Concurrency Control Protocol

Aim: Achieve Serializability
- " Recoverability.

Earlier we studied to check if transaction schedule is serializable or not.
But in CCP, we study how to make the schedule serializable. (Is not to check)
↳ How to make a serializable, and recoverable schedule
How to achieve this ⇒ By using locking Protocol

① Shared exclusive locking
② Shared lock. - Read only
③ Exclusive lock - Read / write.

Eg

T_1	T_2
S(A) ← lock	X(A) ← lock
R(A)	R(A)
U(A) ← Unlock.	W(A)
	U(A) ← Unlock.

only read.

Both read & write.

grant \ Request	S	X
S	✓	X
X	X	X

① granted (S(A))
New Request S(A) → OK

② granted S(A)
Request - X(A) - No of Read/Write Conflict

Shared - Ex lock does not guarantee serializable always.

① May not sufficient to produce Serializable.

→ $T_1 \rightarrow T_2$

$T_2 \rightarrow T_1$

Ans ① $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$.

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

⇒

T_1	T_2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
X(B)	
R(B)	
W(B)	
U(B)	

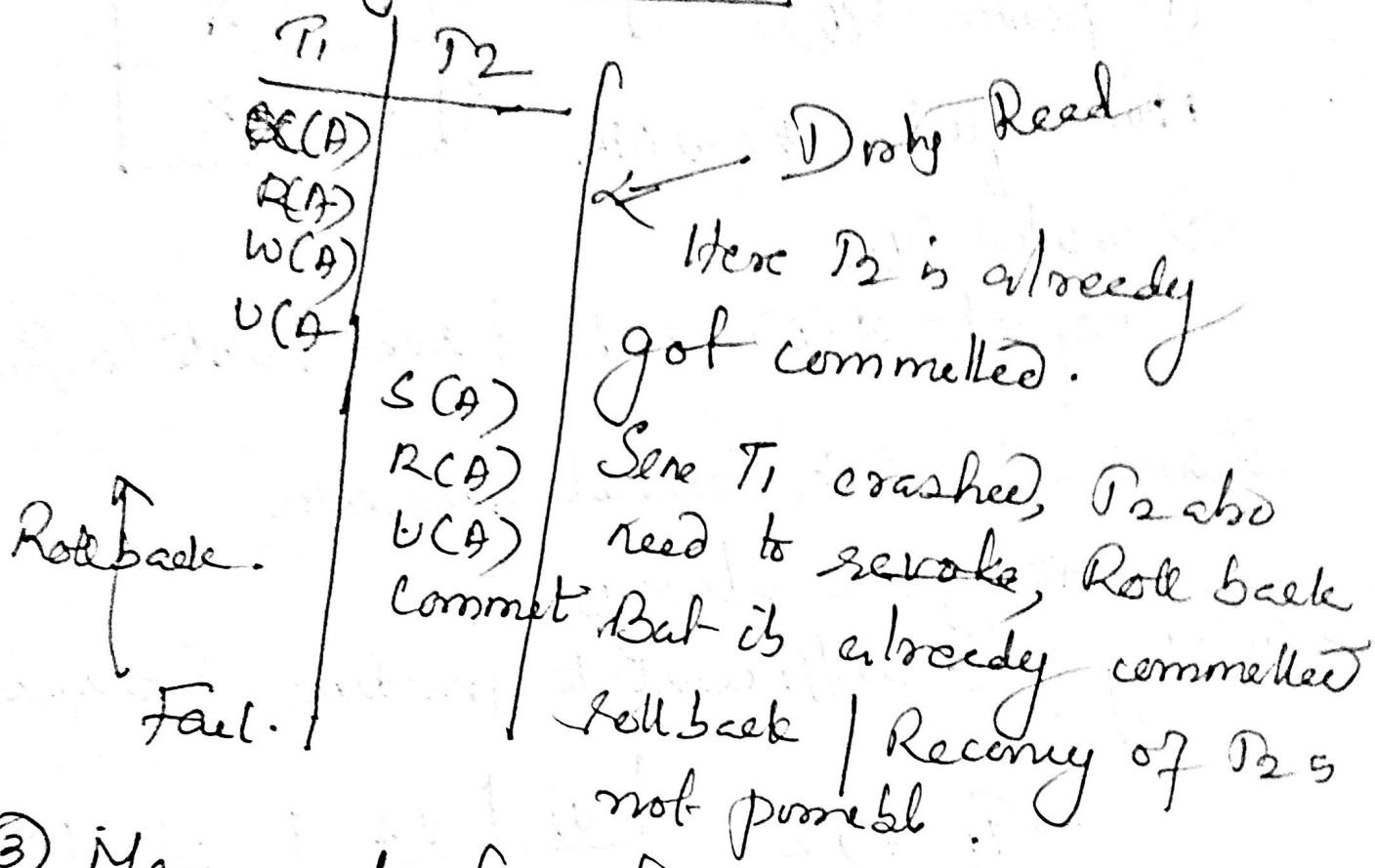
$T_1 \rightarrow T_2$

$T_2 \rightarrow T_1$

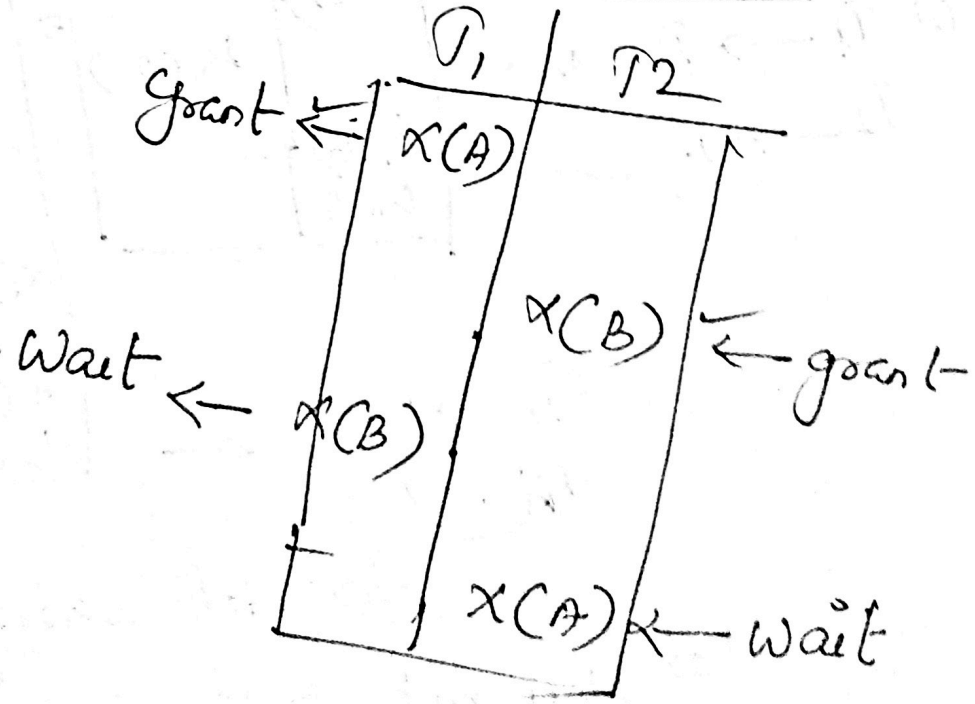
⇒ Not- serializable.

⇒ May not sufficient to Serializable

(2) Not always Recoverable.

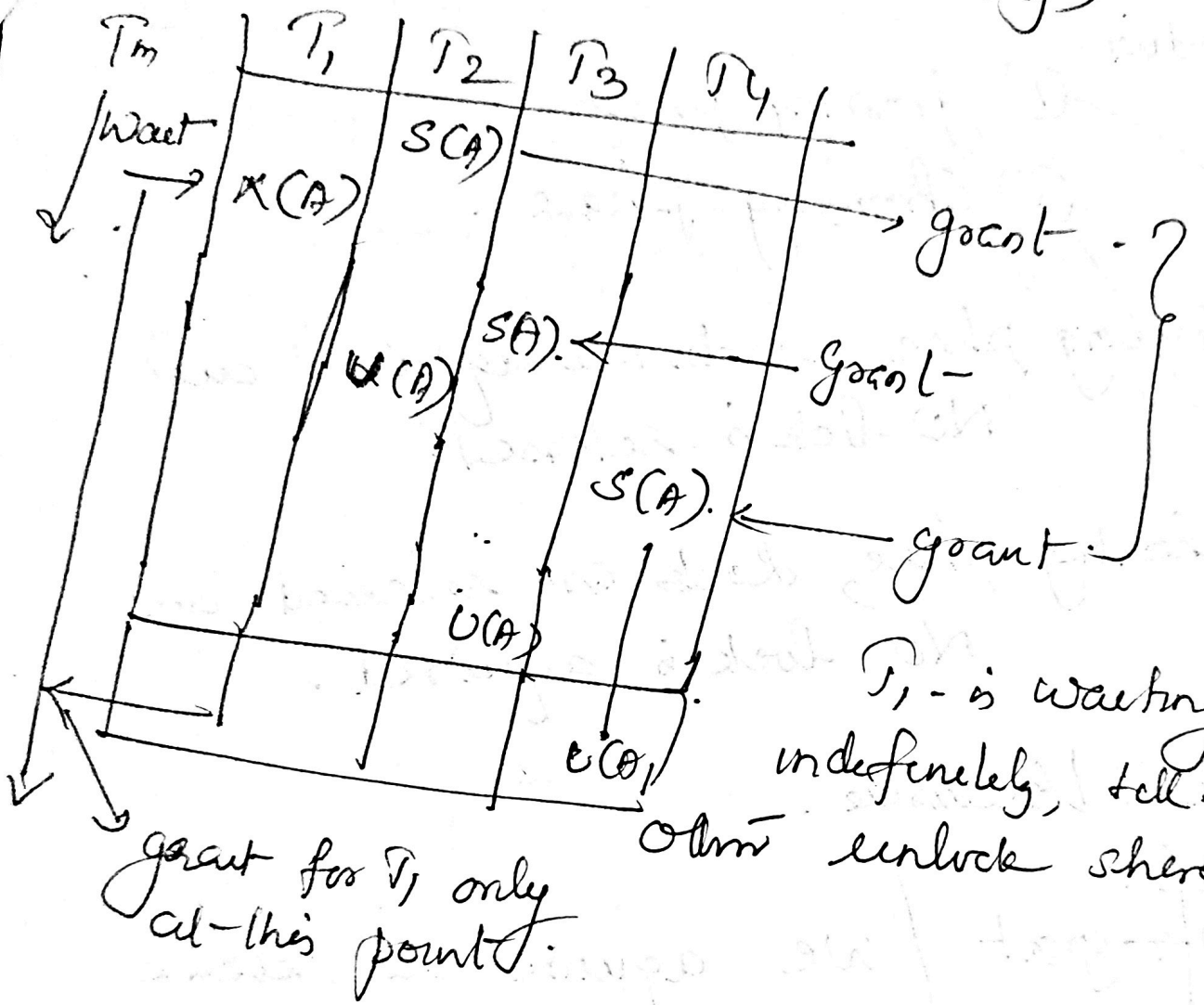


(3) May not free from Deadlock.



(Deadlock ⇒ infinite waiting)

⑤ May not free from starvation
(Not infinite waiting)



2 Phase Locking Protocol 2 PL.

Extension of Simple Shared / Exclusive locking Do

Introduce

(1) Growing phase
(2) Shrinking phase.

(1) Growing phase, locks are aquired and NO lock is released.

Shrinking phase, Locks are released and NO lock is aquired.

(2) Shared / Exclusive.

T1:

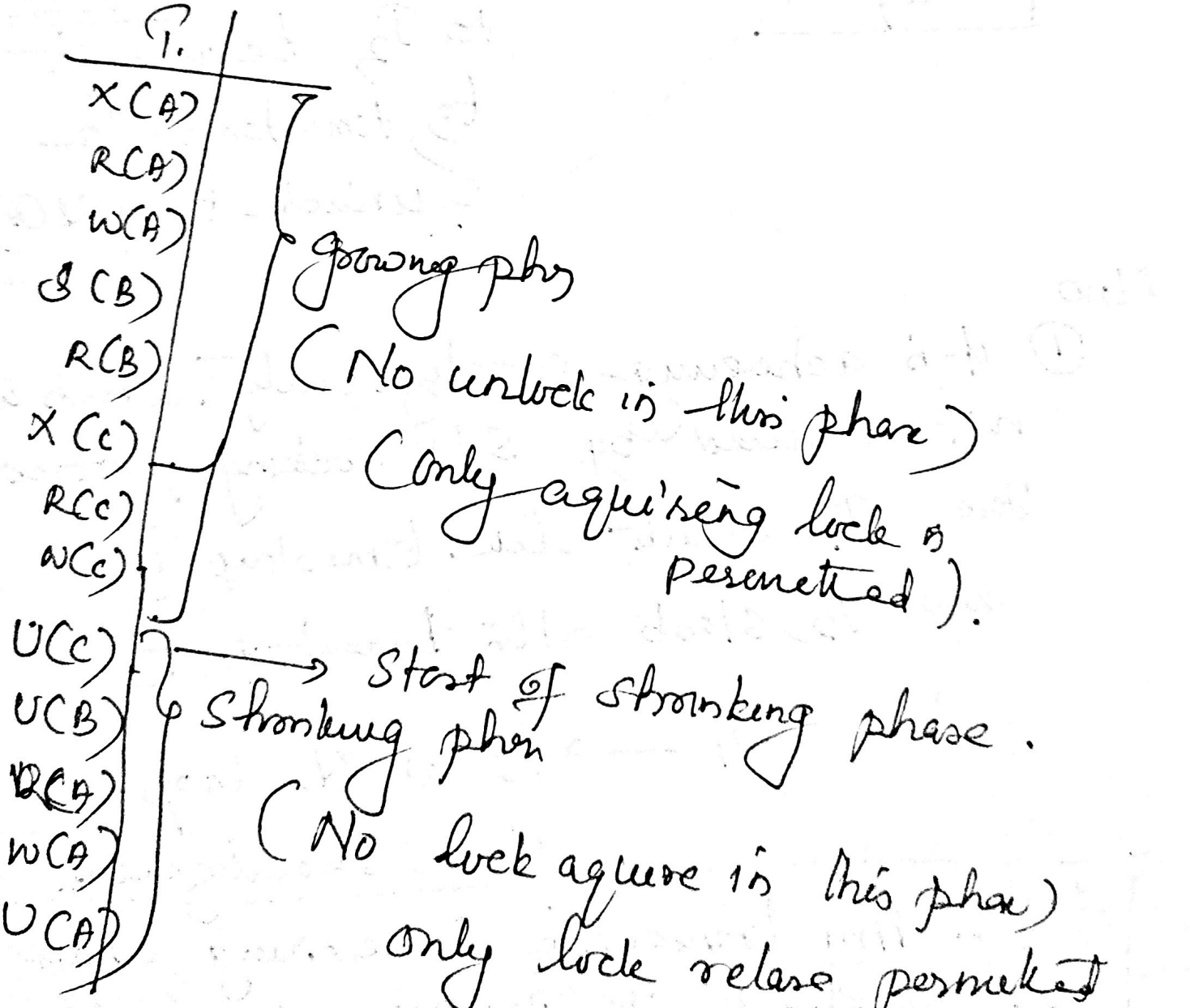
X(A)	← grant
R(A)	
W(A)	
S(B)	← grant
R(B)	
X(C)	← grant
R(C)	
W(C)	
U(A)	← Rel.
W(C)	
U(C)	

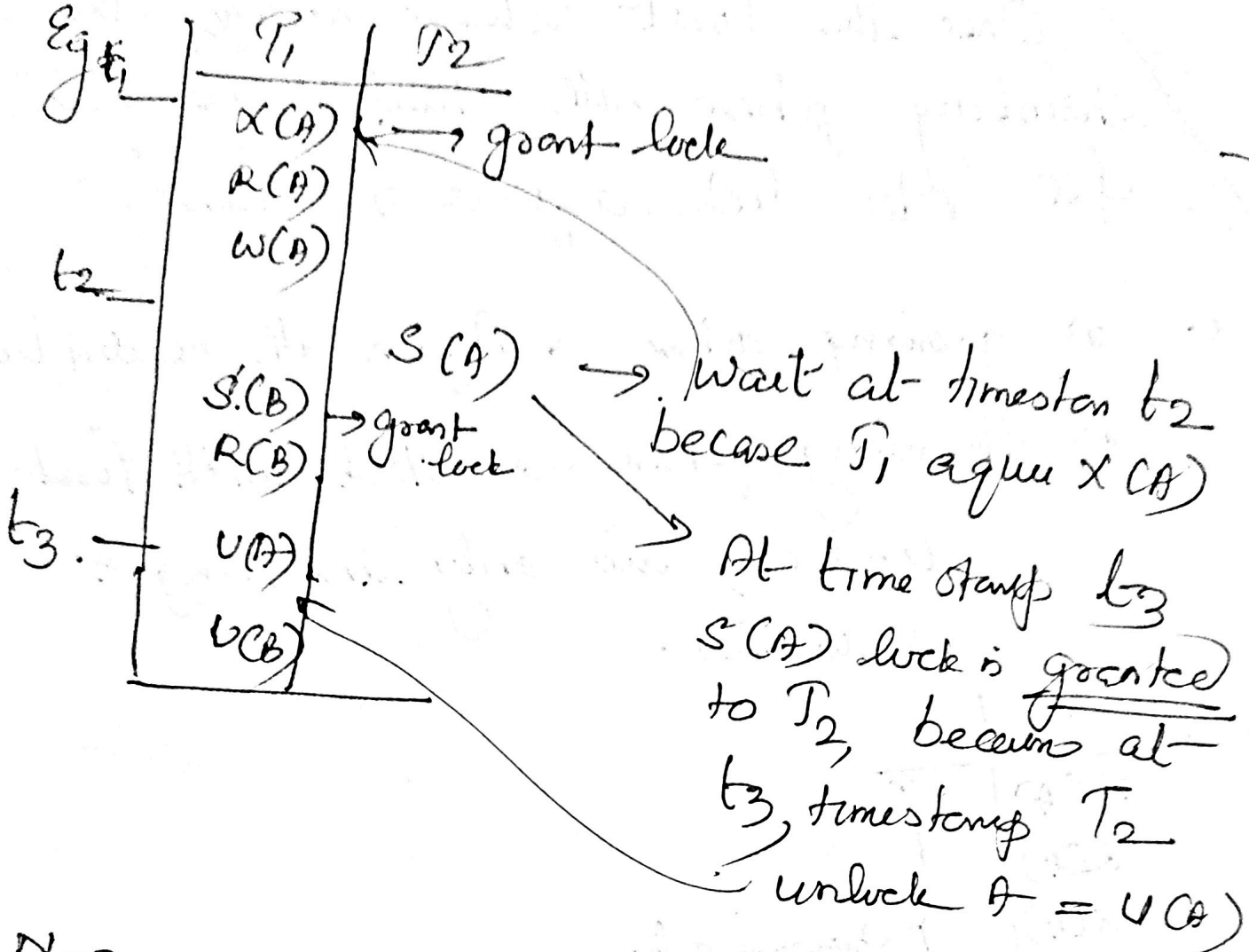
We aquire and release locks when ever needed. this continue late the end of transaction.

(1) 2PL, growing phase work grow as we continue aquiring locks. This phase ends when the first release occurs.

Once the first release occurs, the shrinking phase will start, and there after No lock aquire is allowed.

Q. in growing phase → Aquire all needed lock
in shrinking phase → Starts with first unlock, and only unlocking is permitted.





Now

① It is acheving serializability. coherb was not ensured by S/X locking protocol.

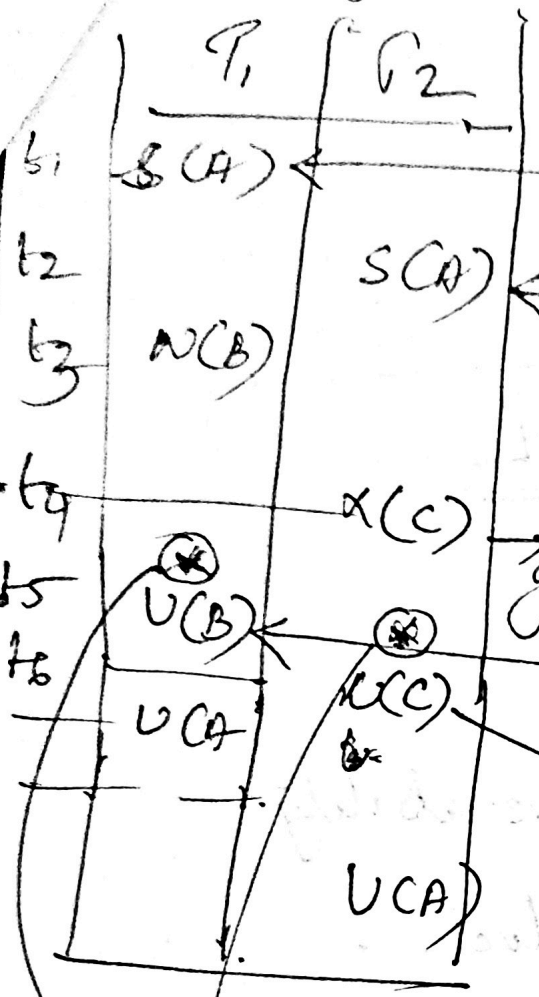
Here T_1 execute like timestamp t_3 and T_2 starts after timestamp t_3

$T_1 \longrightarrow T_2$ ⇒ No loop.

Hence serializable.

Note - Any transaction is reerning under 2PL protocol, it ALWAYS serializable

Eg-2.



start of growing phase of T_2

granted, because S(A) can be given over S(A) of T_1
⇒ Here growing phase of T_2 started at timestamp t_2

start of shrinking phase of T_1.

Start of shrinking phase of T_2

Here is the lock point of T_1, i.e the time when shrinking phase starts.

Here is the lock point of T_2, i.e the time when shrinking phase of T_2 starts.

~~AJ~~ S

Devdutt 24

~~Ajay PA~~ 63

Madhav Rajan

Adv | and Limitation of 2PL

Adv :- Always serializable.

Limitation.

1. Not free from irrecoverability
2. Not free from deadlock.
3. Not free from starvation
4. Not free from Cascading Rollback

OR

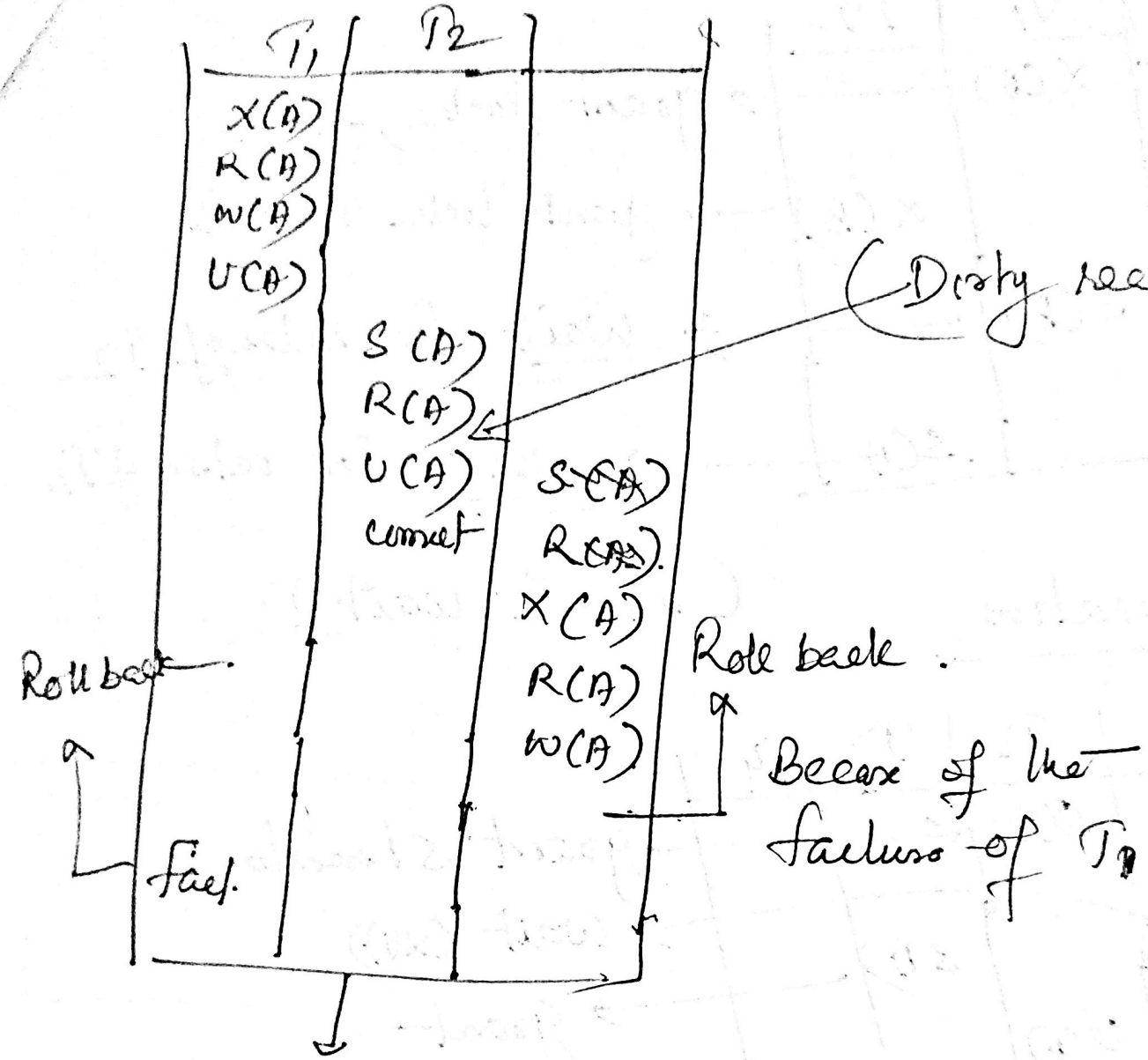
(1)

T_1	T_2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	S(A)
	R(A)
	U(A)
fail	commit

Rollback

Similar to S/E example.
T_2 commited first,
T_1 fail, T_1 rollback
But T_2 already commited so not able to Rollback.

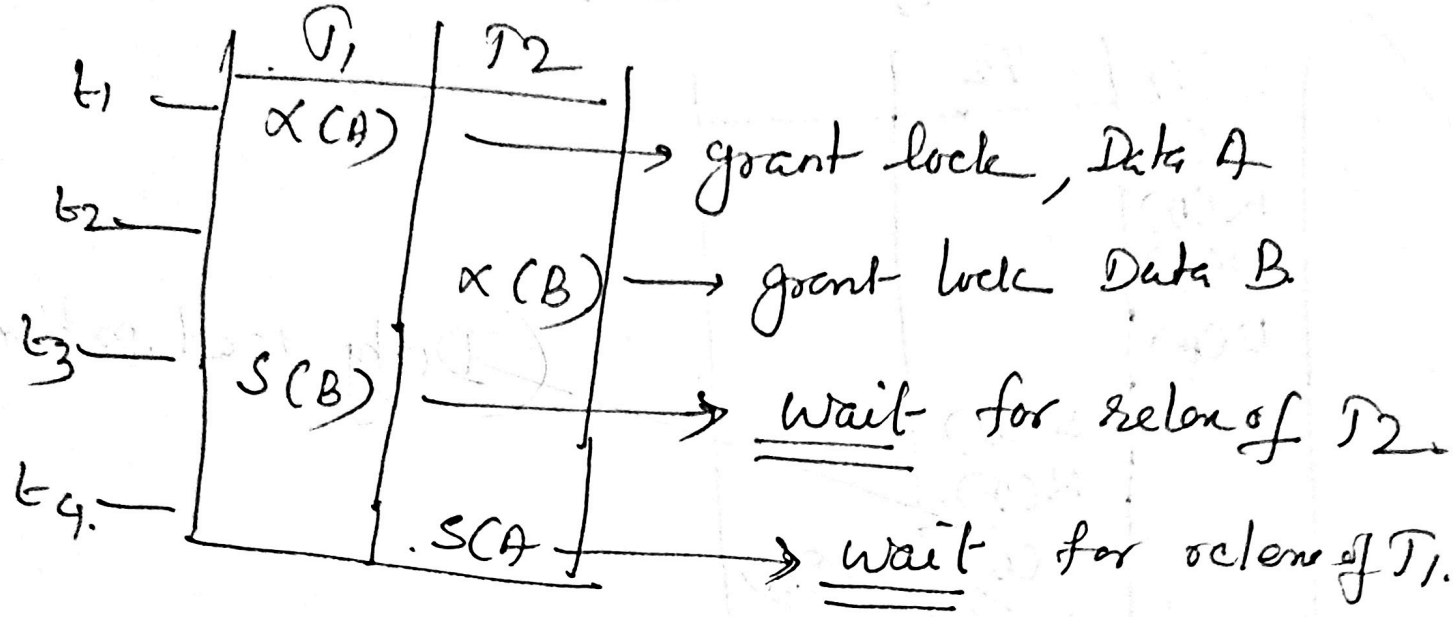
④ Cascading rollback



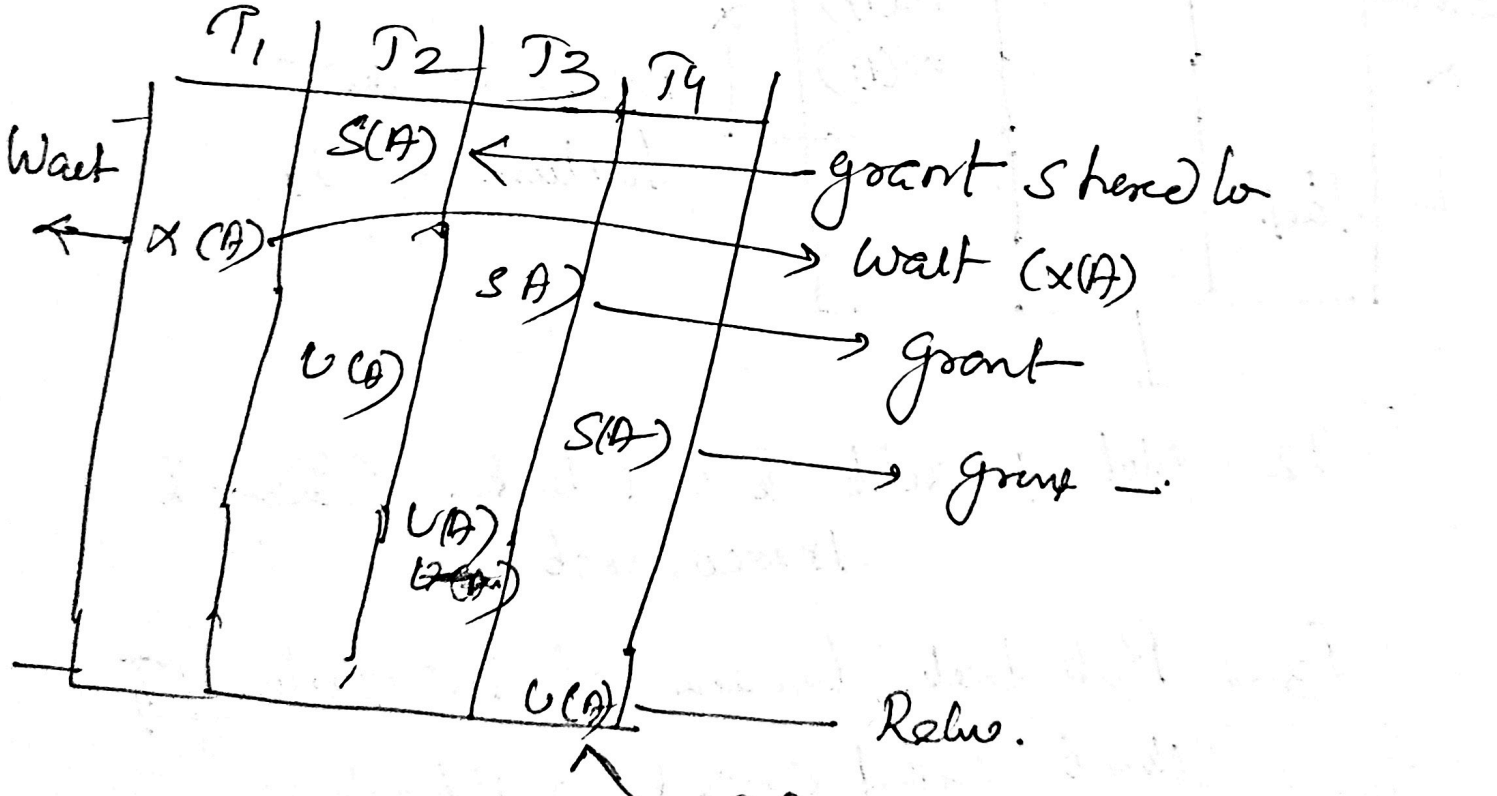
T_2 - Not possible to roll back (Dirty R Irrecoverable.

T_3 - Rollback because of the failure of T_1 This is called cascade rollback.

③ Dead lock.



④ Starvation (infinite wait).



a T1 X(A) wait till T4 relase A.

Strict 2PL and Rigorous 2PL

Basic 2PL has 4 demerits.

1. Irrecovarability
2. Cascading roll back
3. Deadlock
4. Starvation

Strict 2PL fix (1) and (2)

Rig. To achieve this, certain restrictions are added to Basic 2PL. That is

> for Strict-2PL ⇒ it should satisfy Basic 2PL and all exclusive locks should hold until commit / Abort.

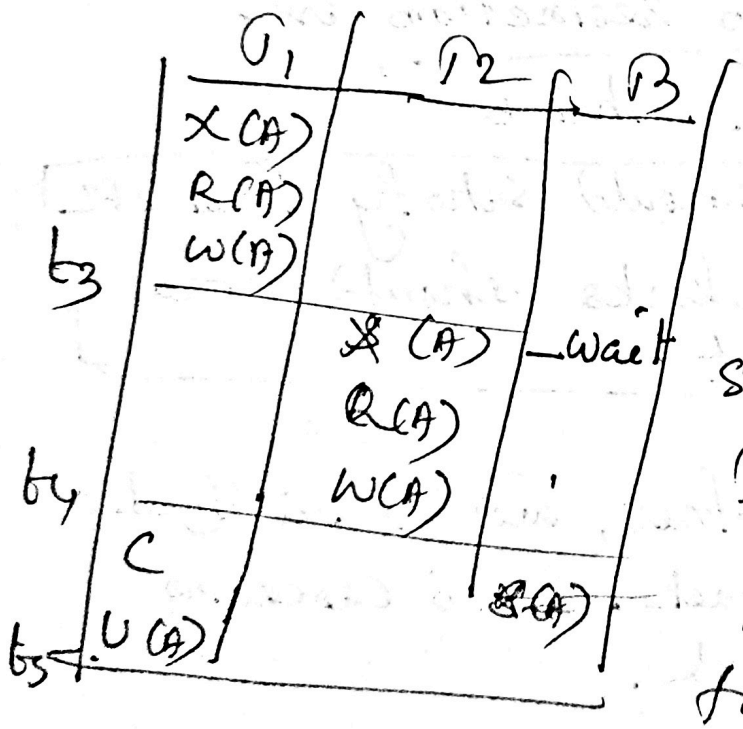
T_1	T_2	T_3	
X(A)			
R(A)			
W(A)			
	S(A)		wait
	R(A)		→ At this point T_2 S(A) is granted. This also can T_3 (S(A) granted.
Rollback U(A)		S(A)	→ granted
fail..		R(A)	

T_1 fails, and T_2 and T_3 also roll back. This is cascading roll back.

If T_1 before t_4 after commit then T_2.

Here T_2 and T_3 are not reading from DB because, T_1 has not commited yet.

⇒ If T_1 unlock after commit, T_2 and T_3 will read from DB, and can be. rollback undone rollback is not needed because of T_1 failure.



T_1 commit at timestamp t_4, follow by U(A) so at timestamp t_5 T_2 X(A) is granted

Now T_2 is reading from DB not, that's not the value written by T_1 in log.

Hence no need to rollback for T_2. Thus eliminating Cascade rollback problem.

Remove irrecoverability

Since T_1 unlock will happen at commit or abort. So T_2 lock granted after T_1 commit.

∴ The first one to write commit first. This remove irrecoverability.

So Strict 2PL are,

① Always cascadeless.

② Recoverable.

Strict 2PL = Base 2PL + hold exclusive lock until commit / Abort

---

Rigorous 2PL [Limitation - Deadlock, Starvation]

① Satisfy Strict 2PL + Hold shared lock until transaction commit / Abort

[That is the restriction is imposed on shared lock also]

Rigorous 2PL

(1) Must support Strict 2PL

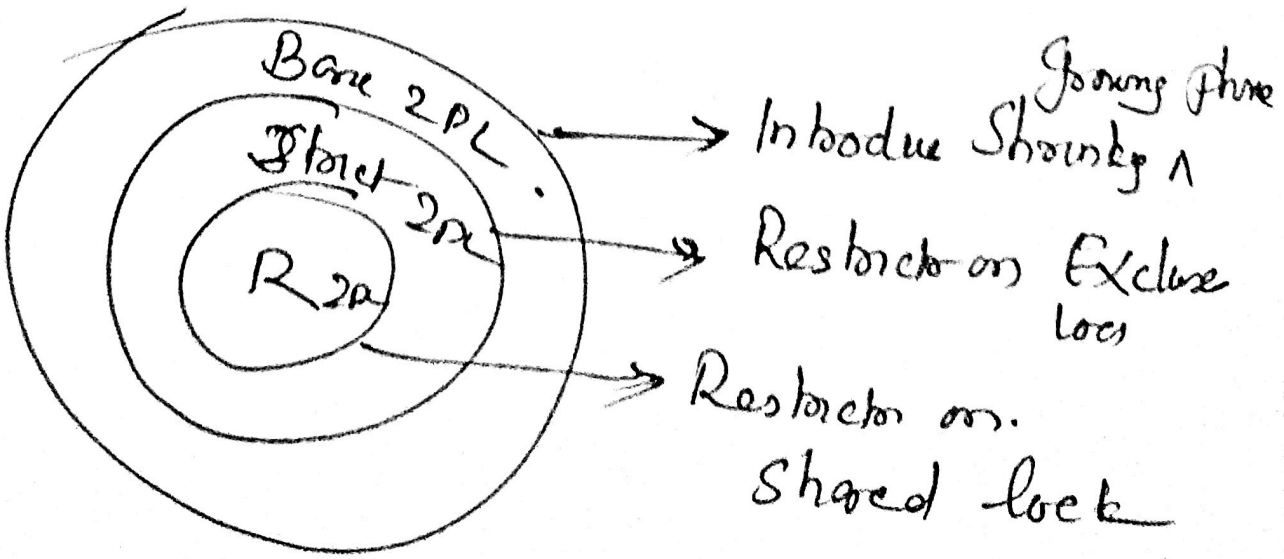
(2) Restriction applied to shared lock also shared lock is also released after commit

This does not answer. fix

(1) Deadlock (2) starvation

This fixes.

(1) Irrecoverability (2) Cascade Rollback

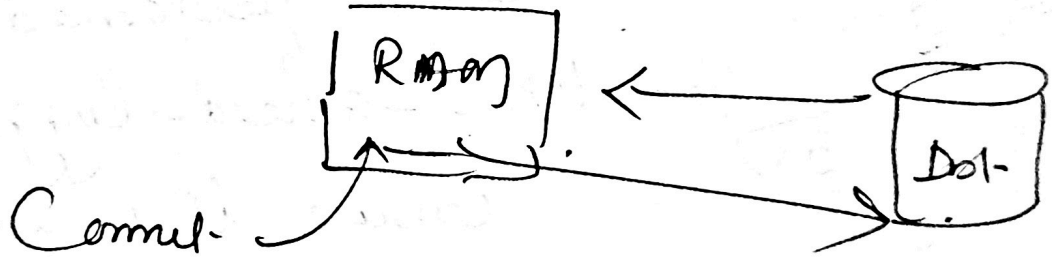


Failure classifications:

① → Logical Errors / Transaction failure.
→ Transaction can not proceed because of bad input, or overflow etc.
→ May be re-execute

② → System crash.
→ H/W malfunction
→ loss of RAM content,

Transaction Recovery



Data lose does not occur

③ Disk failure.
→ Loss of Data
Ex- Multiple copies of data in Backup disk.

To ensure consistency and atomicity where there will be an algorithm to recover from failure.

How to Recover from failure.

Using an algorithm, run on failure case to perform recovery

This Algo will have 2 parts.

① Actions at normal transaction
② Actions at failure (After failure)

① Those actions included

① — logging - Maintain a log that records the sequence of actions performed in transaction

② — Write-ahead-logging - (WAL)
Ensure that all log records associated with a transaction are written into stable storage (non volatile) before that page is updated into Disk.
This ensure log records are durable on failure.

③ — Checkpointing → Create checkpoints in log. Which will act as reference for recovery.

During checkpoint, DBMS writes all data pages to disk, and mark the reference on checkpoint in log

② Actions at failure...

① – Analysis – Analyse the log

② – Redo – Redo transactions that are committed but not updated to Disk

③ – Undo. Transactions there are not commited are rollbacked. (Partially completed transactions are rollbacked to ensure consistancy).

– Restart – After redo and undo DBMS ~~restart~~ restarts the system & resume.

So combining ① Actions at transaction and ② Actions after failure, a recovery algorithm ensure database to restore to a consistant state.

Log Based Recovery

Eg - I transaction to withdraw 5000, and transaction failed at the end state.
My net balance should be not be affected.
The recovery is based on logs.
The type of log records are,

$\langle T_i$ start$\rangle$ T_i started.

$\langle T_i, X_j, V_1, V_2\rangle$ T_i performed write on data X_j. The write val. value of X_j before transaction V_1 value after transaction V_2.

$\langle T_i$ Commit$\rangle$

$\langle T_i$ Abort$\rangle$

(The purpose is to find the old value for roll back.
The older value $\Rightarrow V_1$
The new value V_2.

If transaction fails X_j will be rollback to V_1

Database Modifications

① log is created prior to DB modifications

② Log Record enable below on Recovery

(1) Redo ⇒ Commited but not updated to DB. Redo that transaction

(2) Undo ⇒ Not commited. (Partially commited / Completed)

This enforce consistency.

Undo Transactions performed when Tr is aborted / failed, and there are not commited

Redo Transactions, there are commited but not updated into Disk

For both Redo, Undo, log is used to identify transaction, status, and values. (older values to rollback)

③ Once Redo / undo is performed, the data is updated to Disk.

Deferred Modification

Data is written into disk after committing transaction.
One transaction has generally one commit so deferred modification updates at the end of transaction. It will not write any intermediate result values.

Immediate Modifications

→ DB is updated in between transactions.
→ Not waiting for commit, or abort transaction before committing.

Check point - Checkpointing

Another type of entry is checkpoint. It is just like other records; It is an entry in log similar to other type of records we found earlier.
Checkpoint is a log indicate at that point, the system writes out all DBMS buffers that have been modified to disk.

So essentially at checkpoint all data is written into disk. It comes

Checkpoint comes periodically in "Immediate modification"

It is really a checkpoint record, rather than a reference in log file.

The checkpoint record holds Transaction IDs of active transactions, those have done database write at that point. So using checkpoint, system can identify the transactions during recovery.

<Checkpoint- T_1 T_2 T_2 ... T_3>

Transaction IDs.

Sequence of Actions in Checkpointing:

Steps performing during checkpointing.

Checkpointing → is a process of taking checkpoints by DBMS at specific intervals or when specific conditions are met.

Checkpointing involve

① Determine checkpoint timing.
(When to initiate checkpointing)
May be based on elapsed time, no of transactions processed etc.

② Trigger Checkpoint - Once conditions are met, DBMS initiate checkpoint process.

③ Execute checkpoint Actions -
→ flush data to disk
→ Write a checkpoint-record in log
→ Update checkpoint information.
(Transaction id etc).

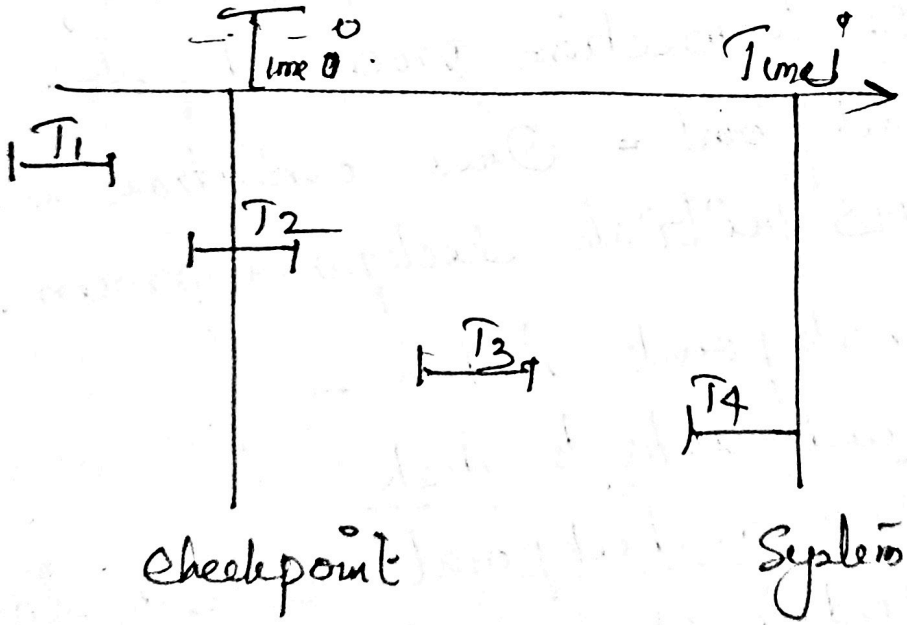
④ Resume normal operation.

[Checkpointing can be Time based, Transaction Based. etc]

④ Transaction are not allowed during checkpointing to perform any DB write or writing to log file during checkpointing

⑤ After system crash occurs, the DBMS system scans log file to locate the last checkpoint entry (<checkpoint>)

Recovery Procedure.



T4 - Started after checkpoint but not commited before failure.

T1 - Already commited, (Before checkpoint)

T2 - Commited but not updated

T3 - Commited but not updated

T4 - Not commited

T1 T2 T3 T4 are Transactions.

Recovery.

Redo → T_2 & T_3, Because commited but not updated to disk..

Update to disk is done at next checkpoint, which is expected to be taken by DBMS

→ So redo T_2 and T_3

Undo → T_4, which is not commited.

i.e. Partially completed transcetons

So undo T_4.

T_1 → Can be ignored, as it is commited before checkpoint, and updated to disk at the given checkpoint.

→ So it can be ignored.

Algo: During recovery need to consider only the most recent transactions, that started before checkpoint

① Scan backwards from failure to find the last checkpoint. (Most recent checkpoint)

<checkpoint, L>

② Continue scanning backward to get <T_i start> in log.

③ Need only consider transaction started before check point

④ for all transaction with NO <Commit> execute undo(T_i)

⑤ All transaction with a <T_i Commit> execute redo(T_i)

These are the cases for immediate modification where check disk update is done periodically.

Next Recovery based on deferred update.

Deferred Modification → Data is updated to Disk after the final commit.

↳ Deferred update postpone any actual update to the database on disk until the transaction commits.

↳ After commit, the value in log is written to DB.

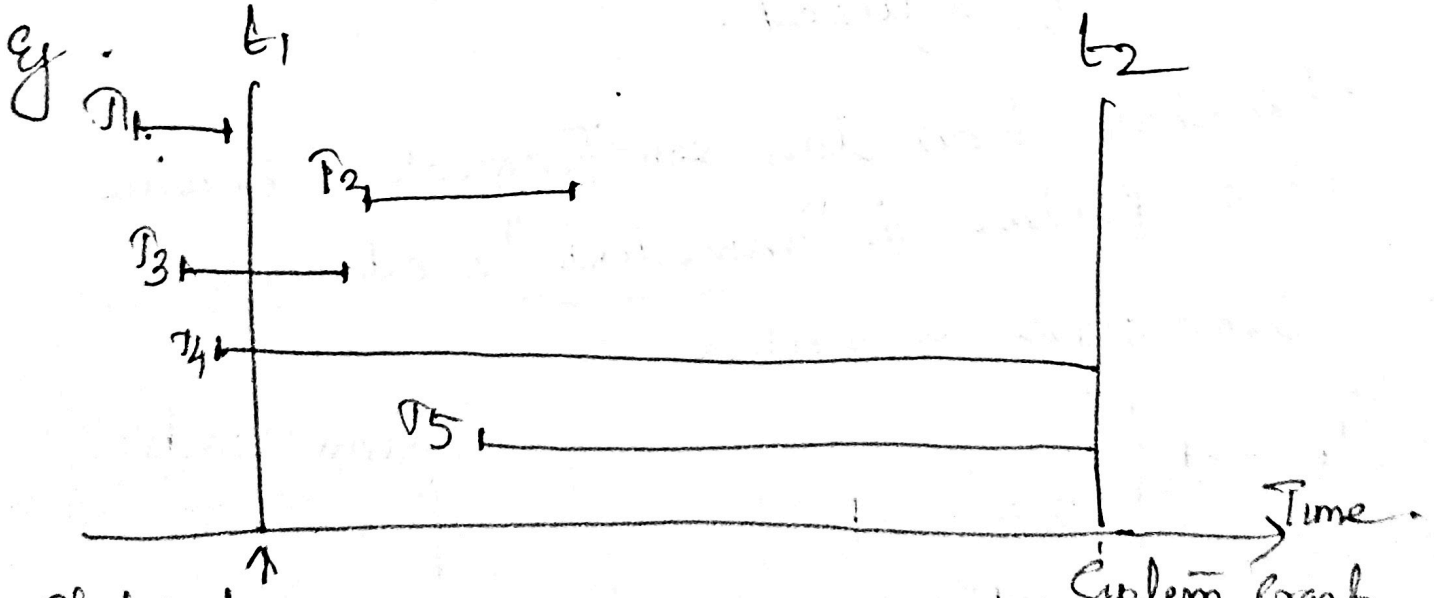
Here UNDO is NOT required, because the transaction failed before it commits and write data into DB in Disk. Since no dB update on Disk is performed, nothing to roll back or undo.

Nexet case → when checkpoint is reached.
DBMS write all data into DB in Desk.

Worst Case → Crashes before checkpoint.
Ideally all commelled transactions write data into DB in Desk at checkpoint.

So if crashes before checkpoint, REDO all commelled transactions, to be precex all WRITE operations of all commelled transcetom.

There can be uncommelled transcetom, They are effectively cancelled and must be resubmelle or re executed.



Some Transcat Commelled and Taken checkepoint to write DB

T_1 → Commelled before checkpoint
T_2, T_3 → Also Commelled but not written to DB in Desk
T_4, T_5 → Active Transaction, that is not commelled.

Actions

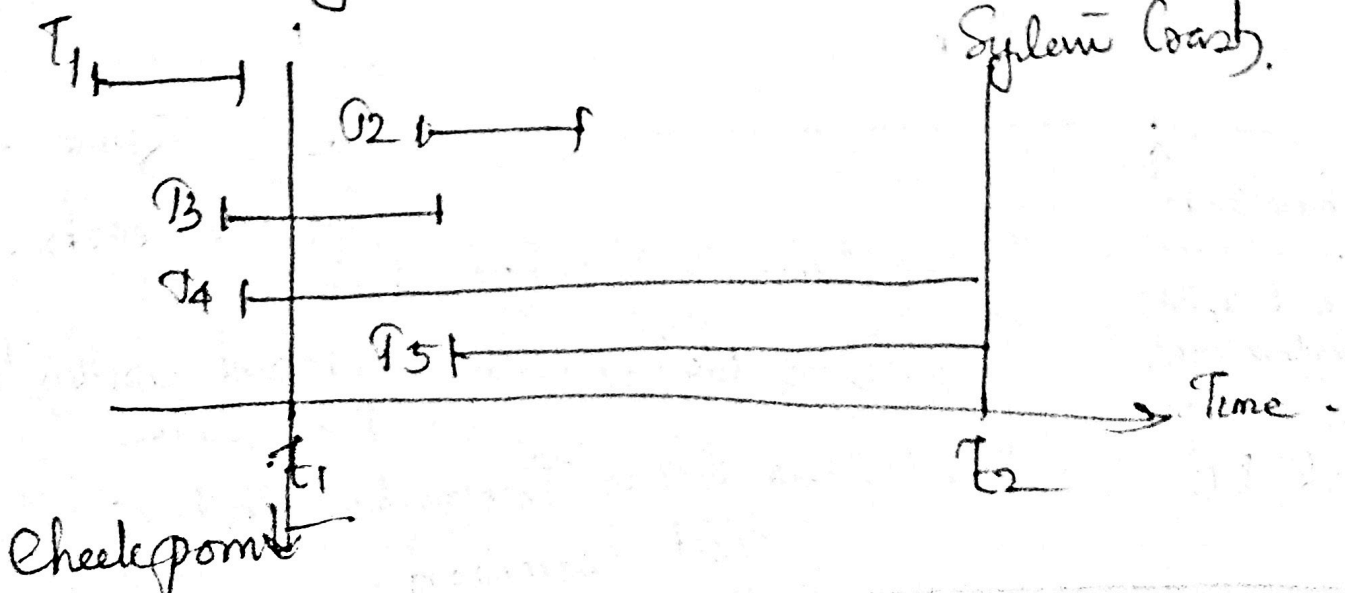
T_1 → Already committed and written to Disk at checkpoint.

T_2, T_3 → Committed after checkpoint but not written to DB in disk. Before that the system crashed. So need to REDO.

T_4, T_5 → Not committed, active transactions. Since not committed, database is not updated.. Hence ignore. So it may be resubmitted or re-executed if required.

Now check how this ~~sch~~ Transactions behave on a failure in "immediate" modification.

Same figure as earlier



T_1 : Already committed before checkpoint, and at checkpoint update DB in Disk.

T_2 : Committed after checkpoint, but not written data into DB in Disk.

T_3 : Started before checkpoint, and at checkpoint, some write operations may have written to DB in Disk

T_4 : Similar to T_3, but T_4 is not committed.

T_5 : ~~Active. Too~~ Not committed, (The Nothing is written to disk because started after checkpoint.

T_1 : Ignore.

T_2 : Redo since it is committed.

T_3 : Redo " " "

T_4 and T_5 :- Not committed, Before commit, it failed, Undo (Rollback)