- **Transaction**: Transaction is an executing program that forms a logical unit of database processing.

OR

logical work

or

logical operation

→ According to general computation principle (Operating system) we may have partially executed program, as the level of atomicity is instruction i.e., either an instruction is executed completely or not.

I_1 a=5, b=10
I_2 sum = a+b
⋮
I_n print(sum)

But in DBMS view, user performs a logical work which is atomic in nature i.e., either operation is executed or not executed, there is no concept like partial execution.

e.g: Transaction T_1 which transfer 50 units from account A to B.

T_1
R(A)
A = A−50
W(A)
R(B)
B = B+50
W(B)

In this transaction if a failure occurs after R(B) then the final status of the system will be inconsistent as 50 units are debited from account A but not credited in account B.

Here for consitency,

before $(A+B)$ == after $(A+B)$

To basic operations are :-

READ(X): Accessing the database item x from disk (where database stored data) to memory variable also name as X.

WRITE(X): Writing the data item from memory variable X to disk.

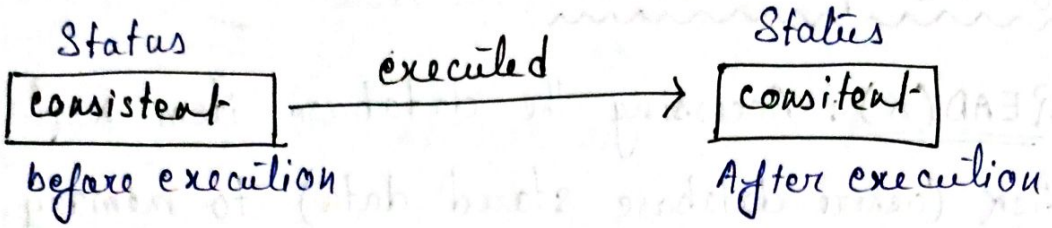
- **Desirable properties of transaction :**

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions :

1) **Atomicity**: As a transaction is set of logically related operations, either all them should be executed or none. (Partial execution should not be allowed).

→ It is the responsibility of recovery control manager / transaction control manager of DBMS to ensure atomicity.

2) **Consistency**: A transaction is consistency preserving if its complete execution takes the database from one consistent state to another.



→ The preservation of consistency of database is the responsibility of programmers (users).

3) **Isolation**: A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

→ The isolation property of database is the responsibility of concurrency control manager of database.

4) **Durability or permanency**: The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

→ It is the responsibility of recovery control manager of DBMS

Transaction State :

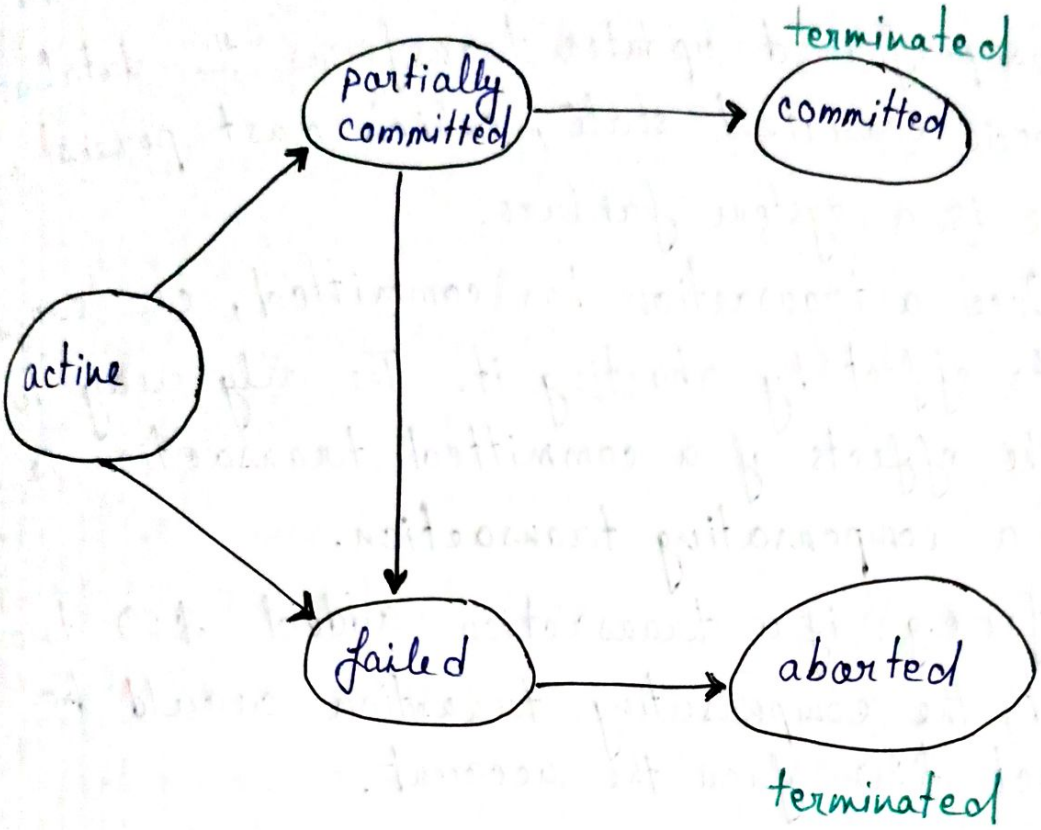


fig: State diagram of a transaction.

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts.

A transaction that completes its execution successfully is said to be committed. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effect by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction.

for e.g: if a transaction added $20 to an account, the compensating transaction would ~~be~~ subtract $20 from the account.

A transaction must be in one of the following states:

- Active: the initial state; the transaction stays in this state while it is executing.
- Partially committed: after the final statement has been executed.
- Failed, after the discovery that normal execution can no longer proceed.
- Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Committed, after success completion.

The state diagram corresponding to a transaction appears in fig. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. ~~When it finishes its~~ At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

(1) It can restart the transaction, but only if the transaction was aborted as a result of some h/w or s/w error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

② It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

- **Concurrent Execution:** Transaction-processing systems usually allow multiple transactions to run concurrently. There are two good reasons for allowing concurrency:

  ① Improved throughput and resource utilization

  ② Reduced waiting time.

Allowing multiple transactions to update data concurrently causes several complications with consistency of data.

Sometimes it is possible that even though individual transaction are satisfying the acid properties, the final status of the system will be inconsistent.

Concurrency Problems in Transactions:

1) Dirty Read Problem
2) Unrepeatable Read Problem
3) Lost Update Problem
4) Phantom Read Problem

1) Dirty read problem / Read-write Problem:

→ In this problem, the transaction reads a data item updated by another uncommitted transaction, this transaction may in future be aborted or failed.

→ The reading transactions end with incorrect results.

T_1	T_2
Read (A)	
write (A)	
	Read (A) ← Dirty Read (Reading from an uncommitted transaction)
	commit
Abort	

A = 10 | T_1: ~~11~~ | T_2: 11 | T_1: 20

2) Unrepeatable Read Problem

→ When a transaction tries to read a value of a data item ~~is~~ twice, and another transaction updates the data item in between, then the result of the two read operation of the first transaction will differ, this problem is called Unrepeatable read problem or Non-repeatable read problem.

T_1	T_2
Read (A)	
	Read (A)
	write (A)
Read (A)	

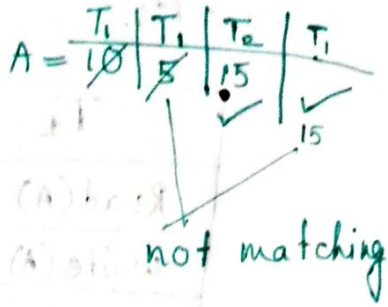
A = 10 | T_1: 10 | T_2: ~~10~~ | T_2: 12 | T_1: 12 ?

for same transaction, for same data item value is different

3) Lost update problem / Write - Write problem:

→ If there is any two write operation of different transaction on same data value, and between them there is no read operations, then the second write over-writes the write of previous transaction.

T_1	T_2
Read (A)	
Write (A)	
	Write (A)
	Commit
Commit	



4) Phantom read problem: It occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

T_1	T_2
Read (A)	
	Delete (A)
Read (A)	

- **Schedule:** When two or more transaction executed together or one after another then they can be bundled up into a higher unit of execution called schedule.

→ A schedule of n transaction $T_1, T_2, \ldots T_n$ is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule S.

→ However, schedule for a set of transaction must contain all the instruction of those transaction, and for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i.

- **Types of Schedule:**

1) **Serial Schedule:** A serial schedule consists of sequence instruction belonging to different transactions, where instructions belonging to one single transaction appear together. Before complete execution of one transaction another transaction cannot be started.

S

T_0	T_1
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

→ For a set of n transactions, there exist n! different valid serial schedules. Every serial schedule lead database into consistent state. Throughput of system is less.

2) Non-serial schedule: A schedule in which sequence of instructions of a transaction appear in the same order as they appear in individual transaction but the instructions may be interleaved with the instructions of different transactions i.e., concurrent execution of transactions takes place.

S

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

→ Total no. of schedules for n different $T_1, T_2, T_3, \ldots T_n$ where each transaction contains $n_1, n_2, n_3, \ldots n_4$ respectively

$$= \frac{(n_1 + n_2 + n_3 \cdots + n_n)!}{n_1!\ n_2!\ n_3! \cdots n_n!}$$

Hence, total no. of non-serial schedule

$$= \text{Total no. of schedules} - \text{no. of serial schedules}$$

$$= \frac{(n_1 + n_2 + n_3 + \dots + n_n)!}{n_1!\, n_2!\, n_3! \dots n_n!} - n!$$

- **Conclusion of Schedules:**

→ From the above discussion we understand that a serial schedule will always be consistent, so if somehow we are prone that a non-serial schedule will also have same effects as of a serial schedule then we will be sure that this particular non-serial schedule will also be consistent.

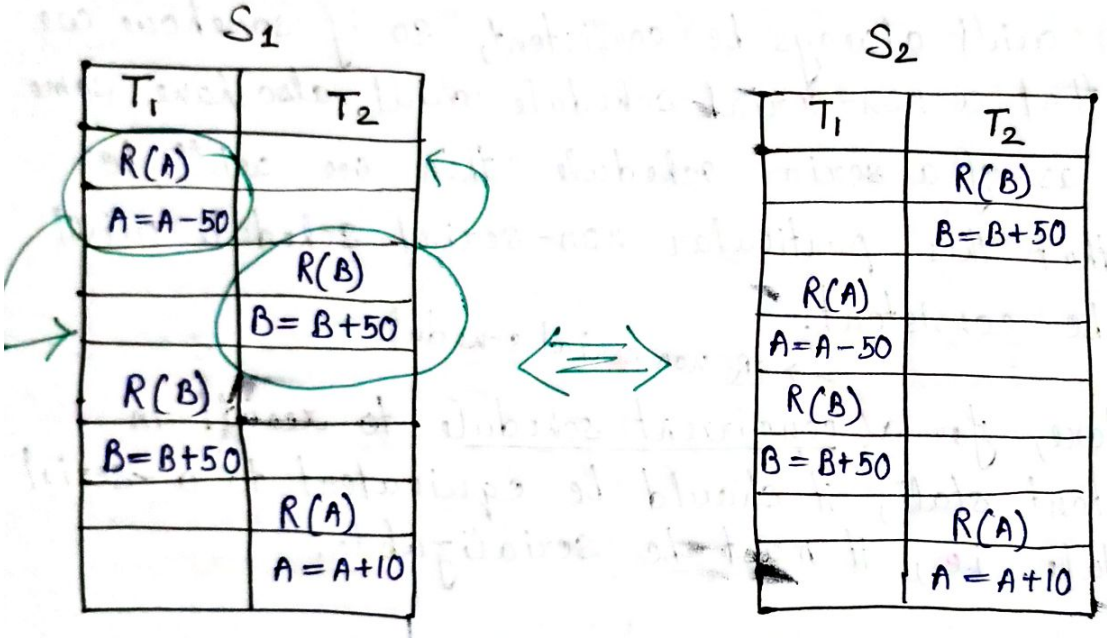
→ Therefore, for a concurrent schedule (OR non serial schedule) to result in consistent state, it should be equivalent to a serial schedule. i.e., it must be serializable.

- **Conflicting Instructions:** Let I and J be two consecutive instructions belonging to two different transactions T_i and T_j in a schedule S, the possible I and J instruction can be as:

I = Read(Q), J = Read(Q) → non-conflicting
I = Read(Q), J = Write(Q) → conflicting
I = Write(Q), J = Read(Q) → conflicting
I = Write(Q), J = Write(Q) → conflicting

→ So, the instructions I and J are said to be conflicting, if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

- Conflict equivalent: If one schedule can be converted to another schedule by swapping of non-conflicting instruction then they are called conflict equivalent schedule.



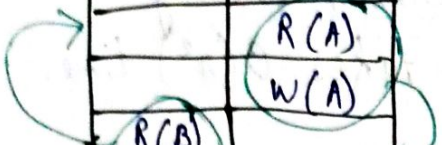
- Conflict Serializable: The schedules which are conflict equivalent to a serial schedule are called conflict serializable schedule.

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

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

S_1

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



S_2

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

- **Precedence graph for determining conflict serializability of a schedule:**

→ A precedence graph consists of a pair G (V, E)

→ V = set of vertices consisting of all the transactions participating in the schedule.

→ E = set of edges consists of all edges $T_i \rightarrow T_j$, for which one of the following conditions holds:

i) T_i executes write(Q) before T_j executes read(Q)

ii) T_i executes read(Q) before T_j executes write(Q)

iii) T_i executes write(Q) before T_j executes write(Q)

→ If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then in any serial schedule S' equivalent to S, T_i must appear before T_j.

→ If the precedence graph for S has no cycle, then schedule S is conflict serializable, else it is not. This cycle detection can be done by cycle detection algorithms, one of them based on depth first search takes $O(n^2)$ time.

→ The serializability order of transaction of equivalent serial schedule can be determined using topological order in a precedence graph.

Q. Let $r_i(z)$ and $w_i(z)$ denote read and write operations respectively on a data item z by a transaction T_i. Consider the following two schedules.

S_1: $r_1(x)$; $r_1(y)$; $r_2(x)$; $r_2(y)$; $w_2(y)$; $w_1(x)$

S_2: $r_1(x)$; $r_2(x)$; $r_2(y)$; $w_2(y)$; $r_1(y)$; $w_1(x)$

Determine S_1 and S_2 are conflicting serializable or not?

→ S_1

$r_1(x)$ S_1 $\not$

S_1

T_1	T_2
$r_1(x)$	
$r_1(y)$	
	$r_2(x)$
	$r_2(y)$
	$w_2(y)$
$w_1(x)$	

Cycle exists

∴, Conflict serializable.

S1 is

S_2

T_1	T_2
$r_1(x)$	
	$r_2(x)$
	$r_2(y)$
	$w_2(y)$
$r_1(y)$	
$w_1(y)$	

No cycle

∴ S_2 is not conflict serializable

S_2

T_1	T_2
$r_1(x)$	
	$r_2(x)$
	$r_2(y)$
	$w_2(y)$
$r_1(y)$	
$w_1(y)$	

No Cycle

∴ S_2 is not conflict serializable.

Practice Questions

Q. Conflict serializable or not?

1) $S: r_1(x);\ r_2(x);\ w_1(x);\ r_3(x);\ w_2(x)$

→ Not conflict serializable

2) $S: r_2(x);\ w_2(x);\ r_3(x);\ r_1(x);\ w_1(x)$

→ Conflict serializable

3) $S: r_2(x);\ r_1(x);\ w_2(x);\ r_3(x);\ w_1(x)$

→ Not conflict serializable

4) $S: r_3(x);\ r_2(x);\ r_1(x);\ w_2(x);\ w_1(x)$

→ Not conflict serializable.

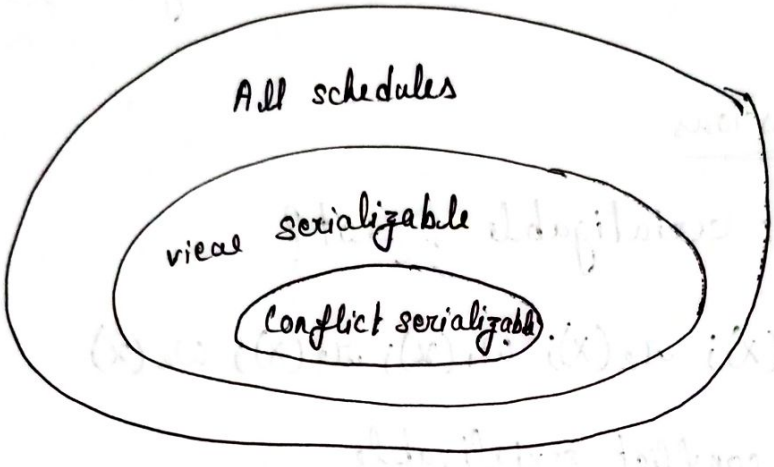
5) $S: r_1(x);\ r_2(y);\ r_3(y);\ w_2(y);\ w_1(x);\ w_3(x);\ r_2(x);\ w_2(x)$

→ Not conflict serializable.

- **View Serializable:**

→ If a schedule is not conflict serializable, still it can be consistent.

→ A weaker form of serializability called view serializability.



→ If a schedule is conflict serializable then it will also be view serializable, so we must check view serializability only if a schedule is not conflict serializability.

→ If a schedule is not conflict serializable then it must have atleast one blind write to be eligible for view serializable. and if does not contain any blind write then it can never be view serializable.

→ Blind Write: When a transaction performs write operation without reading.

e.g:

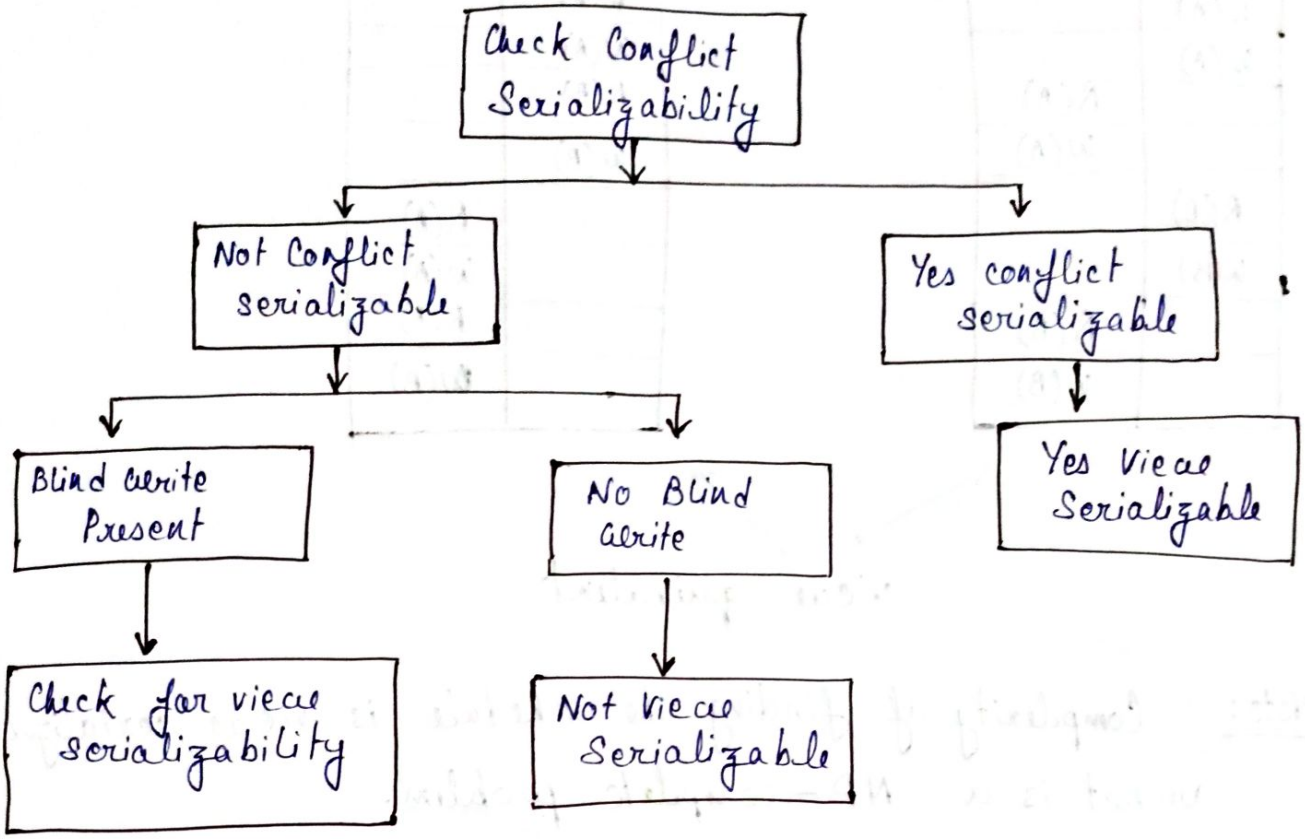
S

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

$W_3(x)$ is a blind write, as there is no read before write

$W_2(x)$ is not blind write, as read operation has been performed $R_2(x)$ before $W_2(x)$.

- Flowchart for finding serializability :



- View Equivalent: Two schedules S and S' are view equivalent, if they satisfy following conditions:

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

2) If a transaction T_i in schedule S reads any data item Q, which is updated by transaction T_j. then a transaction T_i must in schedule S' also read data item Q updated by transaction T_j in schedule S'.

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

S

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

S'

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

view equivalent

Note: Complexity of finding the schedule is view serializable or not is a NP-complete problem.

- View Serializable: A schedule S is a view serializable if it is view equivalent to a serial schedule.

- Algorithm to check View Serializability:

Step 1: If conflict serializable, end, otherwise continue. it is view serializable

Step 2: Check blind write present or not.

~~Step~~ If blind write present, continue. otherwise stop, schedule is not view serializable

Step 3: Draw edge from intial ~~write~~ read of T_i to initial write of T_j.

Step 4: Draw an edge from all transaction to Final write.

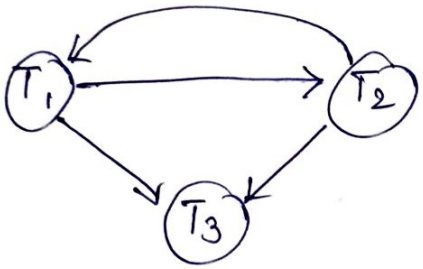
Step 5: If cycle does not exist then it is view serializable.

e.g: Check S_1 is view serializable or not?

S_1

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

→ Step 1: Check conflict serializable or not?



cycle exist not serializable.

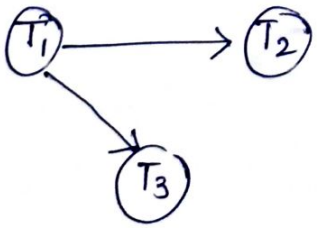
Step 2: Yes blind write present.

W(A) of T_2 is blind write

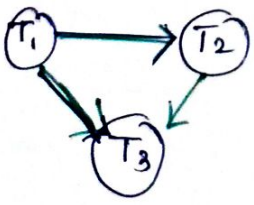
W(A) of T_2 is blind write.

Step 3: Initial read R(A) of T_1, initial write W(A) of T_2, initial write of W(A) of T_2.

∴

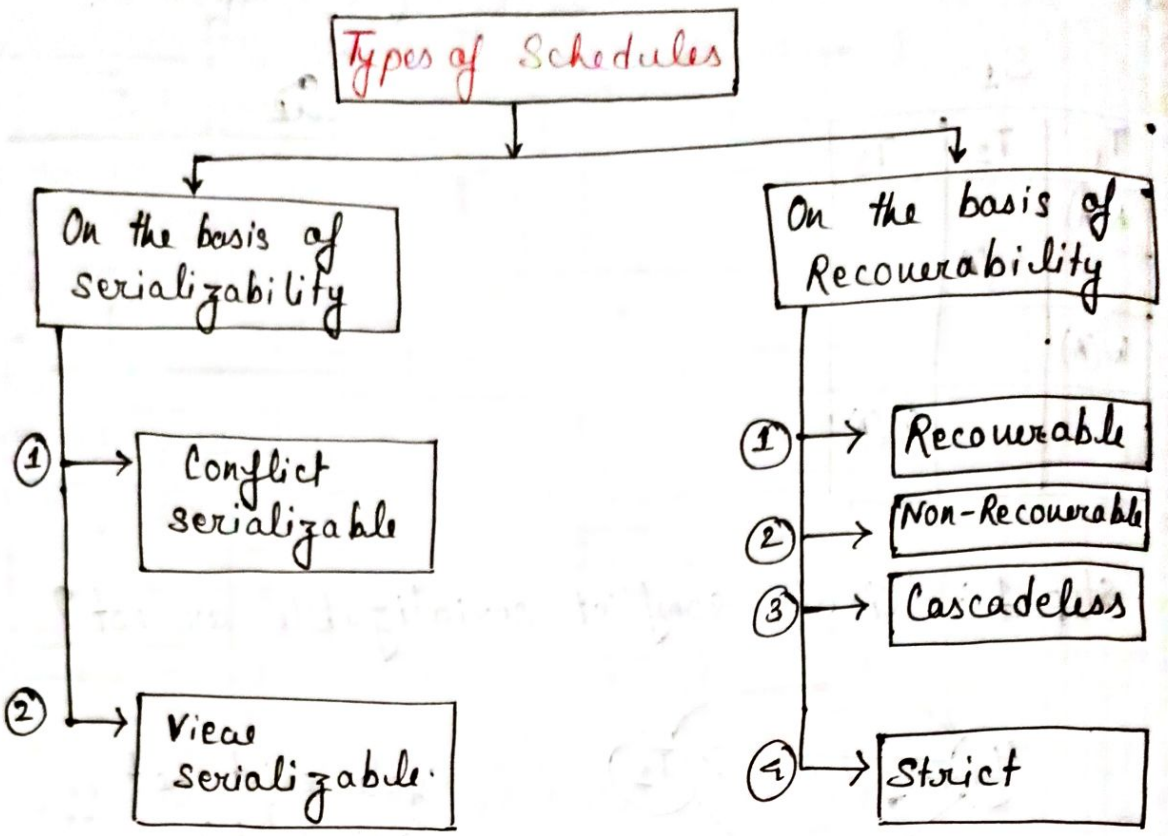


Step 4. Final write W(A) of T_3



Cycle does not exist

∴ S_1 is view serializable.



- **Non-Recoverable Schedule:**

→ A schedule in which for each pair of transaction T_i and T_j, such that if T_j reads a data item previously written by T_i, then the commit or abort operation of T_j appears before T_i. Such a schedule is called Non-Recoverable schedule.

→ In simple words, if T_j does dirty read from T_i then T_i should perform commit or abort first. to be recoverable.

e.g:

S

	T_1	T_2
t_1	R(x)	
t_2	W(x)	
t_3		R(x)
t_4		C
t_5	C	

~~Read~~ dirty read

* Dirty Read: Reading from an uncommited transaction.

from T_1

In the above example, T_2 has done dirty read and immediately after that T_2 committed also. But due to some reason at time t_4 T_1 rollba-cks and undo all the operations and redo again all the operations. Now the problem is T_2 cannot recover its mistake because it already performed commit operation.

- Recoverable Schedule:

→ A schedule in which for each pair of transaction T_i and T_j, such that if T_j reads a data item previously written by T_i, then the commit or abort of T_i must appear before T_j, such a schedule is called Recoverable schedule.

e.g:

S

	T_1	T_2
t_1	R(x)	
t_2	W(x)	
t_3		R(x)
t_4	C	
t_5		C

dirty read

→ In the above example, dirty read order and commit order is same. Hence, schedule is recoverable. Even though T_2 has done dirty read, if T_1 tries to rollback at time t_3, T_2 ~~will also~~ can also rollback at time t_4.

Q. Following schedule is recoverable or not?

T_1	T_2	T_3	T_4
	R(x)		
		W(x)	
		C	
W(x)			
C			
	W(Y)		
	R(Z)		
	C		
			R(x)
			R(Y)
			C

→ No dirty read, hence, recoverable.

* dirty read: Reading from an uncommitted transaction.

Q Consider the schedule:

S: $r_2(x)$; $r_1(x)$; $r_2(y)$; $w_1(x)$; $r_1(y)$; $w_2(x)$; a_1; a_2

a_1, a_2 are abort operation.
Check whether the schedule is recoverable or not?

→

S

T_1	T_2
	$r_2(x)$
$r_1(x)$	
	$r_2(y)$
$w_1(x)$	
$r_1(y)$	
	$w_2(x)$
a_1	
	a_2

No dirty read, hence recoverable.

Q. $S: R_1(x); W_1(x); R_1(y); R_2(x); W_2(x); C_2; C_1$. Recoverable or not?

→

S

T_1	T_2
$R_1(x)$	
$W_1(x)$	
$R_1(y)$	
	$R_2(x)$
	$W_2(x)$
	C_2
C_1	

T_2 does dirty read from T_1. and committed before T_1. ∴ Non-recoverable.

dirty read order and commit order not same, Hence, non-recoverable.

Practice Questions:

Check recoverable or irrecoverable

1) $S: R_2(x); W_2(x); R_1(y); R_1(x); W_2(x); C_2; C_1$

→ Recoverable

2) $S: R_2(x); W_2(x); R_3(y); R_1(x); R_1(y); W_1(x); W_3(y); R_3(x); R_1(y); C_3; C_2; C_1$

→ Irrecoverable / Non-recoverable.

• Cascading Rollback:

→ It is a phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback. Even if the schedule is recoverable, the commit of transaction may lead lot of transaction to rollback.

→ Cascading rollback is undesirable, since it leads to undoing of a significant amount of work. Uncommitt -d reads are not allowed in cascadeless schedules.

S

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

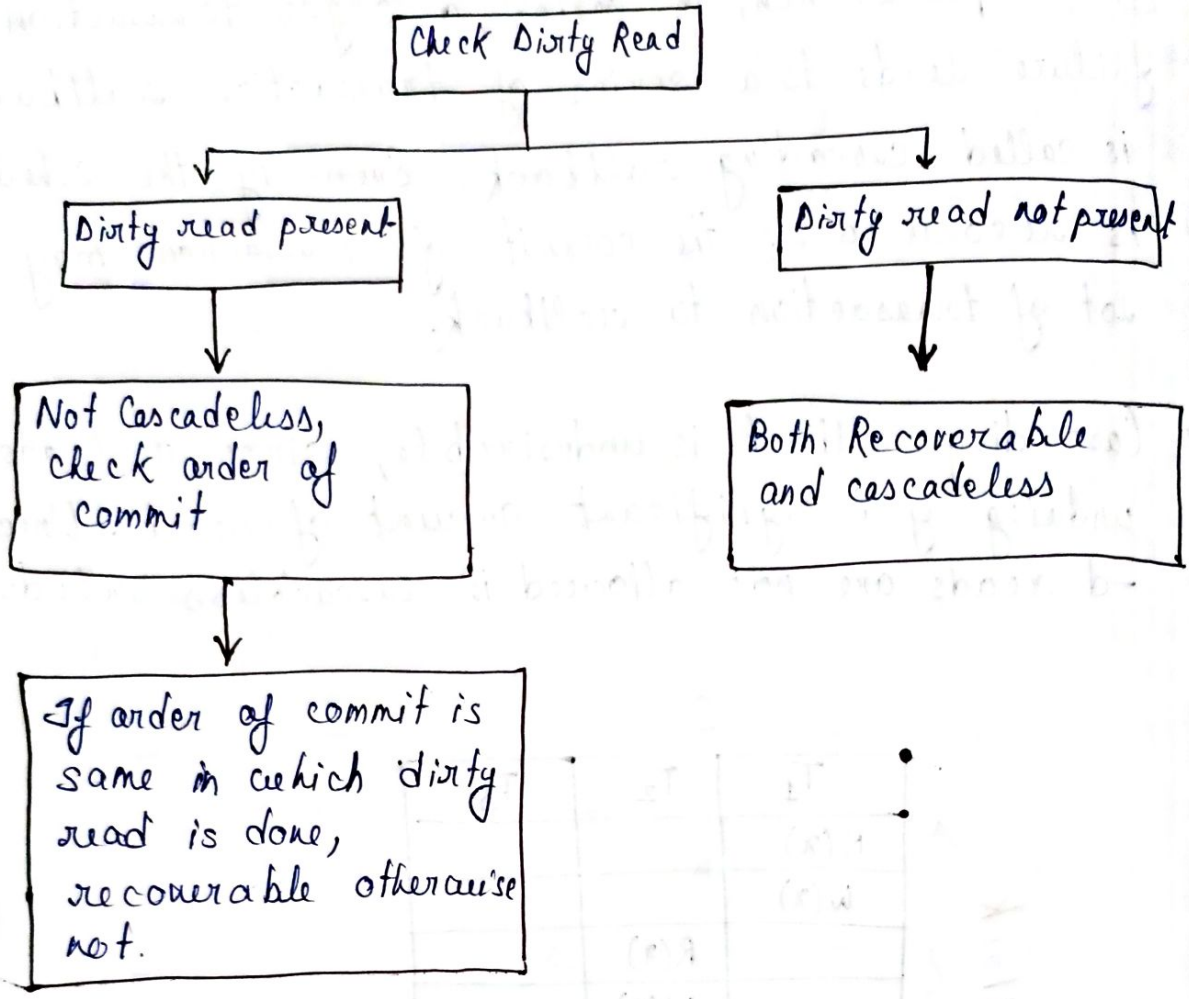
Rollback

• Cascadeless Schedule:

→ To avoid cascading rollback, cascadeless schedule are used.

→ A schedule in which for each pair of transactions T_i and T_j such that if T_j reads a data item previous -ly written by T_i then the commit or abort of T_i must appear before read operation of T_j. Such a schedule is called cascadeless schedule.

- Flow Chart to check recoverability of a schedule:



• Strict Schedule :

→ A schedule in which for each pair of transactions T_i and T_j such that if T_j reads a data item previously written by T_i, then the commit or abort of T_i must appear before read and write operation of T_j.

S_1

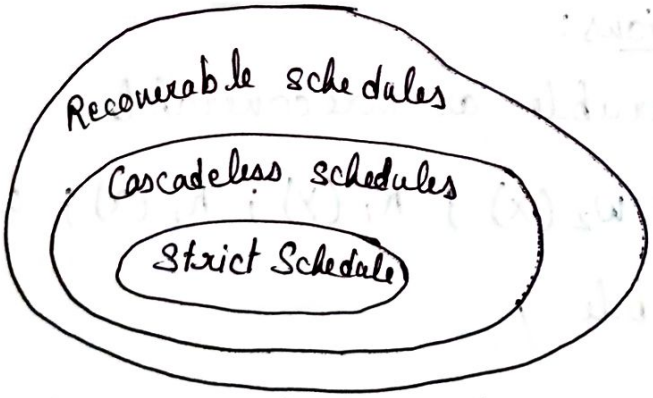
T_1	T_2
R(a)	
W(a)	
	W(a)
C	
	R(a)
	C

not strict ✗

S_2

T_1	T_2
R(a)	
W(a)	
C	
	W(a)
	R(a)
	C

strict ✓



- **Concurrency Control Protocol:**

Till now we have seen that if there is a schedule, how to check whether it will work correctly or not. will it maintain consistency or not.

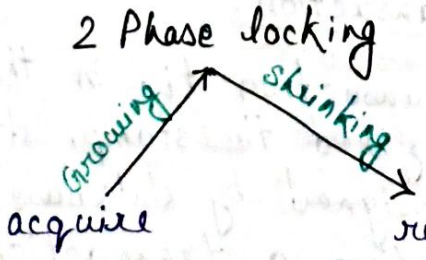
Now let us study some protocols which will guarantee ~~how to design those schedule~~ that schedule will be consistant by ensuring the conflict serializability and other properties.

We understood conflict serializability will ensure consistency. Now how to approach / design a schedule which will guarantee conflict / view serializability.

There are three popular approaches:

1) Time stamping based method: Where before entering the system, a specific order is decided among the transaction, so in case of a clash we can decide which one to allow and which to stop.

2) Lock based method: Where we ask a transaction to first lock a data item before using it. so that no different transaction can use a data at the same time, removing any possibility of conflict.



i) Basic 2PL
ii) Conservative 2PL
iii) Rigorous 2PL
iv) Strict 2PL

3) Validation based :

Majarity of transactions are read only transactions, the rate of conflicts among the transac -tion may be low, thus many of transaction, if executed without the supervision of a concurrency contral scheme, would nevertheless leave the system in a consistent stati.

- Goals of Protocal :

1) Concurrency should be as high as possible, as this the ultimate goal.

2) The time taken by a transaction should be less

3) Desirable properties satisfied by the protocal (ACID), conflict / view serializability etc).

4) Easy to understand and implement.

- Time Stamp Ordering Protocal :

→ Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp), in case of any conflict during the execution order can be decided using the time stamp.

(1) Time stamp for transaction :

→ With each transaction t_i, in the system, we associati a unique fixed timestamp, denoted by TS(t_i). This timestamp is assigned by database system to a transaction at the time a transaction enters into the system. If a transaction has been assigned a time stamp TS(t_i) and a new transaction t_j enters

into the system with a timestamp $TS(t_j)$, then always:

$$TS(t_i) < TS(t_j)$$

Note: (1) Time stamp of a transaction remain fixed throughout the execution.

(2) It is unique means no two transaction can have the same time stamp.

→ The reason only we called time-stamp but not stamp, because for stamping we use the value of the system clock as stamp.

→ The time stamp of the transaction also determines the serializability order. Thus if $TS(t_i) < TS(t_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction t_i appears before transaction t_j.

(2) Time stamp with data item:

In order to assure such schema, the protocol maintains for each data item Q, two timestamp values:

- W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
- R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.
- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

→ Suppose a transaction T_i requests a read (Q):

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

Ex

TS $(T_x) = 10$	TS $(T_i) = 5$
	R(Q) ✓
W(Q)	
	R(Q) ✗

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

TS $(T_x) = 5$	TS $(T_i) = 10$
W(Q)	
	R(Q) ✓ request accepted

TS $(T_x) = 10$	TS $(T_i) = 10$
	W(Q)
	R(Q) ✓ request accepted.

→ Suppose that transaction T_i issues write(Q):

1) $TS(T_i) < R\text{-timestamp}(Q)$, rejected

2) $TS(T_i) < W\text{-timestamp}(Q)$, rejected

3) $TS(T_i) \geq R\text{-timestamp}(Q)$, accepted

4) $TS(T_i) \geq W\text{-timestamp}(Q)$, accepted

Properties of timestamp protocal:

1) It does not gurantee recoverability as there is a chance of dirty read.

2) It may cause starvation.

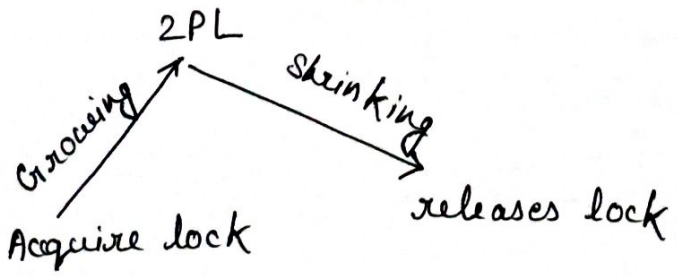
3) It is relatively slow as before executing every instruction we have to check conditions.

4) Time stamping protocal ensure that the schedule designed through this protocal will always be conflict serializable.

- ## 2 Phase locking protocal:

The two-phase locking (2PL) protocal is a concurrency contral technique used in database management system to ensure data consistency and avoid conflicts among concurrent transactions.

The 2PL ~~has~~ follows two rules:



In growing phase, A transaction can acquire new lock during its execution as long as it doesn't release any locks.

In shrinking phase, A transaction can release locks during its execution but it cannot acquire any new locks after releasing any.

→ It ensures conflict serializability.

→ It ensures recoverability as it avoids dirty read.