Transaction Management

Transaction :-

A Transaction is an execution of user program and is seen by the DBMS as a series of actions (or) list of actions.

⇒ There are two operations in Transactions [for accessing DataBase]

(i) Read(x): It Performs reading operation of data item 'x' from DB

(ii) Write(x): It Performs writing operation of data item 'x' to the DataBase

Example of Transaction :-

Let 'T' be a transaction that transfers 1000Rs from Account 'A' to Account 'B'. This transaction can be illustrated as follows.

```
T : [ Read (A);
      A := A - 1000;
      Write (A);

    [ Read (B)
      B := B + 1000;
      Write (B);
```

Properties of Transaction (or) ACID Properties

Every Transaction in DBMS must maintain the following 4 Properties

(i) Atomicity

(ii) Consistency

(iii) Isolation

(iv) Durability

(i) Atomicity: (All actions (or) Non actions)
operations

⇒ If you are doing any database transactions, All the operations should be executed (or) None operations should be executed.

⇒ Here "all the operations in the transaction" is considered as Single unit (or) Atomic task.

⇒ If System fails while performing transaction, The System need to be rollback to its previous (or) original state.

Eg: Consider a transaction to transfer 500Rs from Account A to B

Assume initial balance of Account A is 2000
" B is 1000

T:
R(A)
A := A-500 } Task1: Deduct 500 from Account A
W(A)
- - - - - - - -> At this stage, if System fails (A=1500, B=1000)
R(B)
B := B+500 } Task2: Add 500 to Account B
W(B)

⇒ If all operations are executed Successfully then the database is in Consistent state [A=1500, B=1500 A+B=3000 Same as orginal state]

⇒ For example, "Task-1 Successfully completed while Task2 fails." In this case, A=1500, B=1000 A+B=2500 which is not acceptable in banking System, So if failure occurs then we need to rollback the DB to its original state (or) previous Consistent state.

(ii) Consistency:

⇒ The DataBase must be consistent before and After execution of transaction, It refers to Correctness of the database

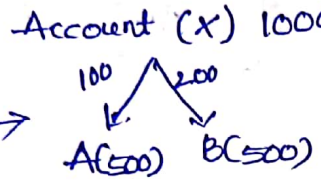
⇒ To preserve the consistency of the Database, The execution of all transactions should be in isolated form: i.e no other transactions executed parallely.

Eg: Consider a transaction to transfer ₹500 from Account A to B.

[Same as above Transaction example in Atomicity]

Here the consistency requirement is the sum of account A & B be unchanged by the execution of transaction.

Isolation:



⇒ If you are performing multiple transactions on single database item. then operations from any transaction should not interfere with operation in other transaction

⇒ The execution of all transactions should be in isolated form.
i.e for every pair of transactions, one transaction should start its execution, only when other transaction finished its execution.

Eg: T_1: Transfer 100 from X to A

T_2: " 200 " X to B

T_1	T_2
R(X) 1000	
X:=X-100	
W(X)(900)	
	R(X) x=900
	X:=X-200
	W(X)(700)
R(A) 500	
A:=A+100	
W(A)(600)	
	R(B) 500
	B:=B+200
	W(B) (700)

✓

T_1	T_2
R(X) 1000	(same time) interfere
	R(X) 1000
X:=X-100	
	X=X-200
W(X) 900	
	W(X) (800)

X

finall Account X contains 900 (or) 800

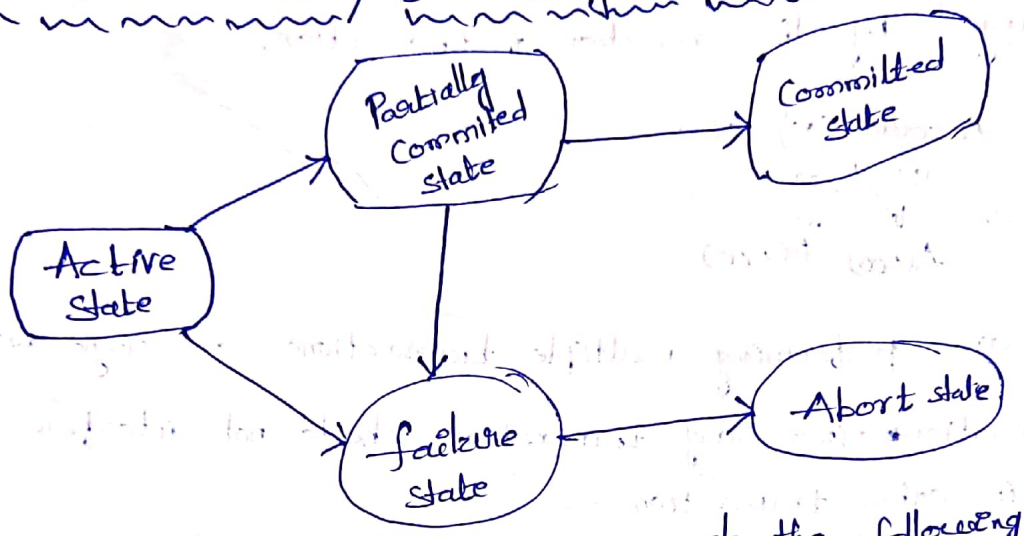
It's not correct

(iv) Durability:

⇒ Once a transaction is completed successfully, the changes it has made into database should be permanent even if there is a system failure.

⇒ The recovery manager of DBMS ensures the durability of transaction.

States of Transaction / State Diagram of Trs:



A transaction must be in one of the following states.

(i) Active state:-
⇒ This is initial state of the transaction, the transaction stays in this state while it is starting its execution.

(ii) Partially committed state:-
⇒ This state occurs before the last statement of the transaction has been executed.

(iii) Failed state:
⇒ This state occurs after discovering that "Normal execution can no longer proceed".

(iv) Abort state:
⇒ This state occurs after the transaction has been rolled back and the database has been restored to its original state. [start of Tr]

(v) Committed state:
⇒ This state occurs after the successful completion of the Transaction.

schedule:

When several transactions are executed concurrently then the order of execution of various action is known as schedule.

Types of schedule:

A schedule that contains either abort or commit statement for each transaction, is complete schedule, otherwise, it's incomplete schedule.

Ex:

T_1	T_2	T_3
		Read(C)
Read(A)		Write(C)
Write(A)		
commit		
	Read(B)	
	Write(B)	
	Abort	
		Read(D)
		Write(D)
		commit

Serial schedule:

If the actions of different transactions aren't interleaved (parallel)
i.e., transactions are executed from start to end, one by one, then we call it as a serial schedule.

Ex:

T_1	T_2
Read(A)	
Write(A)	
	Read(B)
	Write(B)
	commit

Parallel schedule:

If the actions of different transactions are interleaved, i.e., some actions of one transaction are executed 1st and followed by some actions of other transaction executed and followed by some actions of 1st executed and followed then we call it as parallel schedule.

Advantages of Concurrent Execution:

- It helps in reducing waiting time.
- Improves throughput (no. of transactions executed per unit).

Problems caused by interleaved execution (or) Anomalies due to interleaved executing:

(i) WR conflicts (Reading uncomitted data) or dirty read problem
(ii) RW conflicts (unrepeatable reads).
(iii) WW conflicts (lost update or overwritting uncomitted data).

WR conflicts:

A transaction T_2 to read a database object A that has been just modified by another transaction T_1 which has not yet comitted. Such a read is called dirty read. It leads to inconsistent database.

Ex: A = 500

T_1 : add Rs. 100 to account A

T_2 : add Rs. 200 to account A

T_1	T_2
R(A)	
A := A + 100	
W(A)	
	R(A)
	A := A + 200
	W(A)
	commit
fail (or) abort	

RW conflicts:

A transaction T_2 could change the value of object x that has been read by transaction T_1 and T_1 is still in progress.

- If T_1 tries to read the value of x again, it will get different result ~~over~~ even though it has not modified x in the mean time. This is called unrepeatable read problem.

Ex:

X : 500

T_1 (-100)	T_2 (-200)
R(x)	
↳ 500	
	R(x)
	X := X-200
	W(x)
	commit
X := X-100	
W(x)	
commit	

WW conflicts:

Ex:

T_1 (-100)	T_2 (-200)
R(x)	
X := X-100	R(x)
	X := X-200
W(x)	
commit	
	W(x)
	commit

A transaction T_2 could overwrite the value of an object X which has already been modified by transaction T_1 while T_1 is still in progress.

Serializability :-

→ Some non-serial schedules may lead to inconsistency of DataBase

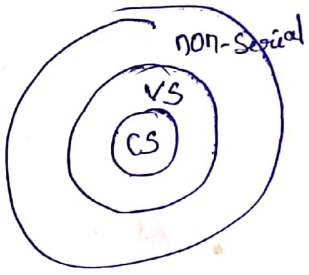
→ "Serializability" is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of DataBase

Serializable schedules :-

If a given non-serial schedule(s) of 'n' Transactions is equivalent to some serial schedule of same 'n' Transactions, Then the schedule S is called Serializable Schedule.

Types of Serializability :-

1. Conflict Serializability
2. View Serializability



1. Conflict Serializability :-

⇒ Conflict equivalent :- Two schedules are said to be conflict equivalent iff the order of any 2 conflicting operations are same in both schedules

⇒ If a non-serial schedule is conflict equivalent to some serial schedule then we call it as "Conflict Serializable schedule." (or)

If a non-serial schedule can be converted into some serial schedule by swapping its non-conflicting operations then we call it as "Conflict Serializable Schedule."

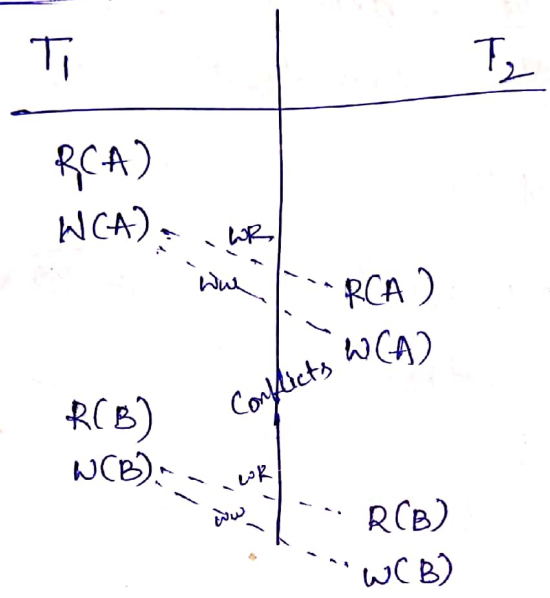
Conflict Operations :-

Two operations are called as conflicting operations, if the following conditions true for them.

- ⇒ Both the operations belongs to different transactions
- ⇒ Both the operations on the same data item
- ⇒ At least one of the two operations is a write operation

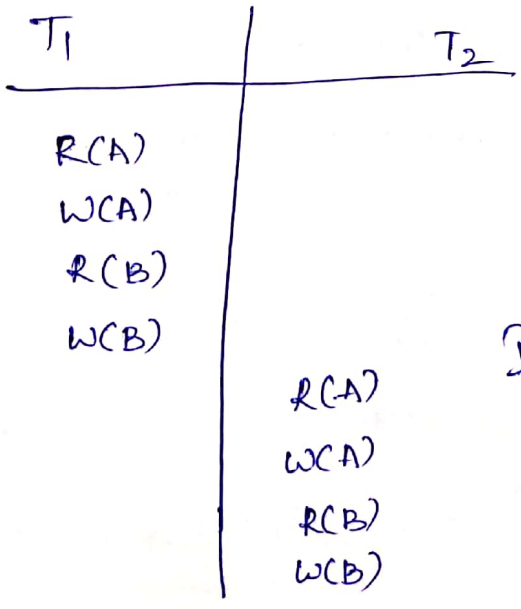
i.e: RW, WR, WW

Example: Schedule S



} Schedule 'S'

⇒ After swapping the non-conflicting operations the Schedule is



It is a Serial Schedule

∴ The above Schedule 'S' is Conflict Serializable Schedule

Conflict Serializability Testing :-

We use precedence graph to test whether a Schedule is Conflict Serializable or Not.

Steps

1. Creating a precedence graph by drawing one node for each transaction [No. of Nodes = No. of Transactions]
2. Draw an edge for each Conflict operations [$T_i \rightarrow T_j$] This ensures that T_i gets executed before T_j
3. If graph has "No cycles" in it, then given Schedule is Conflict Serializable otherwise Not.

Eg: Check whether the given Schedule 'S' is Conflict Serializable or Not

S: $R_1(A)$ $R_2(A)$ $R_1(B)$ $R_2(B)$ $R_3(B)$ $W_1(A)$ $W_2(B)$

Solution:

⇒ No. of Transactions 3

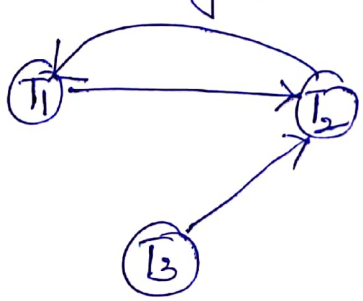
⇒ Conflicting operations

(i) $R_2(A)$ $W_1(A)$ $\quad T_2 \rightarrow T_1$

(ii) $R_1(B)$ $W_2(B)$ $\quad T_1 \rightarrow T_2$

(iii) $R_3(B)$ $W_2(B)$ $\quad T_3 \rightarrow T_2$

⇒ Precedence graph is

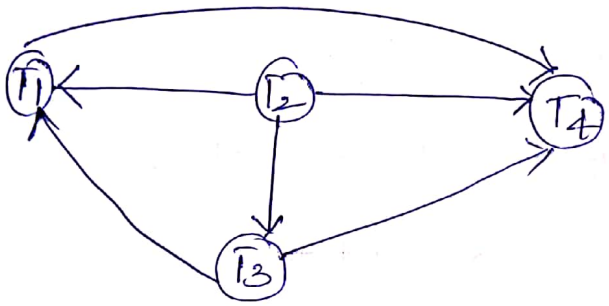


⇒ clearly, there exists a cycle.

∴ The given Schedule 'S' is not Conflict Serializable.

Eg2:

T_1	T_2	T_3	T_4
	R(X)		
		W(X)	
W(X)			
	W(Y)		
	R(Z)		
			R(X)
			R(Y)



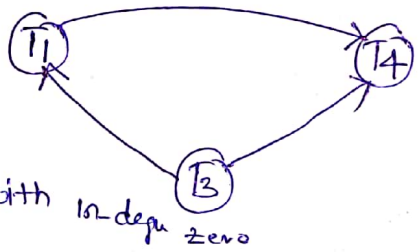
No cycle

∴ The above Schedule is Conflict Serializable.

Possible Serial orders:

→ Find the Node with in-degree zero, and remove from it the graph.

T_2 graph is



→ again find the Node with in-degree zero

$T_2 \rightarrow T_3$

Illy $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$

(2) View Serializability:-

⇒ View Equivalent: Two Schedules S_1 and S_2 are said to be View equivalent if the following 3 conditions hold true for them

(i) If transaction T_i reads the value of object A initially in schedule S_1, then in schedule S_2 also T_i must perform the initial read of Database object 'A'

i.e: "Initial reads must be same for all the data items"

(ii) If transaction T_i reads the value of object 'A', that has been updated by T_j in schedule S_1, then in schedule S_2 also transaction T_i reads object A that has been updated by transaction T_j.

i.e. "Write-read" sequence must be same

(iii) For each data object/item 'A' the transaction T_i that performs final write on 'A' in S_1, then in schedule S_2 also T_i must perform final write on 'A'

i.e. "Final writes must be same for all the data items"

⇒ "If a non-Serial Schedule is View equivalent to some Serial Schedule then we call it as "View Serializable Schedule"

⇒ All Conflict Serializable Schedules are View Serializable Schedules But All View Serializable Schedules may (or) may not be Conflict Serializable Schedules

View Serializability Testing :- steps

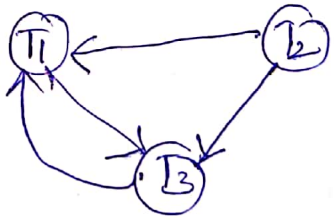
1. check whether the given Schedule is Conflict Serializable (or) not
2. if it is Conflict Serializable then it is surely View Serializable. stop the process
3. if it is not Conflict serializable then check if ∃ any "blind write"
4. if does not exist any blind write. then Schedule is not View Serializable
5. if there exist blind write, we go for dependency graph method.
6. Dependency Graph Method :-
   (i) By Using View equivalance 3 conditions, write all the dependencies
   (ii) Then, draw a graph Using those dependencies
   (iii) if there exists 'no cycle' in the graph then the Schedule is View Serializable otherwise not

{ Blind Write:
{ Perform Write operation
{ w/o reading is called
{ Blind write

Example:

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

Sol: (i) Check for Conflict Serializable Using precendence graph.



→ graph has 'cycle' so it is not Conflict Serializable.

(ii) check for blind writes, yes blind write exists ($W_3(A)$)

(iii) Dependency graph:

	A	
Initial Read:	T_1,	$T_1 \rightarrow T_3$
Update :	T_3, T_4	
Final Write	T_1	$(T_2, T_3) \rightarrow T_1$

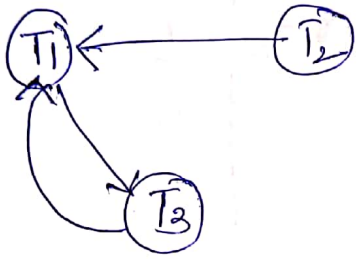
→ T_1 initially reads 'A' and T_3 initially updates A, so T_1 must execute before T_3 ∴ $T_1 \rightarrow T_3$

→ Final update (write) on A is made by the transaction T_1

∴ T_1 must execute after all other transactions. Thus we get the dependency $(T_2, T_3) \rightarrow T_1$

→ Does not exists write-read sequence.

∴ The dependency graph is



There exists a cycle in it

∴ The given schedule is Not View Serializable.

Eg: 2

S: $R_1(A)\ W_2(A)\ R_3(A)\ W_1(A)\ W_3(A)$

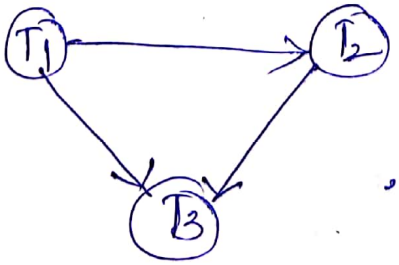
T_1	T_2	T_3
R(A)		
	W(A) → blind write	
		R(A)
W(A)		
		W(A)

→ Initial Reads : T_1, Initial update: T_2 ∴ $T_1 \rightarrow T_2$

Final Write : T_3 ∴ $(T_1, T_2) \rightarrow T_3$

Write-read : $T_2 \rightarrow T_3$

Dependency graph:



No cycle in it

∴ given Schedule is 'View Serializable'

→ Serialization order

$T_1 \rightarrow T_2 \rightarrow T_3$

Eg: 3

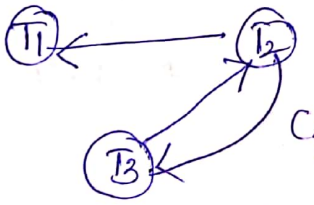
T_1	T_2	T_3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
		W(C)

	A	B	C
initial value Read	T_1, T_2	T_3, T_2	T_2
initial Update	T_1	T_2	T_3
Final write	T_1	T_2	T_3

A: $T_2 \rightarrow T_1$

B: $T_3 \rightarrow T_2$

C: $T_2 \rightarrow T_3$



Cycle formed

So The above is Not View Serializable.

Types of Schedules

- Serial
- Non-Serial
  - Serializable
    - Conflict S
    - View S
  - Non-Serializable
    - Recoverable
      - Cascading Schedule
      - Cascadeless schedule
      - Strict schedule
    - Non-Recoverable

* Non-Recoverable Schedules :-

If in a Schedule

→ A transaction perform a dirty read operation from an Un Committed transaction. [Dirty read Problem]

→ And Commits Before the transaction from which it has read the value [redacted]

then Such Schedule is Known as Non-Recoverable Schedule.

eg:

T_1	T_2
R(X)	
X=X+100	
W(X)	
	R(X)
	X=X+200
	W(X)
	Commit
R(Y)	
Rollback	

Committed Transactions Can't Rollback to its previous state.

T_2 Commits Befor T_1 ∴ The value that T_2 read Now stands to be incorrect, T_2 can't recover since it has already committed.

Recoverable Schedules :-

If in a Schedule.

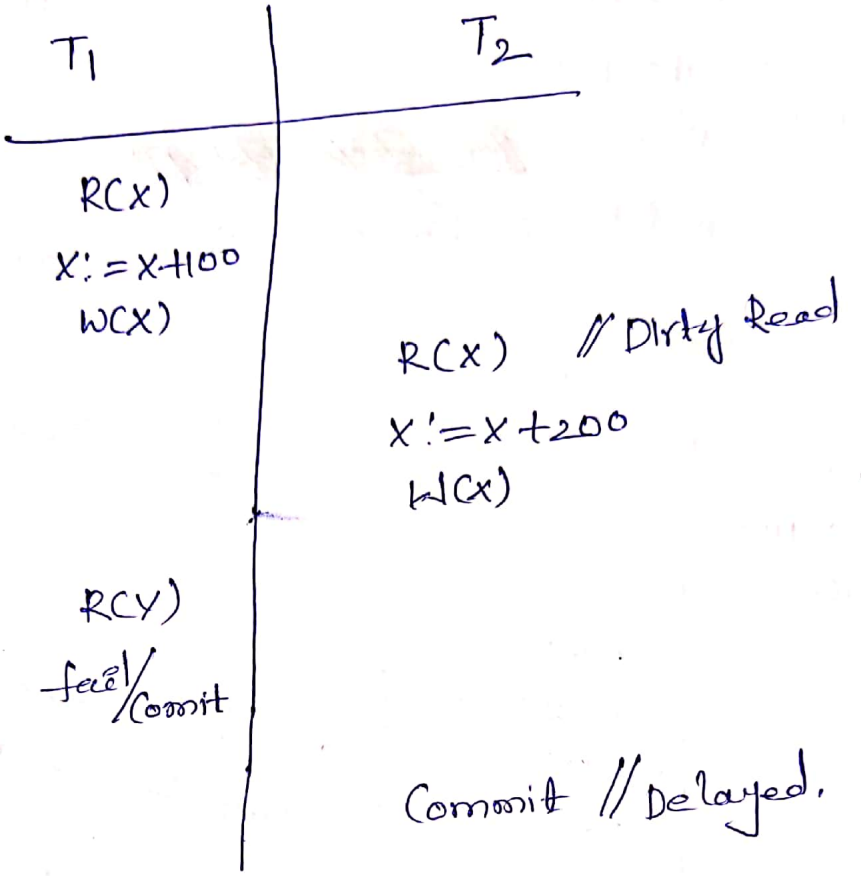
→ A transaction Performs a dirty read operation from an Un Committed transaction.

→ And it's Commit operation is delayed till the UnCommitted transaction either Commit (or) rollbacks

then Such a Schedule is Known as Recoverable Schedule.

→ Here the commit operation of the transaction that performs dirty read is delayed. this ensures that it still has a chance to recover if the uncommitted transaction fails later.

Eg:



Recoverable Schedule

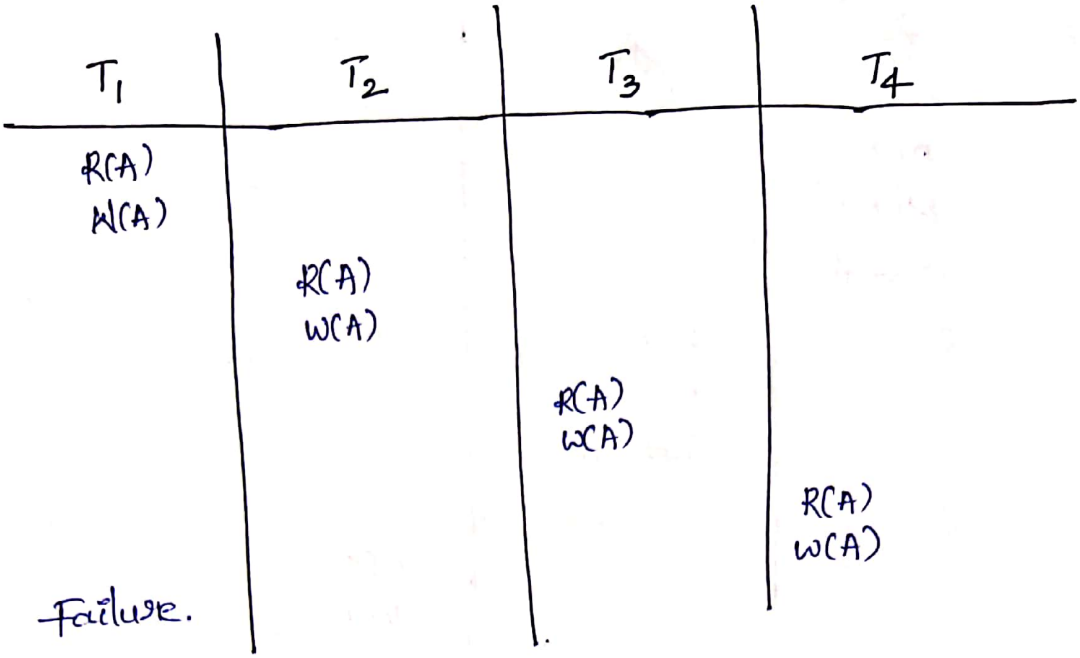
Types of Recoverable Schedules:-
1. Cascading Recoverable
2. Cascadeless "
3. Strict Recoverable Schedule

1. Cascading Recoverable Schedule:-

⇒ If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort. then such a schedule is called as "Cascading Schedule" (or) "Cascading Rollback/Abort"

⇒ It simply leads to the wastage of CPU time

Eg:



Cascading Recoverable Schedule

⇒ Here: T_2 depends on T_1

T_3 " T_2

T_4 " T_3

⇒ In above Schedule

(i) The failure of T_1 causes T_2 to rollback

(ii) The Rollback of T_2 " T_3 "

(iii) " T_3 " T_4 "

Such a rollback is called as "Cascading Rollback"

(2) Cascadeless Recoverable Schedule:

⇒ If in a Schedule, A transaction is not allowed to read a data item until the last transaction that has ~~written~~ Perform write operation, it is committed (or) aborted, then such a Schedule is called as "Cascadeless Recoverable Schedule".

i.e Cascadeless schedule allows only committed read operations

∴ It avoids cascading rollback and thus saves CPU time.

Eg:

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

Cascadeless Schedule

Note: It allows committed read & uncommitted write operations.

Eg:

T_1	T_2
R(A)	
W(A)	
	W(A) // uncommited write operation (Blend write)
Commit	

This is also Cascadless Schedule

3. Strict Recoverable Schedule :-

⇒ If in a Schedule, A transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed (or) aborted, then such a Schedule is called a Strict Schedule.

(or)

⇒ Strict Schedule allows only Committed read & write operations

⇒ It implements more restrictions than Cascadeless Schedule

Eg:

T_1	T_2
R(A)	
W(A)	
Commit/Rollback	
	R(A)/W(A)

Strict Schedule

⇒

Concurrency Control Protocols:-

⇒ Concurrency control is a process of managing Parallel/Simultaneous execution of transactions in shared database. to achive Socializability, Recoverable Schedules

⇒ Need of Concurrency Control:

(i) To enforce isolation - - - -→ (Refer ACID properties)
(ii) To preserve database consistency
(iii) To resolve Conflicts (RW, WR, WW Conflicts)

⇒ Three methods for Concurrency Control

1. Lock based protocols ← 2PL, strict 2PL
2. Time stamp based protocols → Basic Protocol, Thomas Write Rule
3. Validation based protocols

(1) Lock based Protocols:-

→ DBMS must be able to ensure that only Socializable, recoverable Schedules are allowed. by adopting some locking methods.

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

⇒ locks are of two kinds:

1. Binary locks : Data item can be locked in two states
   (i) lock(A) (ii) Unlock(A).
2. Shared/Exclusive locks

2 modes

1⇒ Shared lock (S) :-

→ Data item can only be read

→ This lock is requested using lock-S instruction.

2⇒ Exclusive lock (X) :-

→ Data item can be both read as well as written,

→ This lock is requested using Lock-X instruction.

⇒ Lock requests are made to concurrency-control manager,

⇒ Transaction can proceed only after request is granted.

⇒ lock Compatibility Matrix:

	S	X
S	True	false
X	false	false

→ Any no. of transactions can hold shared locks on an item

→ But, if any transaction holds an exclusive lock on the item, no other transaction may hold any lock on the item.

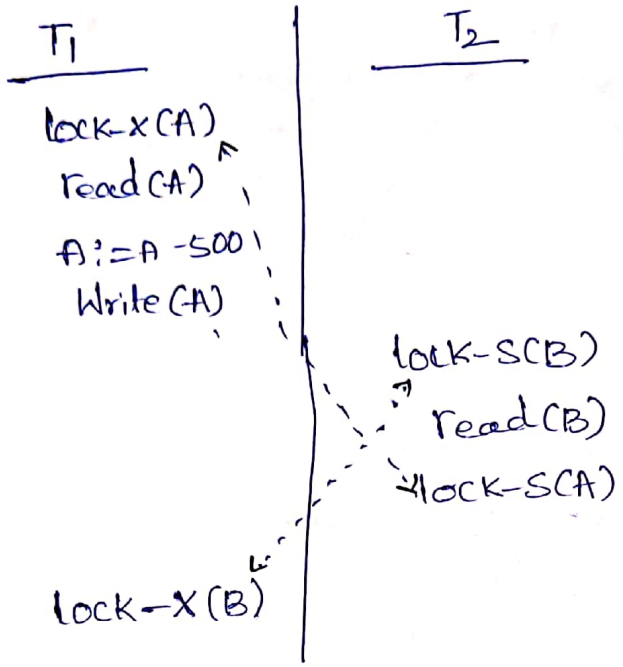
⇒ If a lock can't be granted, the requesting transaction is made to wait till all incomplete locks held by other transactions have been released, The lock is then granted.

Eg: T0: Lock-S(A);
read(A);
Unlock(A);
lock-S(B)
read(B);
unlock(B)

display(A+B)
:
Commit

→ locking as above is not sufficient to guarentee Serializability - "if A and B get updated in blw the read of A and B. the displayed sum would be wrong.

→ consider the partial schedule.



/*

→ "Here execution of lock-S(A) in T_2 is waits for T_1 to release it's lock on 'A'; and while executing lock-X(A) in T_1 is wait for T_2 to release its lock on 'B'" Such a situation is called "Dead lock"

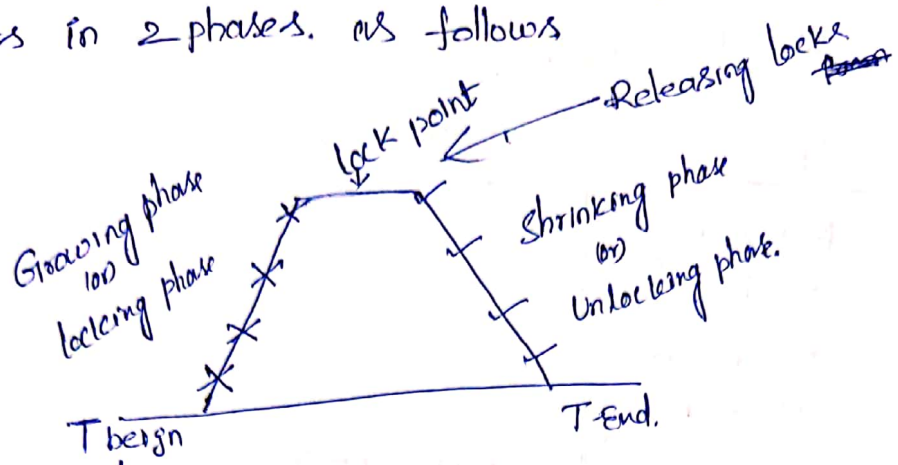
→ To Handle a dead lock one of T_1 or T_2 must be rolled back and its locks released. */

→ A locking Protocol is a set of rules followed by all transactions while requesting and releasing locks. locking protocols restrict the set of possible Schedules.

→ 2 types of locking protocols
(1) 2-phase locking
(2) Strict 2-PL

1. Two-phase locking protocol (2PL):-

→ 2PL requires that each transaction ensures lock & unlock requests in 2 phases. as follows



(i) Growing phase:-

→ Transaction may obtain locks

→ " may not release locks

(ii) Shrinking phase:

→ Transaction may release locks

→ Transaction may not obtain any new locks

⇒ Lock point: The point where a transaction obtain its final lock

⇒ If the transactions are executed in the order of their lock points then "Conflict Serializability" can be achieved.

⇒ Two-phase locking does not ensure freedom from deadlocks

→ Cascading roll-back is possible under 2PL. To avoid this follow a modified protocol called Strict 2-PL:

Lock conversion:

(i) Upgrading of lock: from S(A) to X(A) is allowed in growing phase

(ii) Downgrading of lock: from X(A) to S(A) must be done in Shrinking phase

2. Strict Two-phase locking protocol :- (Strict 2PL)

phase I : [The 1st phase of Strict 2PL is similar to 2PL.] i.e In the 1st phase If a transaction T wants to read an object A, it 1st requests S-lock. But if a transaction T wants to modify an object A, it 1st requests X-lock

phase II : All locks held by a transaction are released when transaction Committed 'or' Rollbacked.

Ex: schedule

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

⇒ This schedule is not allowed by Strict 2PL Because X-lock acquired by T_1 on object A Can't be released until T_1 ends [Either Commit or Abort] So, T_2 can't acquire X-lock on object A as it is held by T_1

⇒ However this schedule is allowed by 2PL, as it is allowed to release the lock before T_1 ends in 2PL. Only constraint is that once it release lock, It Can't acquire again.

Strict 2PL

T_1	T_2
X(A)	
R(A)	
W(A)	
⋮	X(A) ← waiting for T_1 to release lock
	R(A)
	W(A)
⋮	

2PL

T_1	T_2
X(A)	
R(A)	
W(A)	
Unlock(A)	
⋮	X(A)
	R(A)
	W(A)
	Commit
Commit/Abort	

"Release a lock once complete it work"

⇒ In constrast to 2PL, Strict-2PL does not release a lock after using it, It holds all locks untill the Commit point and release all the locks at a time.

(II) Time Stamp Based Protocols :-

→ The Time-stamp based method uses a "timestamp" to serialize the execution of concurrent transactions.

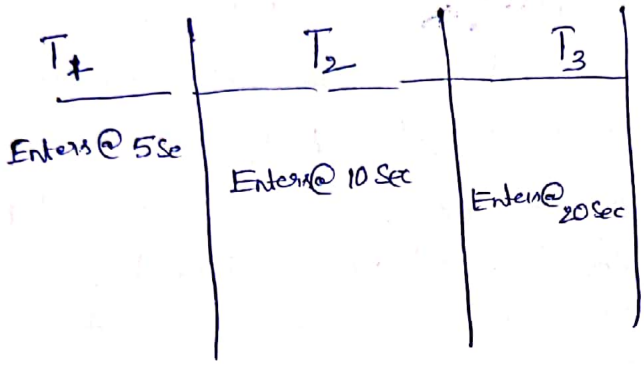
→ This protocol ensures that every conflicting read and write operations are executed in timestamp order.

→ "Time stamp is a tag that can be attached to any transaction, which denotes arrival time of transaction"

→ This protocol uses the system time (or) logical count as a Timestamp.

→ The older transaction is always given priority in this method.

⇒



Then (older) Time stamp(T_1) = 5

Time stamp(T_2) = 10

Time stamp(T_3) = 20 (younger)

⇒ This protocol use two Timestamp values relating to each data item 'x'

(i) Read_TS(x); The largest timestamp of any transaction that executed Read(x) operation successfully

(ii) Write_TS(x); The largest timestamp of any transaction that executed Write(x) operation successfully.

⇒ Basic Time ordering Protocol

(i) A transaction T_i issues a read(x) operation;

if (TS(T_i) < Write_TS(x)) ⇒ T_i older than the last transaction that wrote the value of 'x', request will fail

Abort T_i and Rollback.

else {

read(x);

read_TS(x) = Max(TS(T_i), Read_TS(x))

}

(ii) A transaction T_i issues a write(x) operation:-

if ($TS(T_i) < read\_TS(x)$ OR $TS(T_i) < write\_TS(x)$)

Abot T_i and Rollback.

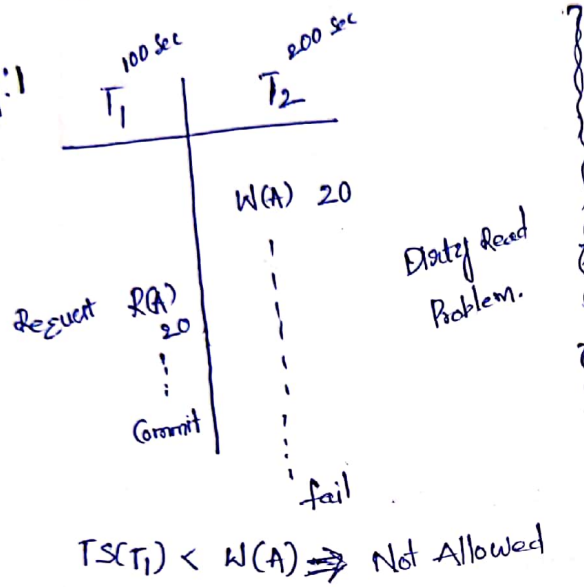
else <- - - - - - - - [$TS(T_i) \geq R(x)$ and $TS(T_i) \geq W(x)$]

{ write(x) performed

$write\_TS(x) = TS(T_i)$;

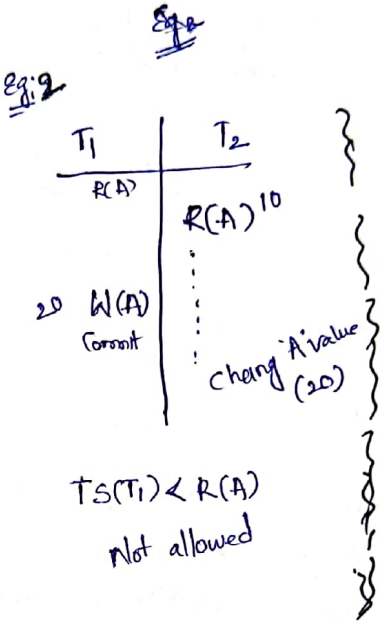
}

Eg:1



$TS(T_1) < W(A) \Rightarrow$ Not Allowed

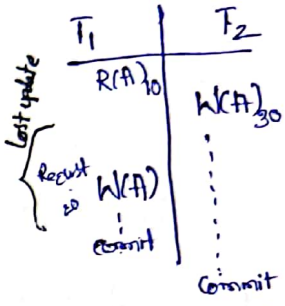
Eg:2



$TS(T_1) < R(A)$

Not allowed

Eg:3



$TS(T_1) < W(A)$

Not Allowed

* Thomas Write Rule :-

-> It is the modification to Basic Timestamp ordering, in which rules for Write operation are slightly different from Basic Timestamp ordering protocol.

-> A transaction T_i issues a Write(x) operation

if (~~Read~~ $TS(T_i) < read\_TS(x)$)

* Abort T_i and Rollback.

else if ($TS(T_i) < write\_TS(x)$)

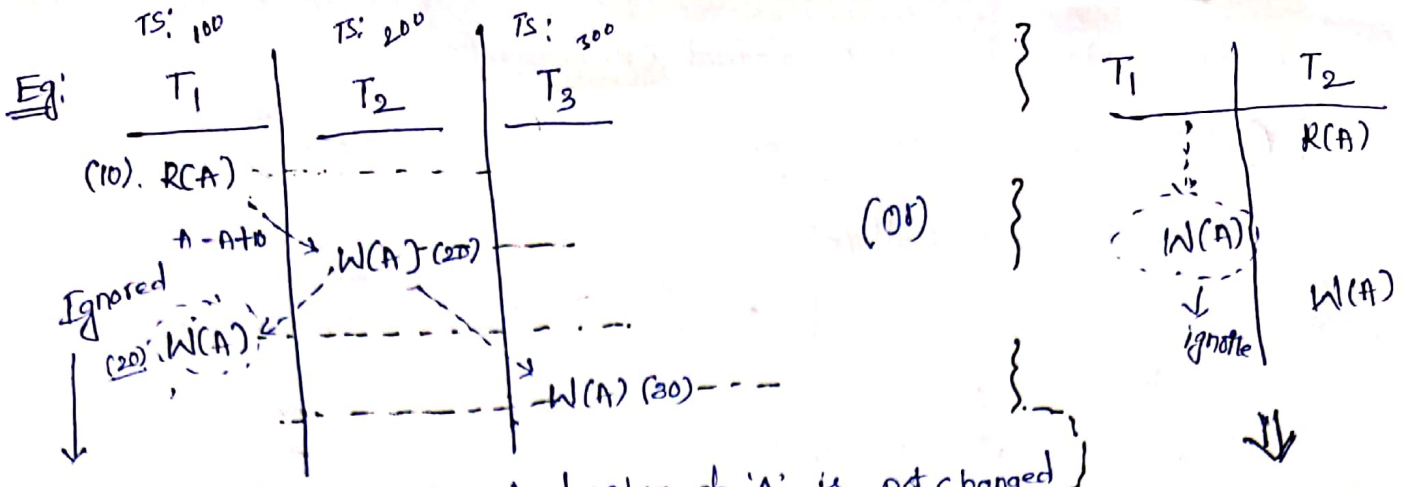
* Ignore the Write operation but continue the Execution of Transaction

else

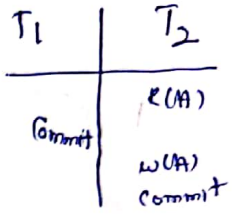
{ Write(x) Performed

$Write\_TS(x) = TS(i)$;

}



Even ignored this, the final value of 'A' is not changed



Conflict Serializable

III. Validation Based Protocol :-

(or)

Optimistic Concurrency Control Technique

→ In this protocol, The transaction is executed in the following 3 phases

1. **Read phase :-**
   → The transaction executed, It is used to read the values of various data items and store them in temporary local variables and It can perform all the write operations on temporary local variables w/o update to actual DB.

2. **Validation phase :-** → Transaction performs validation Test.
   → Before transaction commits, DBMS check whether the transaction conflicts with any other concurrent executing Transaction [for Serializability]
   → ~~If conflict, the transaction is aborted, local variables space and restarted for execution.~~

3. **Write phase :-**
   → If the validation determines that there is no possible conflicts, then the temporary results are written to the actual DataBase Otherwise The transaction is rolled back.

Validation conditions :

→ To perform the validation test, we need to know the following Timestamps

(i) Start (T_i) : The time when T_i started its execution
(ii) Validation(T_i) : The time when T_i finished its read phase and started its validation phase.

(iii) Finish(T_i): the time when T_i finished write phase.

→ For every pair of transactions T_i and T_j where $TS(T_i) < TS(T_j)$ one of the following validation conditions must hold.

(i) Finish(T_i) < Start(T_j) :→ Since T_i completes its execution before T_j started. The Serializability order is maintained.

(ii) Start(T_j) < Finish(T_i) < Validation(T_j):

→ If T_j started before Finish(T_i), the validation phase of T_j should occur after T_i finishes

~~→ Read set(T_j) is disjoint with write set(T_i)~~

Example

T_1	T_2
Read(B)	
	Read(B)
	B = B+50
	Read(A)
	A = A−50
Read(A)	
pass <Validate>	
display(A+B)	
	<Validate> pass
	Write(A)
	Write(B)

Refer Textbook

IV Multiple Granularity Locking (MGL):-

→ Multi version Concurrency Control Protocal:

→ Deadlock Prevention Alg ((i) Wait-die, (ii) Wound-wait)

→ Dead lock detection using waits-for graph.