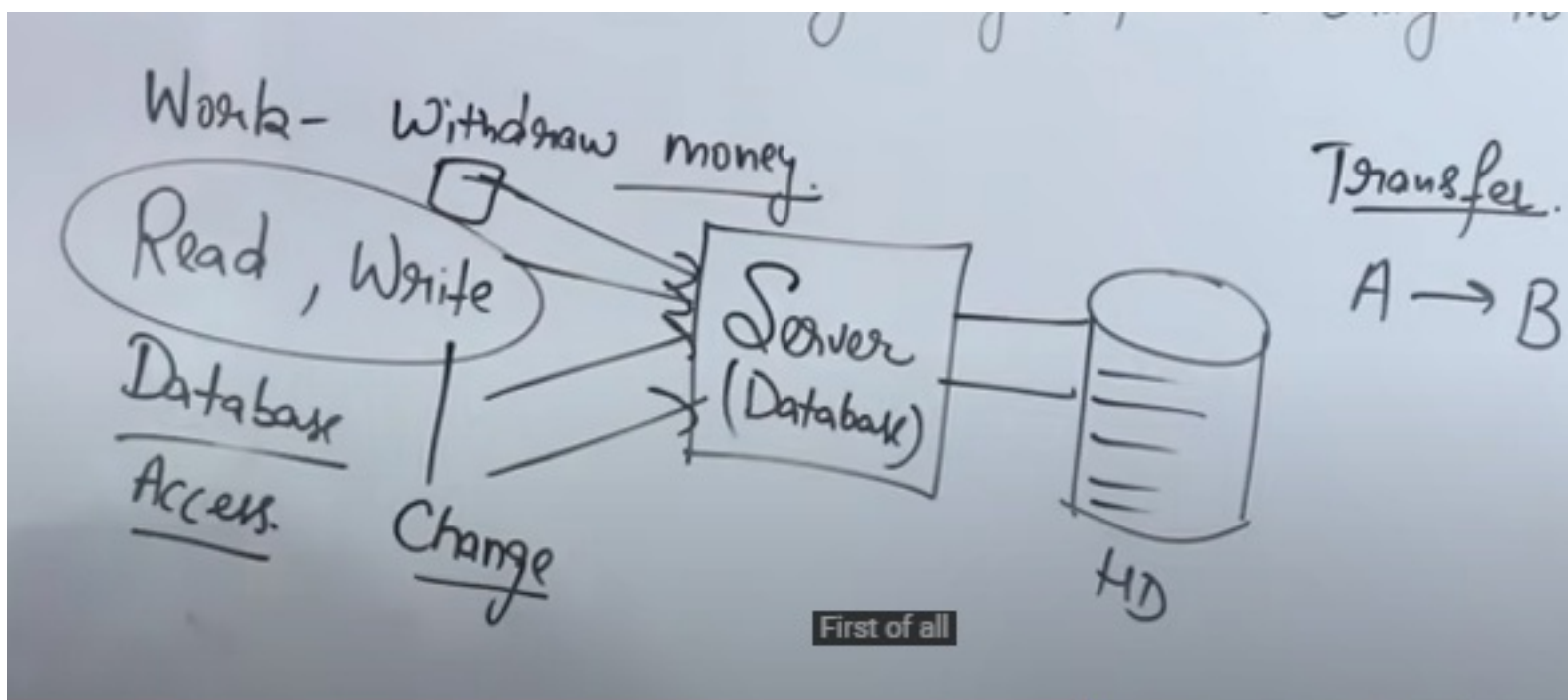




## DBMS NOTES – II

### TRANSACTION CONCURRENCY

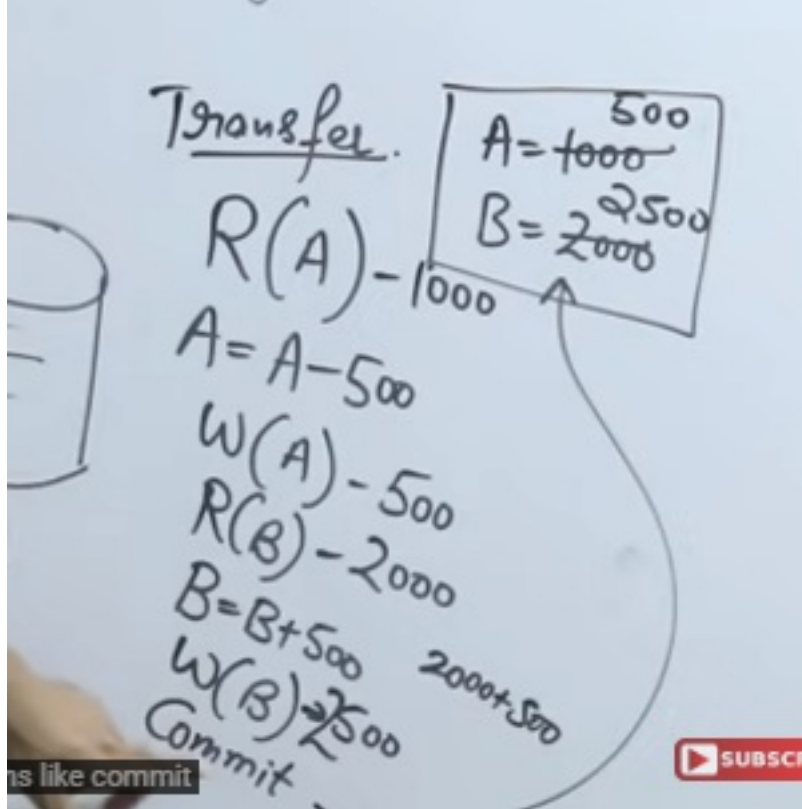
It is a set of operations used to perform a logical unit of work.  
A transaction generally represents change in database.



Lets say I want to transfer money from a to b

So first of all I would have to read the data of a

So following processes will occur in the RAM :



Reading hogi pehle a k account k baare mei

Then updation hoga if 500 has to be transferred

Commit is used to save all the changes in data in the database



## ACID properties

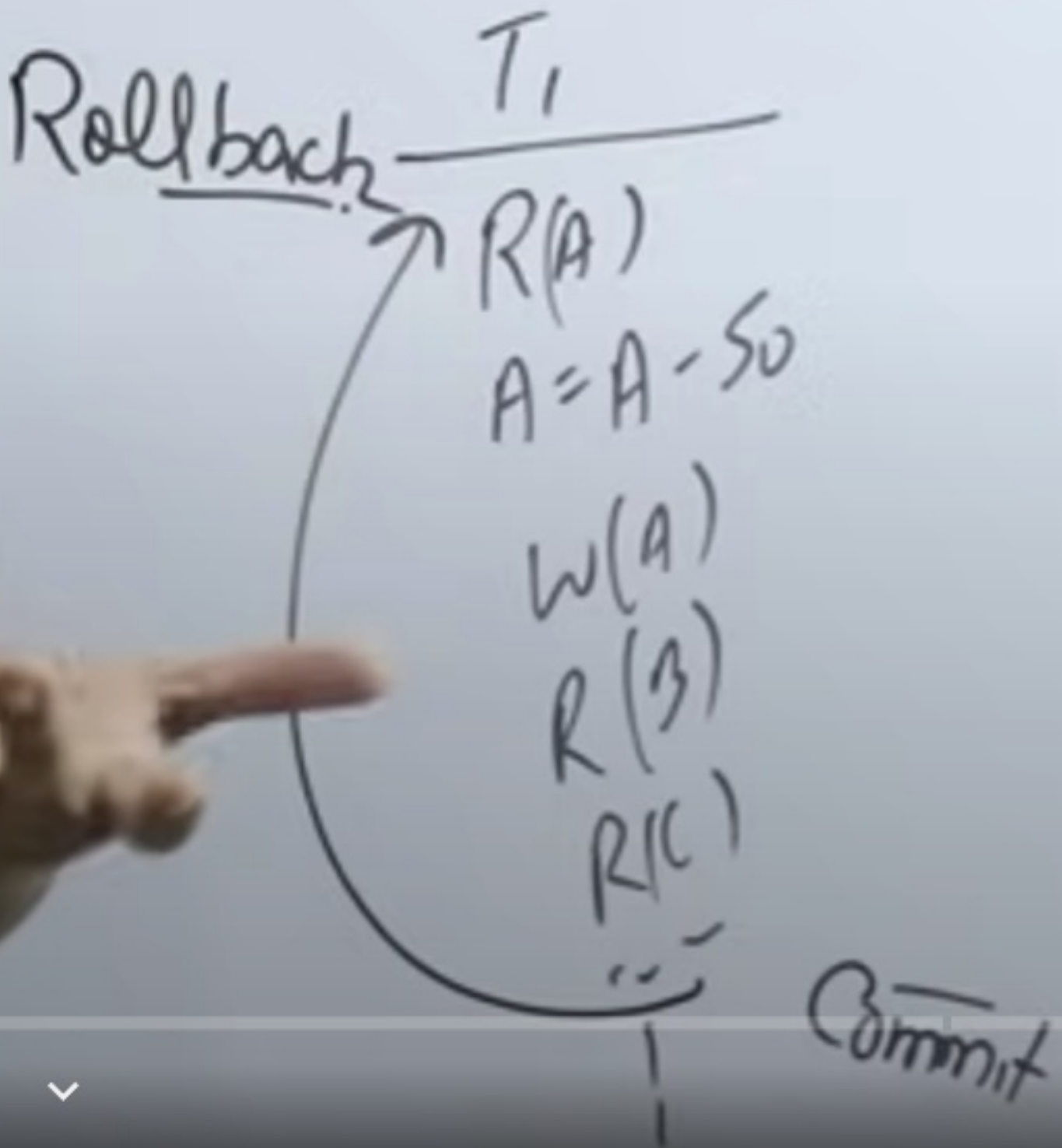


### Atomicity Consistency Isolation Durability

- Conceptual hai
- **ATOMICITY**

Either all the operations will be executed Or none

Eg :

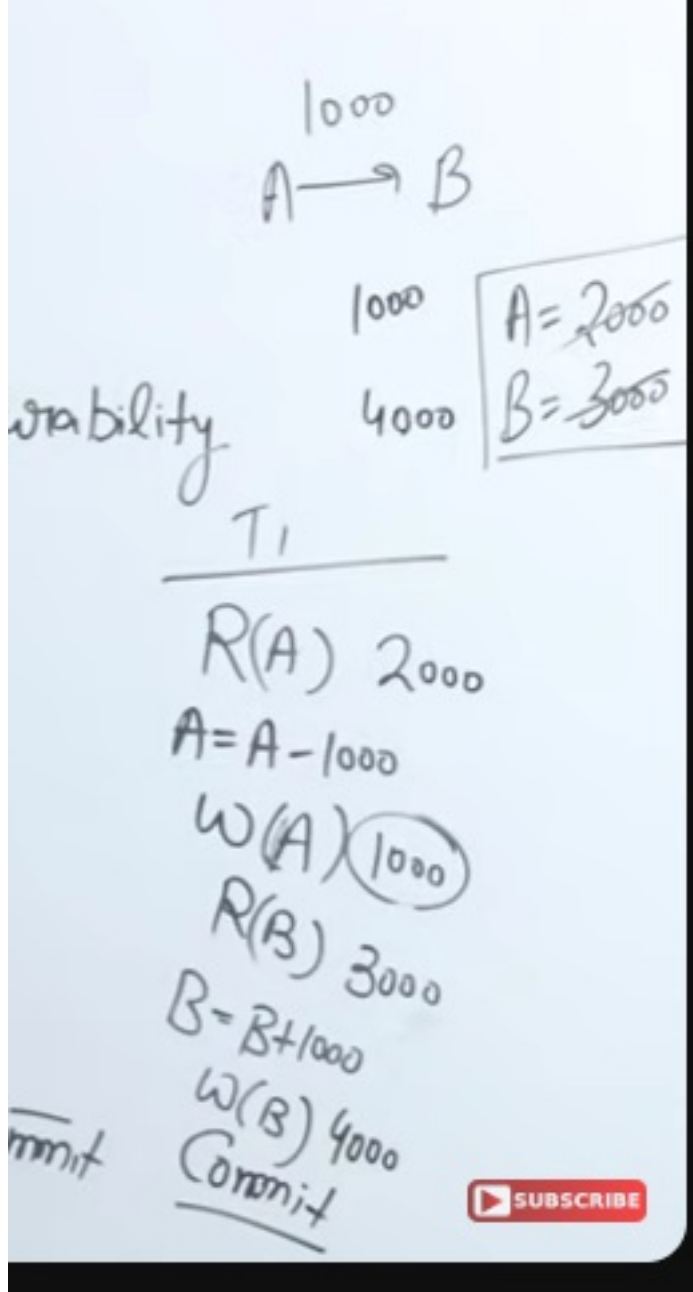


In  $T_1$  transaction if before commit an error occurs then the transaction 'rolls back' such that no operation is executed

A failed transaction can never be resumed it can only restart.

- **CONSISTENCY**

Before transaction starts and after it is completed, the sum of total money should be same.



Eg :

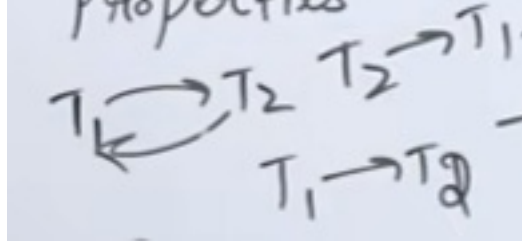


So basically before and after the transaction the total money remained same i.e 5000 ( $1000+4000 = 2000+3000$ )

- ISOLATION (imp)**

Parallel schedule ko serial schedule mei convert krna chaahte

Serial schedule is always consistent isliye it is preferred.



will talk about it more



- **DURABILITY**

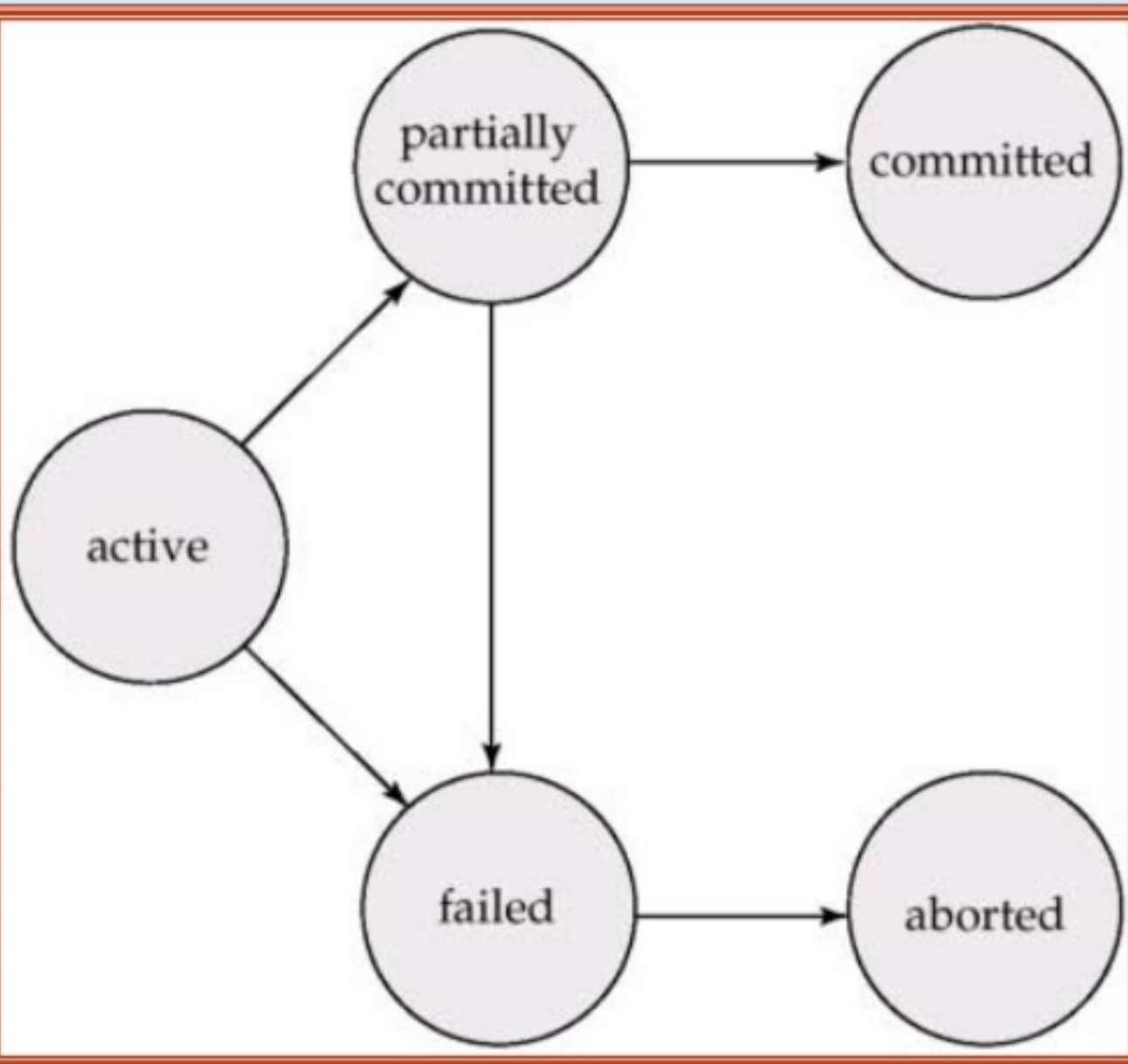
All changes should be permanent

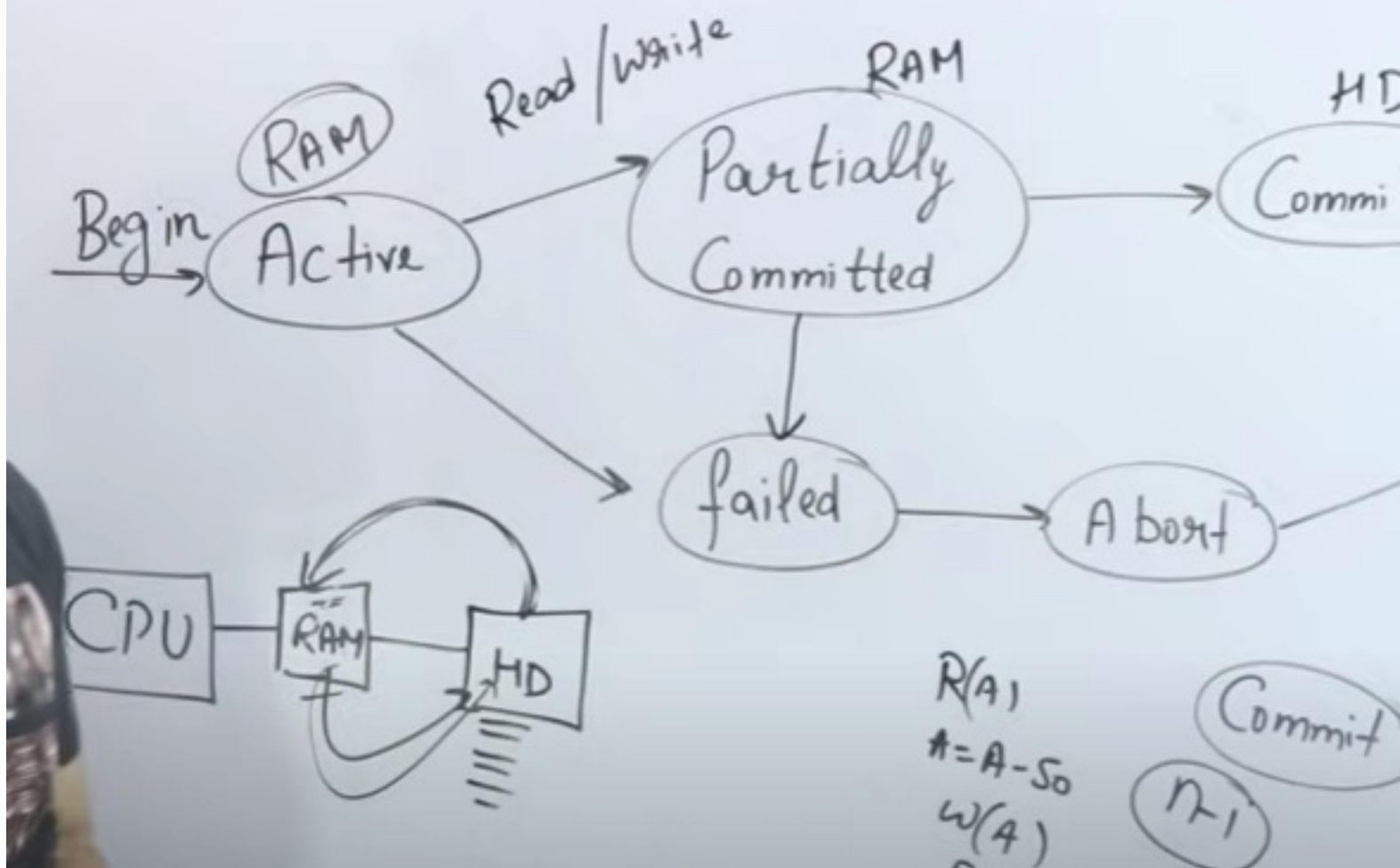
Isliye commit krne k baad hard disk mei save hota hai data jo ki permanently rehta.

## **TRANSACTION STATES**

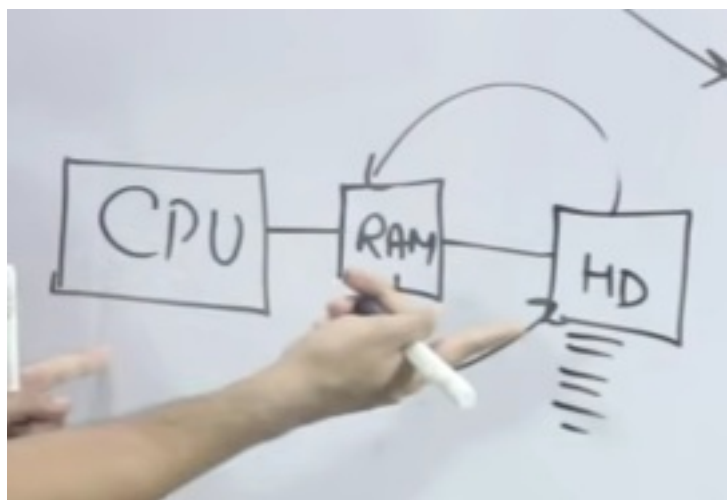


**active , partially committed , failed , committed , aborted**





- When we start executing a particular transaction it gets in the active state and goes to ram



- Partially Committed means all the operations have been performed just one is left i.e is COMMIT. RAM mei cheezein horha hoti
- Committed tab hota jab sab execute hojata sab including commit and the data is stored in hard disk
- Terminated : all the shared resources are freed
- Failed : beech mei fail hogya
- Abort : Kill / Restart

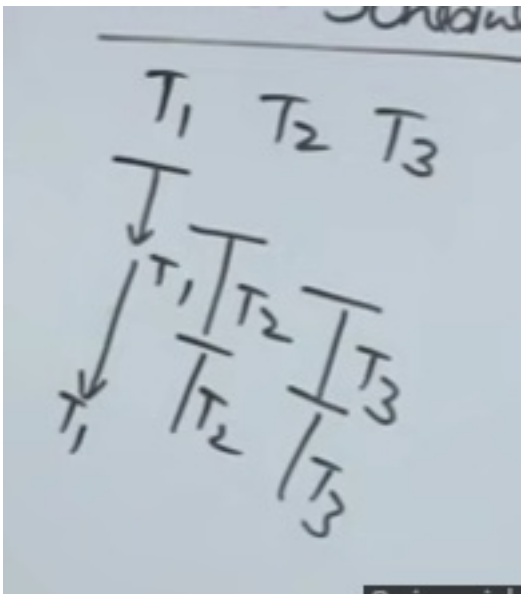
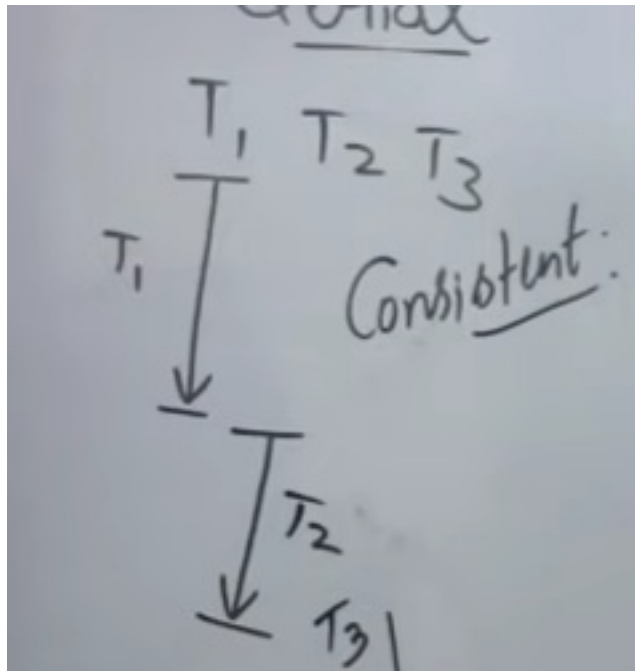
 **SCHEDULE**

**serial VS parallel**

it is chronological execution of multiple transactions.

Serial Parallel

One transaction is fully committed then only Multiple transactions can start together and another one begins get parallely executed.







Waiting time hojaata kaafi. Throughput increases.

Throughput decreases (number of transactions Nowadays used very much Executed per unit time)



**CONCURRENT EXECUTION**

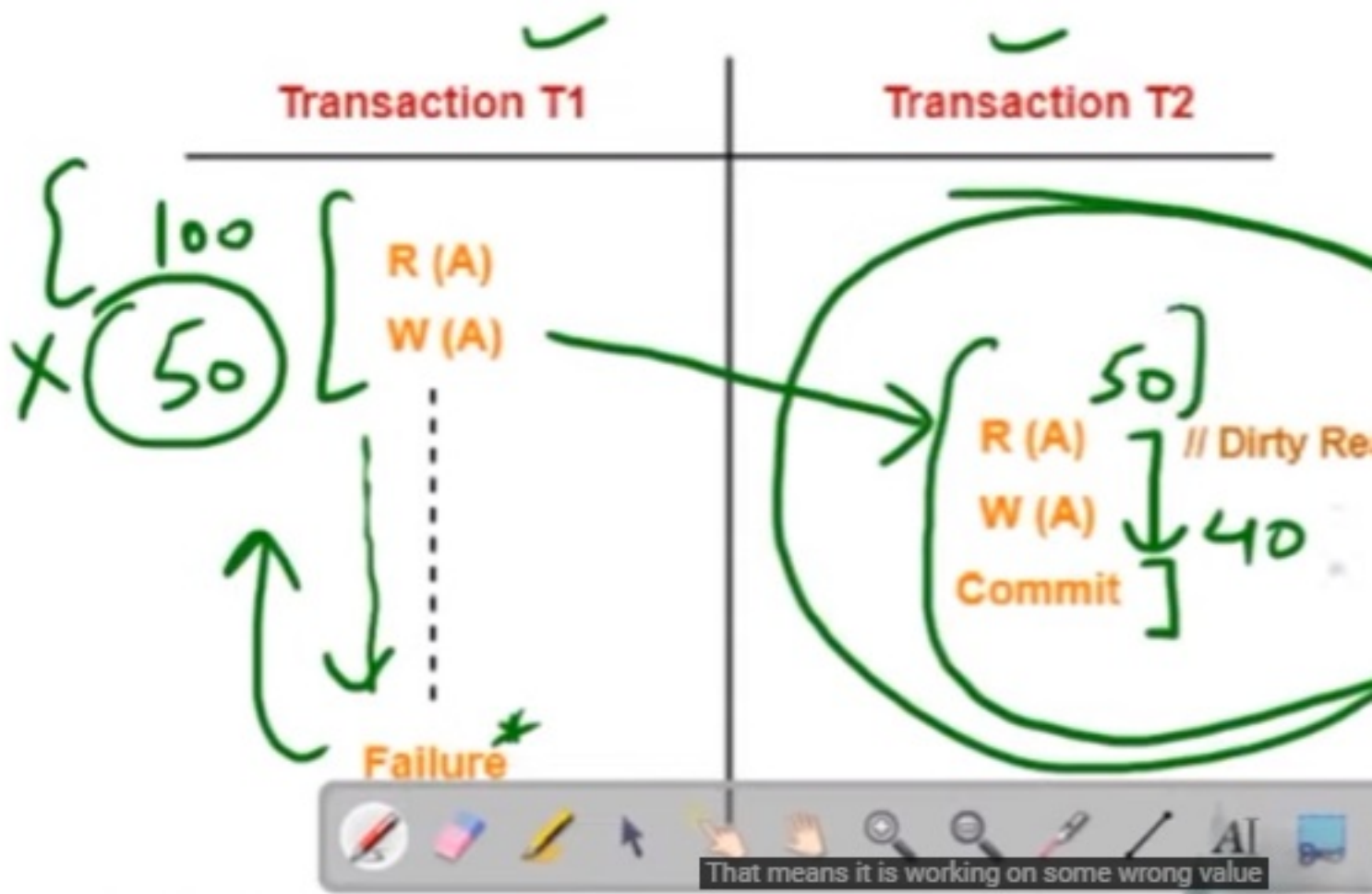


**CONCURRENCY** : when multiple transactions run at a single time (parallel scheduling).

Various problems occur due to concurrency :

- Dirty read
- Incorrect summary
- Lost update
- Unrepeatable read
- Phantom read

# Dirty Read or Uncommitted Read



Important sirf write/read hi hai



**Write-read conflict(dirty read problem)**



# Write-Read Conflict (Dirty Read Problem)

$A = 70$

S

W → R

$T_1$   $T_2$

70 R(A)

70-50  
20 A = A - 50;

30 W(A)

X

R(A) 20 Dirty Read

A = A \* 2; 40

W(A) 40  
Commit

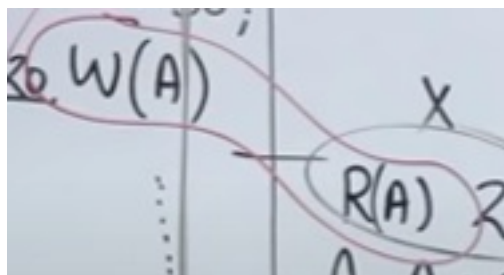
\* R(B)

\* W(B)

Time

But this transaction only failed

The main problem was write k baad read aana



This concept will be used in serializability



Read-Write conflict (unrepeatable read)

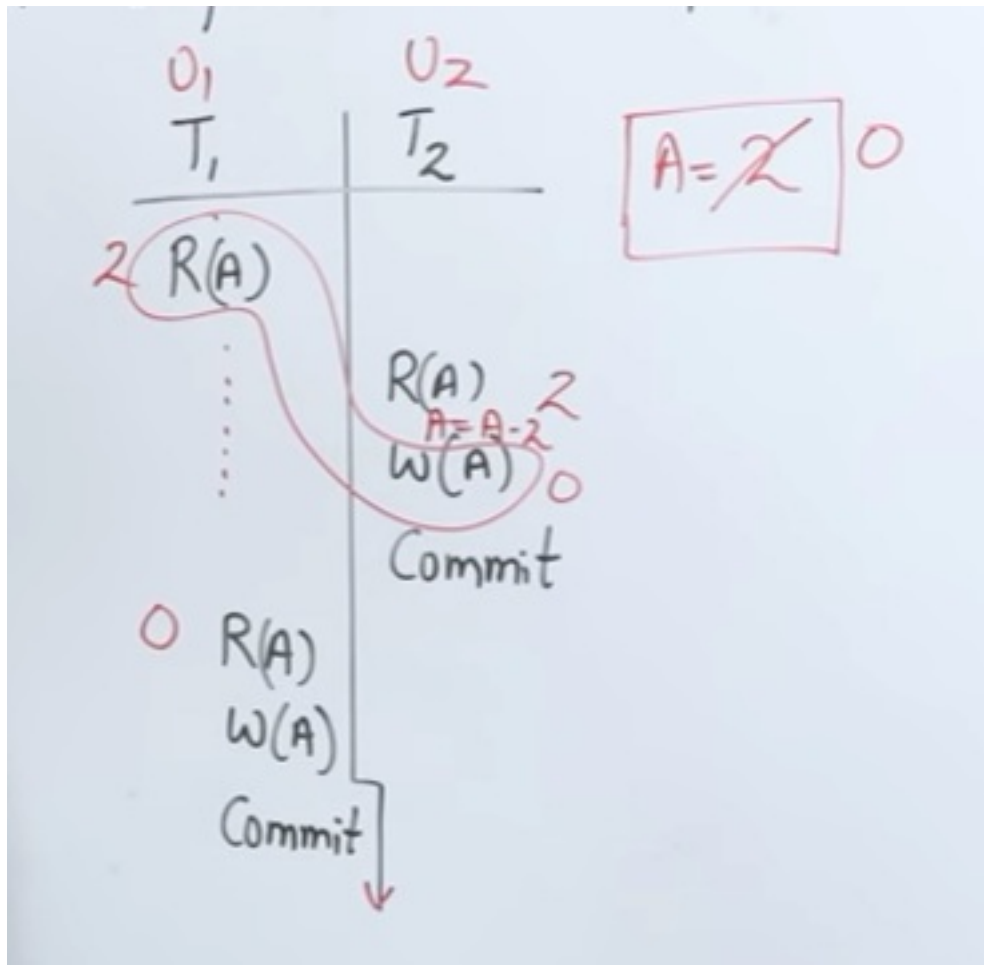


Read = W(A)

Same Data

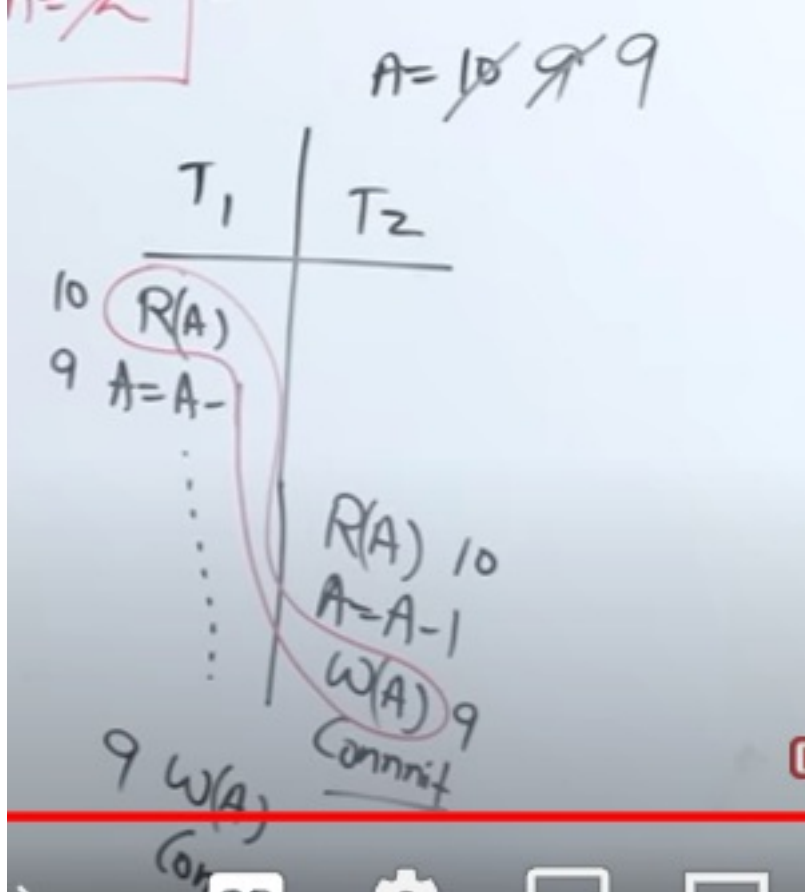
$R(A)$	$R(A)$
$R(A)$	$W(A)$
$W(A)$	$R(A)$
$W(A)$	$W(A)$

Problem occurs only in 3 cases when write is involved



Read – write conflict occurred and  $t_1$  read wrong value so it had to be aborted

Library example :



2 books got issued but database has 9 books instead of 8



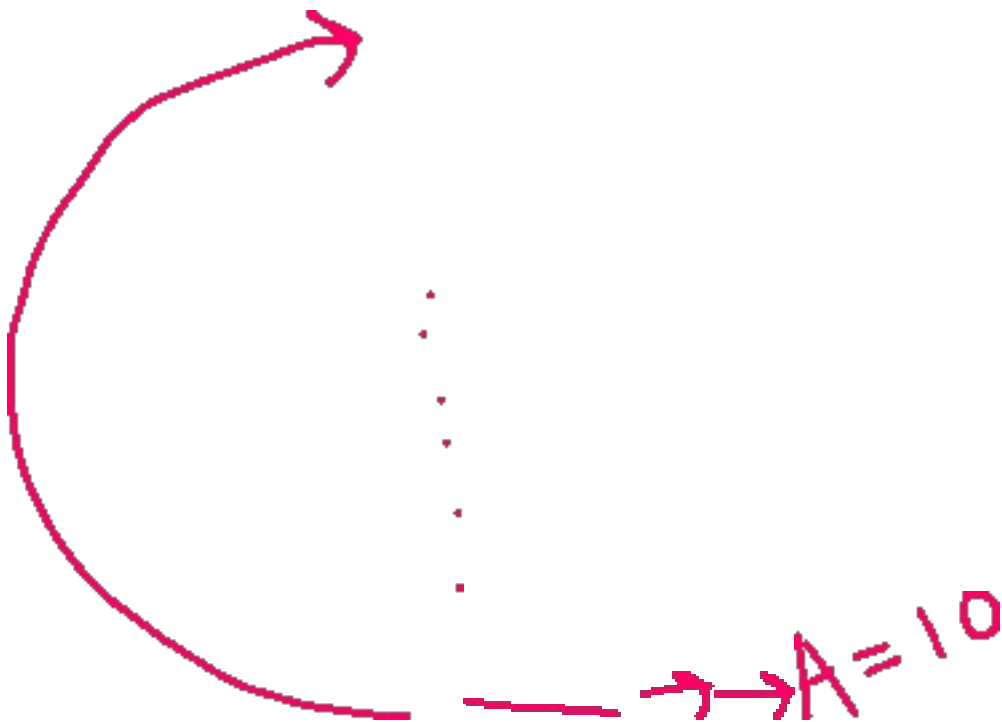
**irrecoverable VS recoverable schedules**

while talking of schedules we talk about serializability and recoverability

**IRRECOVERABLE SCHEDULE**

S

$T_1$	$T_2$
$R(A)$	
$A = A - 5$	
$W(A)$	
	$R(A)$
	$A = A - 2$
	$W(A)$
	Commit
$R(B)$	
* fail	



Let initially value of A be 10

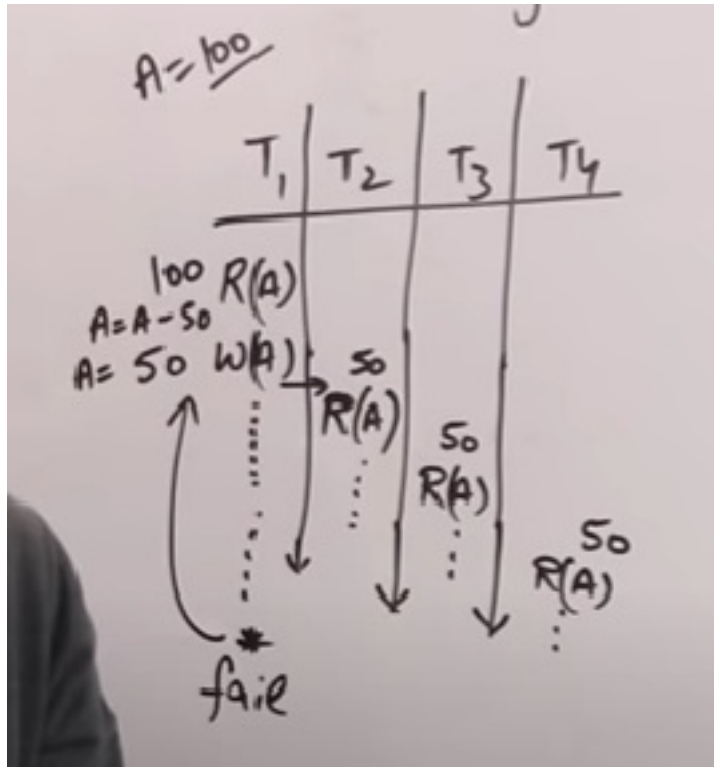
Jaise hi  $T_1$  transaction fails then as per atomicity property ,  $T_1$  will rollback

After it is rolled back , A's value will be updated to 10 again

**BUT NOW** the changes made by  $T_2$  are lost forever , they cant be RECOVERED

## cascading schedule VS cascadeless schedules

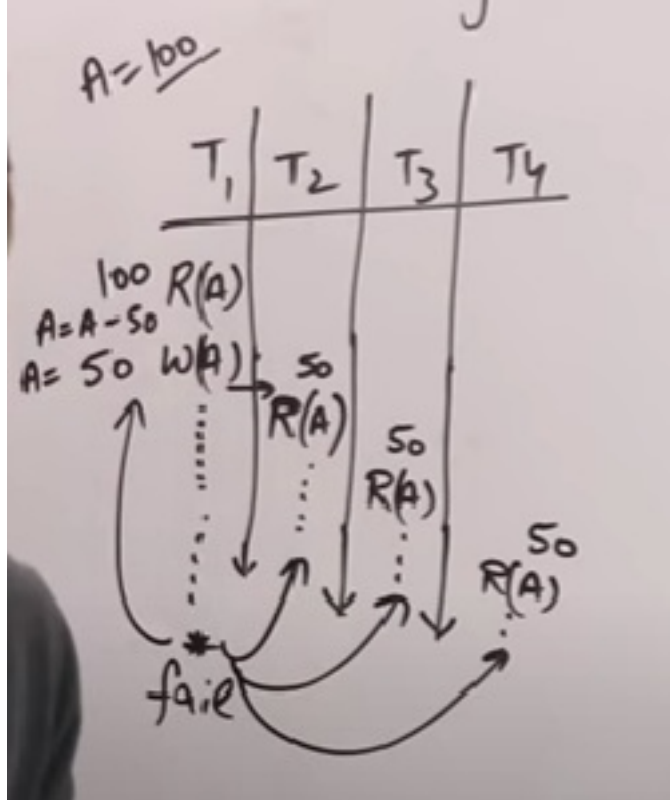
cascading : due to occurrence of one event , multiple events are automatically occurring.



As  $T_2, T_3, T_4$  are working on the invalid value of  $A$ , they have to be aborted as well !

This is called cascading





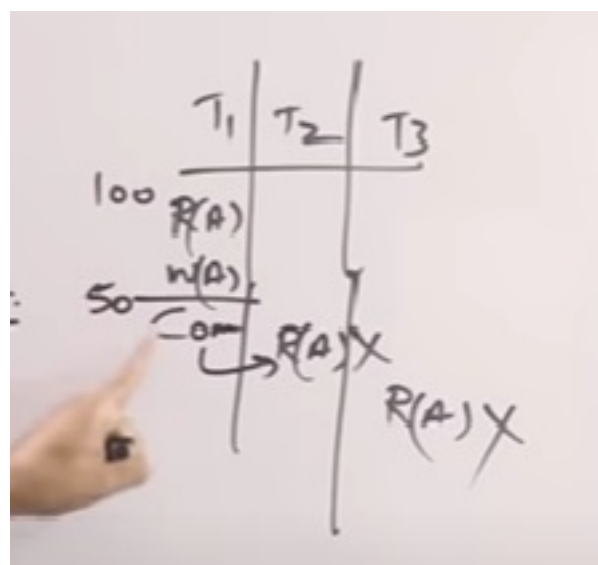
Ek  $t_1$  abort hua to usne baaki sbako bhi kr dia

Disadvantage : cpu utilisation nhi kia proper, cpu cycle waste, degraded performance

CASCADELESS SCHEDULE ?

How to remove cascading

Until the  $T_1$  transaction gets committed or aborted , no read(a) would be performed in the subsequent transactions. They can only be read after  $T_1$  is committed.



$R(A)$ $W(A)$	$R(A)$
------------------	--------

Cascading :

$R(A)$ $W(A)$ Com	$R(A)$
-------------------------	--------

Cascadeless :

Cascadeless mei write write problem aati hai. Cascadeless sirf write k baad read aane pe read ko nhi chalata but write k baad write mei usme dikkat aati

$A = 100$

$T_1$	$T_2$
$100 R(A)$ $A = A - 10$ $90 W(A)$ $R(A) \times$	$100 R(A)$ $A = A - 20$ $W(A) 80$

Diagram showing a write-read conflict in  $T_1$  (marked with an X) and a write-write conflict in  $T_2$  (marked with a star and a dashed arrow pointing to the value 80).

Write k baad write strict recoverable batata hai



**SERIALIZABILITY**

Can a schedule become serializable or not?

S	
T <sub>1</sub>	T <sub>2</sub>
$\begin{cases} R(A) \\ W(A) \end{cases}$	

$\begin{cases} R(A) \\ W(A) \end{cases}$



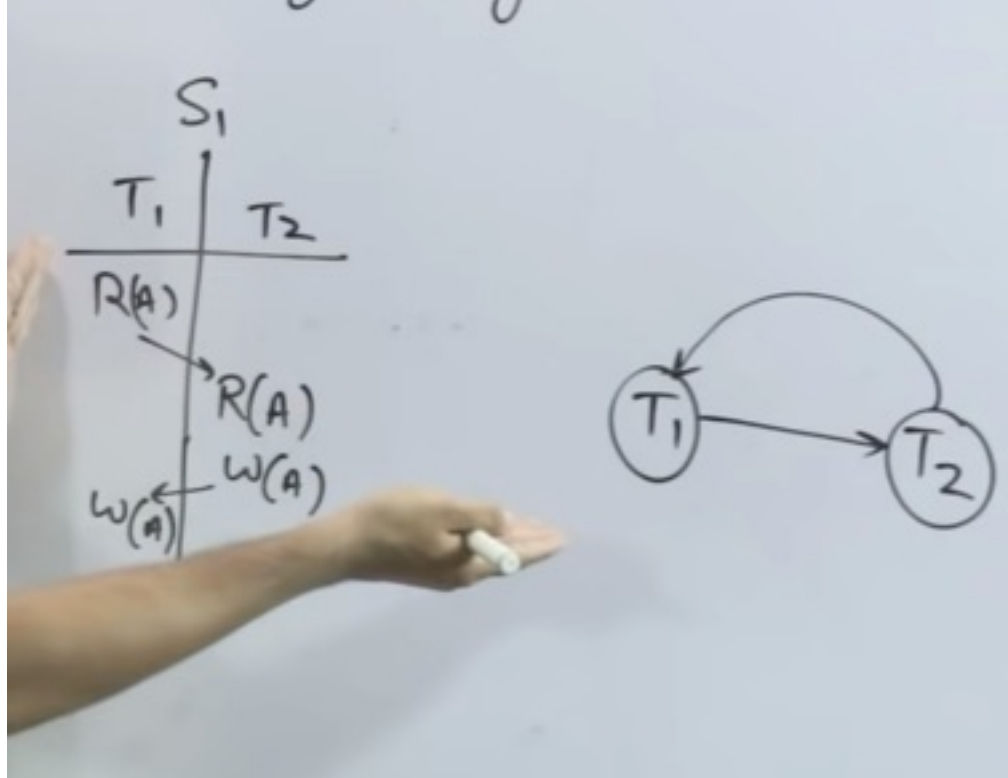
S	
T <sub>1</sub>	T <sub>2</sub>
	$\begin{cases} R(A) \\ W(A) \end{cases}$

$\begin{cases} R(A) \\ W(A) \end{cases}$



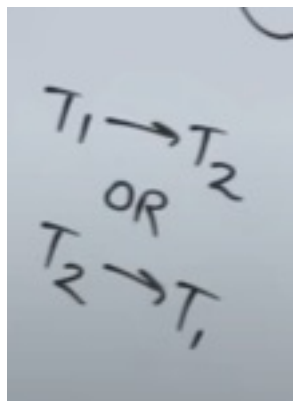
Both the above schedule are serial , so they don't cant be serialized ,obviously only parallel can be

You just need to find a serial clone of a parallel schedule.



Parallel schedules have a loop

To make the above schedule serial we can either:



Serializability < Conflict View

2 types of serializability : Conflict and View

Another eg :

$S$

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

$T_1 \rightarrow T_2 \rightarrow T_3$

$T_1 \rightarrow T_3 \rightarrow T_2$

$T_2 \rightarrow T_3 \rightarrow T_1$

$T_2 \rightarrow T_1 \rightarrow T_3$

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$



**conflict equivalent**

we would need to check whether 2 schedules are conflict equivalent or not

R(A)	R(A)	Non Conflict Pairs
R(A)	W(A)	
W(A)	R(A)	Conflict Pairs
W(A)	W(A)	
R(B)	R(A)	Non Conflict
W(B)	R(A)	
R(B)	W(A)	
W(A)	W(B)	

First of a

When variables are same and the operation includes read and write then only a conflict occurs.

Check conflict equivalence of  $s$  and  $s'$

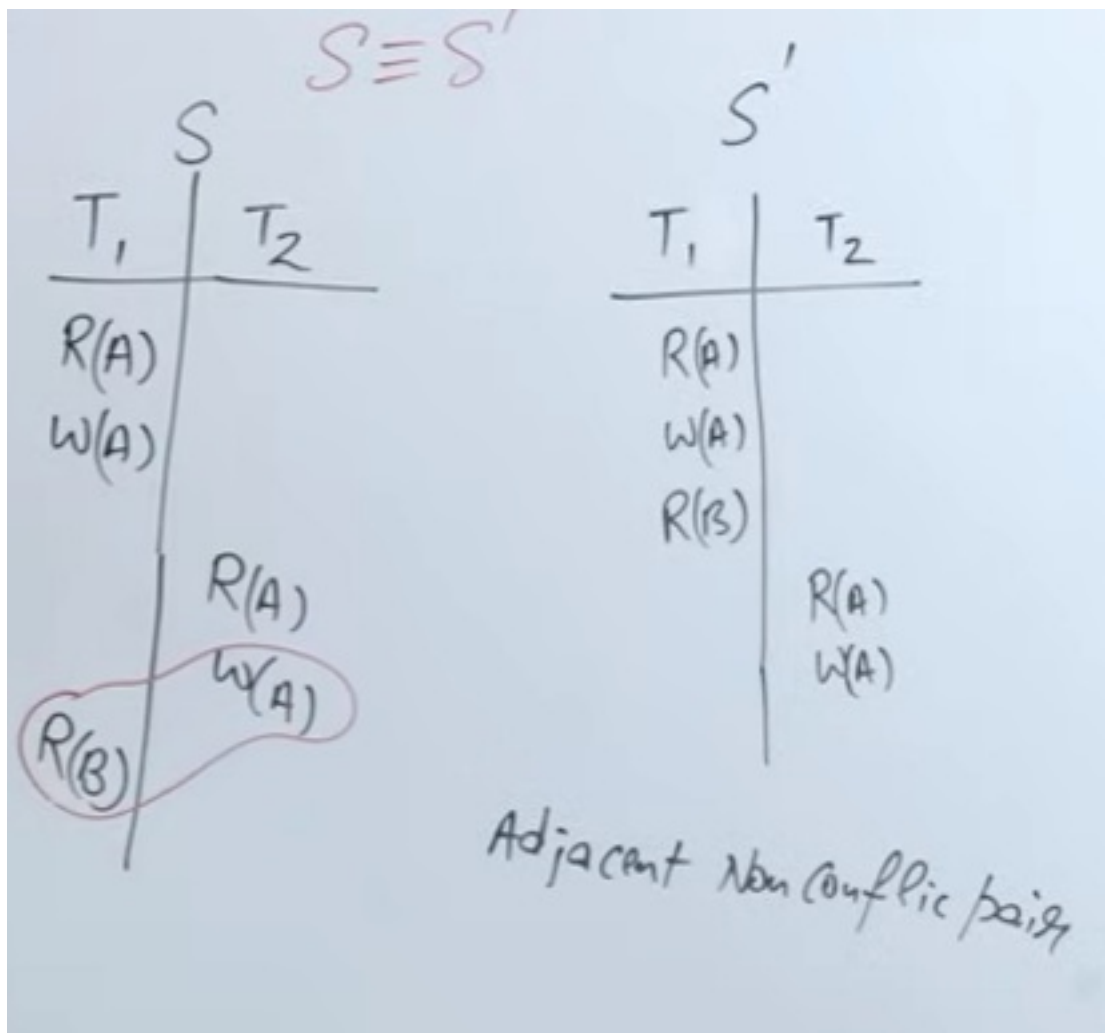
$S \equiv S'$

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

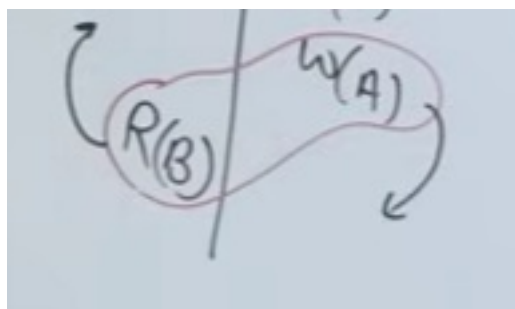
$S'$	
$T_1$	$T_2$
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

R(B) of  $T_1$  is displaced

We need to check for non conflict pair



Swap the non conflict vaalon ko



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



Again an adjacent non conflict pair detected , so swap again

$S' \equiv S$

$S$		$S'$	
$T_1$	$T_2$	$T_1$	$T_2$
$R(A)$		$R(A)$	
$W(A)$		$W(A)$	
$R(B)$		$R(B)$	
	$R(A)$		$R(A)$
	$W(A)$		$W(A)$

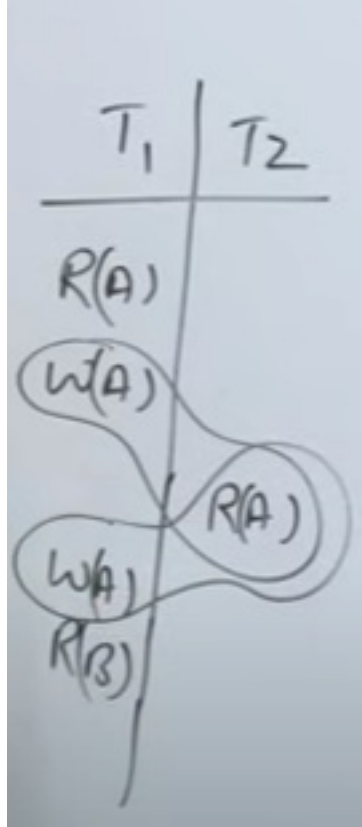
Adjacent Non Conflict pair

Both the schedules became equivalent

Therefore they were equivalent

If we need to find equivalent for this :





There are 2 adjacent conflict pairs , so no swapping can be done

**If a schedule's conflict equivalent exists , then it will always be serializable**

There are other better methods to find conflict equivalents.



**how to find conflict equivalent?**



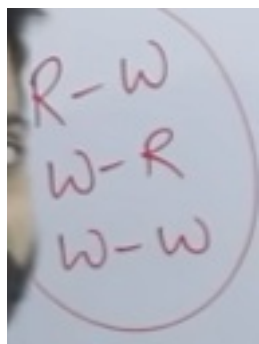
# Conflict Serializability

- Check Conflict pairs transactions and data

$T_1$	$T_2$	$T_3$
$R(x)$		$R(y)$
		$R(x)$
	$R(y)$	
	$R(z)$	
	$w(z)$	$w(y)$
$R(z)$		
$w(x)$		
$w(z)$		

We need to draw a precedence graph.

- The number of vertices would be equal to the total transactions given.
- Check for the following conflict pairs and draw edges

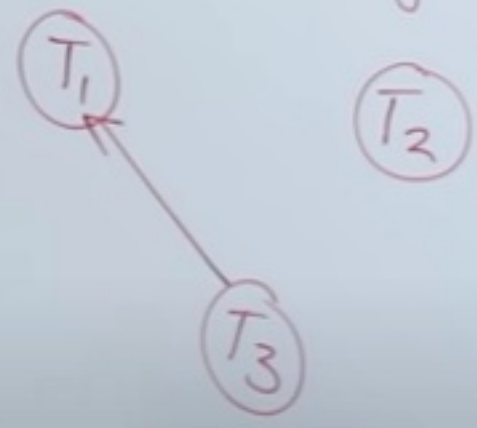


- Here the first conflict occurred here :

S

$T_1$	$T_2$	$T_3$
<del><math>R(x)</math></del>		<del><math>R(y)</math></del> $R(x)$
	$R(y)$ $R(z)$	$w(y)$
	$w(z)$	
$R(z)$ $w(x)$ $w(z)$		

- Check Conflict between transactions and  
Precedence graph



You have to make this a directed graph

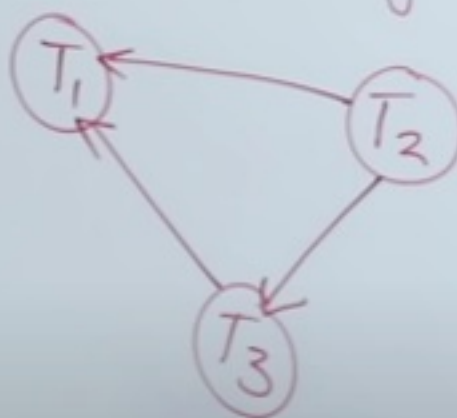
So we drew a directed edge from T3 to T1.

- Also, remember write ka conflict read aur write dono hota hai !
- This would be the resulting graph :

Conflict = Serializable

$T_1$	$T_2$	$T_3$
<del><math>R(x)</math></del>		$R(y)$ <del><math>R(x)</math></del>
	<del><math>R(y)</math></del> <del><math>R(z)</math></del>	
	<del><math>W(z)</math></del>	<del><math>W(y)</math></del> $R(y)$ $W(y)$
$R(z)$ $W(x)$ $W(z)$		

- Check Conflict for transactions and  
Precedence graph



Now its operation are only left

- Check for loop/ cycle -> if doesn't exist -> conflict serializable -> equivalent serial schedule exists -> consistent. If loop exists -> check for view serializability (later)
- What will be the serial schedule of the above parallel schedule? let's find it
- Find the vertex with indegree 0
- **$T_2 \rightarrow T_3 \rightarrow T_1$**
- Basically do topological sort by updating the indegrees and removing the vertex along with connected edges.

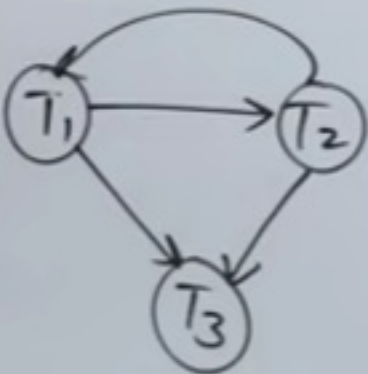
S

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

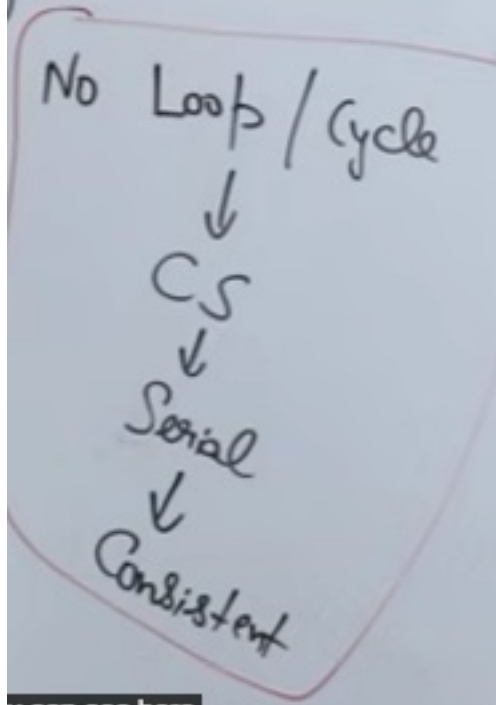
Check wheather Schedule Or not ?

Ans :

No it is not. But we don't know yet whether it can be serialized or not. Non conflict serializable ka mtlb ye nhi hota ki serializable hi nhi ho ♀



Non Conflict Serializable



To check non conflict serializable schedule we use VIEW SERIALIZABLE method.

→ view serializable

[Redacted]

do they match ? :

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>
R(A)		
W(A)	W(A)	
		W(A)

Or not ?

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>
R(A)		
W(A)		
	W(A)	
		W(A)

Lets find out

Lets assume A's value to be 100

Check wheather Schedu  
Or not ?

$T_1$	$T_2$	$T_3$
100 R(A)	$A = A - 40$ W(A) - 60	
$A = A - 40$ W(A) 60 - 40 = 20		$A = A - 20$ W(A)

$T_1$	$T_2$	$T_3$
100 R(A) 60 W(A)	20 W(A)	0 ✓ W(A)

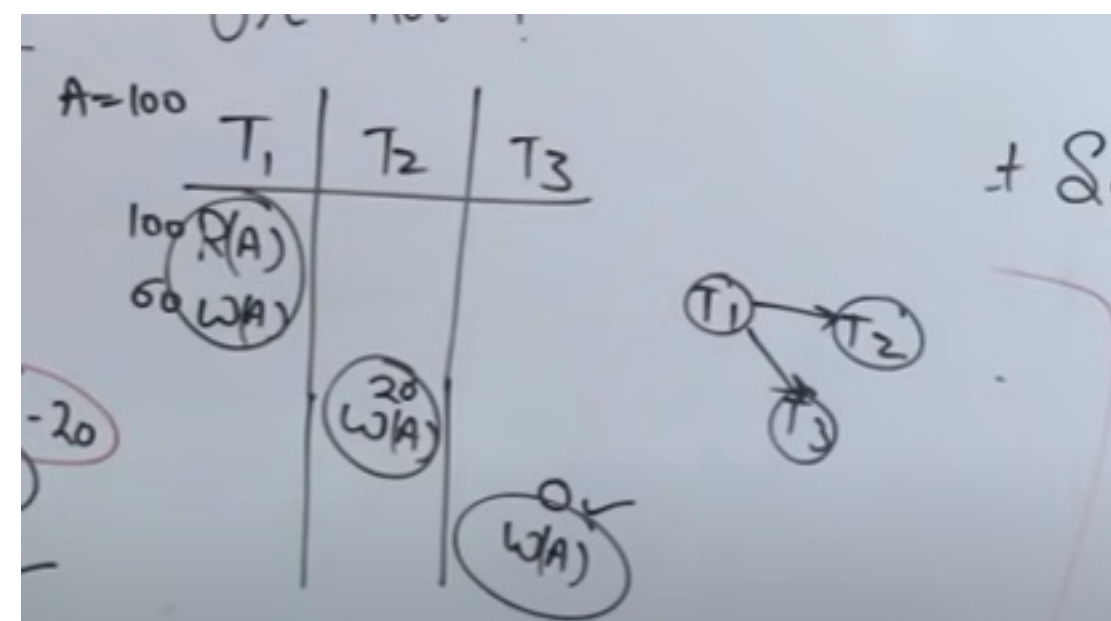
So in this what was final value of A was 0

Both are equivalent as dono ka output match horha hai

But they are NOT CONFLICT EQUIVALENT

THEY ARE VIEW EQUIVALENT.

Precedence graph bnake bhi dekhlo



CONCURRENCY CONTROL PROTOCOL



IT AIMS TO SERIALIZE AND MAKE A TRANSACTION RECOVERABLE.

SHARED – EXCLUSIVE LOCKING.

⇒ Lec-87: Drawbacks in Shared/Excl

Shared – Exclusive

→ Shared Lock (S) ⇒ if t<sub>1</sub>

→ Exclusive Lock (X) ⇒ if t<sub>1</sub>  
whi

\* Problems in S/X locking

1) May not sufficient to be  
Serializable schedule.

2) May not free from In

3) May not free from de



4) May not free from Sto

Varun Sing



0:03 / 10:59 • Introduction

COMPATIBILITY TABLE :

\*SHARED CAN BE GRANTED OVER SHARED\*

A handwritten compatibility table for S/X locks. The table is a 2x2 grid. The columns are labeled 'Request' with 'S' and 'X' above them. The rows are labeled 'grant' with 'S' and 'X' to the left of them. The cells contain the following values: (S, S) is 'Yes' with a checkmark, (S, X) is 'No', (X, S) is 'No', and (X, X) is 'No'.

	Request S	Request X
grant S	Yes	No
grant X	No	No

PROBLEMS WITH S/X LOCK

(A)

$T_1$	$T_2$
$R(A)$ $W(A)$	$T_1 \rightarrow T_2$ OR $T_2 \rightarrow T_1$
$R(B)$ $W(B)$	$R(A)$

(A)

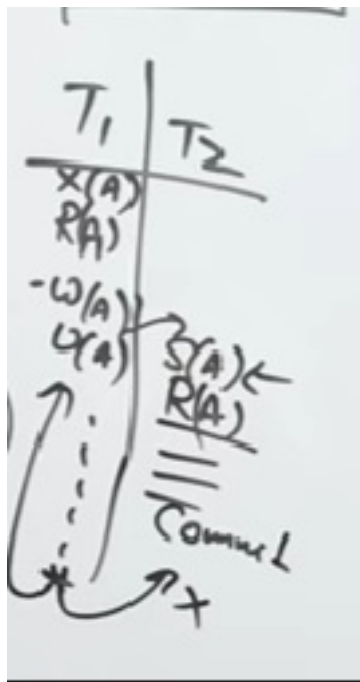
$T_1$	$T_2$
$X(A)$ $R(A)$ $W(A)$ $U(A)$	$S(A) \leftarrow$ $R(A)$ $U(A)$
$X(B)$ $R(B)$ $W(B)$ $U(B)$	

-> NOT SERIALIZED

write both.

\* Problems in S/X locking

- 1) May not sufficient to produce only Serializable schedule.
- 2) May not free from Inrecoverability
- 3) May not free from dead lockz. (R)
- 4) May not free from Starvation. R/A

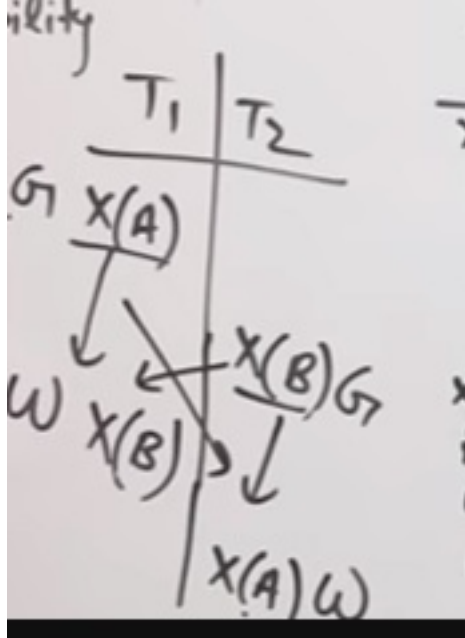


2<sup>ND</sup> POINT HOGYA PROVE.

(DIRTY READ HOGYA)

NOW LETS DISCUSS WHATS DEADLOCK

Do log resources ki wait krre aur dono infinite loop mei wait krre.

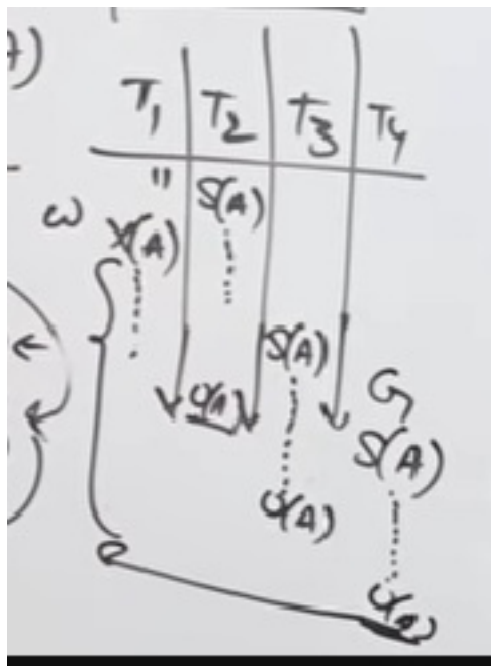


This is deadlock situation

STARVATION?

STARVATION IS LIKE DEADLOCK ONLY PR NOT FOR INFINITELY.

Since shared can be granted above a shared lock so T1 will have to wait till T4 's shared lock has been unlocked.



Since we know when t1 will be granted the X(A) , it is called starvation.

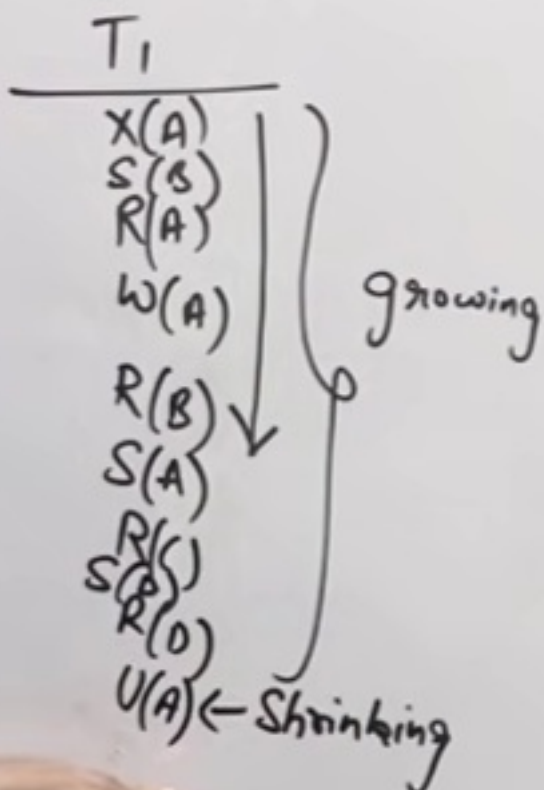
→ **CONCURRENCY CONTROL PROTOCOL**



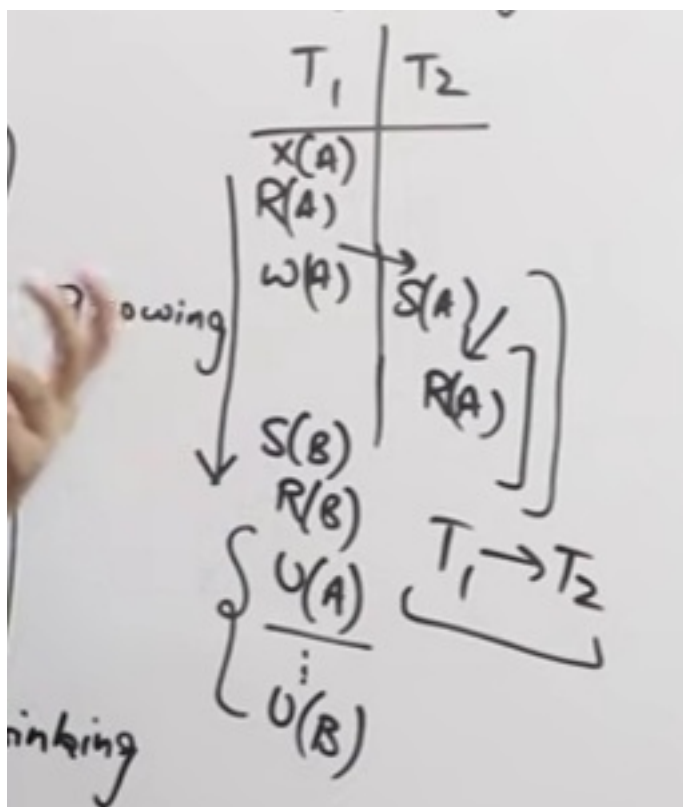
**2 PHASE LOCKING(2PL)**

# MODIFICATION OF S/X ONLY

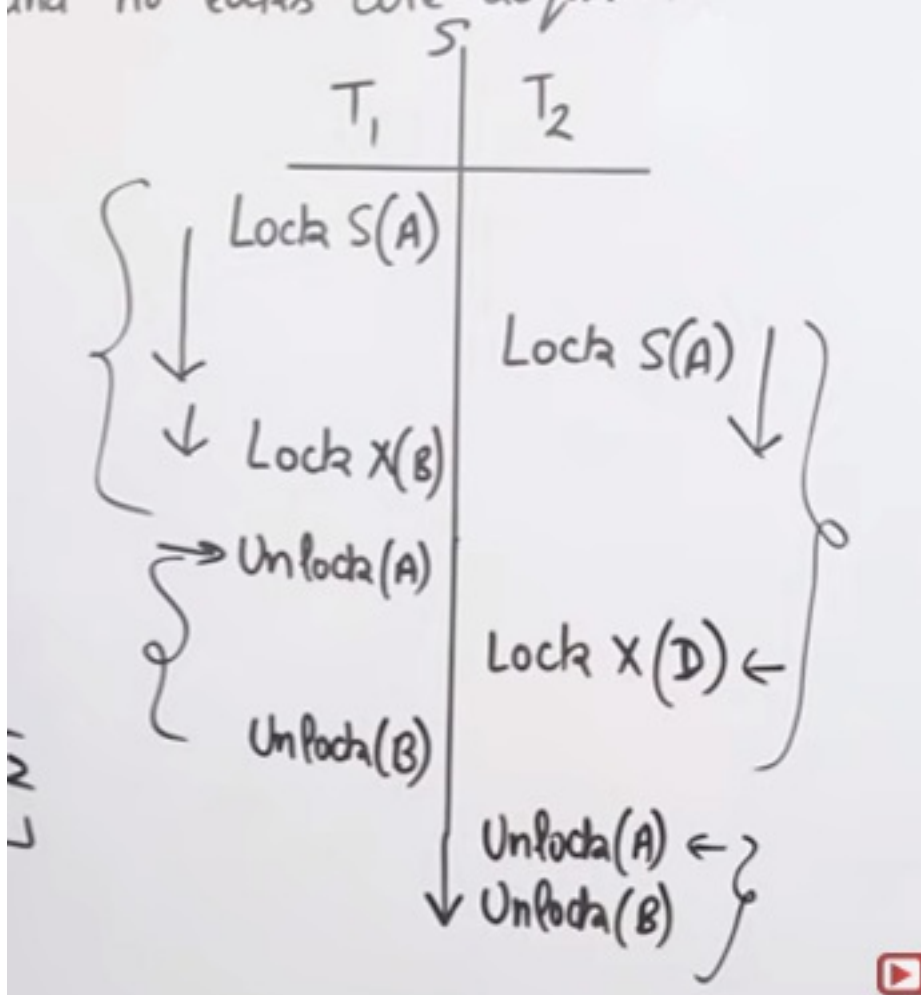
- Growing phase: locks are acquired and no locks are released
- Shrinking phase: locks are released and no locks are acquired



After the shrinking phase no locks will be acquired but only released  
Through this, we can achieve serializability and hence consistency.

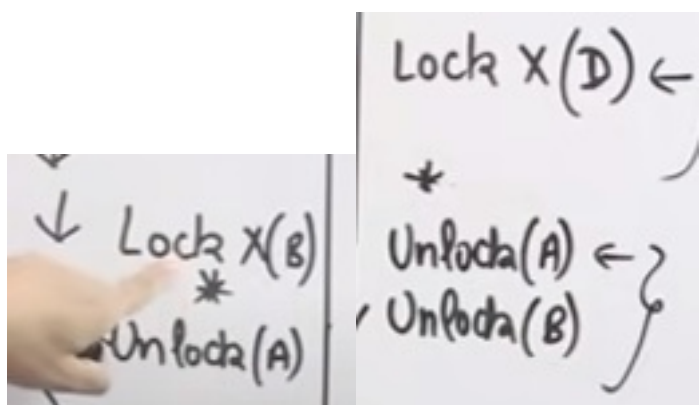


All the 2PL transactions are consistent.



To check what serializability is achieved we see where is the lock point.

Lock point -> pehla unlock hota hai jaha pe



So  $t_1 \rightarrow t_2$

Kyunki  $t_1$  ka lock point pehle aaraha.

 **DRAWBACKS IN 2PL**



## 2PL (2 phase locking)

Advantages: Always ensures Serializability

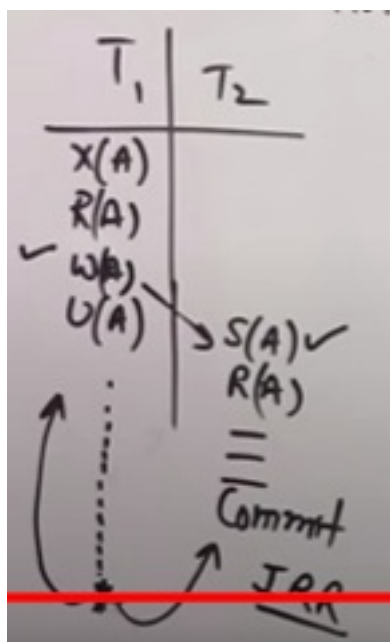
Drawbacks: May not free from irrecoverability

Not free from deadlocks

Not free from starvation

Not free from Cascading Rollback

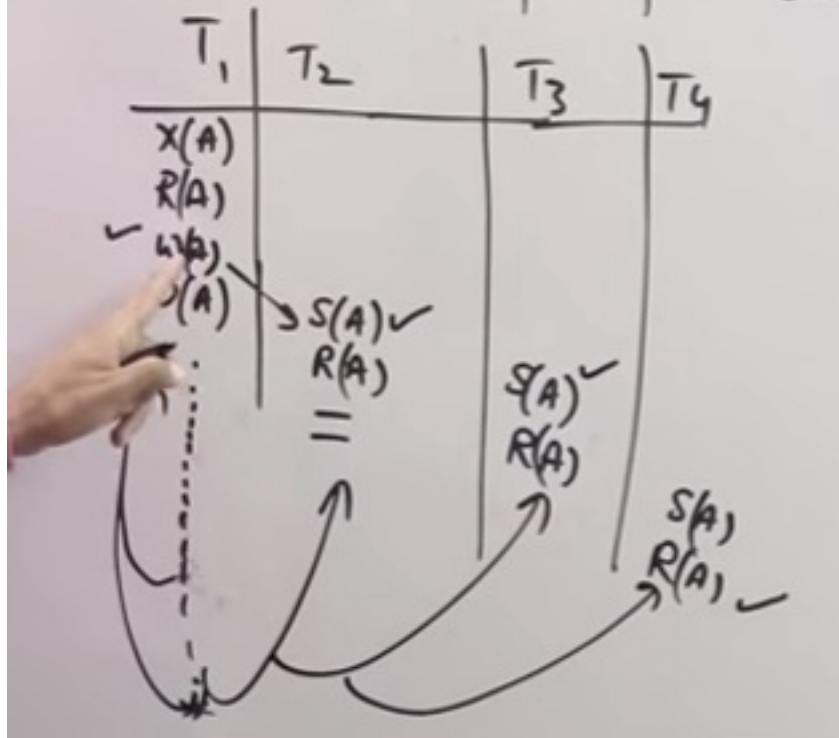
$T_1$  |  $T_2$



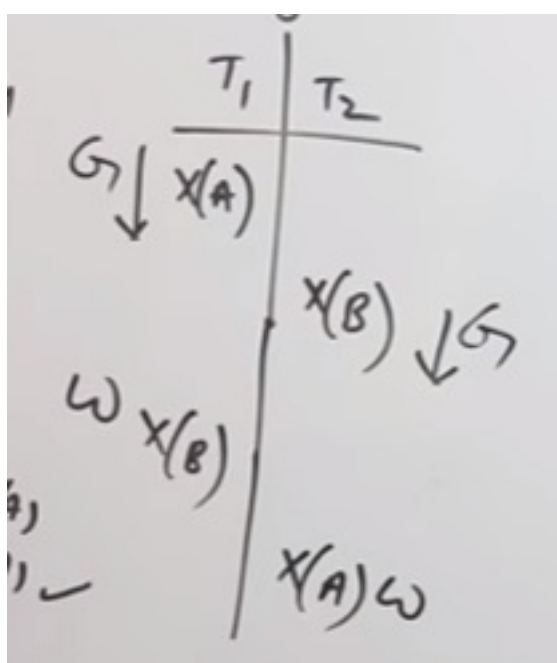
-> IRRECOVERABILITY

CASCADING ROLLBACK :

Not free from Cas

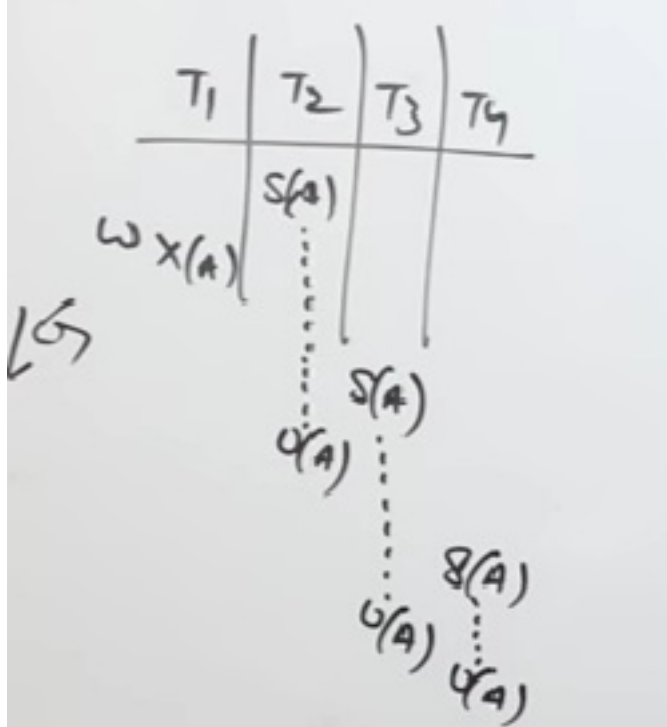


DEADLOCK :



STARVATION:

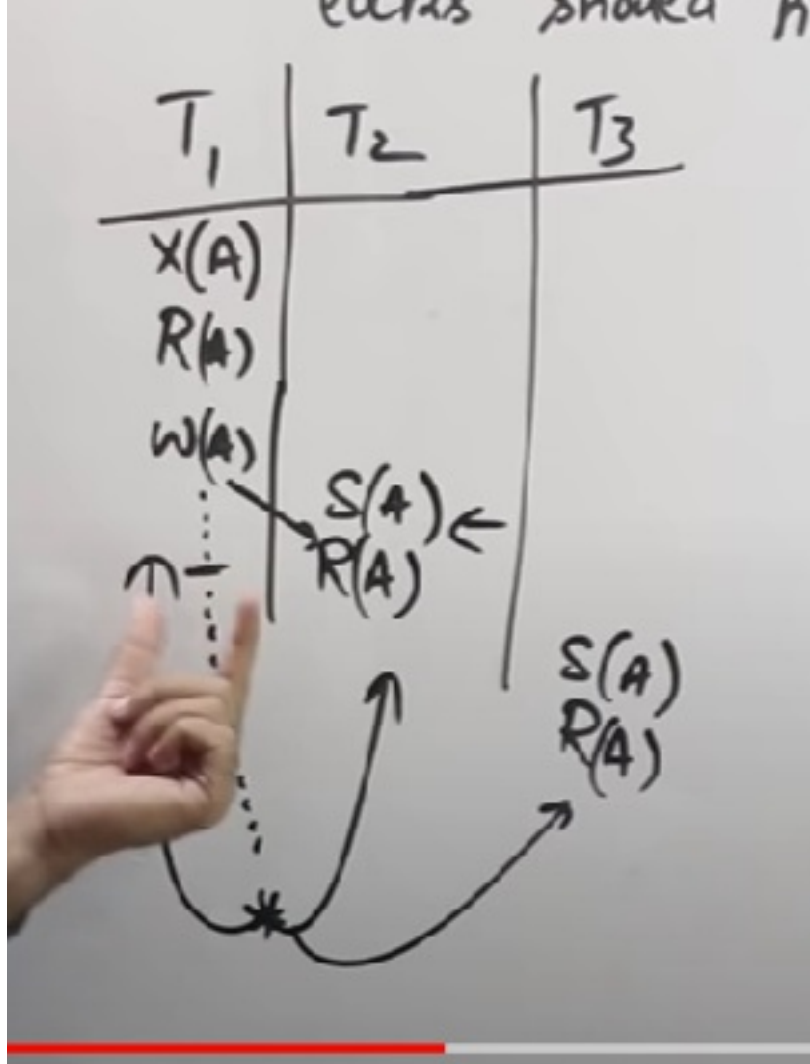




### EXTENSIONS OF 2PL :

Strict 2PL : It should satisfy the basic 2PL and all exclusive locks should hold untill commit/Abort

Rigorous 2PL : It should satisfy the basic 2PL and all shared locks should hold untill commit/Abort.

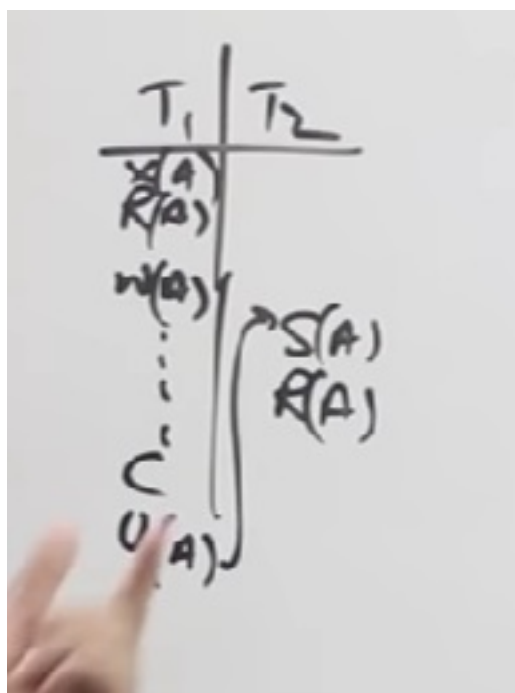


Yaha pe abhi humne strict 2PL nhi kra hai use

So cascading rollback hogya sab

But

Cascading rollback issue is resolved if we use strict 2PL

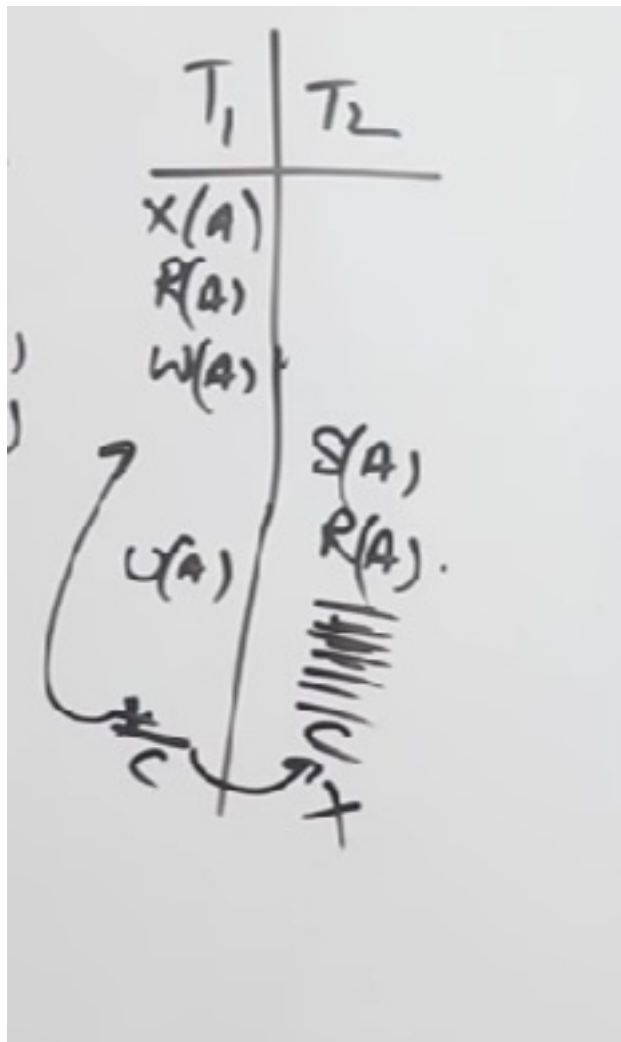


Here  $T_1$  will first get committed then only  $U(a)$  hoga and after that  $T_2$  mei  $R(a)$  will

be read from database only.

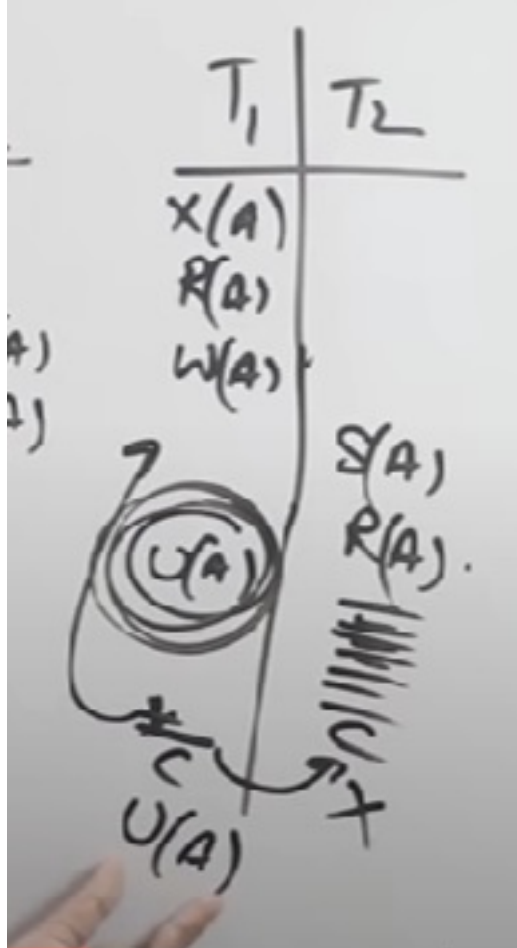
## STRICT 2PL ALWAYS PRODUCES CASCADELESS

Lets talk about irrecoverability now :



Yaha pe  $T_2$  commit hogya to rollback to ho ni skta , so  $T_1$  k roll back k kaaran irrecoverable hogya saara system

On applying strict 2 PL :



Unlock hua nhi to  $t_2$  ka  $S(A)$  grant nhi hua aur ussey pehle hi system abort krgya hehehe so recoverable bngya sab !

**So strict 2pl se cascadeless k saath saath recoverable bhi bngya**

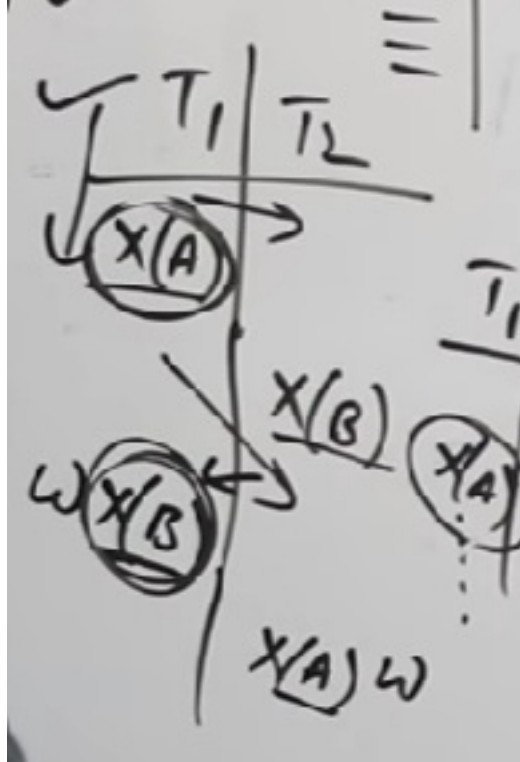
Cascadeless  
Strict Recoverable.

**Rigorous 2 pl is stricter**

**Deadlock aur starvation nhi solve krte but ye**

**There exists something called CONSERVATIVE 2 PL as well**

It says ki pehle transition ko saare locks grant krdo so it wont have to wait for other transitions and therefore deadlock problem solve hojayega



## TIMESTAMP ORDERING PROTOCOL

YE BHI EK CONCURRENCY CONTROL PROTOCOL HAI  
(isme don't look at conflicts sab old young ka khel hai)

'Timestamp Ordering Protocol'

- Unique value assign to every transaction
- Tells the order (when they enters into system)
- $Read\_TS(RTS) = \text{Last (latest) transaction no. which performed Read successfully}$
- $Write\_TS(WTS) = \text{Last (latest) transaction no. which performed Write successfully}$

10:00	10:10
$T_1$	$T_2$
100	200
Older	Younger

Dus bahe agar ek transaction aaya usey maine timestamp dedia 100 ka aur later jo aaya usey 200 ka issey dono ki age maalum pd gyi

Younger	Younger	10	20	30	c)
$T_1$	$T_2$	$T_3$			
$R(A)$	$R(A)$	$R(A)$			
$RTS(A) = 30$					

last ← TS ←

10	Older	20	30	Younger	Younger
$T_1$	$T_2$	$T_3$			
$W(A)$			$RTS(A) = 30$		
			$WTS(A) = 20$		
			$W(A)$		
			$W(A)$		

CONFLICT → TS → D —

\* Rules:

1) Transaction  $T_i$  issues a Read(A) operation

a) if  $WTS(A) > TS(T_i)$ , Rollback  $T_i$

b) Otherwise execute R(A) operation

Set  $RTS(A) = \max\{RTS(A), TS(T_i)\}$

2) Transaction  $T_i$  issues Write(A) operation

a) if  $RTS(A) > TS(T_i)$  then Rollback  $T_i$

b) if  $WTS(A) > TS(T_i)$  then Rollback  $T_i$

c) otherwise execute write(A) operation

Set  $WTS(A) = TS(T_i)$

30

$T_3$



So pehle deal with the older one

Neglect krdo conflict ko bs older vaale ko pehle execute krna hai no matter what

which performed

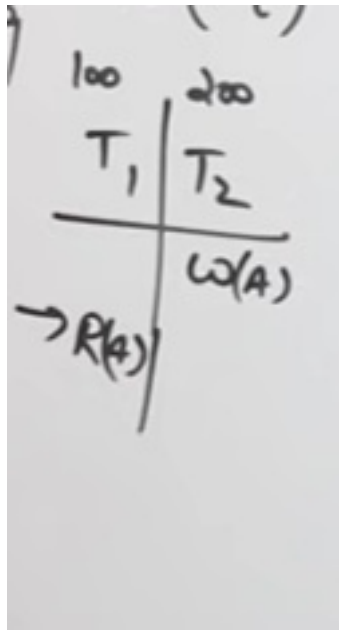
100 $T_1$	200 $T_2$	you
	R(A) ✓	
→ W(A)		



Rules state pehle aai hu transaction baad mei operation nhi krni chahiye !

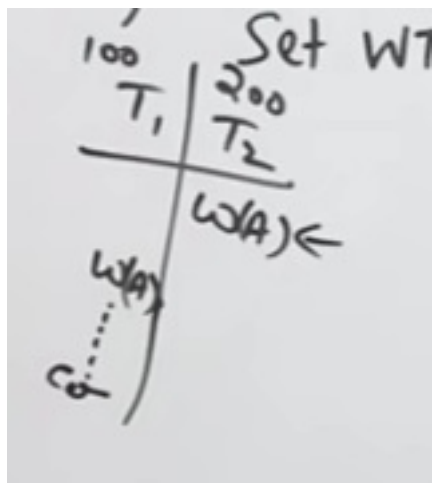
So idhr T1 ko roll back krna pdega

Similarly



A handwritten table with two columns. The first column is headed '100' and contains 'T<sub>1</sub>'. The second column is headed '200' and contains 'T<sub>2</sub>'. A horizontal line is drawn below the headers. Below 'T<sub>1</sub>', there is an arrow pointing to the left towards the text 'R(A)'. Below 'T<sub>2</sub>', there is the text 'W(A)'.

100	200
T <sub>1</sub>	T <sub>2</sub>
→ R(A)	W(A)



A handwritten table with two columns. The first column is headed '100' and contains 'T<sub>1</sub>'. The second column is headed '200' and contains 'T<sub>2</sub>'. A horizontal line is drawn below the headers. Below 'T<sub>1</sub>', there is a dashed line leading to a curved arrow pointing to the text 'W(A)'. Below 'T<sub>2</sub>', there is the text 'W(A)' with an arrow pointing to the left.

100	200
T <sub>1</sub>	T <sub>2</sub>
W(A)	W(A) ←



ALWAYS TREAT OLDER ONE FIRST.

BASICALLY OLDER KUCH BHI YOUNGER SE BAADD MEI KRE TO USKE PROCESS KO ROLLBACK KRDO

**NUMERICAL**



$(100)_{T_1}$	$(200)_{T_2}$	$T_3(300)$
$R(A)$		
$W(C)$	$R(B)$	
$R(C)$		$R(B)$
	$W(B)$	$W(A)$

	A	B	C
RTS	0	0	0
WTS	0	0	0

$$0 > 100$$

Otherwise execute  $R(A)$  operation  
 Set  $RTS(A) = \text{Max}_i \{ RTS(A), TS(T_i) \}$

	A	B	C
RTS	$\overset{100}{\emptyset}$	0	0
WTS	0	0	0

$$0 > 200$$

$$\text{Set } RTS(A) = \text{Max}_g \{ RTS(A), TS(T_i) \}$$

	A	B	C
RTS	$\frac{100}{\emptyset}$	$\frac{200}{\emptyset}$	0
WTS	0	0	0

SIMILARLY DO FOR ALL ITHET OPERATIONS

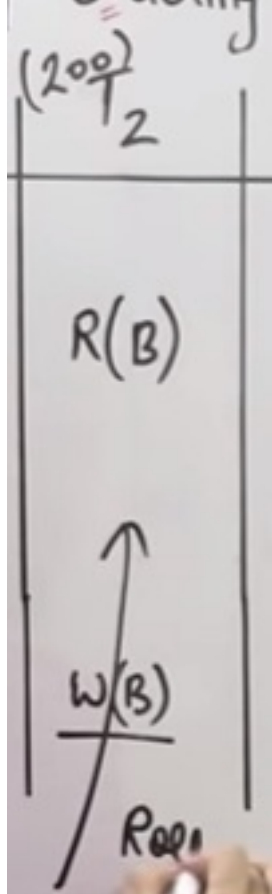
### 'Time stamp Ordering Protocol'

$(100)$ old $T_1$	$(200)$ $T_2$	$T_3(300)$ youngest
$R(A)$	$R(B)$	$0 > 100$ $0 > 200$ $0 > 100$ $0 > 100$ $0 > 300$ $100 > 100$ $[300 > 100]$
$\rightarrow W(C)$		$R(B) \leftarrow$
$\rightarrow R(C)$	<u><math>W(B)</math></u>	$W(A)$

	A	B	C
RTS	100	<del>200</del> 300	<del>100</del> 100
WTS	0	0	<del>100</del> 100

\* Rules:

- 1) Transaction  $T_i$  is  
 { a) if  $WTS(A)$   
 b) otherwise execute  
 Set  $RTS(A)$
- 2) Transaction  $T_i$  is  
 a) if  $RTS(A) > T$   
 b) if  $WTS(A) > T$   
 c) otherwise execute  
 Set  $WTS(A)$



T2 WILL NOW RESTART BAAD MEI SABSE

♡. mnd

# 'Timestamp Ordering Protocol'

(100)  
oldest  $T_1$

(200)  
 $T_2$

$T_3$ (300)  
youngest

R(A)

R(B)

→ W(C)

→ R(C)

2000  
W(B)

(1000)

R(B) ←

(W(A))

Rollback

$0 > 100$

$0 > 200$

$0 > 100$

$0 > 100$

$0 > 300$

$100 > 100$

$[300 > 200]$

\* Rules:

1) True

{ a)

{ b)

2) True

a) if

b) if

c) ot

	A	B	C
RTS	100 Ø	<del>200</del> 300 Ø	Ø 100
WTS	0	0	Ø 100

$100 > 300$   
 $0 > ?$

3

300