INTRODUCTUCTION TO TRANSACTION CONCURRENCY

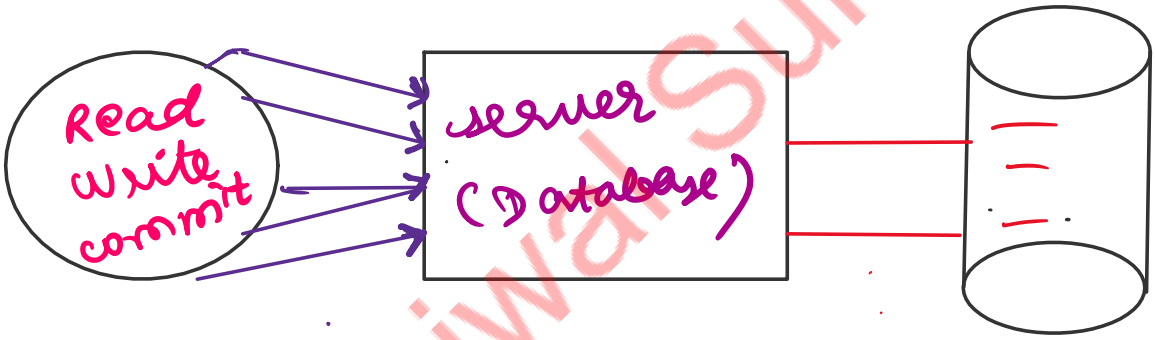
15 October 2023 12:03

Transactions

↳ It is a set of operations used to perform a logical unit of work.

A transaction generally represents a change in database.

Work → withdraw money



Transfer A → B

Database Access change

Example: A = 1000, B = 2000

Transfer 500 from A → B.

1. R(A) → A = 1000
2. A = A - 500
3. W(A) → A = 500
4. R(B) → B = 2000
5. B = B + 500
6. W(B) → B = 2500

③ W(A) ← A = ...

[All in RAM only]

Finally ⑦ commit ← save it in the database

[R → read, W → write]

ACID PROPERTIES

15 October 2023 13:27

(A)tomicity (C)onsistency

(I)solation (D)urability

ACID

Atomicity

↳ either all or none

In a transaction, if any one step fails before commit

ROLLBACK is done.

↳ as if no statement in the transactions were executed.

A failed transaction cannot be recovered, it will always restart.

T1

R(A)
A = A - 50
W(A)
R(B)
:

---

consistency

Before the transaction starts, and after the transaction ends, the sum of the total money should be the same.

1000

be the same.

Eg: A = 2000
B = 3000

$A \xrightarrow{1000} B$

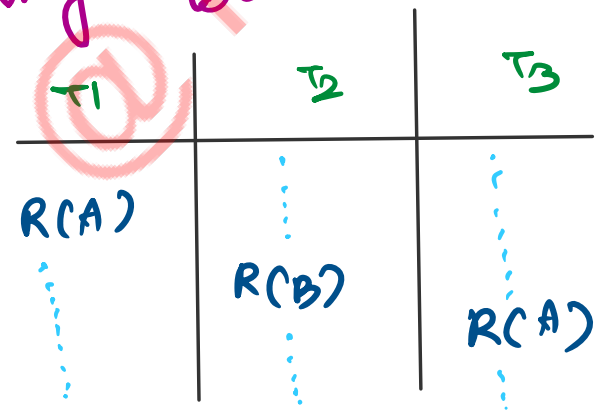
R(A) → A = 2000
A = A - 1000
W(A) → A = 1000
R(B) → B = 3000
B = B + 1000
W(B) → B = 4000
commit

Before transaction:
sum = A + B
= 2000 + 3000
= 5000

After transaction:
sum = 1000 + 4000
= 5000
(same)

Isolation

many transactions running parallely



$T_1 \rightarrow T_2$ (or)
$T_2 \rightarrow T_1$

can a parallel schedule be converted into a serial schedule in real time (only conceptually) ?

into a [illegible]

(only conceptually) ?

[serial schedule → always consistent]

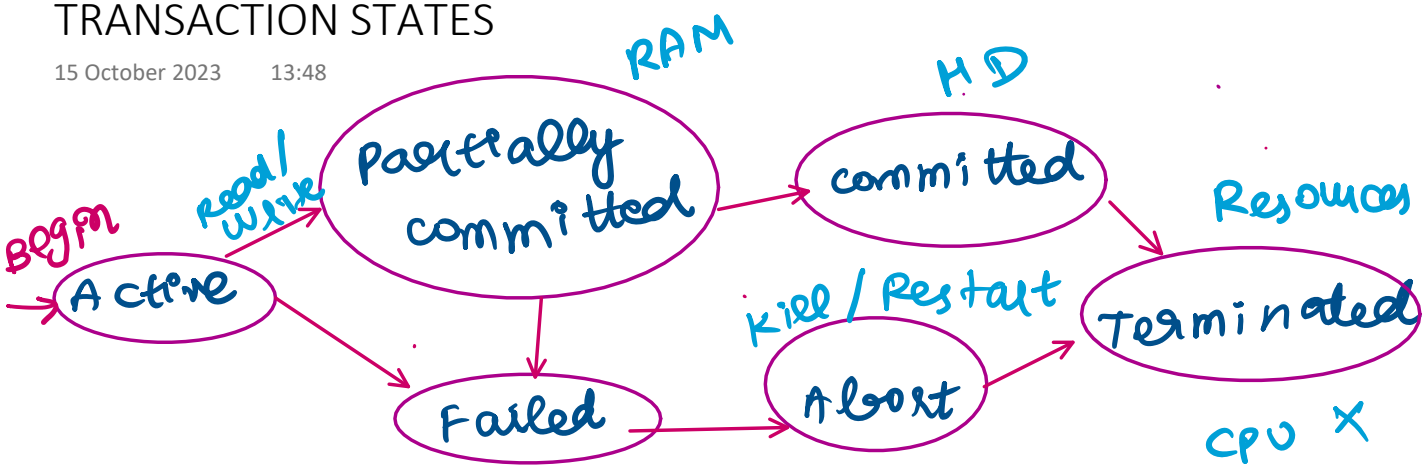
Durability

↓

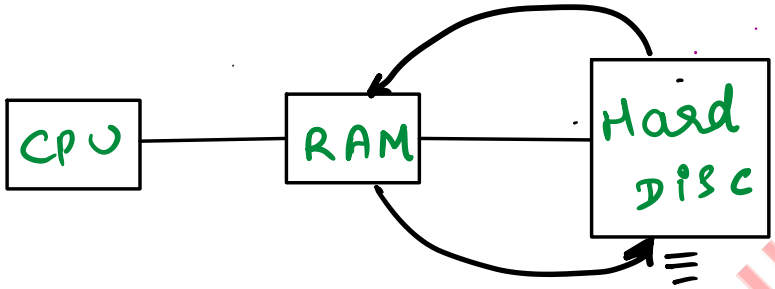
whatever changes are performed on the database must be permanent.

TRANSACTION STATES

15 October 2023 13:48



CPU X
RAM X
registers X



R(A)
A = A - 50
W(A)
⋮

n-1 operations done

nth operation → commit → Hard Disc

SCHEDULE

15 October 2023 16:22

Schedule

↳ It is the chronological execution sequence of multiple transactions.

T_1 T_2 T_3 .... T_n

---

serial schedule : → (eg ATM)

T_1 T_2 T_3

Any transaction that starts, completes fully without any interruptions.

↳ so secure.

serial transactions are always consistent.

But the problem is waiting time → even though T_1, T_2, T_3 arrived at the same time → T_2 & T_3 had to wait for T_1, then T_3 had to wait for T_2.

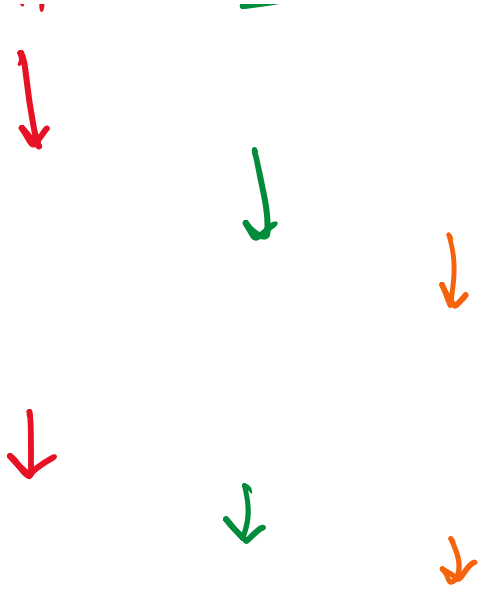
---

parallel schedule :

T_1 T_2 T_3

1

↓ eg online banking system



eg online [illegible] system

Here waiting time ↓

→ performance ↑

→ very high throughput ↑

→ no. of transactions executed / time

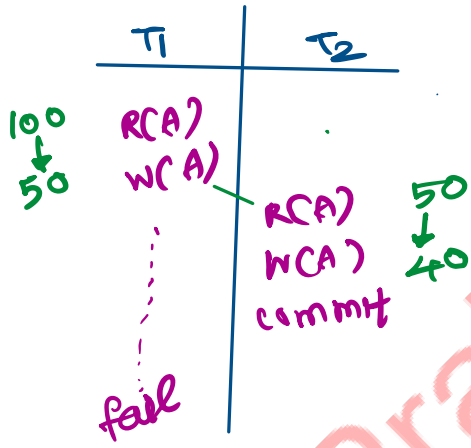
PROBLEMS IN CONCURRENCY

15 October 2023 16:33

(parallel schedule)

* dirty read
* incorrect summary
* lost update
* unrepeatable read
* phantom read

Dirty Read / uncommitted read / RAW



dirty read → value should be 100 as T1 failed, but it is taken as 50 instead.

Incorrect summary

T1	T2	A=1000, B=1000, C=1000
	s = 0	s = 0 (sum)
	avg = 0	avg = 0
	read(c)	T2 R: C=1000
	s = s+c	s = 1000
read(A)		T1 R: A = 1000
A = A-50		

read(A)

A = A-50

write(A)

read(A)

S = S+A

read(B)

S = S+B

avg = S/3

commit

read(B)

B = B+50

write(B)

commit

T_1 W : A = 950

T_2 R : A = 950

S = 1950

T_2 R : B = 1000

S = 2950

avg = 983.33

T_1 R : B = 1000

T_2 W : B = 1050

Lost Update

Alice

Q : 7

L : 5

Begin

Select

Update qty = 6

lost ← commit : Q = 6

L = 5

Bob

Begin

Select

Update qty = 10

commit : Q = 10

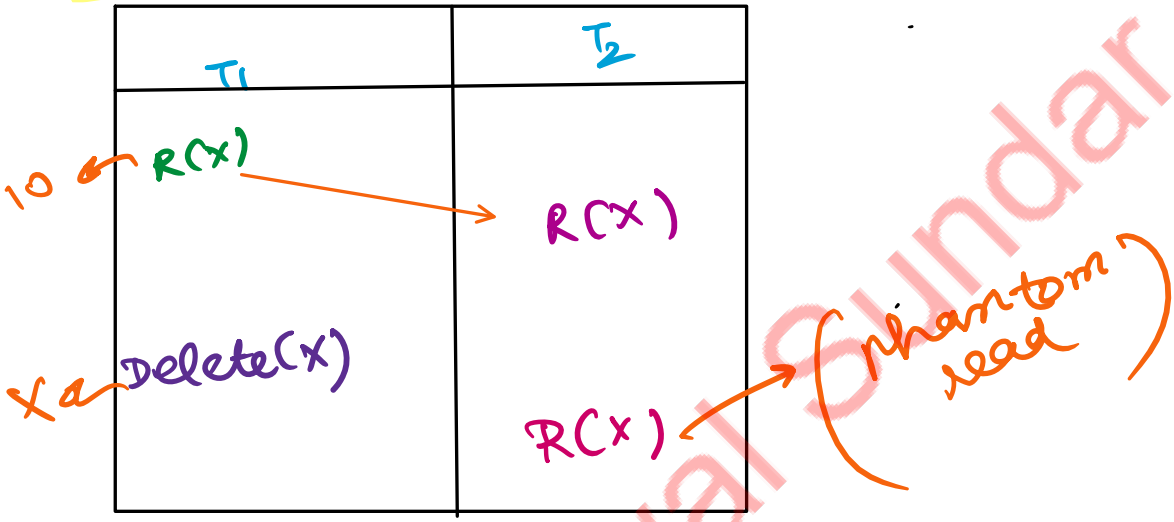
Q → quantity

L → likes

lost ← .... L = 5

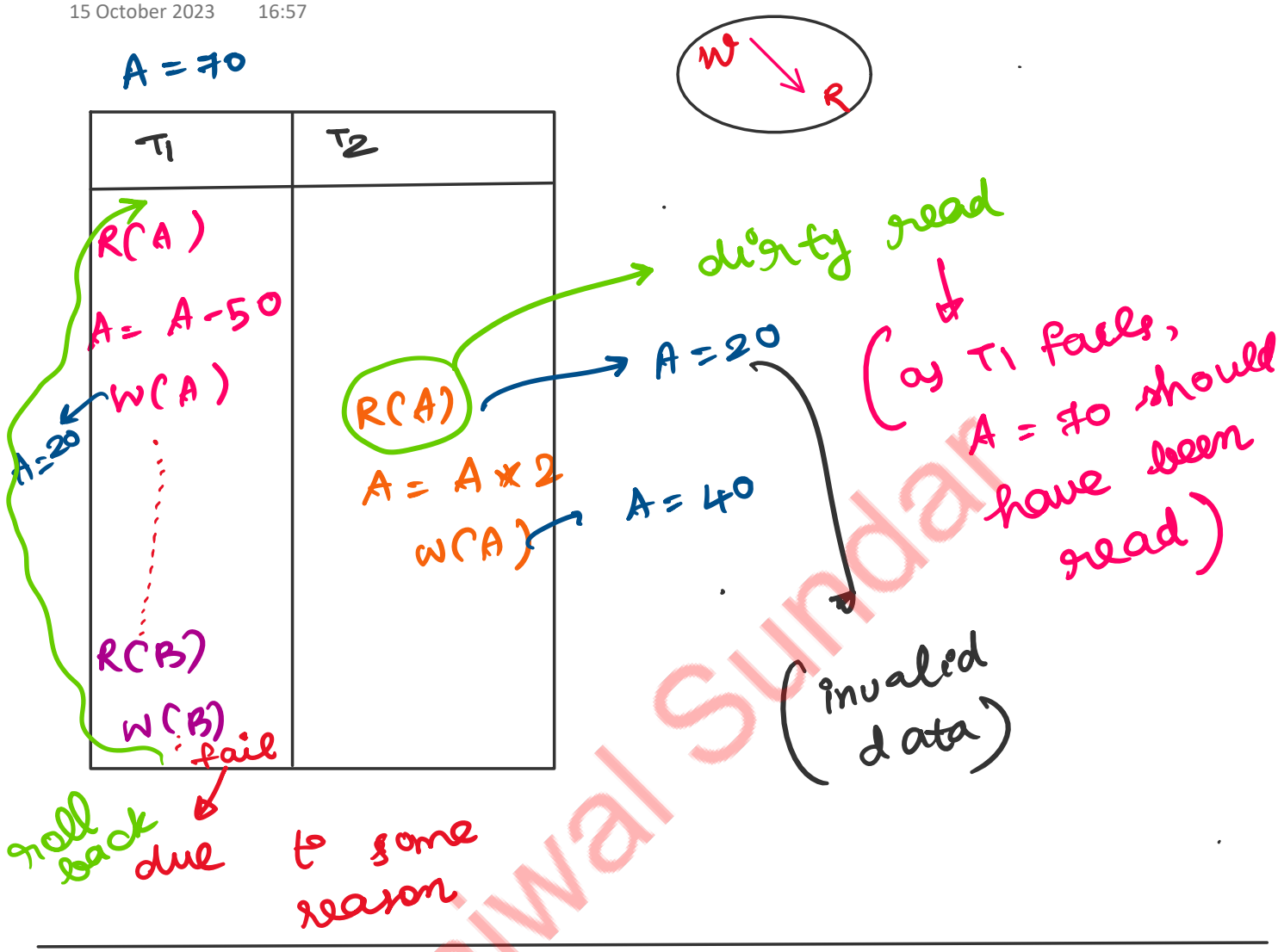
commit : Q = 10
L = 5
↓
overwrites previous commit

Unrepeated read



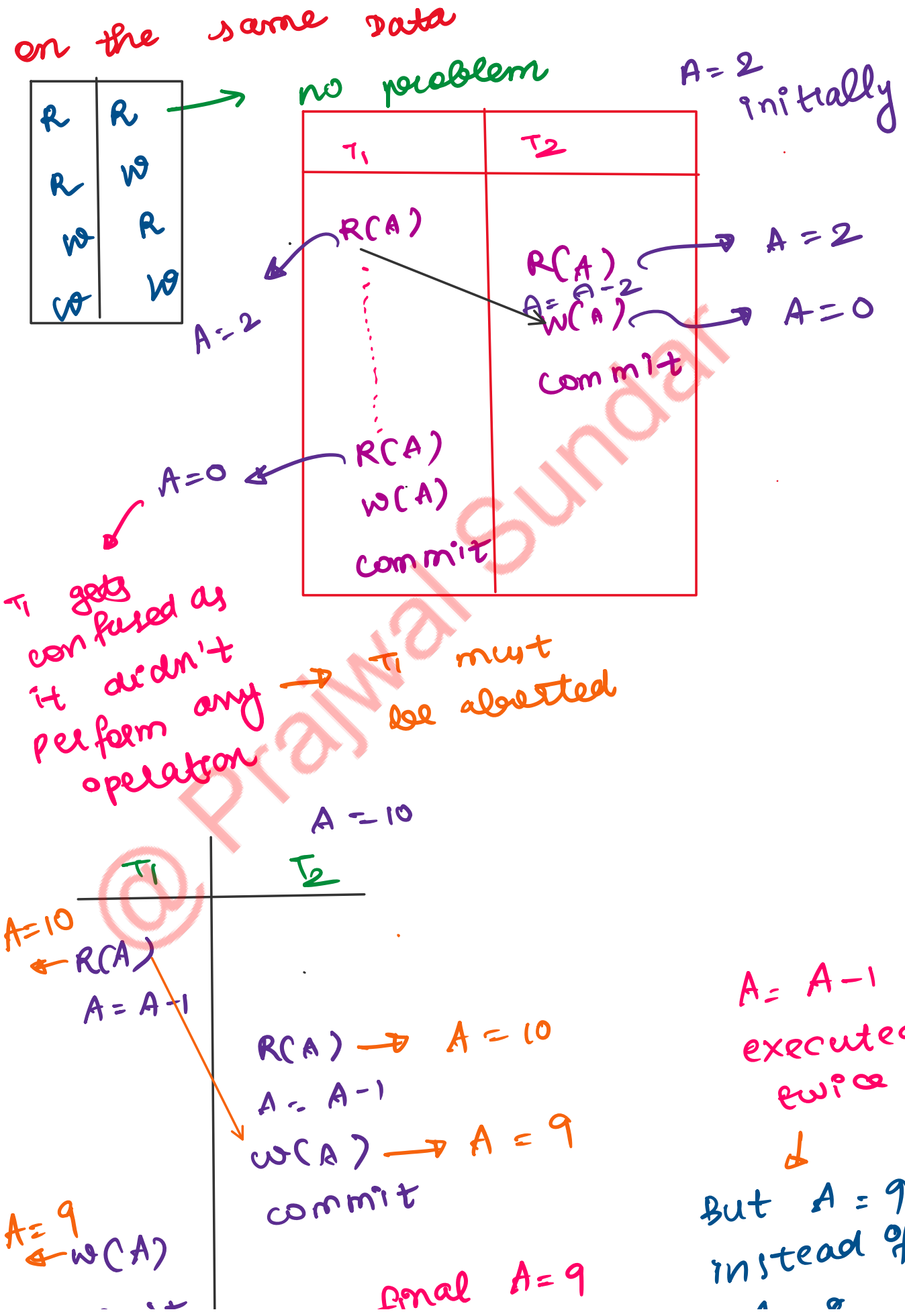
WRITE-READ CONFLICT [DIRTY READ PROBLEM]

15 October 2023 16:57



READ-WRITE CONFLICT [UNREPEATABLE READ]

15 October 2023 17:04



W(A)

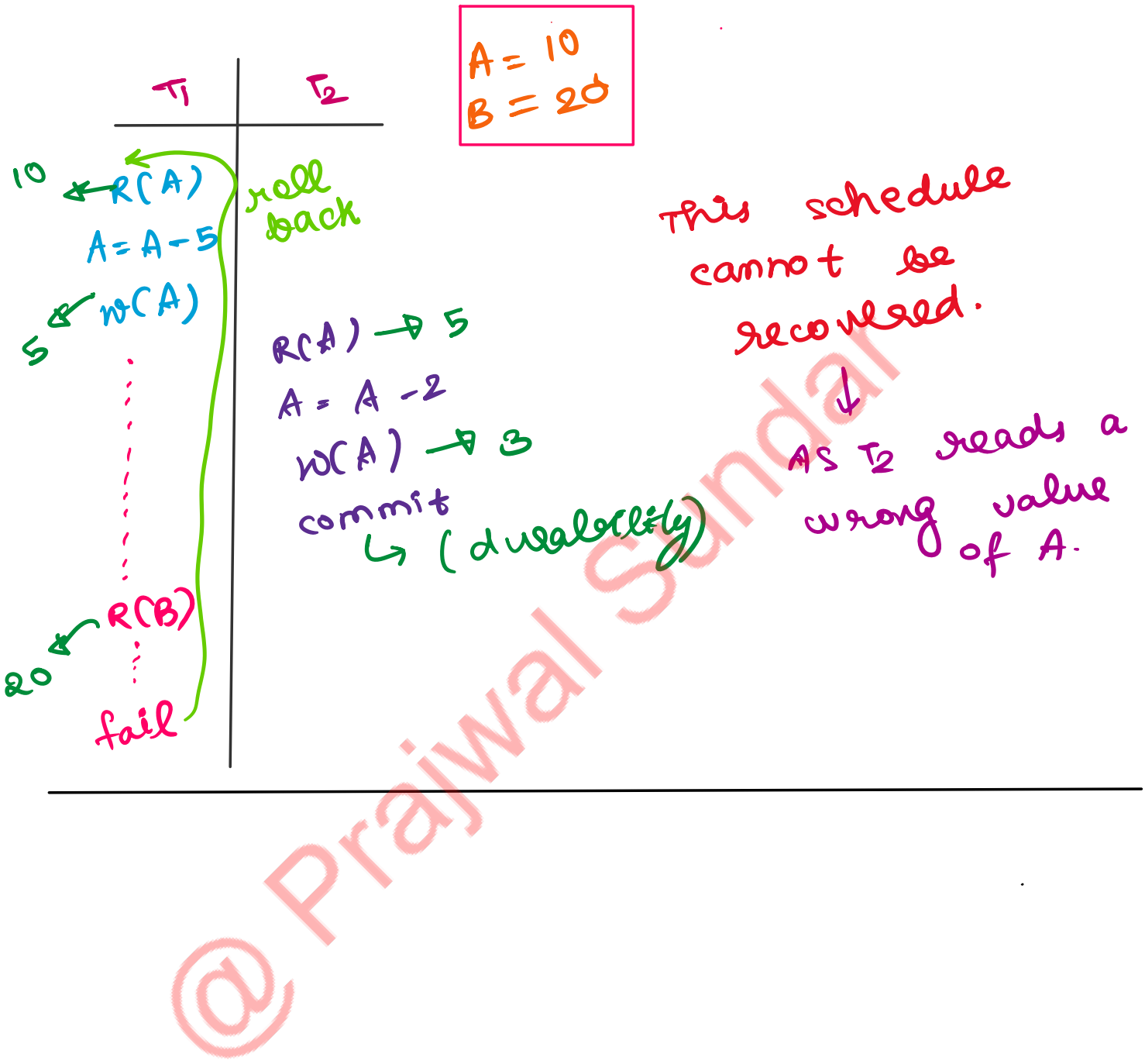
commit

final A = 9

instead A = 8

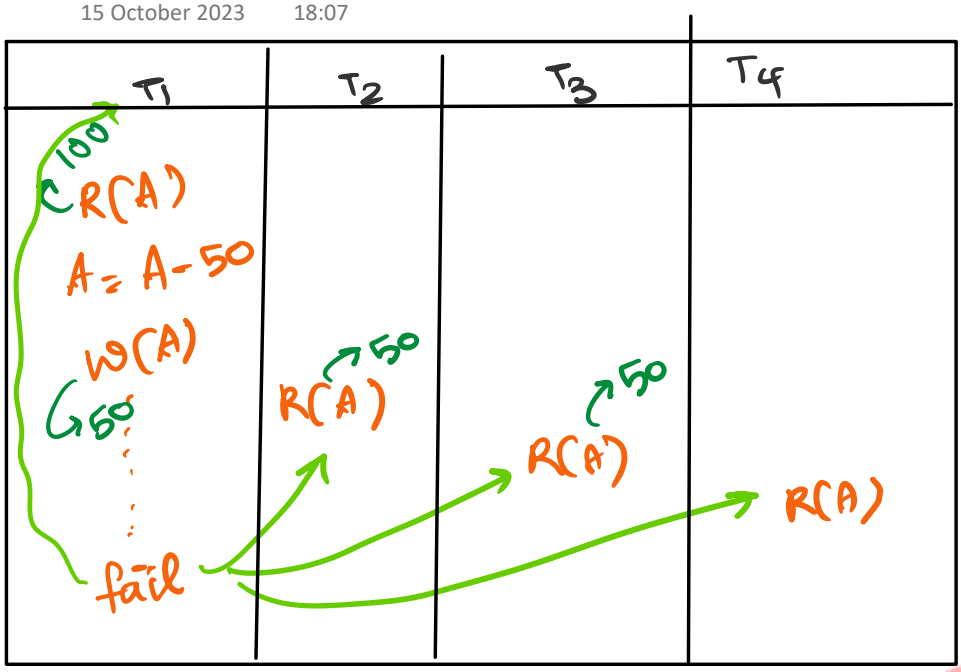
IRRECOVERABLE SCHEDULE → (recoverability)

15 October 2023 18:02



CASCADING VS CASCADELESS SCHEDULE

15 October 2023 18:07



while performing rollback

↓

rollback all connected transactions [cascade]

↳ CPU utilization ↓

Performance is degraded

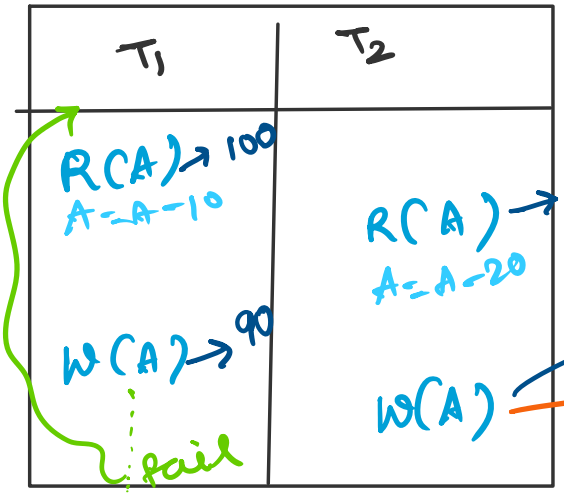
T_1	T_2	T_3
R(A)		
W(A)	R(A) X	
		R(A) X

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

while T_1 is operating on data A, do not allow other transactions to access it until either

* T_1 commits (OR) * T_1 fails

A = 100



A = 100

[WW problem still exists]

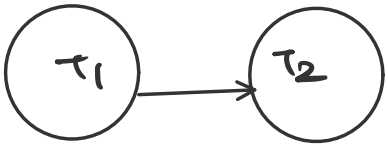
lost update problem

SERIALIZABILITY

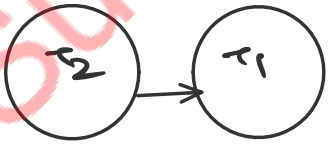
15 October 2023 18:30

↳ ability to make a schedule serializable

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



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



these 2 schedules are already serial schedules → no need to serialize / check serializability.

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

↓

(parallel schedule)

checking for serializability

↓

check if an equivalent schedule of the given parallel schedule can be obtained or not.

Types → conflict
→ view

(parallel schedule)

Types → view

[convert to $T_1 \rightarrow T_2$ (OR) $T_2 \rightarrow T_1$]

Another example :

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

parallel schedule

$3! = 6$ cases

T_1 T_2 T_3

T_1 T_3 T_2

T_2 T_1 T_3

T_2 T_3 T_1

T_3 T_1 T_2

T_3 T_2 T_1

if the given parallel schedule can be converted into any one of these 6 combinations, it is serializible.

CONFLICT EQUIVALENT SCHEDULE

15 October 2023 18:56

R(A) R(A) → non conflict pair

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

conflict pairs

R(B) R(A)
R(B) W(A)
W(B) R(A)
W(B) W(A)

non-conflict pairs

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)

Is S ≡ S' ?

↓

check

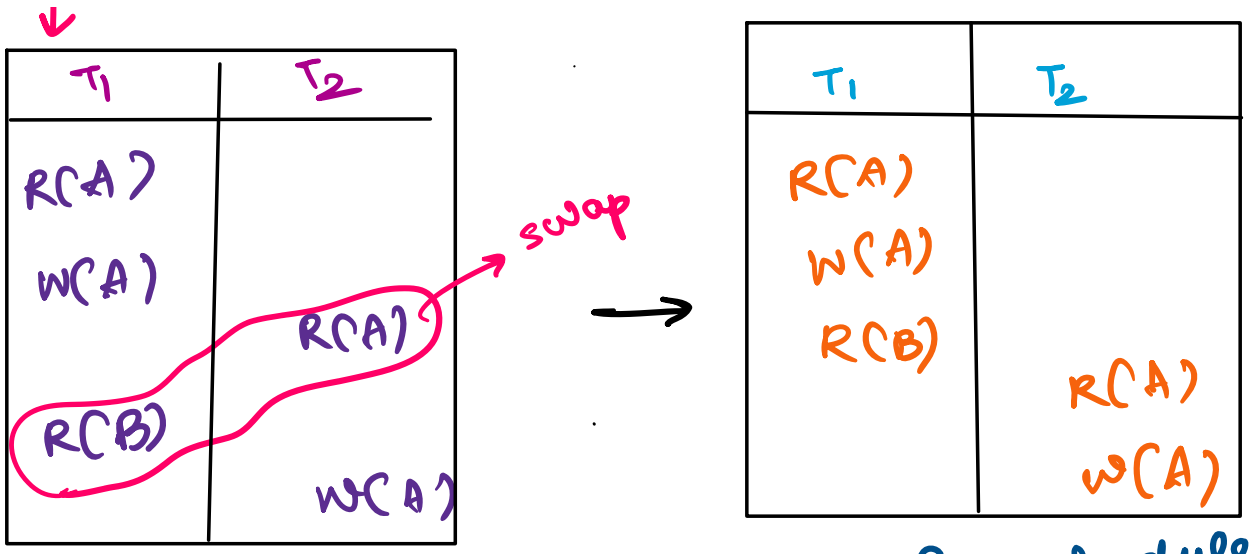
Method :

swap adjacent non-conflict pairs

↓

S'

T_1	T_2



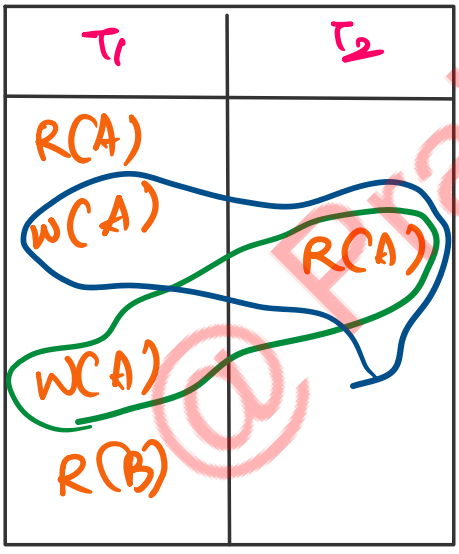
serial schedule obtained.

$S \equiv S'$ → obtained

∴ S is conflict serializable.

---

check for conflict serializibility:



conflicting pairs ↓ no change in positions possible

→ not conflict serializable

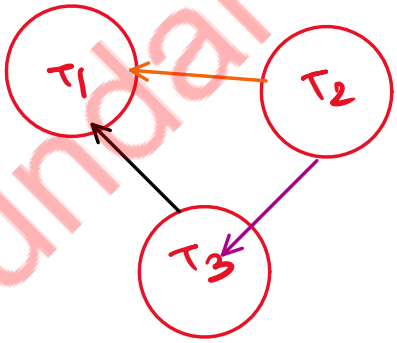
CONFLICT SERIALIZIBILITY

15 October 2023 19:29

check conflict pairs in other transactions and draw edge [RW, WR, WW]

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

precedence graph :



check for loop / cycle in the graph

If no → conflict serializable ✓

↓

serializable ✓

↓

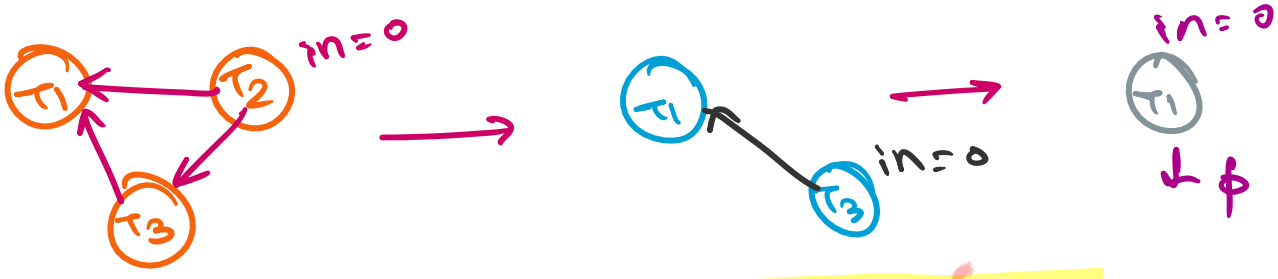
consistent ✓

To find which order, use :

topological sort

[ ] pick vertex with in-degree = 0

[choose vertex with in-degree = 0
and erase it from the graph]



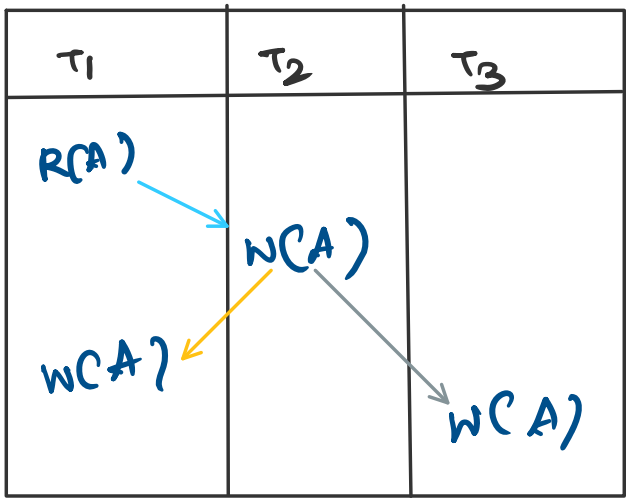
serial order : $T_2 \rightarrow T_3 \rightarrow T_1$

Equivalent serial schedule :

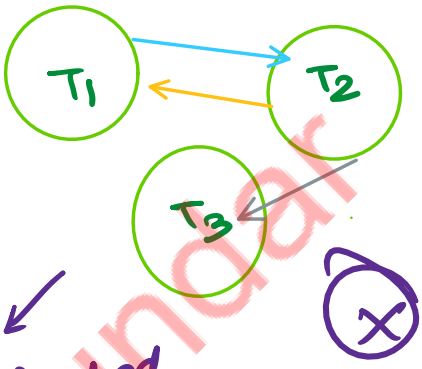
T_1	T_2	T_3
	R(Y)	
	R(Z)	
	W(Z)	
		R(Y)
		R(X)
		W(Y)
R(X)		
R(Z)		
W(X)		
W(Z)		

WHY VIEW SERIALIZIBILITY?

15 October 2023 19:49



check whether given schedule is conflict serializable or not



Loop Detected.

Topological sort is not possible

It is non conflict serializible.

↳ may / may not be serializible.

[If no loop / cycle → CS → serial → consistent]

To check serializibility

↳ view serializibility is used.

A = 100

T1	T2	T3
100 R(A)	A=A-40 W(A) 60	
A=A-40 W(A)		A=A-20 W(A)

≡

T1	T2	T3
100 R(A) A=A-40 W(A) 60		
	A=A-40 W(A) 20	
		A=A-20 W(A) 0

Final value of A is the same.

They look same → view equivalent ✓

But not conflict equivalent ✗

SHARED-EXCLUSIVE LOCKING

15 October 2023 20:24

Shared Lock (S)

- If transaction locked data item in shared mode, then it is allowed to read only.

Exclusive Lock (X)

- If Transaction lock data item in exclusive mode, then it is allowed to read and write both.

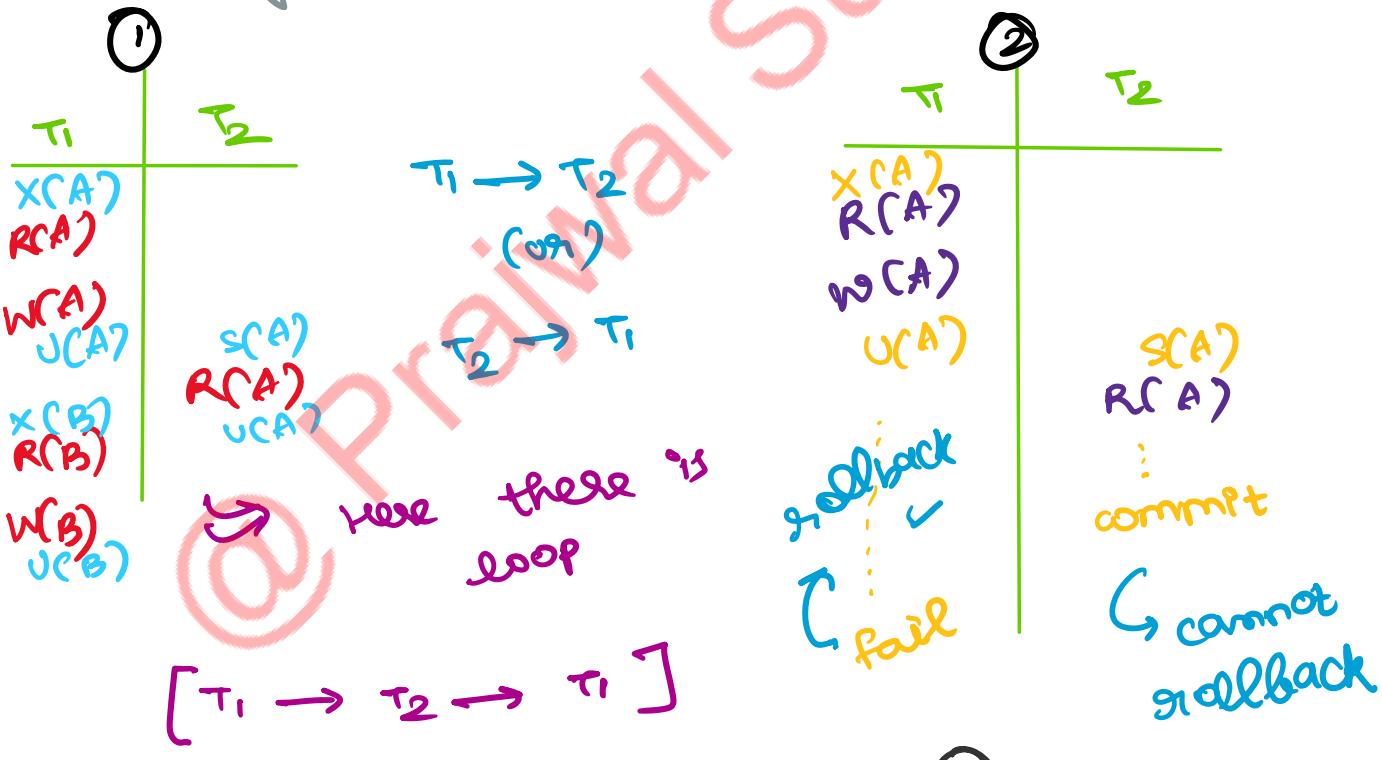
Grant \ Request	S	X
S	Yes	NO
X	NO	NO

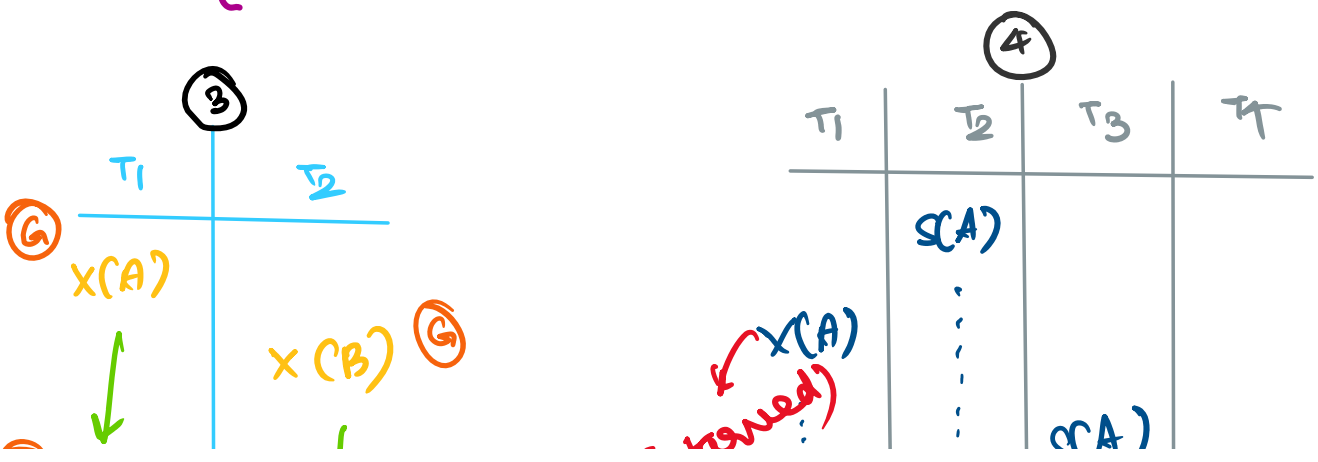
DRAWBACKS IN SHARED / EXCLUSIVE LOCKING PROTOCOL

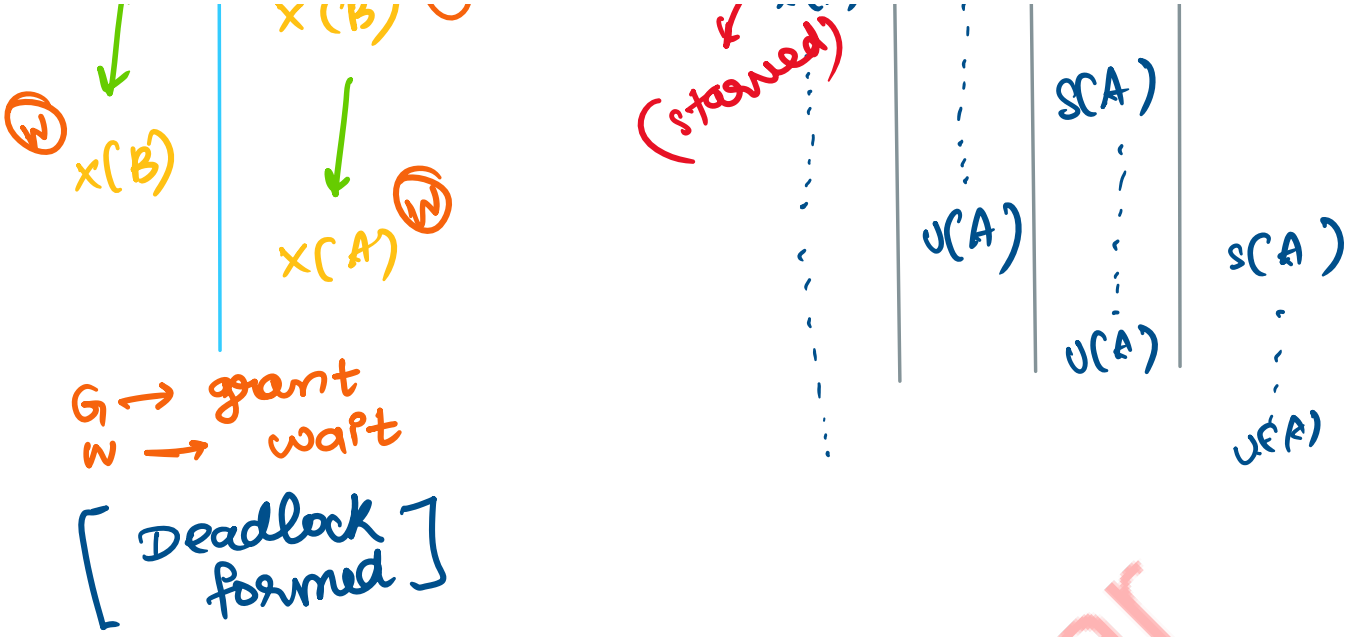
04 November 2023 11:00

Problems in S/X Locking :

1. May not be sufficient to produce only serializable schedule.
2. May not be free from irrecoverability.
3. May not be free from deadlock.
4. May not be free from starvation.





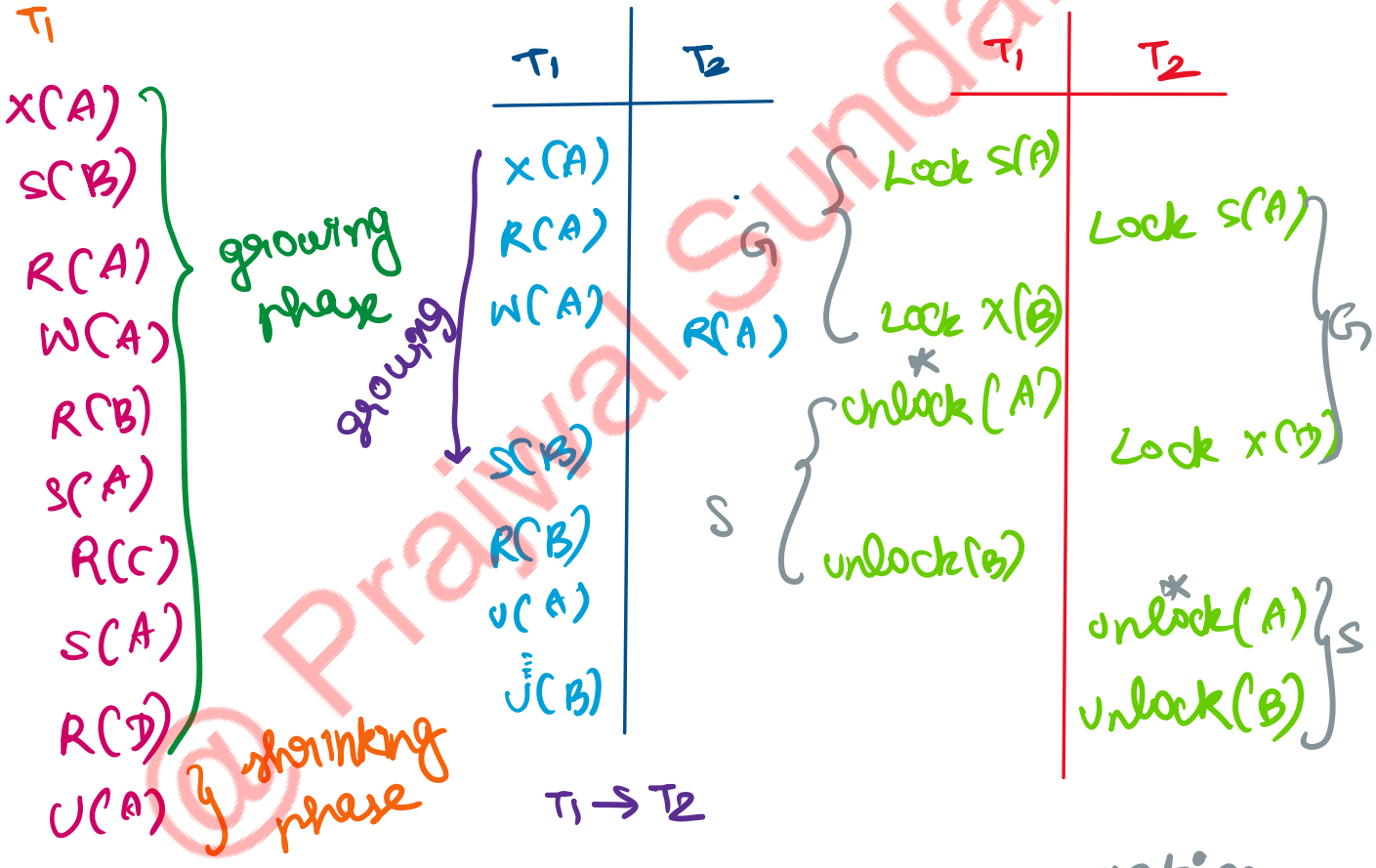


2 PHASE LOCKING

04 November 2023 11:14

Growing phase
↳ Locks are acquired and no locks are released.

shrinking phase
↳ Locks are released and no locks are acquired.



Lock point : Point in a transaction where the first unlock is done.

DRAWBACKS IN 2 PHASE LOCKING PROTOCOL

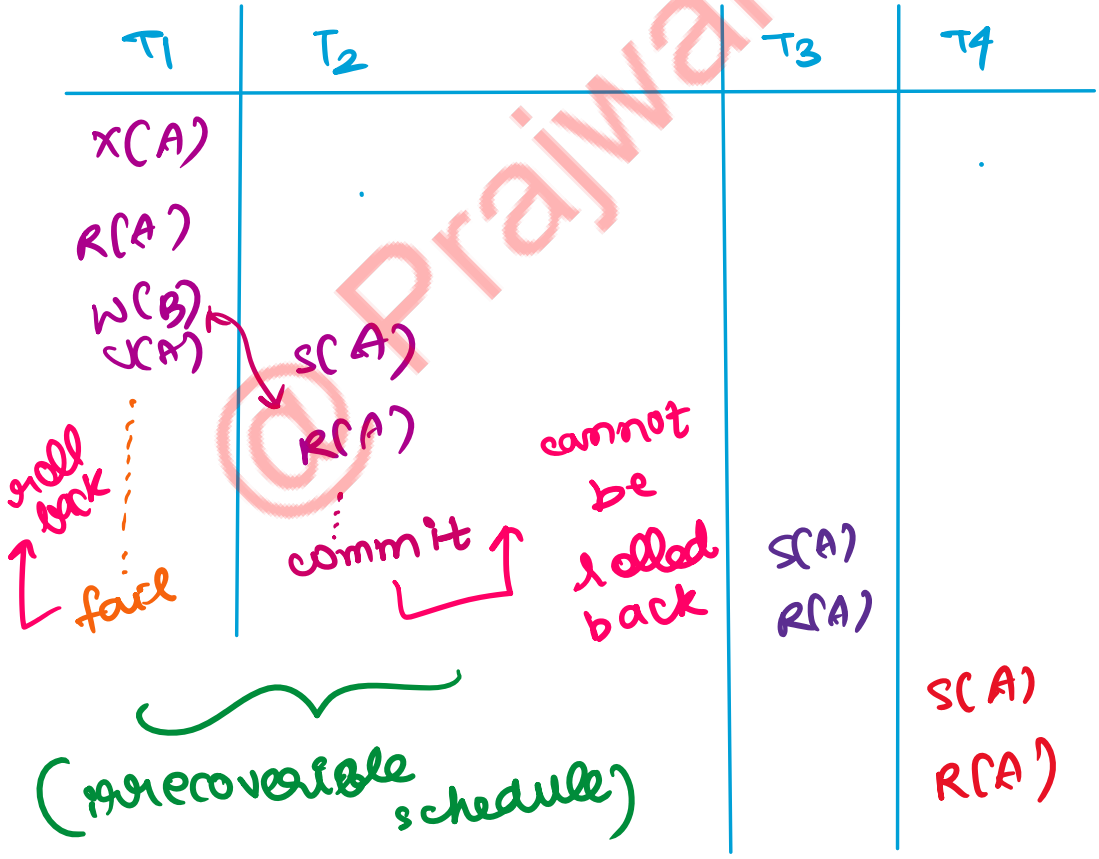
04 November 2023 18:39

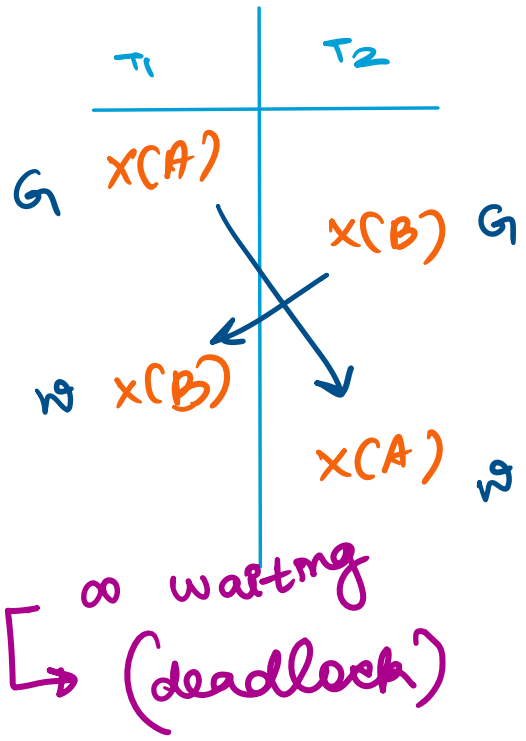
Advantages :

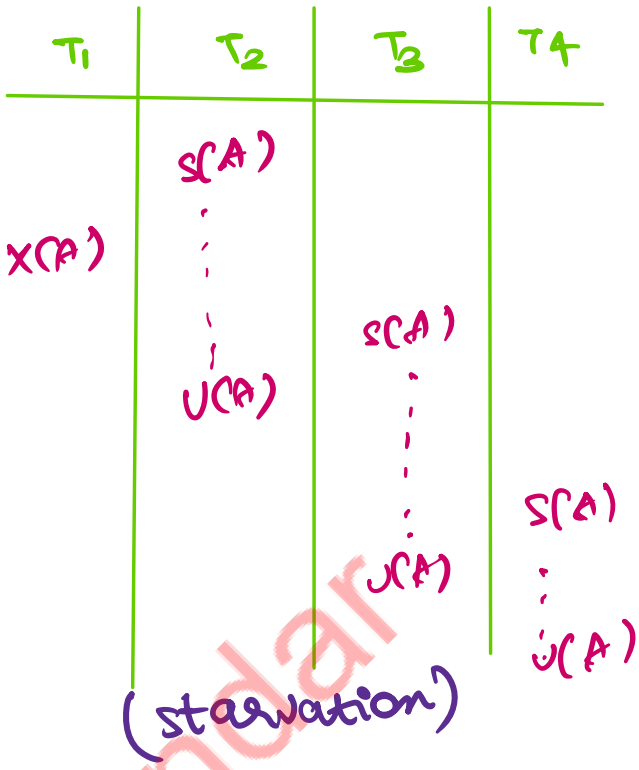
↳ Always ensures serializability

Disadvantages :

1. May not be free from irrecoverability.
2. Not free from deadlocks.
3. Not free from starvation.
4. not free from cascading rollback.







STRICT 2PL, RIGOROUS 2PL AND CONSERVATIVE 2PL

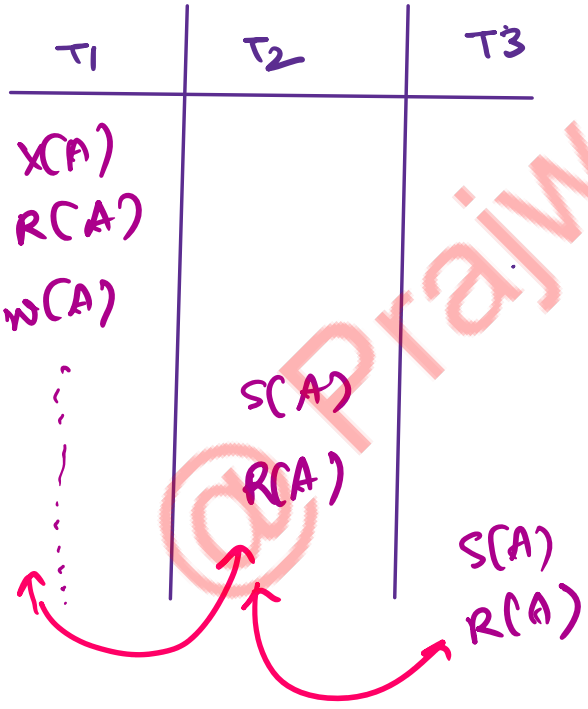
04 November 2023 19:29

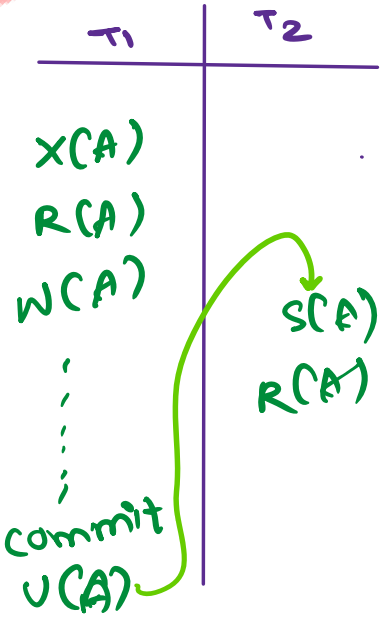
Strict 2PL

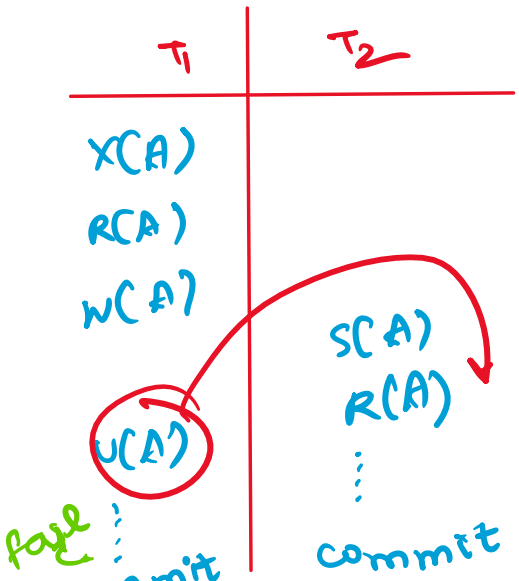
↳ It should satisfy the basic 2PL and all exclusive locks should hold until commit / abort.

Rigorous 2PL

↳ It should satisfy the basic 2PL and all shared, exclusive locks should hold until commit / abort.





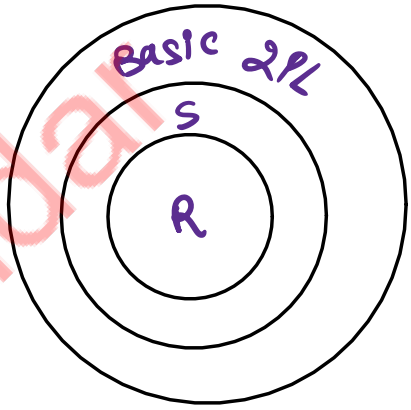


causes problem

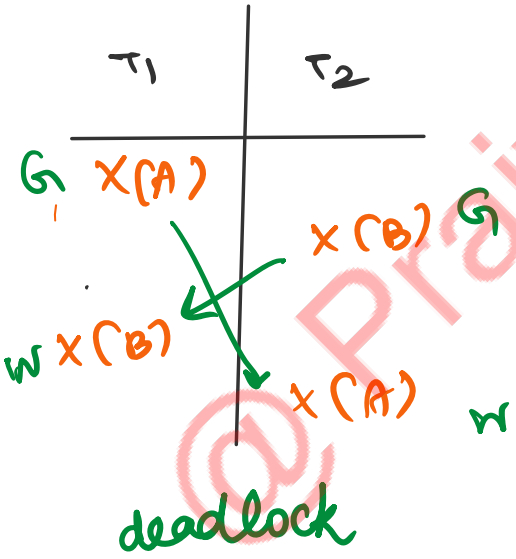
soln: unlock only after commit

Advantages:

1. cascadeless
2. strict recoverable



S → strict
R → rigid



conservative 2PL

↳ when a transaction begins, it should take all locks before starting.
[not practically possible]

TIMESNAP ORDERING PROTOCOL

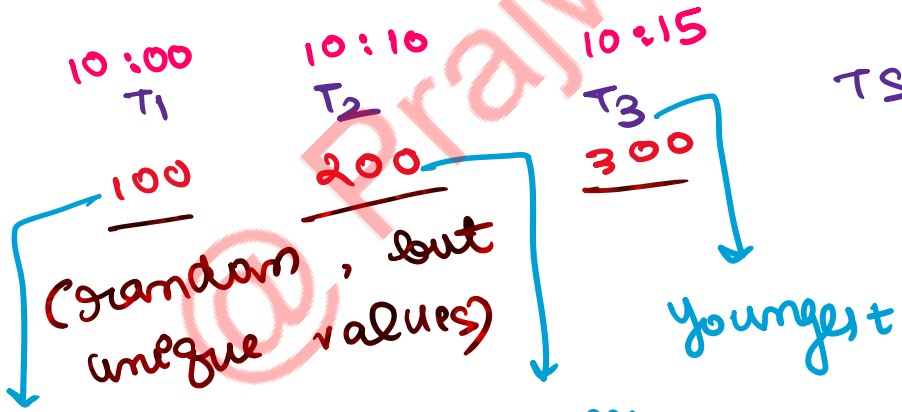
04 November 2023 20:05

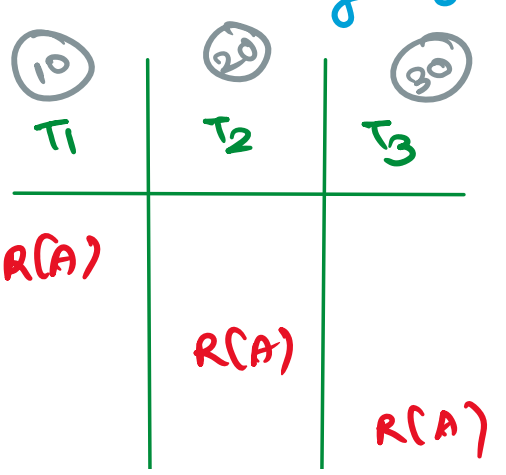
→ unique value assigned to every transaction

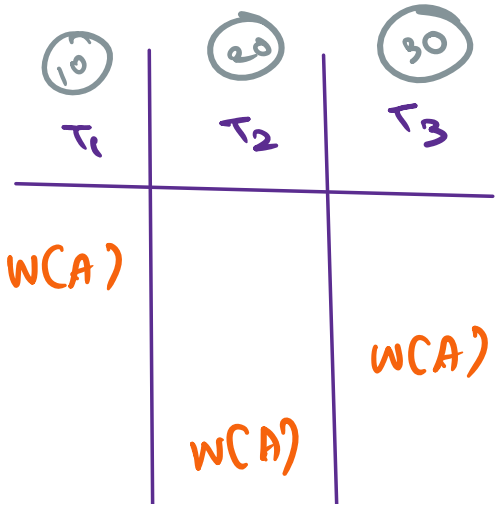
→ tells the order (when they enter into the system)

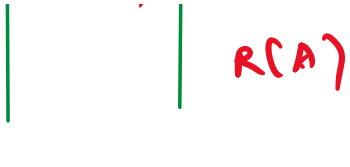
→ Read-TS (RTS) = Last (latest) transaction no. which performed read successfully.

→ Write-TS (WTS) = Last (latest) transaction no. which performed write successfully.









RTS(A) = 30

WTS(A) = 20

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), rollback T_i

b) if WTS(A) > TS(T_i), rollback T_i

c) otherwise execute write(A) operation

set WTS(A) = TS(T_i)