Module 6-Transaction Management

Transaction:

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:

➢ Failures of various kinds, such as hardware failures and system crashes

➢ Concurrent execution of multiple transactions

For example, transaction to transfer $50 from account A to account B:

1. read(A) → 200
2. A := A – 50 → 200 - 50
3. write(A) → 150
4. read(B) → 200
5. B := B + 50 → 200 + 50
6. write(B) → 250
7. COMMIT

Inconsistent

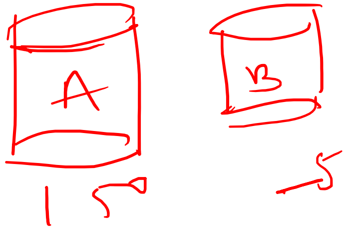
Prior

$A + B = 200 + 200 = 400$

Post

$A + B = 150 + 250 = 400$

Consistent



- ACID Property
- A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- Atomicity: Either all operations of the transaction are properly reflected in the database or none
- Consistency: Execution of a transaction in isolation preserves the consistency of the database
- Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions .That is, for every pair of transactions Ti and Tj , it appears to Ti that either Tj , finished execution before Ti started, or Tj started execution after Ti finished
- Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Transaction State Diagram

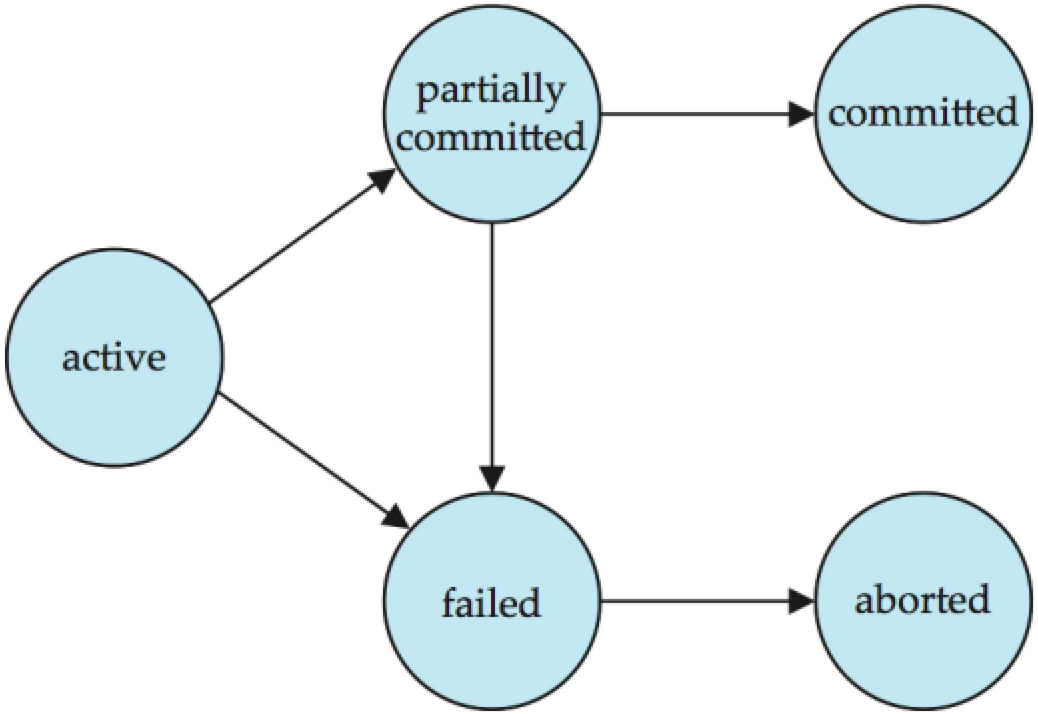
1.Active :The initial state; the transaction stays in this state while it is executing

2. Partially committed :After the final statement has been executed

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

4. Aborted :After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted: 1.Restart the transaction – can be done only if no internal logical error 2. Kill the transaction

5. Committed :After successful completion


partially committed
committed
active
failed
aborted

- Concurrent Executions
- Multiple transactions are allowed to run concurrently in the system. Advantages are:
- Increased processor and disk utilization, leading to better transaction throughput.For example, one transaction can be using the CPU while another is reading from or writing to the disk
- Reduced average response time for transactions: short transactions need not wait behind long ones
- Concurrency control schemes – mechanisms to achieve isolation That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- Schedule – 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
- A transaction that successfully completes its execution will have a commit instructions as the last statement
- By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B
- An example of a **serial** schedule in which T_1 is followed by T_2 :

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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write A	
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

- Serializability
- Basic Assumption – Each transaction preserves database consistency
- Thus, serial execution of a set of transactions preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
- 1. conflict serializability
- 2. view serializability

- Conflict Serializability
- 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 T_2 follows T_1, by a series of swaps of non-conflicting instructions.
  - Swap T1.read(B) and T2.write(A)
  - Swap T1.read(B) and T2.read(A)
  - Swap T1.write(B) and T2.write(A)
  - Swap T1.write(B) and T2.read(A)

These swaps do not conflict as they work with different items (A or B) in different transactions.

- Therefore, Schedule 3 is conflict serializable:

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

Schedule 3

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

Schedule 5

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

Schedule 6

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule <T_3, T_4>, or the serial schedule < T_4, T_3 >

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

→

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

- Are all serializable schedules conflict-serializable? No.
- Consider the following schedule for a set of three transactions.
  - $w_1(A)$, $w_2(A)$, $w_2(B)$, $w_1(B)$, $w_3(B)$

S

T_1	T_2	T_3
$w_1(A)$		
	$w_2(A)$	
	$w_2(B)$	
$w_1(B)$		
		$w_3(B)$

serial schedule

- We can perform no swaps to this:
  - The first two operations are both on *A* and at least one is a write;
  - The second and third operations are by the same transaction;
  - The third and fourth are both on *B* at least one is a write; and
  - So are the fourth and fifth.
  - So this schedule is not conflict-equivalent to anything – and certainly not any serial schedules.
- However, since nobody ever reads the values written by the $w_1(A)$, $w_2(B)$, and $w_1(B)$ operations, the schedule has the same outcome as the serial schedule:
  - $w_1(A)$, $w_1(B)$, $w_2(A)$, $w_2(B)$, $w_3(B)$

Serial schedule

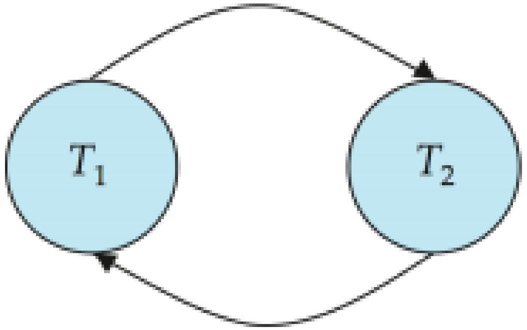
T_1	T_2	T_3
$w_1(A)$		
$w_1(B)$		
	$w_2(A)$	
	$w_2(B)$	
		$w_3(B)$

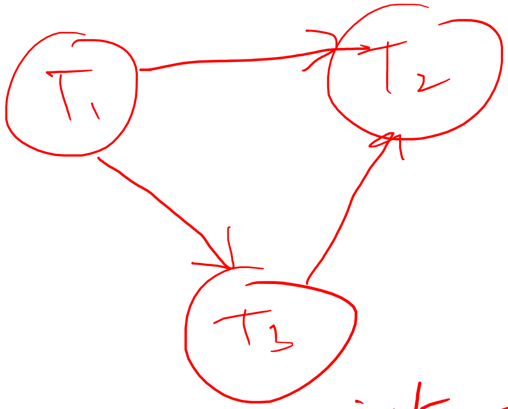
- Precedence Graph

  - Consider some schedule of a set of transactions T_1, T_2, ..., T_n
  - **Precedence graph**
    - A direct graph where the vertices are the transactions (names)
  - We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier
  - We may label the arc by the item that was accessed
  - **Example**

Eg

T_1	T_2	T_3
R(x)		
	R(y)	
		R(y)
	W(y)	
W(x)		
		W(x)
	R(x)	
	W(x)	

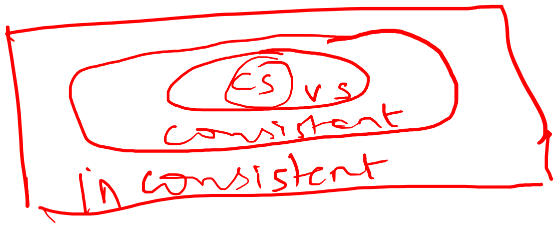




If cycle exist then it is not conflict serializable schedule.

- View Serializability
- 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’ ⍰ As can be seen, view equivalence is also based purely on reads and writes alone



Every conflict serializable schedule is view serializable &
if view serializable is conflict then there will be blind write

S	
T_1	T_2
R(a)	
W(a)	
	R(a)
	W(a)
R(b)	
W(b)	
	R(b)
	W(b)

S'	
T_1	T_2
R(a)	
W(a)	
R(b)	
W(b)	
	R(a)
	W(a)
	R(b)
	W(b)

View Serializability —

1.

S			S'
T_1	a	100	T_1
T_2	b	100	T_2

2.

S		S'
T_2	a	T_2
T_2	b	T_2

3.

S		S'
T_2	a	T_2
T_2	b	T_2

Conflict Serializable

Y — View seriable

No — Blind write

Y — Check for VS

N — Not VS

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Below is a schedule which is view-serializable but *not* conflict serializable

T_{27}	T_{28}	T_{29}
read (*Q*)		
	write (*Q*)	
write (*Q*)		
		write (*Q*)

2

- What serial schedule is above equivalent to?
  - T_{27}–T_{28}–T_{29}
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T_{29} performs the final write of Q in both schedules
- T_{28} and T_{29} perform write(Q) operations called **blind writes**, without having performed a read(Q) operation
- Every view serializable schedule that is not conflict serializable has **blind writes**

- Concurrency Control
- A database must provide a mechanism that will ensure that all possible schedules are both:
- Conflict serializable
- Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability after it has executed is a little too late!
- Tests for serializability help us understand why a concurrency control protocol is correct
- Goal – to develop concurrency control protocols that will assure serializability

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using

lock-X instruction

2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction

A transaction can unlock a data item Q by the unlock(Q) Instruction

Lock requests are made to the concurrency-control manager by the programmer

Transaction can proceed only after request is granted

- Example

- Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2.
  - Transaction *T*1 transfers $50 from account *B* to account *A*.
  - Transaction *T*2 displays the total amount of money in accounts *A* and *B*, that is, the sum *A* + *B*
  - Suppose that the values of accounts *A* and *B* are $100 and $200, respectively

*T*1:	T2:
lock-X(*B*);	lock-S(A);
read(*B*);	read(A);
B := *B* − 50;	unlock(A);
write(*B*);	lock-S(B);
unlock(*B*);	read(B);
lock-X(*A*);	unlock(B);
read(*A*);	display(A + B)
A := *A* + 50;	
write(*A*);	
unlock(*A*);	

- If these transactions are executed serially, either as *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300

- The Two-Phase Locking Protocol
- This protocol ensures conflict-serializable schedules
- Phase 1: Growing Phase:

➢ Transaction may obtain locks

➢ Transaction may not release locks

- Phase 2: Shrinking Phase:

➢ Transaction may release locks

➢ Transaction may not obtain locks

- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points
- That is, the point where a transaction acquired its final lock
- In Growing phase, a transaction will acquire locks but will not release any locks, while in Shrinking Phase a transaction will release locks but will not acquire any locks.

Given the following two transactions identify which is valid for the transaction T_1 and T_2

T1	T2
lock-X (A)	lock-S (A)
read (A)	read (A)
A:=A-0.2*A	unlock (A)
write (A)	lock-S (B)
lock-X (B)	read (B)
read (B)	unlock (B)
B:=B+0.2*A	display (A+B)
unlock (A)	
unlock (B)	

T1 follows Two phase locking protocol, T2 does not follow Two phase locking protocol.

Deadlocks

- Two-phase locking *does not* ensure freedom from deadlocks

T3:

```
lock-X(B);
read(B);
B := B − 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A)
```

T4:

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

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)	

- Observe that transactions *T3* and *T4* are two phase, but, in deadlock

- Deadlock Handling
- 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
- Deadlock prevention protocols ensure that the system will never enter into a deadlock state.
- Some prevention strategies :
- Require that each transaction locks all its data items before it begins execution(predeclaration)
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order

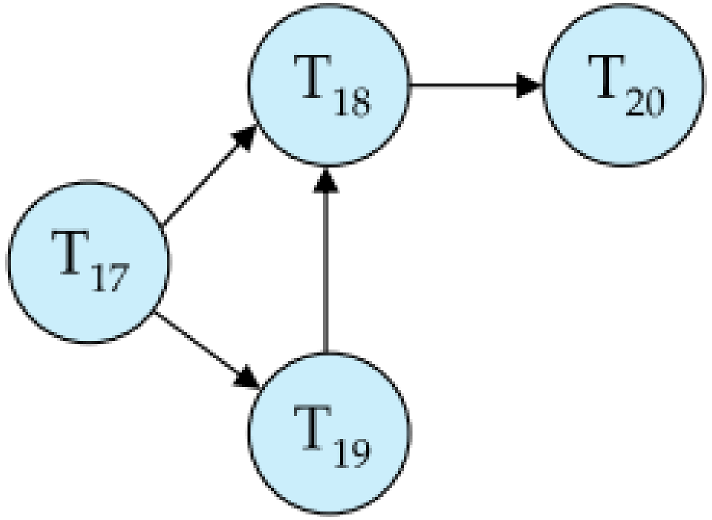
- Deadlock Prevention
- Following schemes use transaction timestamps for the sake of deadlock prevention alone
- wait-die scheme — non-preemptive
- Older transaction may wait for younger one to release data item. (older means smaller timestamp)
- Younger transactions never wait for older ones; they are rolled back instead
- A transaction may die several times before acquiring needed data item
- wound-wait scheme — preemptive
- Older transaction wounds (forces rollback) of younger transaction instead of waiting for it
- Younger transactions may wait for older ones
- May be fewer rollbacks than wait-die scheme

- Deadlock Prevention
- Both in wait-die and in wound-wait schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided
- Timeout-Based Schemes:
- a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval

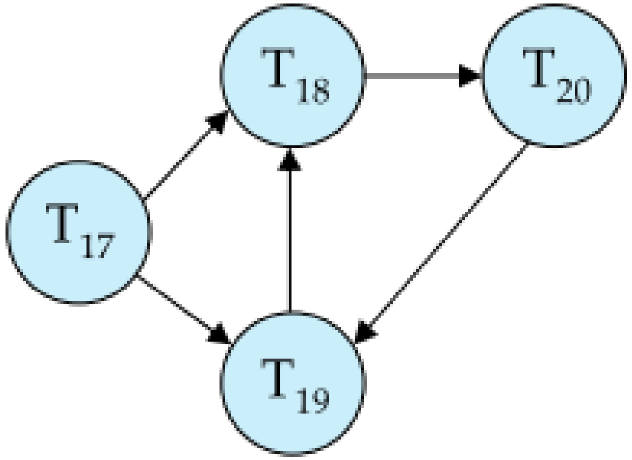
Deadlock Detection

- Deadlocks can be described as a wait-for graph, which consists of a pair G = (V,E),
- V is a set of vertices (all the transactions in the system) E is a set of edges; each element is an ordered pair Ti →Tj
- 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
- When Ti requests a data item currently being held by Tj, then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.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

Deadlock Detection: Example



Wait-for graph without a cycle



Wait-for graph with a cycle

Consider there are two schedules S_1 and S_2 as follows:

T1	T2
lock-X (A)	
read (A)	
write (A)	
unlock (A)	lock-S (B)
	read (B)
lock-S (B)	lock-X (A)
read (B)	read (A)
unlock (B)	unlock (A)
	unlock (B)

S1

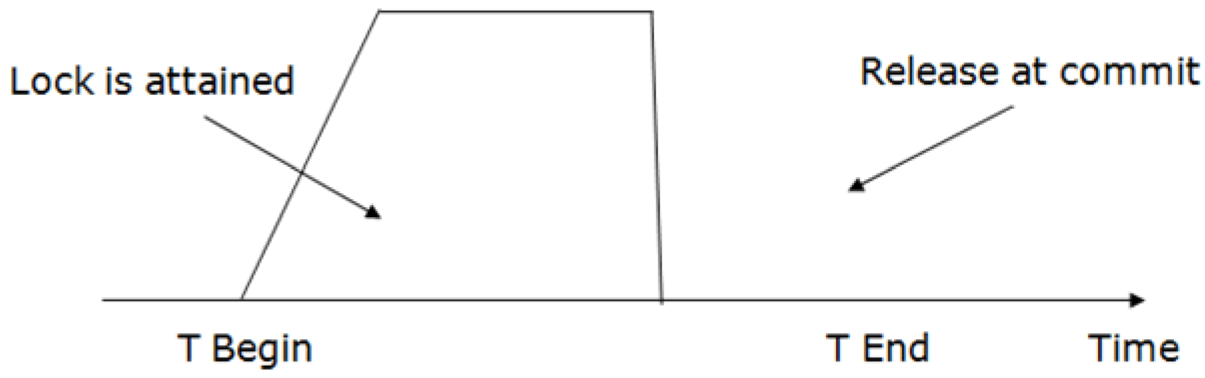
T1	T2
lock-X(A)	
read(A)	lock-X(B)
	read(B)
	write(B)
lock-S(B)	lock-S(A)
read(B)	read(A)
unlock(A)	unlock(A)
unlock(B)	unlock(B)

S2

S2 will suffer deadlock, S1 will not suffer deadlock. In S1, T2 has acquired shared lock on (B), and T1 wants to acquire shared mode lock on (B). More than one transactions are allowed to acquire shared mode lock on the same database, so shared mode lock on (B) is granted to T1 and no deadlock occurs in S1. But in S2, T2 is holding exclusive mode lock on (B) and T1 has requested shared mode lock on (B). While one transaction is holding exclusive mode lock on a particular database, no other transaction can acquire any lock on that database unless the lock is released by the former transaction (which is holding exclusive lock on the database). Unless transaction T1 gets shared mode lock on (B), it will not proceed with the next operations and will not release the exclusive mode lock on (A), which restricts transaction T2 to acquire lock on (A). Both T1 and T2 are waiting for each other to release resources. Hence S2 is going to suffer a deadlock.

Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



- Timestamp-Based Protocols
- Each transaction is issued a timestamp when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that TS(Ti) <TS(Tj).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
- W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully
- R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully

Timestamp-Based Protocols

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order
- Suppose a transaction Ti issues a read(Q)

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

2. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to max(R-timestamp(Q), TS(Ti)).

- Suppose that transaction Ti issues write(Q).
- 1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously,and the system assumed that that value would never be produced.Hence, the write operation is rejected, and Ti is rolled back
- 2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q.Hence, this write operation is rejected, and Ti is rolled back
- 3. Otherwise, the write operation is executed, and W-timestamp(Q) is set to TS(Ti)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits
- But the schedule may not be cascade-free, and may not even be recoverable

Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

- Transaction failure
- System crash
- Disk failure

1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

Logical errors: If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

Syntax error: It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. For example, The system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

System failure can occur due to power failure or other hardware or software failure. Example: Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.

Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

- 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.
- Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.

  <Tn, Start>

- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

  <Tn, City, 'Noida', 'Bangalore' >

- When the transaction is finished, then it writes another log to indicate the end of the transaction.

  <Tn, Commit>

Before Ti execute write(X) a log record <Ti,X,V1,V2>

O N

V1:Old value

V2:New Value

start

commit

A - start
C -
I -
D -
commit

- There are two approaches to modify the database:

1. Deferred database modification:

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Start
A : A - 50
write(A)
B : B + 50
commit

- Recovery using Log records
- 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<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone

start or commit
commit

Redo

Tn start

undo

- Undo(Ti)-
- Restores the value of all data item updated by Ti to their old value going backward from last log record for Ti
- Each time a data item is restored to its old value a special log

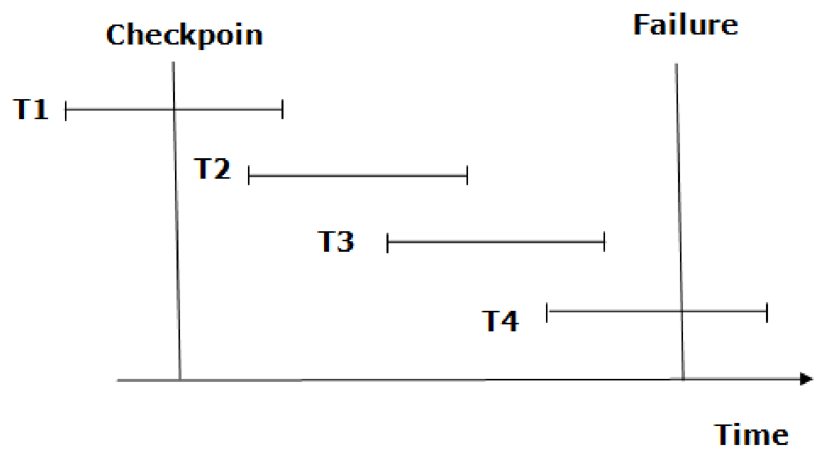
<Ti,X,V> is written out.

- When undo of a transaction is complete a log record <Ti abort> is written out.

- Redo(Ti)-It sets the value of all data items updated by Ti to new value
- No logging is done in this case.

- Checkpoint
- 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 using Checkpoint**
- In the following manner, a recovery system recovers the database from this failure:



•The recovery system reads log files from the end to start. It reads log files from T4 to T1.

•Recovery system maintains two lists, a redo-list, and an undo-list.

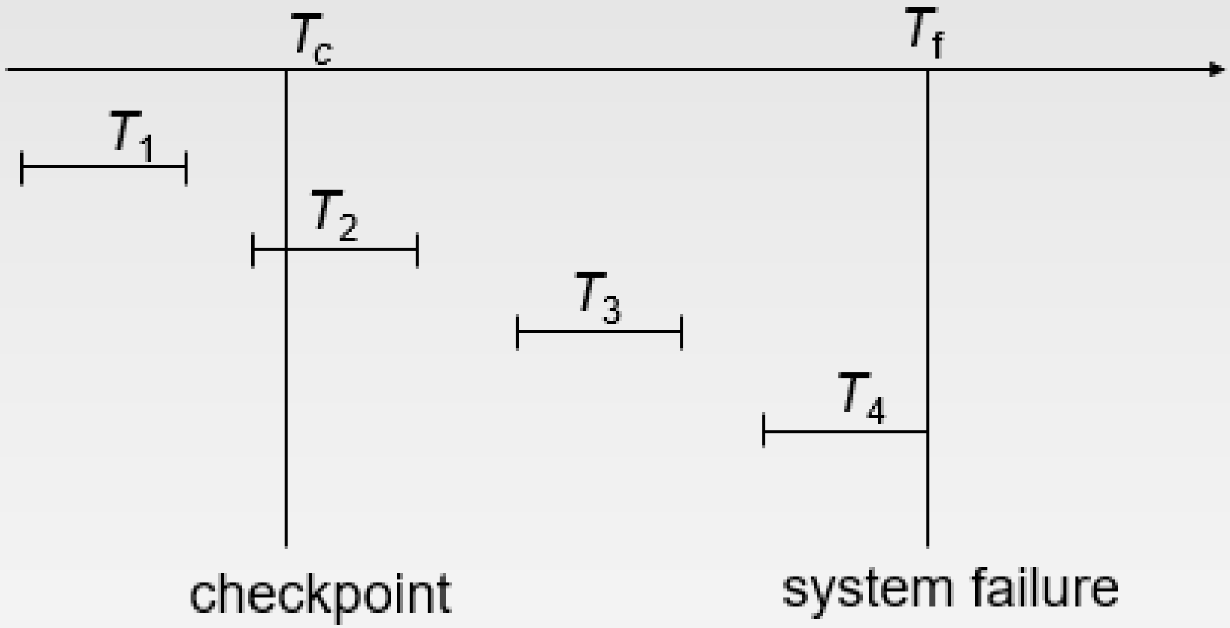
•The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
•**For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.

Start + commit Before CP— Permanently store DB

•The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
•**For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

start + commit after CP to Redo

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone