UNIT-IV

Transaction:

- A transaction is a collection of operations that form a single logical unit of work.
- A transaction is an execution of an user program and seen by DBMS as a series of actions.
- A transaction is written between begin transaction & end transaction.

Transaction operations < read(X), write(X)

- Read(X): It performs reading of data items from the db, to logical buffer (or) program workarea.
- Write(x): Performs writing of data items to the db from local buffer.

Ex: Write a transaction to transfer an amt of Rs. 50/- from A to B.

A $\xrightarrow{50}$ B

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

(To read db obj it is first brought into disk from main memory & copied in prog work area in db. This is done by read operation)

(To write a db obj to memory, copy of the obj is 1st modified and then written to disk. This is done by write operation)

Properties of Transaction:

There are 4 properties of transactions to maintain concurrent access of data and recovery from system failures in DBMS. These properties are called ACID properties.

A - Atomicity
C - Consistency
I - Isolation
D - Durability

Atomicity: Either all operations of the transactions are properly selected in db or none are.

Transactions are incomplete due to 3 reasons:

(i) If a transaction is aborted or terminated for internal reasons (ex: If it is automatically restarted, then it gives incomplete transaction)

(ii) System may crash due to system failure

(iii) Read a value which is not in disk.

Transaction Management Component takes care of Atomicity.

Consistency:

This property states that after the transaction is finished, its db must remain in a consistent state. There must not be any possibility that some data is incorrectly affected by the execution of transaction.

If the db was in a consistent state before the execution of transaction, it must remain in consistent state after the execution of the transaction.

No separate module takes care of consistency. Only sys progs takes care of consistency.

(If Atomicity, Isolation, Durability holds good consistency also holds good).

Isolation:

In a db sys 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 & executed as if it is the only transaction in the sys.

Ex: User A withdraw $100 and user B withdraw $250 from user Z acct, one of the users is required to wait until the other user transaction is completed, avoiding in consistent data. If B is required to wait, then B must wait until A's transaction is completed, & Z's acct balance changes to $900. Now B can withdraw $250 from this $900 balance.

(Even if multiple transactions are executing together none has to be affected by other, then it is logical Isolation)

Concurrency control Module will take care of logical Isolation.

Durability:

This property states that in any case all updates made on the db will persist even if the sys fails and restarts.

If a transaction writes or updates some data in db & commits, that data will always be there in db.

If the transaction commits but data is not written on the disk and the sys fails, that data will be updated once the sys comes up.

States of Transactions:

A transaction in a db can be in one of the following states:

- Active state: It is the initial state of transaction. The transaction will be in this state while it starts execution.
- Partially committed state: This state occurs after the final state of the transaction has been executed. After execution of all operations, the db sys performs some checks. eg: the consistency state of db after applying op of trans on to db.
- Failed state: If a normal execution can no longer proceed it is said to be failed state.
- Aborted: If any checks fails & transaction reached in failed state, the recovery manager rollsback all its operation on the db to make db in the state where it was prior to start of execution of transaction. Transaction in this state are called aborted.

Committed state:

If the transaction executes all its operations successfully it is said to be committed. All its effects are now permanently made on db system.

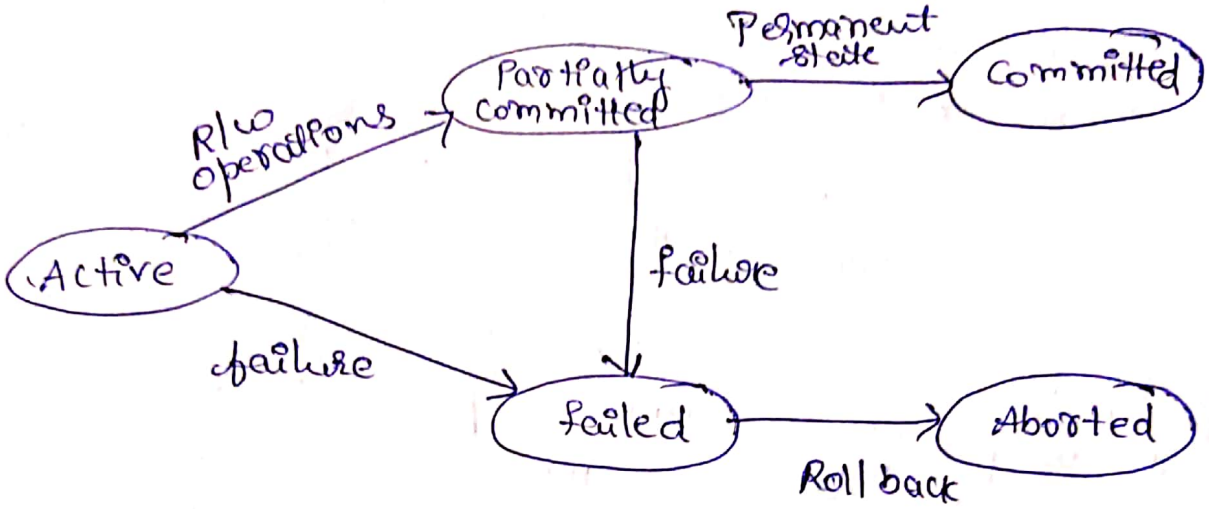


fig: transaction state diagram.

A transaction enters the failed state after the sys determines that the transaction can no longer proceed with its normal execution. Such a transaction must be rolled back. Then it enters the aborted state.
At this point the sys has two options:

(a) Restart the transaction:

If the transaction was aborted due to some H/w failure (or) S/w errors we can restart the transaction. A restarted transaction is considered as new transaction.

(b) Kill transaction:

If some logical errors occurs & if they can be corrected only by rewriting the transaction then it is called kill the transaction.

Implementation of Atomicity & Durability:

The recovery-mgmt component of a db sys implements the support for atomicity & durability.

- The shadow copy scheme which is based on making copies of the db called shadow copies, assumes that only one transaction is active at a time.
- A pointer called db-pointer is maintained on disk, it points to the current copy of the db.
- All updates are made on a shadow copy of the db and db-pointer is made to point to the updated shadow copy only after the transaction reaches commit and all updated pages have been flushed to disk.
- In case transaction fails, old consistent copy pointed to by db-pointer can be used, & the shadow can be deleted.

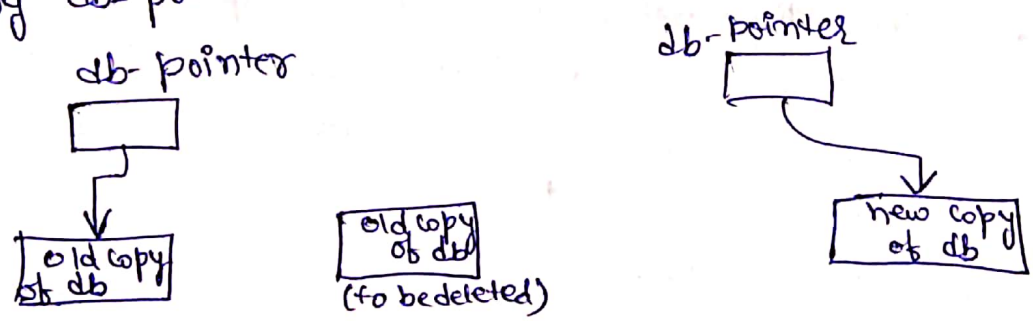


fig (a): Before update fig (b): After update.

Thus the atomicity & durability properties of transactions are ensured by the shadow copy implementation of the Recovery-mgmt component.

Ex: text editors.

But this implementation is extremely inefficient for large dbs. Since executing a single transaction requires copying the entire db.

Storage Structure:

Storage media can be distinguished by their relative speed, capacity & resilience to failure, & classified as volatile storage or nonvolatile storage. or stable storage.

- Volatile storage:
  Information in volatile storage doesnot survive when system crashes.
  Ex: Main memory & Cache memory.
  - Access to volatile storage is extremely fast.

- Non Volatile storage:
  Information in non volatile storage survives when system crashes.
  Ex: Secondary storage devices such as magnetic disk, online storage, magnetic tapes, tertiary storage etc.
  Non volatile storage is slower than volatile storage.

- Stable storage:
  Information residing in stable storage is never lost.
  To implement stable storage, we replicat the information in several non volatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage doesnot cause a loss of information.

(Note:)
For a transaction to be durable, its changes need to be written to stable storage.

Transaction Isolation:

(Concurrent executions)

Multiple transactions which are running at same time is called concurrent execution of transactions.

Allowing multiple transactions cause several good things and bad things.

There are two good reasons for allowing concurrency:

→ Improved throughput & Resource Utilization:

- A transaction involve I/O activity, CPU activity etc.
- The I/O activity can be done in parallel with processing at the CPU.
- The parallelism of the CPU & the I/O sys can therefore be exploited to run multiple transactions in parallel.
- While a Read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases throughput of the sys. (i.e num of transactions executed in a given amt of time).
- The processor & disk utilization also increase. (i.e the processor & disk spend less time idle).

→ Reduced waiting time:

There may be some transactions running on a sys, some short & some long.

- If trans are running serially and a short trans has to wait till long trans complete, this will lead to delays in running transactions.
- If the trans are running concurrently, they share CPU cycles & disk access among them.
- Concurrent execution reduces the avg response time.

→ Schedule:

A schedule is a set of transactions to be executed. They represent the chronological order in which instructions are executed in the system.

There are two types of schedules:

(i) Complete schedule:

A schedule that contains either a abort (or) commit statement is called complete schedule.

(ii) Serial Schedule:

If transactions are executed from start to finish one by one without any interchange (or) interleave then we call the schedule as a serial schedule.

Ex: Banking System.

Let T_1 & T_2 be two transactions that transfers funds from one acct to another.

- T_1 transfers $50 from Acct A to Acct B.

It is defined as

```
T1 : Read (A)
     A := A - 50
     Write (A)
     Read (B)
     B := B + 50
     Write (B)
```

- T_2 transfers 10% of the balance from Acct A to Acct B.

```
T2 : Read (A)
     Temp := A * 0.1
     A := A - temp
     Write (A)
     Read (A)
     B := B + temp
     Write (B)
```

Suppose current values are Acct A = $1000, Acct B = $2000 and two transactions are executed in order T_1 followed by T_2.

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)

fig: Schedule 1 - T_1 followed by T_2.

The final values of Acct A and Acct B after execution are $855 and $2145.
Thus total amt A+B is preserved after the execution of both transactions.

Now T_2 followed by T_1

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)	

fig: Schedule-2 T_2 followed by T_2 (Serial Schedule)

These schedules are serial: Each serial schedule consists of a sequence of instructions from various transactions.

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

If two transactions are running concurrently, the operating sys may execute one transaction for a little while, then perform a context switch, execute the 2nd trans for some time, & then switch back to 1st trans for some time and so on. (CPU time is shared among all transactions).

In general, it is not possible to predict how many instructions of a trans will be executed before the CPU switches to another transaction.

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)

fig: Schedule 3 – Concurrent schedule equivalent to serial schedule – 1

- After this execution takes place, we arrive at some state as schedule 1. The sum(A+B) is preserved.

- Not all concurrent executions results in a correct state.

	T_1	T_2	
1000	Read(A)		
950	A := A-50		
		Read(A)	1000
		temp := A*0.1	100
		A := A-temp	900
		write(A)	900
		Read(B)	2000
950	write(A)		
2000	Read(B)		
2050	B := B+50		
2050	write(B)		
		B := B+temp	2100
		write(B)	2100

Fig: Schedule-4 a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final vals are Acct A & B are $950 & $2100.

This final state is <u>inconsistent</u> state. as sum(A+B) is not preserved by execution.

Note: We can ensure consistency of the db under concurrent execution by making sure that any schedule that executed has same effect as a schedule that could have occurred without any concurrent execution.

Serializability: is a concurrency scheme where the concurrent transaction is equivalent to one that executes the transactions serially.

The db sys must control concurrent execution of transactions, to ensure that the db state remains consistent.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs & how operations of various transactions interact. For this reason we will consider only two operations:

Read and write.

The different forms of schedule equivalence

- Conflict Serializability
- View Serializability

Conflict Serializability

- Let us consider a schedule S in which there are consecutive instructions I_i, I_j of Transactions T_i and T_j respectively ($i \neq j$).
- If I_i and I_j refer to different data items, then we can swap, I_j & I_j without affecting the results of any instruction in the schedule.
- If I_i and I_j refer to same data, then the order of two steps may matter.
- These are 4 cases that we need to consider (or) Anamolies due to Interleaved fashions)

(i) (Read, Read) (I_i = Read (Q), I_j = Read (Q))

order of I_i and I_j doesnot matter, since same value is read by T_i & T_j

(ii) (Read, Write) (I_i = Read (Q), I_j = Write (Q))

order of I_i and I_j matters.

- If I_i comes before I_j then T_i doesnot read the value of Q that is written by T_j in inst I_j
- If I_j comes before I_i then T_i reads the value of Q that is written by T_j.

(iii) (Write, Read) (I_i = Write (Q), I_j = Read (Q))

The order of I_i & I_j matters

(iv) (Write, Write) (I_i = Write (Q) I_j = Write (Q))

The order of instructions doesnot effect either T_i or T_j.

(Note:- I_i and I_j conflict if they are operations by different transactions on the same data item, and atleast one of these instructions is a write operation.

Now we will see examples:

→ (Write Read) (W-R conflict): Reading uncommit data.

WR conflicts is that a transaction T_2 could read a database object A that has been modified by another transaction T_1 which has not yet committed. Such a read is called a dirty read

Ex: ① T_1 transfers Rs. 100 from A to B.
& T_2 increments both A & B by 6%.

suppose actions are interleaved like
(i) T_1 deducts Rs. 100/- from Acct A
(ii) T_2 reads current vals of accts A & B & adds 6% Interest to each.
(iii) T_1 adds Rs. 100 to acct B.

T_1	T_2
R(A)	
A := A - 100	
W(A)	
	R(A)
	A := A + 0.06
	W(A)
	R(B)
	B := B + 0.06
	W(B)
	Commit
R(B)	
B := B + 100	
W(B)	
Commit	

Preserves db consistency.

If for ex value of A written by T_1 read by T_2 before T_1 has completed all changes then db may be in in consistent.

Ex: 2

A=5

T_1	T_2
R(A)	
W(A)	
:	R(A)
:	commit
(failure) Abort	
:	
A=5	A=6

dirty read problem exists.

Ex: 3

T_1	T_2
R(A)	
	R(A)
	W(A)
R(A)	
	commit
W(A)	
Commit	

dirty read problem exists

→ Read Write Conflicts (RW):- Unrepeatable Reads

T_2 could change the value of an obj A that has been read by a transaction T_1 while T_1 is still in progress.

Ex:① T_1 & T_2 reads some value of A say 5.
T_1 has increased A as 6 but before incrementing T_2 occurs & decremented A to 4 which is to be 5. which is incorrect.

T_1	T_2
R(A)	
A=A+1	
	R(A)
	A=A-1

This situation can never arise in serial execution of T_1 & T_2.

Ex: ②

	X=10 T_1	T_2
10	R(X)	
		R(X) 10
15	W(X)	
		R(X) 15

→ **WW (Write Write Conflicts):-** overwriting uncommitted data (Lost update problem)

T_2 could overwrite the value of an obj A, which has already been modified by a Trans T, while T_1 is still in progress.

Ex:① Suppose A & B are two employees, & their salaries must be kept equal.

T_1	T_2
A := 2000	
B := 2000	
	A := 1000
	B := 1000

T_1 followed T_2
∴ both sals are 1000
T_2 followed by T_1
∴ sals are 2000

} such a write is called a blind write

Now consider interleaving actions of T_1 & T_2.

T_1	T_2
	A := 1000
B := 2000	
	B := 1000
	commit
A := 2000	
commit	

The result is not identical A = 2000, B = 1000 & inconsistent state.

Ex: ②

T_1	T_2	T_3
	$R_2(X)$	
$R_1(X)$		
		$W_3(X)$
	$W_2(X)$	

$W_3(X)$ is a blind write as there is no read before write.

Swapping:

Let I_i and I_j be consecutive instructions of a schedule S. If I_i and I_j refer to diff data items, then we can swap I_i and I_j to produce new schedule S'. We expect S to be equivalent to S'.

ex:

T_1	T_2
1 R(A)	
2 W(A)	
	3 R(A)
	4 W(A)
5 R(B)	
6 W(B)	
	7 R(B)
	8 W(B)

fig: schedule (S_5) before swapping.

S_5 (1, 2, 3, 4, 5, 6, 7, 8)
swap

T_1	T_2
1 R(A)	
2 W(A)	
	3 R(A)
5 R(B)	
	4 W(A)
6 W(B)	
	7 R(B)
	8 W(B)

fig: schedule S' after swapping.

swap (1, 2, 3, 5, 4, 6, 7, 8)
swap

T_1	T_2
1 R(A)	
2 W(A)	
5 R(B)	
	3R(A)
	4W(A)
6 W(B)	
	7R(B)
	8 W(B)

S_5(1, 2, 5, 3, 4, 6, 7, 8)
swap.

T_1	T_2
1 R(A)	
2 W(A)	
5 R(B)	
	3R(A)
6 W(B)	
	4W(A)
	7R(B)
	8 W(B)

S_5(1, 2, 5, 3, 6, 4, 7, 8)
swap.

T_1	T_2
1 R(A)	
2 W(A)	
5 R(B)	
6 W(B)	
	R(A) 3
	W(A) 4
	R(B) 7
	W(B) 8

$\therefore S_5 = S_1$

We cannot swap if I_i & I_j have same data item.

I_i : R(A)
I_j : W(A)

conflicting instructions.

(i) belong to diff transaction
(ii) belong to same data

ex:

I_i (T_1)	I_j (T_2)	A = 10
	R(A)	
15 W(A)		
15 W(A)		
	R(A) 15	

(iii) at least one of them is write operation.

I_i (T_1)	I_j (T_2)
	W(A) 10
15 W(A)	
15 W(A)	
	W(A) -10.

all 3 conditions happens then there is a conflicting existing.

Testing for Serializability:

In order to know that a particular transaction can be serialized, we can draw a precedence graph.

Precedence graph:

This is a graph of nodes & vertices, where the nodes are the transaction names & the vertices are attribute collisions.

The schedule is said to be serialized if and only if there are no cycles in the resulting diagram.

- how to draw graph:

(i) Draw a node for each transaction in the schedule.

(ii) Where Transaction T_1 writes to an attribute which transaction T_2 has read from, draw a line pointing from T_2 to T_1 $\quad w(A) \rightarrow R_2(A)$

(iii) Where transaction T_1 writes to an attribute which transaction T_2 has written to, draw a line pointing from T_2 to T_1. $\quad W_1(A) - W_2(A)$

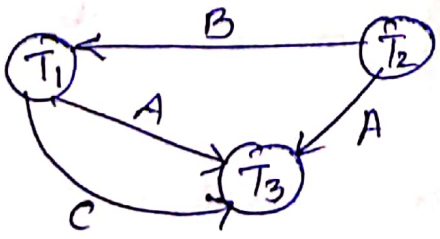
(iv) Where transaction T_1 reads from an attribute which transaction T_2 has written to, draw a line pointing from T_2 to T_1. $\quad R_1(A) \rightarrow W_2(A)$

If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.

ex: ①

T_1	T_2	T_3
R(A)		
R(B)		
	R(A)	
	R(B)	
		W(A)
R(C)		
W(B)		
		W(C)

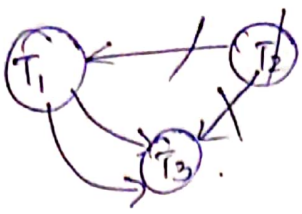
$R_1(A) \rightarrow W_2(A)$
$W_1(A) \rightarrow R_2(A)$
$W_1(A) \rightarrow W_2(A)$



No cycle so it is conflict serializable schedule. Now find serializability order by topological sorting.

- Find indegree of each node.
  $T_1 - 1$, $T_2 - 0$, $T_3 - 3$

So schedule T_2

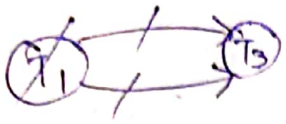


consider only T_1 & T_3.

Indegree $T_1 = 0$

$T_3 = 2$.

So schedule is $T_2 \rightarrow T_1$



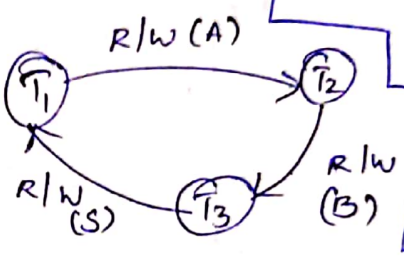
Indegree of T_3 is 0.

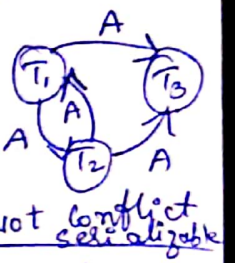
∴ final schedule is

$T_2 \rightarrow T_1 \rightarrow T_3$.

Ex: (4)

T_1	T_2	T_3
R(A)		
	R(B)	
	W(A)	
W(A)		
		W(A)





ex: (2)

T_1	T_2	T_3
R(A)		
	W(A)	
	R(B)	
		W(B)
		R(S)
W(S)		

As there is a cycle this is not conflict serializable.

Ex: (3)

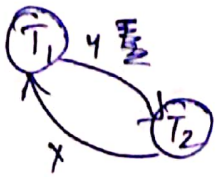
S_1

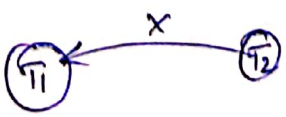
T_1	T_2
R(X)	
R(Y)	
	R(X)
	R(Y)
	W(Y)
W(X)	

S_2

T_1	T_2
R(X)	
	R(X)
	W(Y)
W(Y)	
R(Y)	
W(X)	

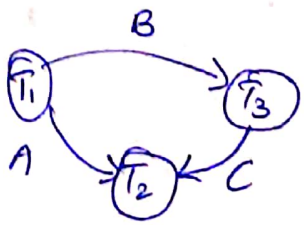
S1





Ex: (5)

T_1	T_2	T_3
R(A)		
R(B)		
	R(A)	
	R(C)	
W(B)		
		R(B)
		R(C)
		W(B)
	W(A)	
	W(C)	



$T_1 = 0$ $T_2 = 2$

$T_3 = 1$

Order is T_1 T_3 T_2

Conflict equivalent:

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

ex: Schedule 1 is conflict equivalent to schedule 3.

T_1	T_2
R(A)	
A = A - 50	
W(A)	
R(B)	
B = B + 50	
W(B)	
	R(A)
	t = A * 0.1
	A = A - t
	W(A)
	R(B)
	B = B + t
	W(B)

Schedule (1)

T_1	T_2
R(A)	
A = A - 50	
W(A)	
	R(A)
	t = A * 0.1
	A = A - t
	W(A)
R(B)	
B = B + 50	
W(B)	
	R(B)
	B = B + t
	W(B)

Schedule (3)

Conflict Serializable:

A schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

The above example schedule (3) is conflict serializable, since it is conflict equivalent to the serial schedule (1).

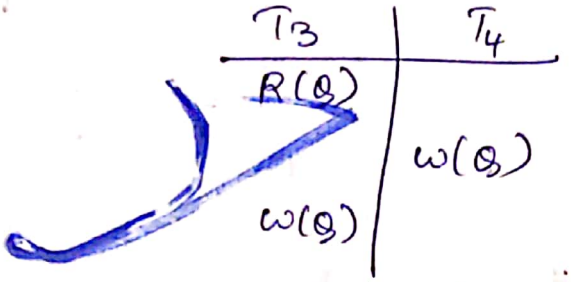
Ex: (2)

S_1		S_2		S_3	
T_1	T_2	T_1	T_2	T_1	T_2
R(A)		R(A)		R(A)	
	W(A)	W(B)		R(B)	
R(B)			R(A)		W(A)
	W(B)		R(B)		W(B)
R-W conflict.		W-R		R-W	

∴ S_1 & S_2 are not conflict equivalent.

∴ S_1 & S_3 are conflict equivalent

Ex: Consider schedule 7 of below fig:



it consists of only the significant operations (i.e read & write) of transactions T_3 & T_4.
This schedule is not conflict serializable, since it is not equivalent to either the serial schedule (T_3, T_4) or the serial schedule (T_4, T_3).

→ View Serializability:

→ View Equivalent:

The schedules S & S' are said to be view equivalent if ~~that~~ three conditions are met:

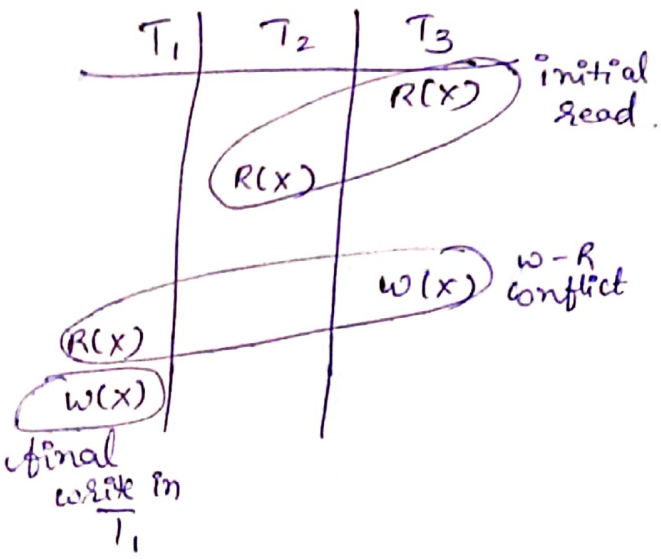
(i) For each data item Q
if T_i reads initial val of Q in schedule S then T_i in schedule S' also read initial value of Q.

(ii) (W-R conflict)
if T_i executes Read(Q) in schedule S, & if that val was produced by a write(Q) operation executed by Transaction T_j then
Read(Q) oper in T_i must in S' also read the val of Q that was produced by the same write(Q) operation of Trans T_j.

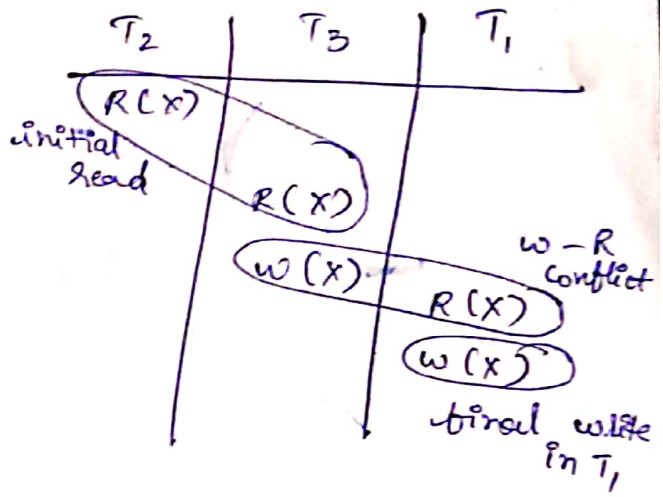
(iii) The trans that performs the final write(Q) oper in S must perform the final write(Q) in S'.

A schedule S is view serializable if it is view equivalent to a serial schedule.

ex: S_1 (Concurrent schedule)



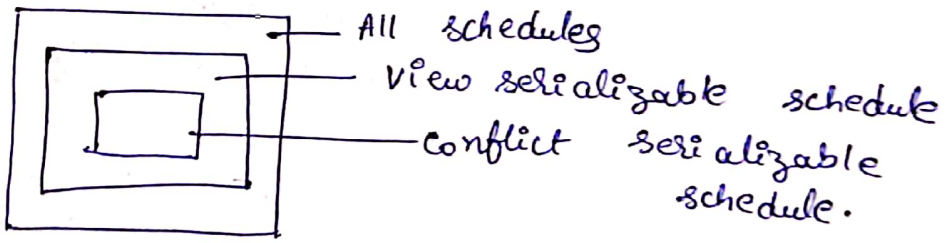
S_2 (serial schedule):



(i) initial Read – ✓ (satisfied)
(ii) W-R conflict – ✓
(iii) Final write – ✓

all 3 conditions are satisfied so it is view serializable.

Note: Any conflict serializable schedule is view serializable. But any view serializable schedule doesnot necessarily be conflict serializable schedule.



Ex: Concurrent

S_1		S_2		Serial schedule S_3	
T_1	T_2	T_1	T_2	T_1	T_2
R(A)		R(A)		R(A)	
A=A-10		A=A-10		⋮	
	R(A)	W(A)			
	T=2×A	R(B)			
	W(A)	B=B+10		W(A)	
	R(B)	W(B)		⋮	
W(A)			R(A)		
R(B)					R(A)
B=B+10					⋮
W(B)					

(i) initial Read – X not satisfied in S_2 & S_3.
so not view serializable schedule.

Ex: View Serializable.

Non-Serial

S1

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

Serial.

S2

T_1	T_2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

Let us check 3 conditions of View Serializability:

→ Initial Read: In schedule S1, T_1 first first Reads(X)
In S_2 also T_1 first Reads(X)
Lets check Y. In S1, T_1 first reads(Y), In S_2 also T_1 first reads Y.
Both data items X & Y initial read is satisfied in S1 & S2

→ Update Read (WR): In S1 T_2 reads the value of X written by T_1. In S2 T_2 reads the X after it is written by T_1.
In S_1 T_2 reads the value of Y written by T_1
In S_2 T_2 reads the value of Y " " ",
∴ This condition is also satisfied.

→ Final write : In S_1, the final write (X) is done by T_2.
In S_2 also T_2 performs final write on X.

In S_1 final write operation on Y is done by transaction T_2.
In S_2 final write on Y is done by T_2.

∴ final write is satisfied.

Since all 3 conditions are satisfied it is (S_1 & S_2) view equivalent schedule.
As it is view equivalent it is view Serializable schedule.

Transaction Isolation & Atomicity:

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.

In a sys that allows concurrent execution, the atomicity property requires that any transaction T_j that is dependent on T_i is also aborted. To achieve this, we need to place restrictions on the type of schedules permitted in the sys.

→ Recoverable Schedule:

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

ex: (1)

T_8	T_9
R(A)	
W(A)	
	R(A)
	Commit
R(B)	

T_9 is dependent on T_8 Non-recoverable schedule. If T_9 commits before T_8.

Suppose that the system allows T_9 to commit immediately after execution of read(A) instruction. Thus T_9 commit before T_8 does.

Now suppose that T_8 fails before it commits. Since T_9 has read the value of data item A written by T_8 we must abort T_9 to ensure transaction Atomicity.

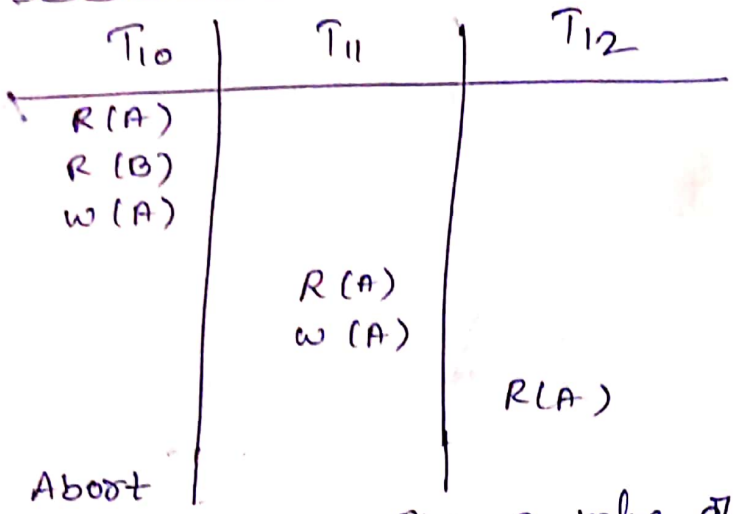
However, T_9 has already committed & cannot be aborted. Thus we have a situation where it is impossible to recover correctly from the failure of T_8.

Ex: (2)

T_1	T_2
1000 R(A)	
900 A = A - 100	
900 W(A)	
	R(A) 900
	A = A + 500 1400
	W(A) 1400
failure point	~~commit~~
commit;	
	commit;

T_1 reads & writes A & that value is read & written by T_2. But later on T_1 fails, so we have to rollback T_1. Since T_2 read the value written by T_1, it should also be rollbacked. As it is not commited we can rollback T_2 as well. So it is recoverable schedule with cascading schedule.

→ Cascadeless schedule:



- Transaction T_{10} writes a value of A that is read by Transaction T_{11}.
- T_{11} writes a value of A that is read by T_{12}.
- Suppose 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, T_{12} is dependent on T_{11}, T_{12} must be rolled back.
- This phenomenon, in which a single transaction failure leads to a series of transaction rollback is called Cascading rollback.
- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur, such schedules are called cascadeless schedules.
- Formally, A cascadeless schedule is one where for each pair of transaction T_i & T_j such that T_j reads data item, previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

(Note:) Every cascadeless schedule is also recoverable schedule.

Cascadeless schedule -

T_{10}	T_{11}
R(A)	
W(A)	
	R(B)
commit	
	R(A)

Transaction Isolation levels:

The isolation levels specified by the SQL standard are as follows:

- Serializable usually ensures serializable execution.
- Repeatable read allows only committed data to be read & further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.
- Read committed allows only committed data to be read, but doesnot require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- Read uncommitted allow uncommitted data to be read. It is the lowest Isolation level allowed by SQL.

All the isolation levels above additionally disallow dirty writes, i-e they disallow writes to a data item that has already been written by another transaction that has not yet committed or Aborted.

Implementation of Isolation levels:

(i) Locking (ii) Timestamps.

Lock Based Protocols:

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes:
  - Shared (S) mode – Data item can only be read. S-lock is requested using lock-S instruction.
  - Exclusive (X) mode – Data item can be both read as well as written. X-lock is requested using lock-X instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

T_j \ T_i	S	X
S	true	false
X	false	false

fig: Lock compatibility matrix.

Shared mode is compatible with shared mode but not with exclusive mode.
At any time several shared-mode locks can be held simultaneously on a particular data item.
A subsequent exclusive-mode lock req has to wait until the currently held shared-mode locks are released.

To access a data item, Transaction must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus T_i is made to wait until all incompatible locks held by other transactions have been released.

Eg:

T_1 (A = 100, B = 200)

lock-X(B)
R(B)
B = B - 50
W(B)
unlock(B)
lock-X(A)
R(A)
A = A + 50
W(A)
unlock(A)

T_2

lock-S(A)
R(A)
unlock(A)
lock-S(B)
R(B)
unlock(B)
display(A+B)

schedule $\langle T_1, T_2 \rangle$ consistent
$\langle T_2, T_1 \rangle$ consistent

(4·7) (22)

T_1	T_2	Concurrency-control manager
lock-X(B)		grant-X(B, T_1)
200 R(B)		
150 B=B-50		
150 W(B)		
unlock(B)		
	lock-S(A)	grant-S(A, T_2)
	R(A) 100	
	unlock(A)	
	lock-S(B)	grant-S(B, T_2)
	R(B) 150	
	unlock(B)	
	display(A+B) 100+150 =250.	
lock-X(A)		grant-X(A, T_1)
100 R(A)		
150 A=A+50		
150 W(A)		
unlock(A)		

fig: schedule 1.

The concurrent ~~sche~~ transactions are in inconsistent state. The reason is Transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state.

transaction must hold a lock on data item as long as it accesses that item. Moreover, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

Suppose now that unlocking is delayed to the end of the transaction.

Trans T_3 corresponds to T_1 with unlocking delayed.

```
T3 : lock-X(B)
     R(B)
     B = B - 50
     W(B)
     lock-X(A)
     R(A)
     A = A + 50
     W(A)
     unlock(B)
     unlock(A)
```

Trans T_4 corresponds to T_2 with unlocking delayed.

```
T4 : lock-S(A)
     R(A)
     lock-S(B)
     R(B)
     display(A+B)
     unlock(A)
     unlock(B)
```

serial schedule $< T_3, T_4 >$ consistent.

Now we will see concurrent schedule.

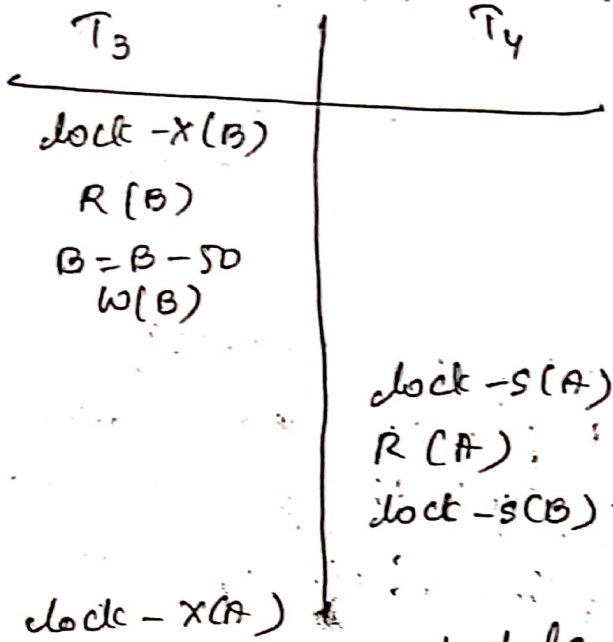


fig: schedule 2

From the above fig since T_3 is holding an X-mode lock on B and T_4 is holding a S-mode lock on B, & ~~is requesting~~ T_4 is waiting for T_3 to unlock B.

similarly T_4 is holding a S-mode lock on A & T_3 is requesting an X-mode lock on A, T_3 is waiting for T_4 to unlock A.

Thus, we have arrived at a state where neither of these transactions can never proceed with its normal execution. This situation is called deadlock. When a deadlock occurs, the sys must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to other transa, which can continue with its execution.

- If we donot use locking, or if we unlock data items as soon as possible after reading or writing them, we may get inconsistent states.
- If we do not unlock a data item before requesting a lock on another data item, deadlock may occur.

The inconsistent states may lead to real-world problems that cannot be handled by the db sys. So each transaction in sys follow a set of rules called locking protocols indicating when a transaction may lock & unlock each of data items.

Granting of Locks:

If a transaction request a lock on data item in a particular mode & no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

eg:

	T_1	T_2	T_3	T_4
		lock-S(Q)		
	lock-X(Q)	⋮	lock-S(Q)	
		released		lock-S(Q)

starvation

Suppose a transaction T_2 has a shared-mode lock on a data item, & another Trans T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a Trans T_3 may request a shared-mode lock on the same data item. The lock req is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, & is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, & each trans releases the lock a short while after it is granted, but T_1 never gets the X-mode lock on the data item. The transaction T_1 may never make progress, & is said to be starved.

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

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

(i) There is no other transaction holding a lock on Q in a mode that conflicts with M.

(ii) There is no other trans that is waiting for a lock on Q, & that made its lock request before T_i

Thus, a lock req will never get blocked by a lock req that is made later.

Two-Phase locking Protocol:

The protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

(i) Growing phase: A transaction may obtain locks, but may not release any lock.

(ii) Shrinking phase: A trans may release locks, but may not obtain new locks.

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

ex: ① T_3: lock-X(B)
R(B)
B = B-50
W(B)
lock-X(A)
R(A)
A = A+50
W(A)
unlock(B)
unlock(A)

T_4: lock-S(A);
R(A)
lock-S(B)
R(B)
display(A+B)
unlock(A)
unlock(B)

T_3 & T_4 are two phase.

ex: ② T_1: lock-X(B)
R(B)
B = B-50
W(B)
unlock(B)
lock-X(A)
R(A)
A = A+50
W(A)
unlock(A)

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

T_1 & T_2 are not two phase.

T_1 / T_2
lock-X(A) / lock-S(B)
lock-X(B) / lock-S(A)

The two phase locking protocol ensures conflict serializability.

The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction.

Now transaction can be ordered according to their lock points.

Two phase locking does not ensure freedom from deadlock.

Ex: T_3 & T_4 are two phase.

ex:

T_1	T_2
lock-X(B)	
R(B)	
B = B - 50	
W(B)	
	lock-S(A)
	R(A)
	lock-S(B)
	⋮
lock-X(A)	

T_1	T_2
lock-S(A)	
	lock-S(A)
lock-X(B) (lockpoint)	
unlock-(A)	
	lock-X(C) (lock point)
unlock(B)	
	unlock(A)
	unlock(C)

This schedule is two-phase but it is in deadlock state.

cascading rollback may occur under two-phase locking.

Ex:

T_5	T_6	T_7
lock-X(A)		
R(A)		
lock-S(B)		
R(B)		
W(A)		
unlock(A)		
	lock-X(A)	
	R(A)	
	W(A)	
	unlock(A)	
		lock-S(A)
		R(A)

each transaction observes the two-phase locking protocol but the failure of T_5 after the R(A) step of T_7 leads to cascading rollback of T_6 & T_7.

Cascading rollbacks can be avoided by making modification of 2-phase locking called strict 2-phase locking protocol.

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

Ex: Strict - 2 Phase locking Protocol

T_1	T_2
lock - S(A)	
R(A)	
	lock - S(A)
lock - X(B)	
unlock(A)	
R(B)	
W(B)	
	R(A)
	unlock(A)
Commit	
unlock(B)	
	lock - S(B)
	R(B)
	unlock(B)
	Commit

It guarantees cascadeless recoverability.

Rigorous two phase locking protocol requires that all locks be held until the transaction commits, In this transaction can be serialized in the order in which they commit.

Ex:

T_1	T_2
lock - S(A)	
R(A)	
	lock - S(A)
lock - X(B)	
	R(A)
R(B)	
W(B)	
commit	
unlock(B)	
	lock - S(B)
	R(B)
unlock(A)	
	commit
	unlock(A) unlock(B)

Lock Conversions:

- It is a mechanism for upgrading a shared lock to an exclusive lock (upgrade). Upgrading can take place in growing phase.
- mechanism for downgrading a X-lock to a shared lock (downgrade). Down grading can take place in shrinking phase.

Ex:

T_8 : R(A_1)
R(A_2)
⋮
R(A_n)
W(A_1)
⋮

T_9 : R(A_1)
R(A_2)
display($A_1 + A_2$)

T_8 and T_9 run concurrently under 2-phase locking protocol.

T_8	T_9
lock-S(A_1)	
	lock-S(A_1)
lock-S(A_2)	
	lock-S(A_2)
lock-S(A_3)	
lock-S(A_4)	
⋮	unlock(A_1)
	unlock(A_2)
lock-S(A_n)	
upgrade(A_1)	

By locking A_1 in shared mode we can achieve concurrency. Thus, we can get no conflict. By completing all the locking we have to convert A_1 into X-lock. Thus we use upgrade, which converts shared to exclusive and then we can write. This is used in growing phase.

Downgrade is used in shrinking phase.

Implementation of locking:

(mechanism to manage the locking requests made by transactions)

- A lock-manager can be implemented as a process that receives messages from transactions & sends messages in reply.
- The lock-mgr process replies to lock-request msgs with lock-grant msgs or roll back (abort) in case of deadlocks.
- Unlock msgs requires only an acknowledgement in response.
- The lock mgr uses this data structure: For each data item i.e currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived.

It uses a hash table, indexed on the name of data item, to find the linked list for a data item. This table is called lock table

Each record of the linked list for a data item ~~this~~ notes which transaction made the request, & what lock mode is requested. The record also notes if the request has currently been granted.

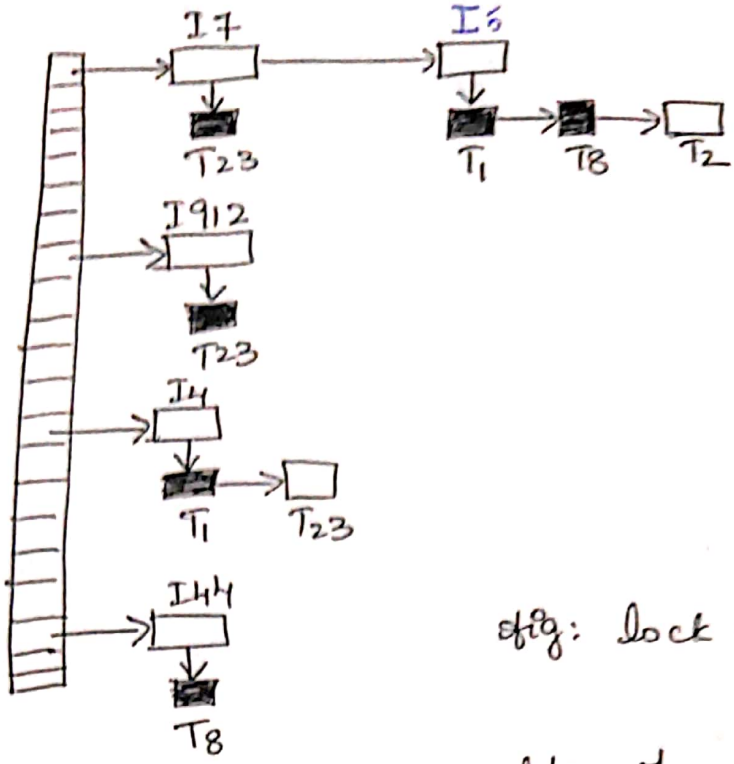


fig: lock table

The above fig shows an example of a lock table. The table contains locks for five different data items, I_4, I_7, I_{23}, I_{44} & I_{912}.

The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items.

Granted locks are filled-in (black) rectangles, while waiting requests are the empty rectangles.

For example, T_{23} has been granted locks on I_{912} & I_7 and is waiting for a lock on I_4.

New requests are added at end of queue & granted if it is compatible with earlier locks

If transaction aborts, all waiting or granted requests of the transaction are deleted;

Graph Based Protocol.

Simplest graph based protocol is Tree locking protocol

Tree locking protocol is used to employ exclusive lock & when the db is in the form of a tree of data items. Tree locking protocol is serializable.

→ Tree based protocol.

- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items can be unlocked at any time.

This protocol ensures Conflict serializability & Deadlock Free schedule.

Here we need not wait for unlocking a Data item as we did in 2-PL Protocol, thus increasing the concurrency.

The prerequisite of this protocol is that we know the order to access a Database items.

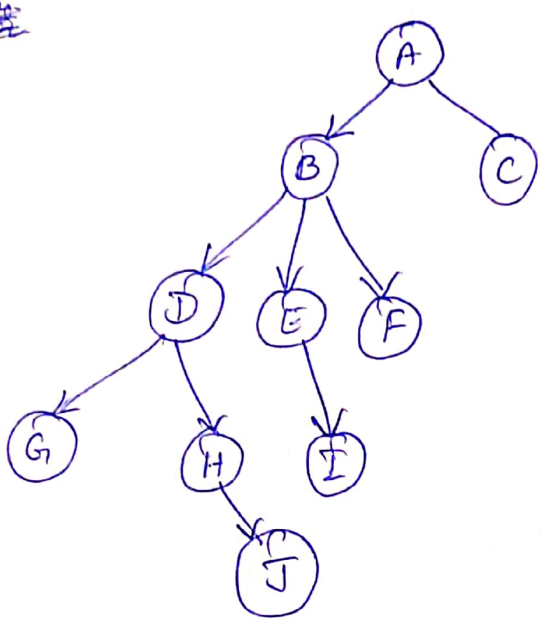
For this we implement a Partial Ordering on a set of Database items (D) $\{d_1, d_2, d_3, \ldots, d_n\}$

The protocol following the implementation of Partial ordering is stated as —

→ if $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.

- Implies that the set D may now be viewed as a directed acyclic graph (DAG) called a database graph

Ex:



Data base graph

Based on the above Database graph we will see a example.

T_1	T_2	T_3	T_4
Lock-X(B)			
	Lock-X(D)		
	Lock-X(H)		
	unlock-X(D)		
Lock-X(E)			
Lock-X(D)			
unlock-X(B)			
unlock-X(E)			
		Lock-X(B)	
		Lock-X(E)	
	unlock-X(H)		
Lock-X(G)			
unlock-X(D)			
			Lock-X(D)
			Lock-X(H)
			unlock-D
			unlock-H
		unlock(B)	
		(E)	
unlock(G)			

Ex: The following 4 transaction follow the tree protocol on previous db graph.

T_1	T_2	T_3	T_4
lock-X(B)	lock-X(D)	lock-X(B)	lock-X(H)
lock-X(E)	lock-X(H)	lock-X(E)	lock-X(J)
lock-X(D)	unlock-X(D)	unlock-X(E)	unlock-X(H)
unlock-X(B)	unlock-X(H)	unlock-X(B)	unlock-X(J)
unlock-X(E)			
lock-X(G)			
unlock-X(D)			
unlock-X(G)			

Advantages:
- Shorter waiting time and increases in concurrency.
- Protocol is deadlock free, no rollback are required.

Disadvantages:
- Protocol doesnot guarantee Recoverability or cascade freedom
- In tree locking protocol a transaction that needs to access data item A & J in the db graph, must lock not only A & J but also data items B, D, H. This additional locking results in increased locking overhead & the possibility of additional waiting time & potential decrease in concurrency.

Time Stamp Ordering Protocol:

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first.

To determine the timestamps of the transaction, this protocol uses system time or logical counter.

Ex: Let's assume there are two transactions T_1 & T_2. Suppose the transaction T_1 has entered the system at 007 sec & transaction T_2 has entered at 009 sec. T_1 has higher priority, so it executes first as it is entered the sys first.

- For each data item we maintain two Timestamps:

W-TS(X): largest timestamp of any transaction that executed write(X) successfully.

R-TS(X): is the largest timestamp of any transaction that executed read(X) successfully.

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

T issues a read(x) operation:

```
if (W-TS(X) > TS(T))
    Abort T & Roll back;
else
{
  Read(X)
  R-TS(X) = TS(T);
}
```

T issues a write(X) operation:

```
if (R-TS(X) > TS(Ti)) OR (W-TS(X) > TS(Ti)) then
    Abort-T & Rollback;
else
{
 Write(X)
 W_TS(X) = TS(Ti);
}
```

Ex:

T_1	T_2
R(x)	
	W(x)
R(x)	

initially RTS = 0 , WTS = 0

TS of $T_1 = 1$

$T_2 = 2$

1) T_1 issues Read operation

WTS(x) > TS(T)

0 > 1 False.

else

Read(x)

R_TS(x) = 1. (i.e TS(T_1))

2) Now Write(x)

R-TS(x) > TS(T_i) or W-TS(x) > TS(T_i)

1 > 2 0 > 2

False

else write(x)

W-TS(x) = 2

3) Read(x)

2 > 1

True

Abort & roll back.

Time stamp protocol ensures freedom from deadlock as no transaction ever waits.

- But these schedules may not be cascade-free & may not even be recoverable.

Thoma's write Rule:

This rule states that in case of

if $TS(T_i) < W\text{-}TS(x)$

Operation rejected & T_i rolled back.

Time stamp ordering rules can be modified to make the schedule view serializable.
Instead of making T_i rolled back, the 'write' operation itself is ignored.

```
If (RTS(x) > TS(x))
    Abort & Rollback.
If (write-TS(x) > TS(x))
    Reject write but commit
    the transaction.
else
    {
    w(x)
    w_TS(x) = TS(Ti);
    }.
```

Ex:

T_1	T_2
w(x)	
	w(x)
w(x)	

This is not necessary.

T_1	T_2
w(x)	
	commit
w(x)	

4·26

Validation Based Protocols also called optimistic concurrency control

It imposes less overhead

Also based on Timestamp Protocol. It has three phases:

1. **Read Phase:** During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variable local to T_i. It performs all the write operations on temporary local variables without update of the actual database.
2. **Validation Phase:** Transaction T_i performs a validation test to determine whether it can copy to database the temporary local variables that hold the result of write operations without causing a violation of serializability.
3. **Write Phase:** If Transaction T_i succeeds in validation, then the system applies the actual updates to the database, otherwise the system rolls back T_i.

To perform the validation test, we need to know when the various phases of transaction T_i took place. We shall therefore associate three different timestamps with transaction T_i.

1. **Start (T_i)**: the time when T_i, started its execution.
2. **Validation (T_i):** the time when T_i finished its read phase and started its validation phase.
3. **Finish (T_i)**: the time when T_i finished its write phase.

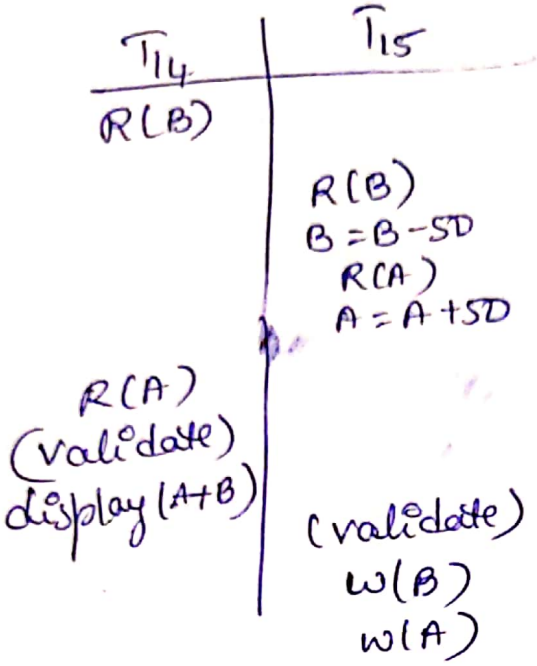
The Validation Test for T_j requires that, for all transaction T_i with $TS(T_i) < TS(T_j)$ one of the following condition must hold

1. Finish (T_i) < Start (T_j): Since T_i completes its execution before T_j started, the serializability order is indeed maintained.
2. Start(T_j)<Finish(T_i) <validation(T_j): The validation phase of T_j should occur after T_i finishes.

Few Points

1. Resolves the cascade rollbacks
2. Suffers from the starvation.
3. Optimistic concurrency control.

Ex: Schedule Produced by Validation:



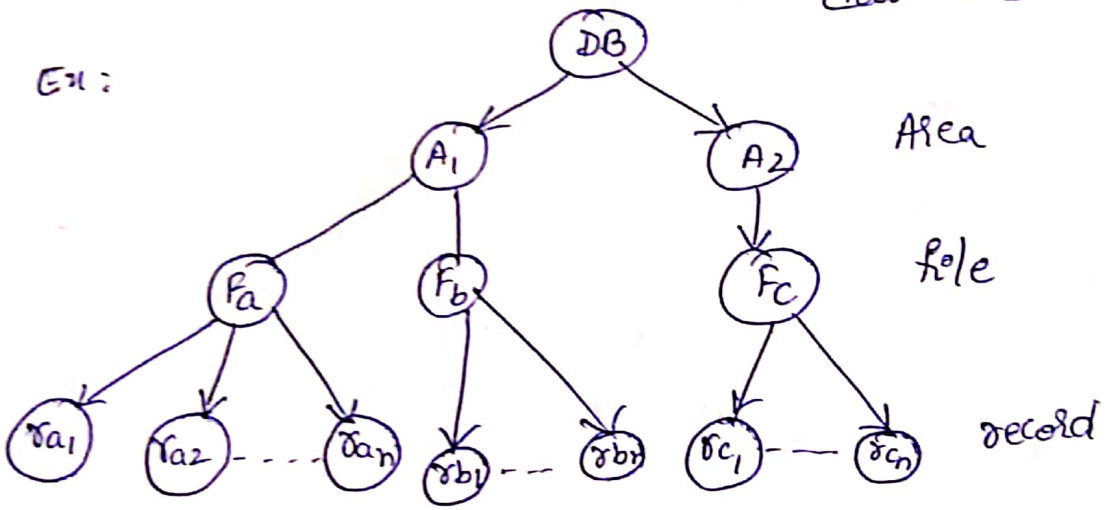
Suppose that $TS(T_{14}) < TS(T_{15})$ then the validation phase succeeds in the above schedule. Note that the writes to the actual variables are performed only after the validation phase of T_{15}. Thus T_{14} reads the old values of B & A & this schedule is serializable.

Multiple Granularity:

Granularity means diff level of data.

- Multiple granularity locks allow us to lock at different granularities (db, tables, pages, tuples).
- This is useful because we can choose deff granularities for diff transactions.

Ex:



Consider the above tree which consists of 4 levels of nodes.

- Highest level represents entire db.
- below it area
- area in turn has file as child (No file is more than in one area).
- file consists of records. (no record can be present in more than one file).

When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in same lock mode.

Consides two cases

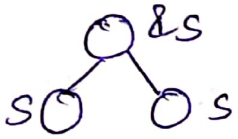
Case-1: If we want to lock ra_2, we have to check from root to node, Is there any anscester locked already.

Case-2: for example ra_1 is locked.
Now if we want to lock DB we can't as ra_1 is already locked.

To address the above problems we introduce Intention lock modes:

In addition to S & X lock modes, there are three additional lock modes with multiple granularity:

- Intension-shared (IS): indicates explicit locking at a lower level of the tree but only with shared locks

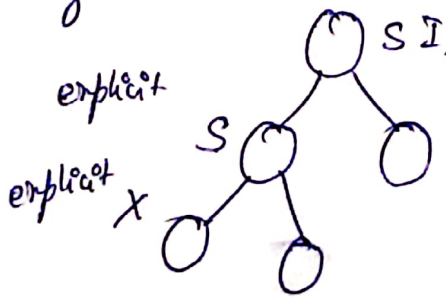


- Intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks.

X - explicit lock.

implicit.
are locked automatically in X-lock.

- Shared & Intention-exclusive (SIX): the subtree rooted by that node is locked explicitly in shared mode & explicit locking is being done at a lower level with exclusive-mode locks.



Compatibility Matrix:

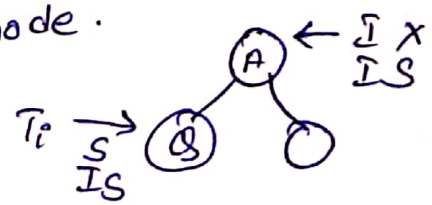
T_i \ T_j	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

Parent locked in	Child can be locked in.
IS	IS, S
IX	IS, S, IX, X, SIX
S	{S, IS} not necessary
SIX	X, IX, {SIX}
X	none

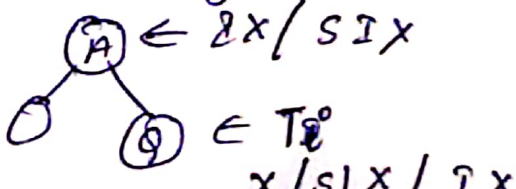
Rules:

Transaction T_i can lock a node Q using following rules:

- lock compatibility matrix must be observed.
- Root of the tree must be locked first, & may be locked in any mode.
- A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.

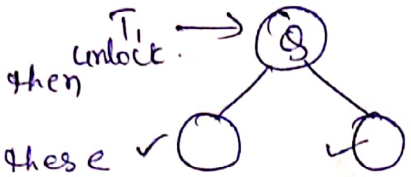


- A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.



— T_i can lock a node only if it has not previously unlocked any ~~node~~ node so far.

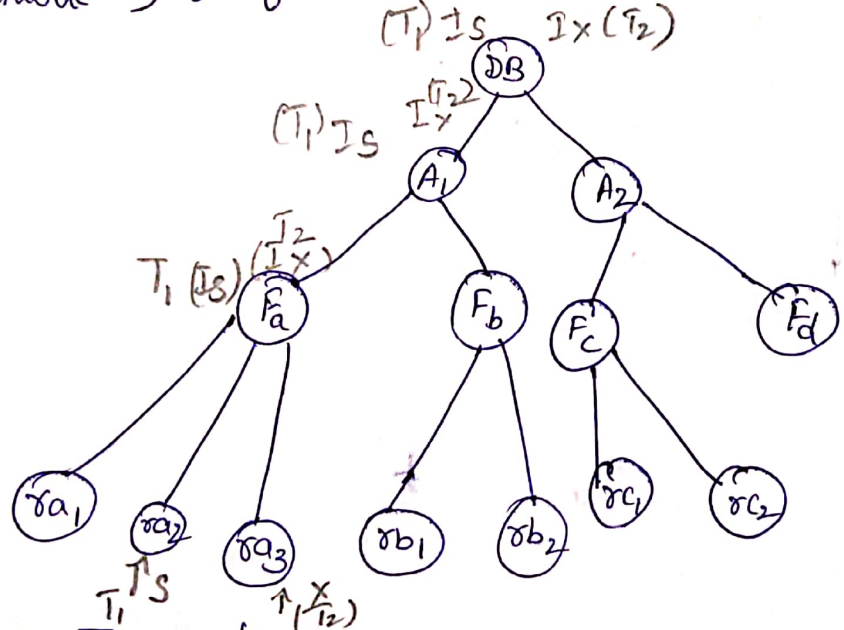
— T_i can unlock a node Q only if one of the children of Q are currently locked by T_i.



must be unlocked by T_1.
then only Q be unlocked.

Lock → root-to-leaf order
unlock → leaf-to-root order.

Ex:



→ (a) T_1 reads ra_2
to apply share lock to ra_2 traverse from root & apply IS to DB, A, F_a.

→ (a) T_2 modifies ra_2 (writing the data).
This will not be obtained as ra_2 is locked in shared mode.

T_2 modifies ra_3.
ra_3 must obtain X lock at upper level.

→ T_3 reads all record on Fa
will not possible to make to shared mode
as it is not compatible with IX.
As shared clock can't be granted &, T_3 doesnot execute.

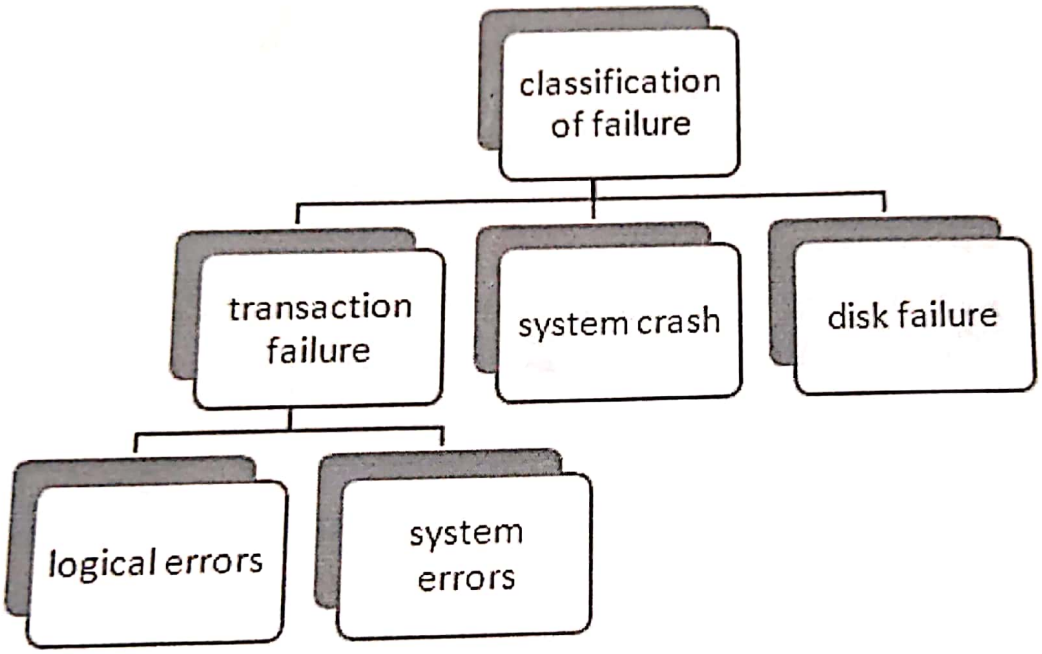
→ T_4 reads entire DB
i.e DB in shared mode not possible as it is
locked by T_2 in IX mode.

Recovery System

Database systems, like any other computer system, are subject to failures but the data stored in it must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Failure Classification:

We generalize a failure into various categories, as follows –



1. Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further.

Reasons for a transaction failure could be –

- **Logical errors** –Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** – Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

2. System Crash

There are problems – external to the system – that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

3. Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories –

- **Volatile storage** – As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** – These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
    - **input**(B) transfers the physical block B to main memory.

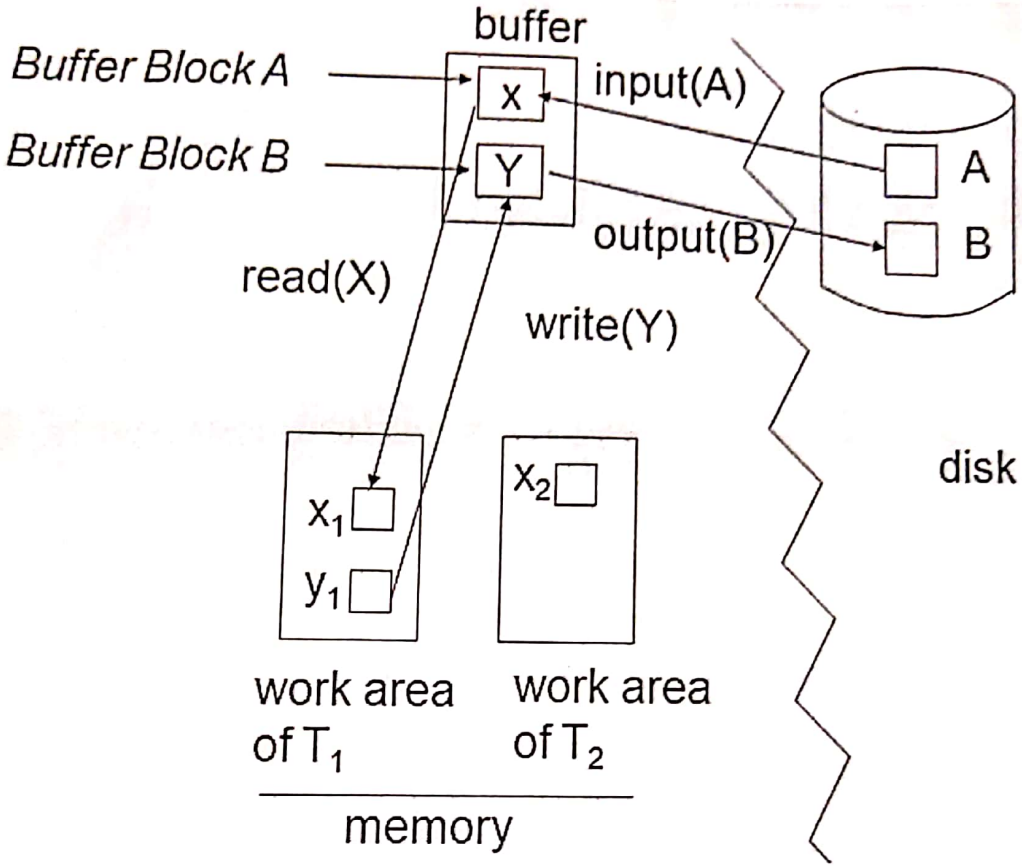
- **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.

  T_i's local copy of a data item X is called x_i.

- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- Transactions
  - Perform **read**(X) while accessing X for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following –

- It should check the states of all the transactions, which were being executed.

- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction –

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction.

- When transaction T_i starts, it **registers itself** by writing a $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record

  $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.

  Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.

- When T_i **finishes it last statement, the log record**

  $<T_i$ **, commit**> is written.

- **< Ti ,abort> Transaction Ti has aborted.**

We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

- **Deferred Modification Technique:** All logs are written on to the stable storage and the **database is updated when a transaction commits.**

Assume that transactions execute serially

- **Transaction starts by writing <T_i *start*> record to log.**
- **A write(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X**

Note: old value is not needed for this scheme

- **The write is not performed on X at this time, but is deferred(postponed).**
- **When T_i partially commits, <T_i commit> is written to the log**
- **Finally, the log records are read and used to actually execute the previously deferred writes.**
- **During recovery after a crash, a transaction needs to be redone if and only if both <T_i start> and<T_i commit> are there in the log.**
- **Redoing a transaction T_i (redoT_i) sets the value of all data items updated by the transaction to the new values.**
- **Crashes can occur while the transaction**
  - **is executing the original updates, or**
  - **while recovery action is being taken**

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0	T_1
read (A)	read (C)
$A := A - 50$	$C := C - 100$
Write (A)	write (C)
read (B)	
$B := B + 50$	
write (B)	

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since <T_0 commit> is present

(c) redo(T_0) must be performed followed by redo(T_1) since <T_0 commit> and <T_i commit> are present

- **Immediate Modification Technique:** the database is modified immediately after every operation.

  <T_i start>

  <T_i, X, V1, V2>

  <T_i commit>

- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Log	Write	Output
<T_0 start>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	$A = 950$ $B = 2050$	
<T_0 commit>		
<T_1 start>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 commit>		
		B_A

Note: B_X denotes block containing X.

Recovery procedure has two operations instead of one:

- **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
- **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

When recovering after failure:

- transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

4.35

Undo operations are performed first, then redo operations.

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Check Points

When a system crash occurs we must consult the log to determine those transactions that need to be redone & those that need to be undone. We need to search entire log to determine this information. There are two difficulties with this approach:

- Searching entire log is time-consuming
- We might unnecessarily redo transactions which have already o/p their updates to the database.

To reduce these problems we introduce checkpoints.

A checkpoint is performed as follows:

- o/p all log records residing in main memory to stable storage.
- o/p all modified buffer blocks to the disk.
- Write a log record <checkpoint> onto stable storage.

During recovery we have to consider only the most recent transaction T_i that started before the checkpoint, & the transactions that started after T_i.

(i) Scan backwards from end of log to find the most recent <checkpoint> record.

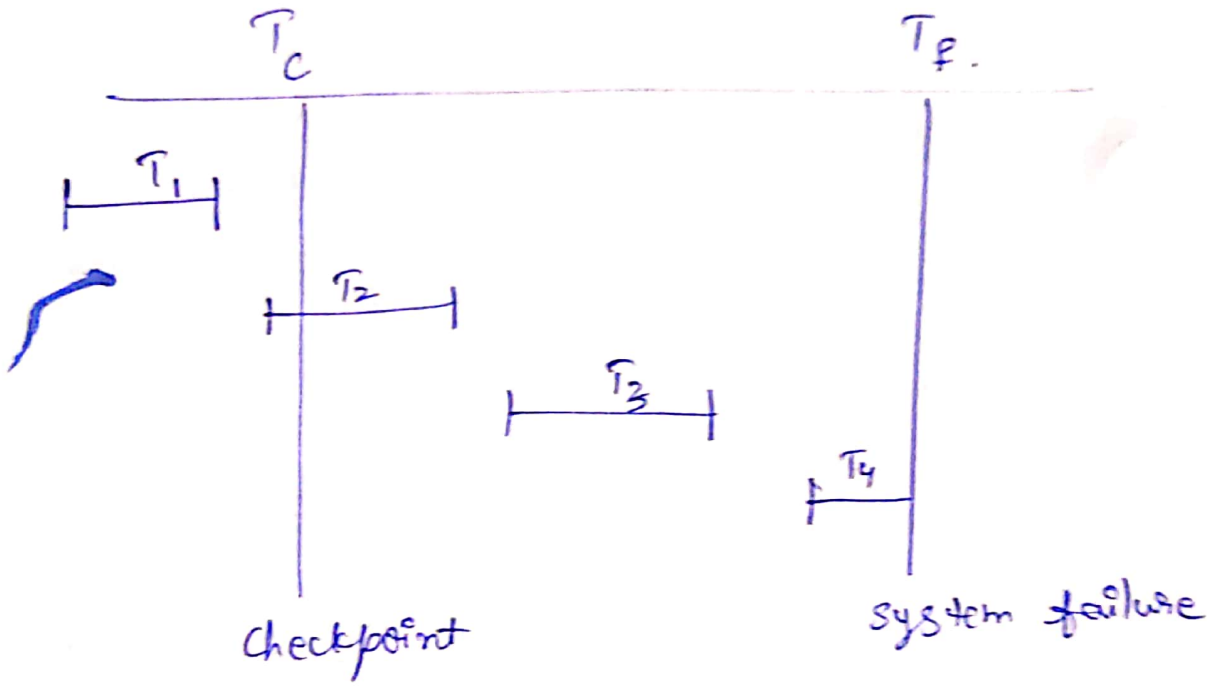
(ii) Continue scanning backwards till a record <T_i start) is found.

(iii) Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, & can be erased whenever necessary.

(iv) For all transactions (starting from T_i or later) with no $\langle T_i \text{ commit} \rangle$ execute undo(T_i).

(v) Scanning forward in the log, for all transactions starting from T_i or later with a $\langle T_i \text{ commit} \rangle$ execute redo(T_i).

Ex:



- T_1 can be ignored (updates already o/p to disk due to checkpoint.
- T_2 & T_3 redone
- T_4 undone.

Ex: if we have T_1 to T_{100} &

$\langle T_{95}, \text{starts} \rangle$
:
$\langle$ checkpoint 5 $\rangle$
$\langle T_{96}, \text{starts} \rangle$
:
$\langle T_{97}$ starts $\rangle$
:
$\langle T_{100}$ starts $\rangle$
:

as recent checkpoint is checkpoint 5, recent transaction is T_{95}, so at the time of recovery $T_{95}, T_{96}, \ldots T_{100}$ need to be considered.

Recovery with Concurrent Transactions:

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

- All trans share a single disk buffer & a single log.
- A buffer block can have data items updated by one or more transactions.

We assume concurrency control using strict-two phase locking:

- i.e the updates of uncommitted transactions should not be visible to other transactions.

otherwise how to perform undo if T_1 updates A, then T_2 updates A & commits, & finally T_1 has to abort?

Checkpoint log record is of the form:

- <checkpoint L>

where L is the list of transactions active at the time of checkpoint.

- We assume no updates are in progress, while the checkpoint is carried out.

When sys recovers from a crash, it first does the following:

- (i) initialize undo-list & redo-list to empty.
- (ii) Scan the log backwards from the end, stopping when the first <checkpoint L> record is found.

For each record found during the backward scan:

- if the record is <T_i commit> add T_i to <redo-list>

- if the record is <T_i start> then if T_i is not in redo-list add T_i to undo list.

- For every T_i in d, if T_i is not in redo-list, add T_i to undo-list.

Once the lists are created, the recovery proceeds as follows:

- The sys again scans the log from most recent record in backward, & performs undo for each log record belongs to T_i in undo list. Log records of redo trans list is ignored.
  The scan stops when <T_i start> records have been found for every trans T_i in undo list.

- The sys locates the most recent <checkpoint L> record on the log.

- The sys scans log in forward from most recent <checkpoint L> record & performs a redo for each log record that belongs to T_i that is an redo list. It ignores log records of trans on the undolist in this phase.

undo - reverse
redo - forward
only operations after <Checkpoint L> has to be undo or

Ex:

<T1 Start>

<write, T1, B, 2, 3>

<start T2>

<Commit, T1>

<write T2, C, 5, 7>

<checkpoint, {T2}>

<start, T3>

<write, T3, A, 1, 9>

<Commit, T3>

<start, T4>

<write, T4 C, 7, 2>

imagine that there is a crash.

undo list ↑ backward. {T4, T2} active transaction. as it is not committed.

redo list → forward. {T3}

As T1 has committed before checkpoint list it does not appear in undo or redo list.

undo undo redo.

order : T4, T2, T3

(undo list first then redo list)

order of operations: (old).

T4 : C : 7.

T2 : X (nothing appears so no operation is done).

T3 : A : 9. redo is done,

we have to redo or undo after the transaction after checkpoint.