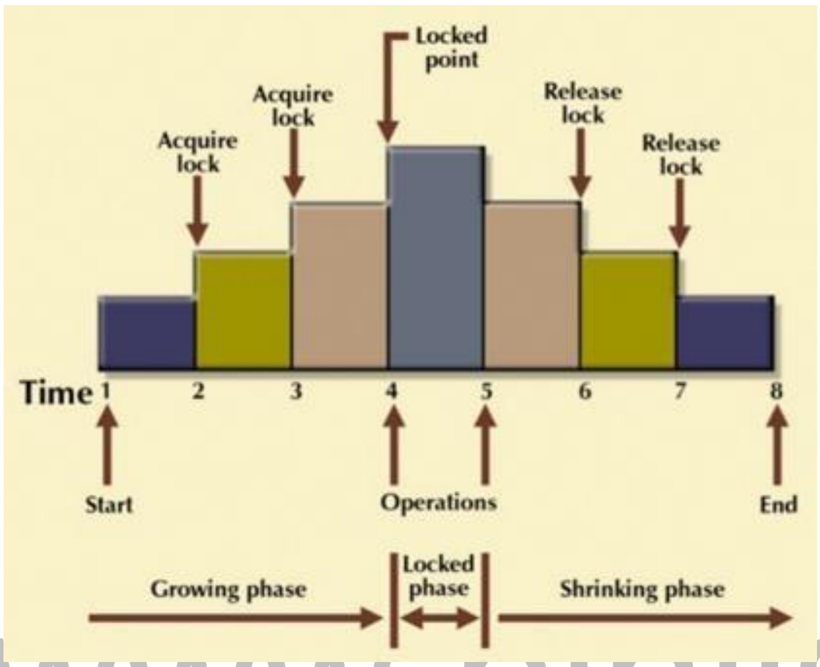
Transaction T2:

- Growing Phase is from steps 2-6.
- Shrinking Phase is from steps 8-9.
- Lock Point at 6

What is **LOCK POINT** ? The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work.

Types of two phase protocol

- Strict two phase locking protocol
- Rigorous two phase locking protocol



Strict two phase locking protocol

This protocol requires not only that locking be two phase, but also all exclusive locks taken by a transaction be held until that transaction commits

Rigorous two phase locking protocol

This protocol requires that all locks be held until all transaction commits.

Consider the two transaction T1 and T2

T1 :

```
        read(a1);
        read(a2);
        .......
        read(an);
    write(a1);
```

T2: read(a1);read(a2);

display(a1+a1);

Lock conversion

- Lock Upgrade
- Lock Downgrade

Lock upgrade:

- Conversion of existing read lock to write lock
- Take place in only the growing phase

```
if  Ti has a read-lock (X) and Tj has no read-lock (X) (i != j)
        then convert read-lock (X) to write-lock (X)
else
        force Ti to wait until Tj unlocks X
```

Lock downgrade:

- Conversion of existing write lock to read lock
- Take place in only the shrinking phase

Ti has a write-lock (X) (*no transaction can have any lock on X*)

convert write-lock (X) to read-lock (X)

T_1	T_2
Lock-S(a_1)	
	Lock-S(a_1)
Lock-S(a_2)	
	Lock-S(a_1)

Lock-S(a_3)
Lock-S(a_4)	

	Unlock(a_1)
	Unlock(a_2)

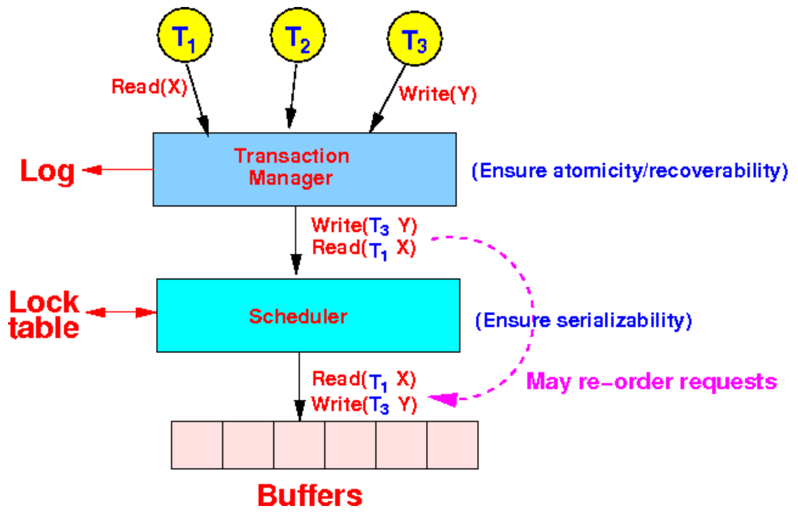
Lock-S(a_1)
Upgrade(a_1)	

Log

- Log is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures.
- Physically, a log is a file of updates done to the database, stored in stable storage.

Log rule

- A log records for a given database update must be physically written to the log, before the update physically written to the database.
- All other log record for a given transaction must be physically written to the log, before the commit log record for the transaction is physically written to the log.
- Commit processing for a given transaction must not complete until the commit log record for the transaction is physically written to the log.
-



System log

- [Begin transaction ,T]
- [write_item , T, X , oldvalue,newvalue]
- [read_item,T,X]
- [commit,T]
- [abort,T]

➢ Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.

➢ Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.

➢ The protocol involves all the local sites at which the transaction executed

➢ Let *T* be a transaction initiated at site *Si*, and let the transaction coordinator at *Si* be *Ci*

Phase 1: Obtaining a Decision (prepare)

- Coordinator asks all participants to *prepare* to commit transaction *Ti*.

- Ci adds the records <prepare *T*> to the log and forces log to stable storage
- sends prepare *T* messages to all sites at which *T* executed

- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <no *T*> to the log and send abort *T* message to *Ci*
  - if the transaction can be committed, then:
  - add the record <ready *T*> to the log
  - force *all records* for *T* to stable storage
  - send ready *T* message to C*i*

Phase 2: Recording the Decision (commit)

- *T* can be committed of *Ci* received a ready *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <commit *T*> or <abort *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site *Si* recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <commit *T*> record: site executes redo (*T*)
- Log contains <abort *T*> record: site executes undo (*T*)
- Log contains <ready *T*> record: site must consult C*i* to determine the fate of *T*.

- If *T* committed, redo (*T*)

- If *T* aborted, undo (*T*)

- The log contains no control records concerning *T* replies that Sk failed before responding to the prepare *T* message from Ci
  - since the failure of *Sk* precludes the sending of such a response *Ci* must abort *T*
  - *Sk* must execute undo (*T*)

Handling of Failures- Coordinator Failure

If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:

1. If an active site contains a <commit *T*> record in its log, then *T* must be committed.
2. If an active site contains an <abort *T*> record in its log, then *T* must be aborted.
3. If some active participating site does not contain a <ready *T*> record in its log, then the failed coordinator *Ci* cannot have decided to commit *T*. Can therefore abort *T*.
4. If none of the above cases holds, then all active sites must have a <ready *T*> record in their logs, but no additional control records (such as <abort *T*> of <commit *T*>). In this case active sites must wait for *Ci* to recover, to find decision.
   - Blocking problem : active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

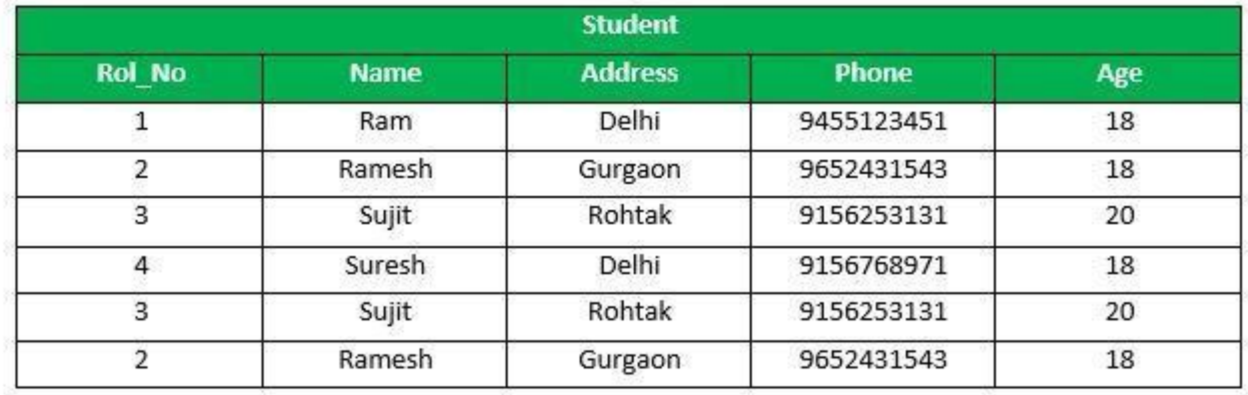
- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
- No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
- Again, no harm results

SAVE POINTS

Before knowing about the save points let us discuss about the commit and rollback operations

COMMIT: If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called **committed.** The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command. **Syntax:**

COMMIT;



Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18

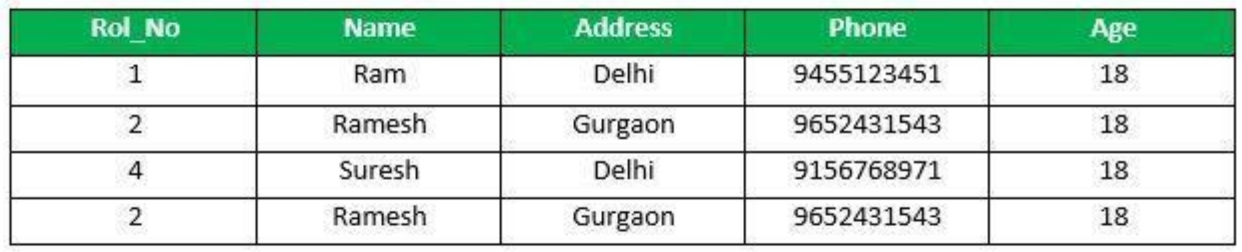
Following is an example which would delete those records from the table which have age = 20 and then COMMIT the changes in the database.

Queries:

COMMIT;

Output:

Thus, two rows from the table would be deleted and the SELECT statement would look like,



Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
4	Suresh	Delhi	9156768971	18
2	Ramesh	Gurgaon	9652431543	18

4. ROLLBACK: If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback**. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued. **Syntax:**

ROLLBACK;

Example:

From the above example **Sample table1**,

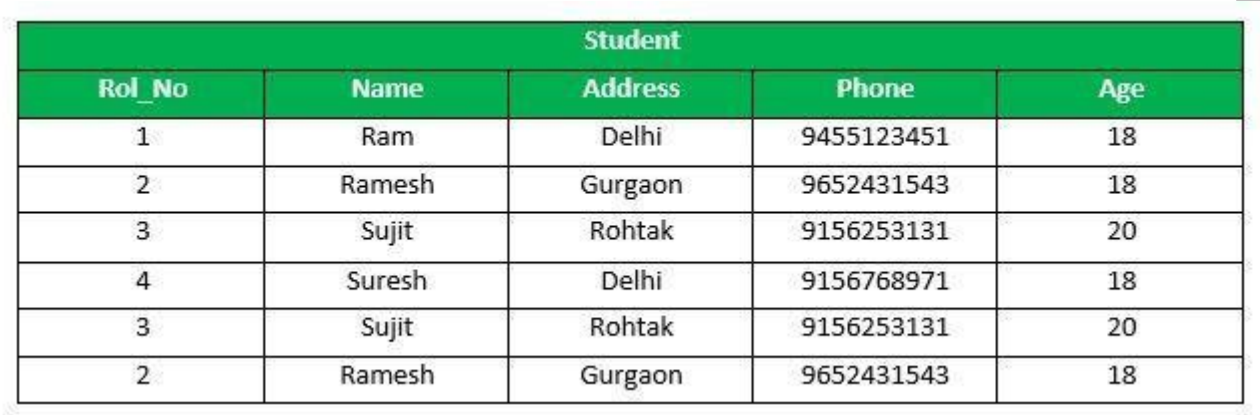
Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database.

Queries:

DELETE FROM Student WHERE AGE = 20;

ROLLBACK;

Output:



Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18

SAVEPOINT: creates points within the groups of transactions in which to ROLLBACK.

A SAVEPOINT is a point in a transaction in which you can roll the transaction [illegible]

Syntax for Savepoint command:

SAVEPOINT SAVEPOINT_NAME;

This command is used only in the creation of SAVEPOINT among all the transactions. In general ROLLBACK is used to undo a group of transactions.

Syntax for rolling back to Savepoint command:

ROLLBACK TOSAVEPOINT_NAME;

We can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example:

From the above example **Sample table1**,

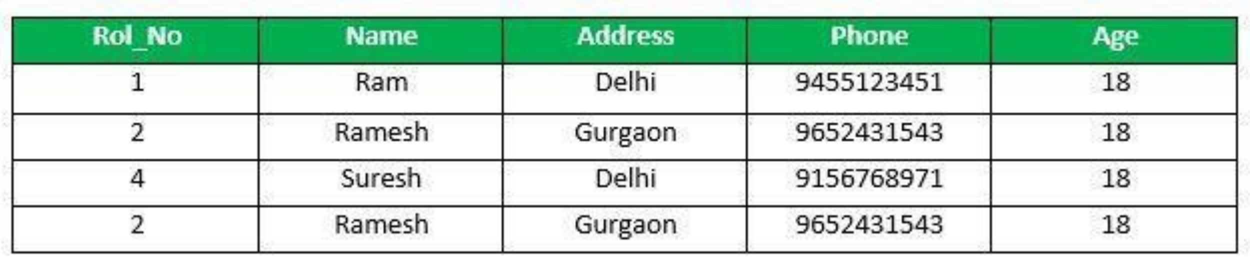
Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database by keeping Savepoints.

Queries:

```
SAVEPOINT SP1;                    //Savepoint
  created. DELETE FROM Student WHERE AGE = 20;
                                  //deleted
  SAVEPOINT SP2;                        //Savepoint created.
```

Here SP1 is first SAVEPOINT created before deletion. In this example one deletion have taken place.

After deletion again SAVEPOINT SP2 is created.



Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
4	Suresh	Delhi	9156768971	18
2	Ramesh	Gurgaon	9652431543	18

Deletion have been taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP1 which is before deletion.

ROLLBACK TO SP1; //Rollback

2. ISOLATION LEVEL

- Degree of interference
- An isolation levels mechanism is used to isolate each transaction in a multi-user environment
- **Dirty Reads**: This situation occurs when transactions read data that has not been committed.
- **Nonrepeatable Reads**: This situation occurs when a transaction reads the same query multiple times and results are not the same each time
- **Phantoms**: This situation occurs when a row of data matches the first time but does not match subsequent times

Types

Higher isolation level (Repeatable read)

- Less interference
- Lower concurrency
- All schedules are serializable

Lower isolation level (cursor stability)

- More interference
- Higher concurrency
- Not a serializable

One special problem that can occur if transaction operates at less than the maximum isolation level (i.e) less then repeatable read level is called phantom problem.

SCHEDULES:

Schedule is a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

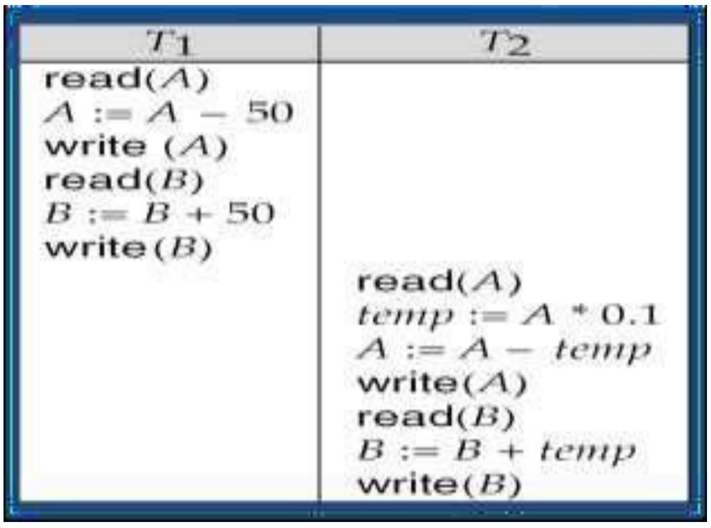
- A schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction.

Serial Schedule

It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

Schedule 1

- Let *T1* transfer 50 from *A* to *B*, and *T2* transfer 10% of the balance from *A* to *B.*
- A serial schedule in which *T1* is followed by *T2* :



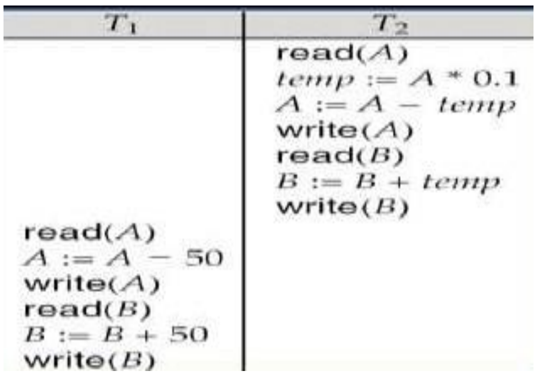
T_1	T_2

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)

Schedule 2

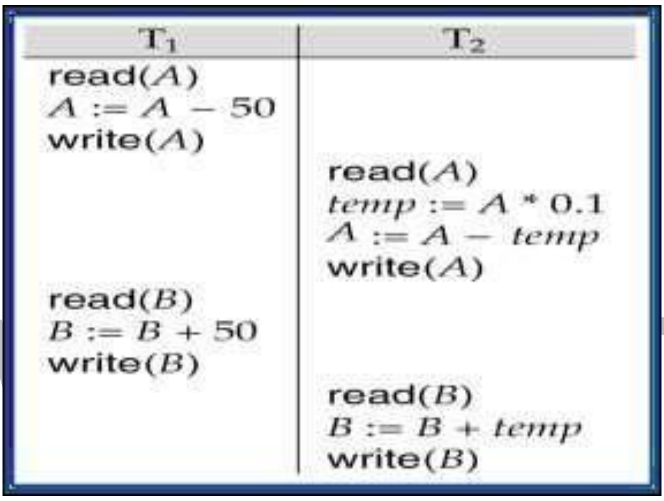
- A serial schedule where *T2* is followed by *T1*



T_1	T_2
	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 3

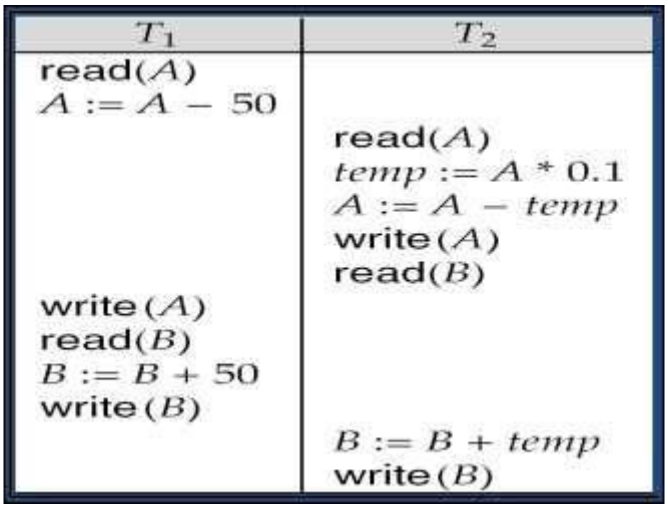
- Let *T1* and *T2* be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.



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)

Schedule 4

The following concurrent schedule does not preserve the value of ($A + B$).



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)

SERIALIZABILITY

- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.
- Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.
  - Serializability is the classical concurrency scheme.
  - It ensures that a schedule for executing concurrent transactions is equivalent

to one that executes the transactions serially in some order.

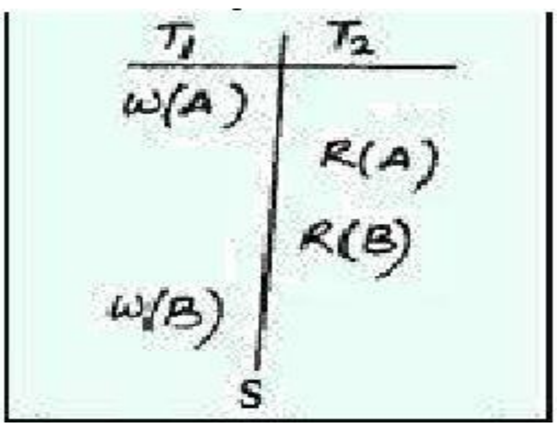
Serializable schedule

If a schedule is equivalent to some serial schedule then that schedule is called Serializable schedule

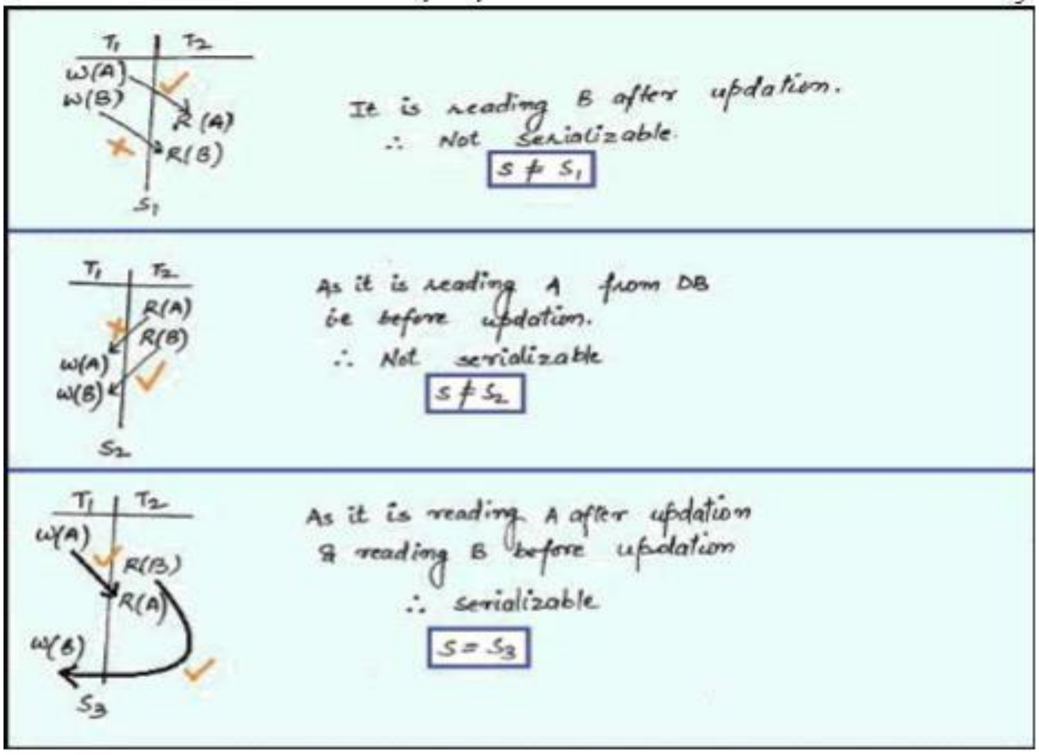
Let us consider a schedule S.

What the schedule S says ?

- Read A after updation.
- Read B before updation.



Let us consider 3 schedules S1, S2, and S3. We have to check whether they are serializable with S or not ?



It is reading B after updation.
∴ Not Serializable
$\boxed{S \neq S_1}$

As it is reading A from DB ie before updation.
∴ Not serializable
$\boxed{S \neq S_2}$

As it is reading A after updation & reading B before updation
∴ serializable
$\boxed{S = S_3}$

Types of Serializability

1. Conflict Serializability
2. View Serializability

1. Conflict Serializable

Any given concurrent schedule is said to be Conflict Serializable if and only if it is Conflict equivalent to one of the possible serial schedule.

Two schedules would be conflicting if they have the following properties

- Both belong to separate transactions.
- Both accesses the same data item.
- At least one of them is "write" operation.

Conflicting Instructions :

Ii and Ij of transactions Ti and Tj respectively, conflict if they are operations by different transaction on the same data item, and at least one of these instruction is write operation.

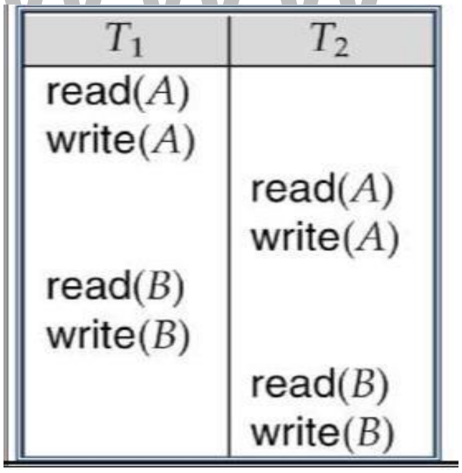
1. Ii = read(Q), Ij = read(Q). Ii and Ij don't conflict.
2. Ii = read(Q), Ij = write(Q). They conflict.
3. Ii = write(Q), Ij = read(Q). They conflict

4. Ii = write(Q), Ij = write(Q). They conflict

Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if

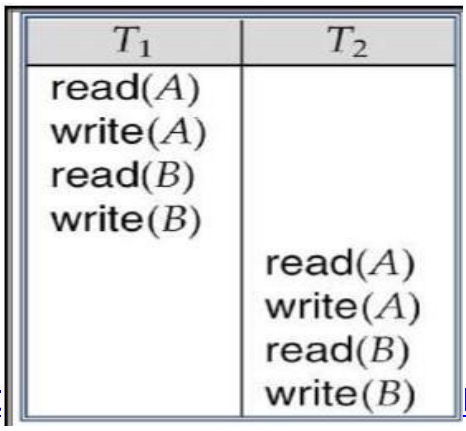
- Both the schedules contain the same set of Transactions.
- The order of conflicting pairs of operation is maintained in both the schedules.
- If a schedule S can be transformed into a schedule S´ by a series of swaps of non- conflicting instructions, we say that S and S´ are conflict equivalent.
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule
- Schedule 3 can be transformed into Schedule 6, a serial schedule where T2 follows T1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule 3



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

Schedule 6



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

2. View Serializable

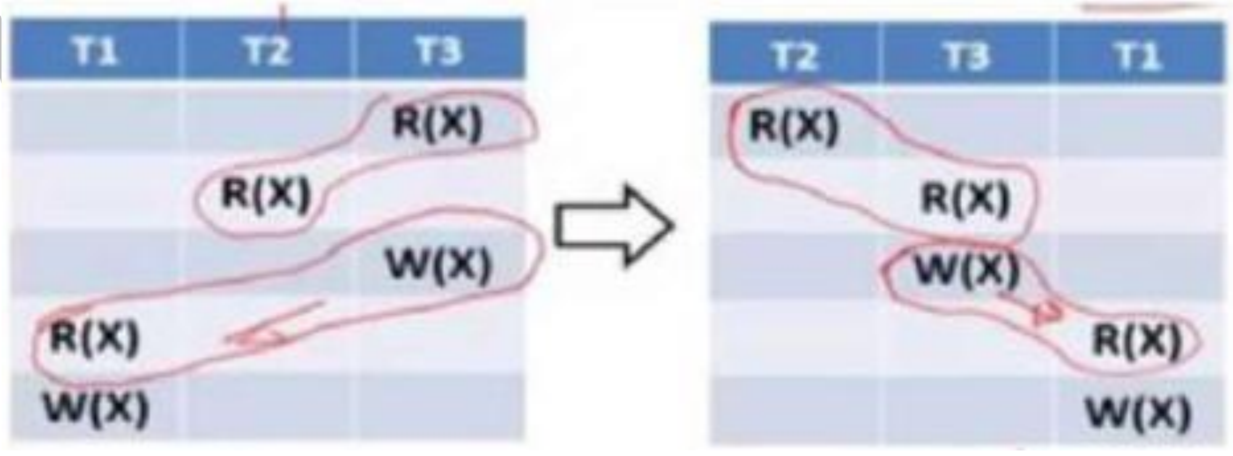
Any given concurrent schedule is said to be View Serializable if and only if it is View equivalent to one of the possible serial schedule.

Let S and S′ be two schedules with the same set of transactions. S and S′ are view equivalent if the following three conditions are met, for each data item Q,

1. If in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.

2. If in schedule S, transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .

3. The transaction (if any) that performs the final write(Q) operation in schedule S must also perform the final write(Q) operation in schedule S'.



T1	T2	T3
		R(X)
	R(X)	
		W(X)
R(X)		
W(X)		

T2	T3	T1
R(X)		
	R(X)	
	W(X)	
		R(X)
		W(X)

TRANSACTION CONCEPTS

A transaction is a collection of operations that forms single logical unit of work.

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friend's account balance
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction

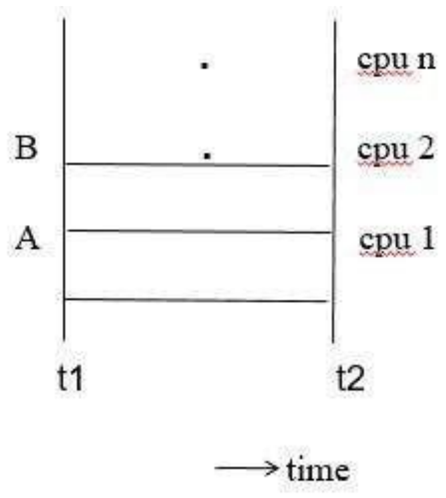
Transaction processing system

- The system with large database and hundreds of concurrent users that are executing database transaction.
- Eg :reservation system , banking system etc

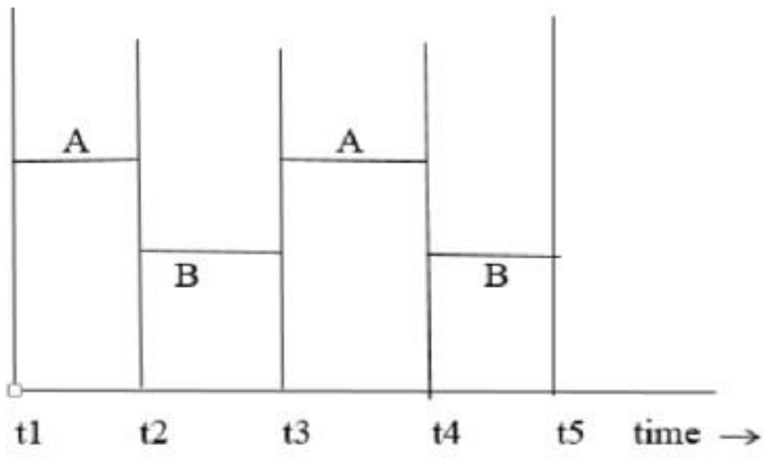
(i) Concurrent access

- Mutiple user accessing a system at the same time
- Single user-one user at a time can use a system
- Multi user-many user use the system at a time.
- It can be achieved by multiprogramming:

(ii) Parallel- multi-users access different resources at the same time.



(iii) Interleaved- Multiple users access a single resource based on time



Transaction access data using two operations

- **Read(x)**

  It transfer the data item x from the database to a local buffer belonging to the transaction that executed the read operation.

- **Write(x)**

  It transfer the data item x from the local buffer of the transaction to the database i.e. it write back to the database

ACID Properties

To ensure the integrity of data during a transaction, the database system maintains the following properties. These properties are widely known as ACID properties:

(i) 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.

(ii) 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.

(iii) 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.

(iv) 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.

Example : Example of Fund Transfer

transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Atomicity requirement

- if the transaction fails after step 3 and before step 6, money will be ―lost‖ leading to an inconsistent database state

- the system should ensure that updates of a partially executed transaction are not reflected in the database

Durability requirement

- once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist even if there are software or hardware failures. **Isolation requirement**

— if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database **T1 T2**

1. **read** (*A*)
2. *A* := *A* – 50
3. **write**(*A*)

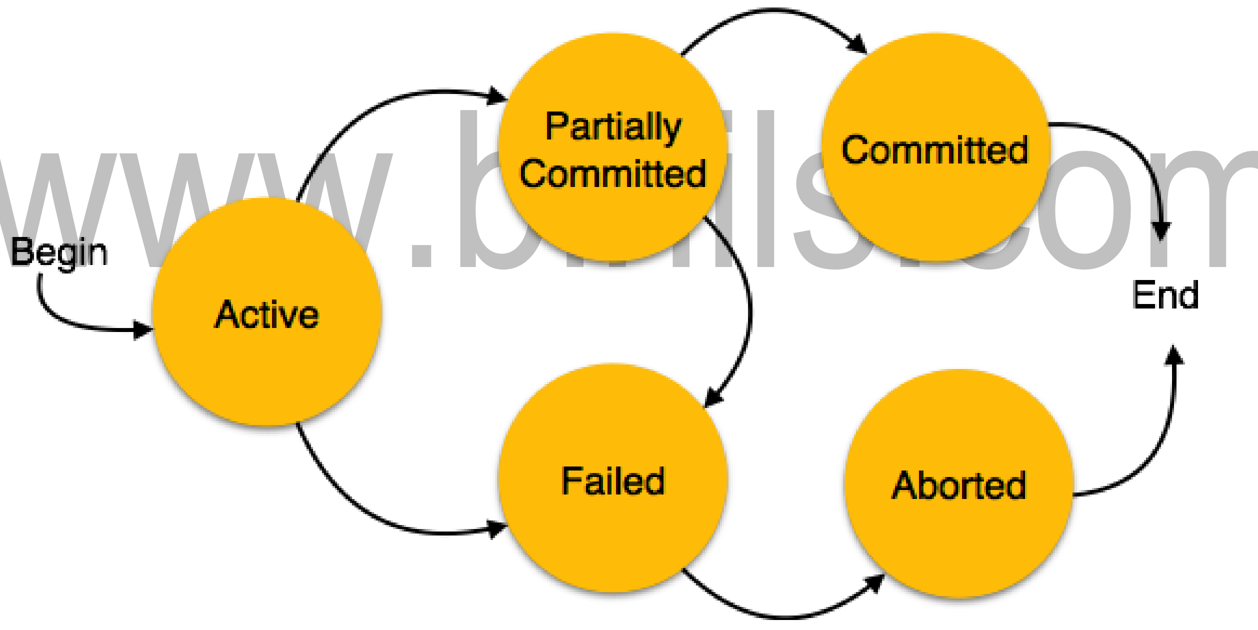
#read(A), read(B), print(A+B)

4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

Isolation can be ensured trivially by running transactions **serially**– that is, one after the other.

States of Transactions

A transaction in a database can be in one of the following states −



- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.

- **Aborted** – If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction.

  Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts –

  - Re-start the transaction
  - Kill the transaction
- **Committed** – If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

TRANSACTION RECOVERY

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures

Recovery using Log records (Log-Based Recovery

- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<Tn, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Example

Begin transaction

Update Acc 1001{balance:=Balance-

100}; If any error occurred then Goto

Undo; End if;

Update Acc

1002{balance:=balance+100}; If any

error occurred then Goto **undo**; End

if;

Commit;

Goto finish;

Undo: rollback;

Finish: return;

Requirement for recovery

- Implicit rollback
- Message handling
- Recovery log
- Statement atomicity
- No nested transaction

Transaction recovery

Database updates are kept in buffer in main memory and not physically written to disk until commit.

System recovery

- **Local failures** –affect only the transaction which the failure has actually occurred.
- **Global failures**- affect all the transaction in progress at the time of failure.
- **System failure** – do not physically damage the DB Eg: power shut down Media failure- cause damage to the DB. Eg: head crash ARIES

Recovery using Checkpoints:

Checkpoints:

- Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.

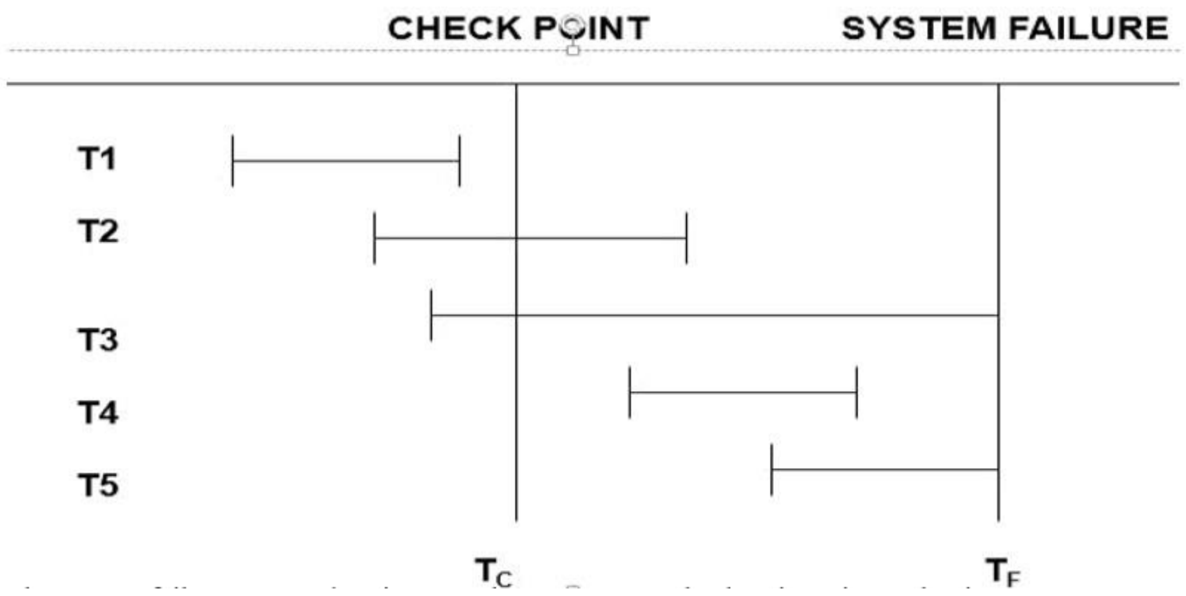
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery Algorithm

ARIES-Algorithm for Recovery and Isolation Exploiting

Semantics ARIES recovery involves three passes

- Analysis pass: Determines the REDO and UNDO lists.
- Redo pass: Repeats history, redoing all actions from REDO List
- Undo pass: Rolls back all incomplete transactions



The system failure occurred at time Tf , the most recent check point prior to the time Tf was taken at a time Tc

- Start with two list of transaction the UNDO and REDO list
- Search forward through the log starting from check point.
- If begin transaction log record is found for transaction(T) add T to UNDO list.
- If commit log record is found for transaction(T),add T to REDO list
- When the end of log record is reached the UNDO and REDO list is identified

UNDO : T3 T2

REDO : T5 T4