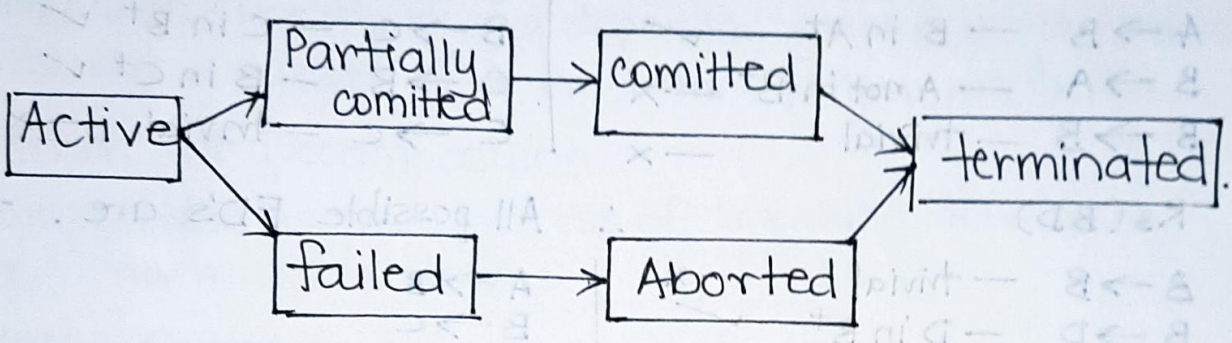
DATABASE TRANSACTION MANAGEMENT

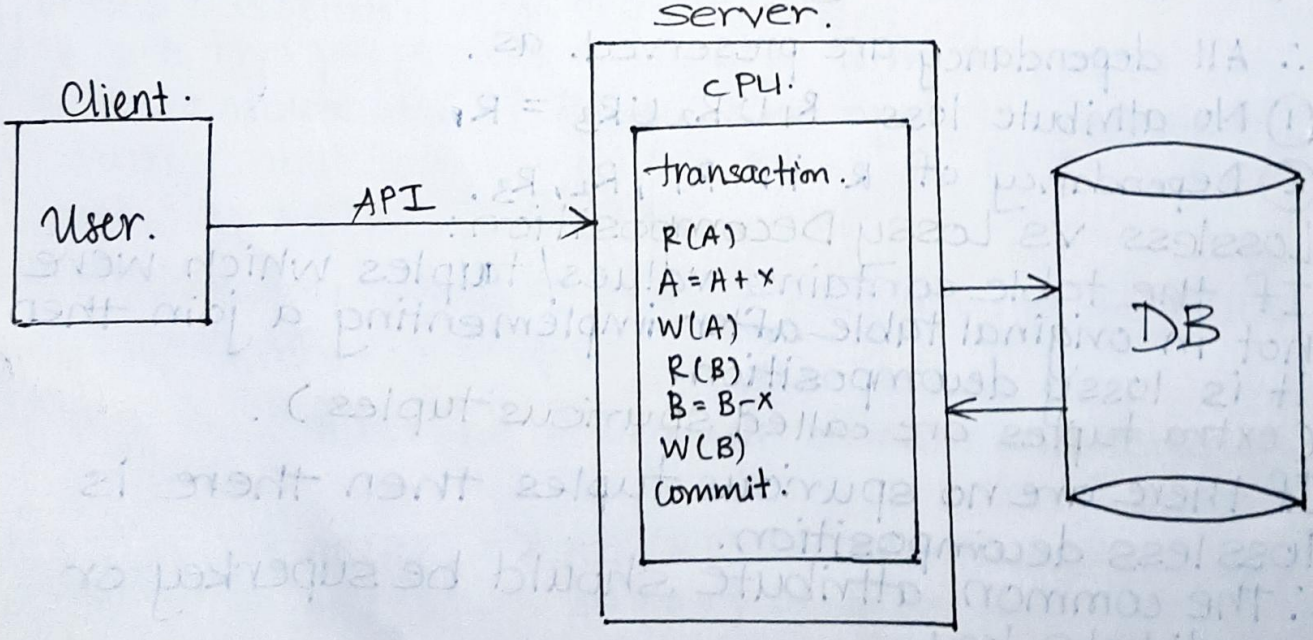
Introduction:

A DB transaction is a set of operation that execute as a single unit. ensuring data integrity.
Transaction should follow ACID property to maintain correctness even in case of system failure or crash

Transaction States



Active : Transaction is currently executing.
Partially Comitted : Execution is complete but not yet comitted.
Comitted : All operations completed successfully and data is committed.
Failed : Transaction cannot proceed.
Aborted : Transaction is rolled back and no changes are applied.



ACID Properties

- **Atomicity** : All actions in a transaction must be complete or none.
- **Consistency** : The DB transitions from one valid state to another
- **Isolation** : Concurrent transaction do not affect each other.
- **Durability** : Once committed, change are permanent. even after failure.

CONCEPT OF SCHEDULE

A schedule is an ordered sequence of operations from multiple transaction.

It ensures that transaction execution maintains database consistency, and the operations are executed in a way that prevent conflicts and anomalies.

- **SERIAL SCHEDULE**

A serial schedule is a transaction schedule where transaction are executed one after the other without interleaving. This ensures there is no conflicts or inconsistency, and the outcome is predictable, reflecting what would happen in a purely sequencial execution. example: ATM

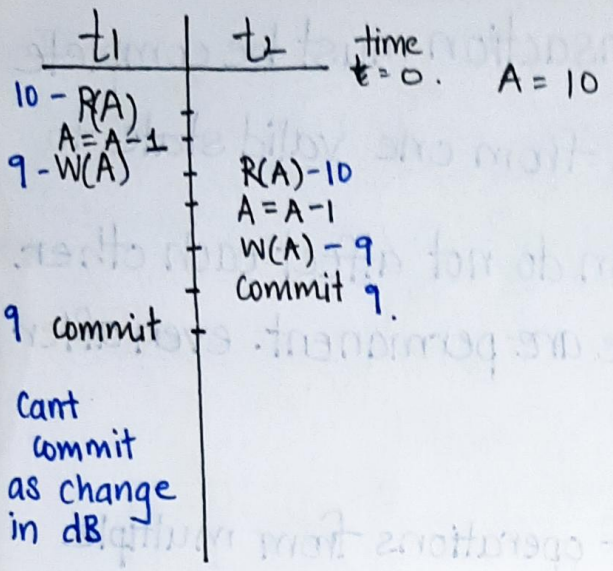
- **PARALLEL SCHEDULE**

A parallel schedule is a transaction schedule where transaction are executed in parallel with task switching. This increases Performance & decreses waittime example: Banking APP

Serial schedule	Parallel schedule.
$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots T_n$.	$T_1 \rightarrow$ $T_2 \rightarrow$ $T_3 \rightarrow$ $\vdots$ $T_n \rightarrow$

time

Read write Conflict or Unrepeatable read



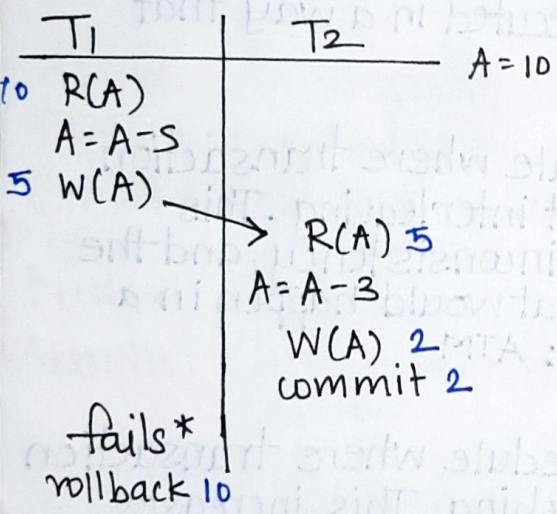
As in the example.

T_1 reaches partially commited state at time t=3

but T_2 executes at t=3 and reaches commited state at t=6 thus making change in DB

Hence the T_1 transaction is now a wrong transaction and needs to be rolled back. as final state of DB has A=9 when logically it should be A=8.

Irrecoverable Schedule.



As in example.

T1 executes and writes A as 5

T2 reads this value and executes then commits and leaves.

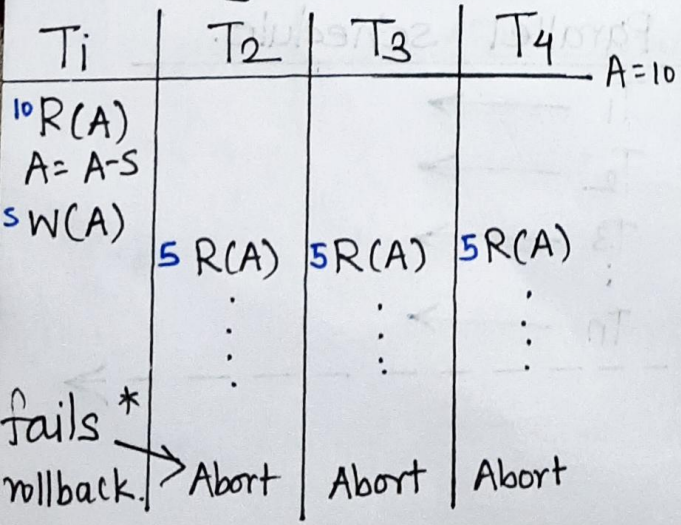
but T1 doesnt complete its task and fails

Thus we roll back T1

T2 commited to A as 2 but due to rollback A is now 10

thus task T2 becomes non existent and irrecoverable.

Cascading Abort.



When reader writer problem occurs i.e T_2, T_3, T_4 read from T_1 and if T_1 fails then the read of T_2, T_3, T_4 would be of wrong data

Thus T_1 sends abort messages to T_2, T_3, T_4

This is called cascading aborts.

Cascadeless Schedule: This is a way of handling Cascaded Aborts. It says when a transaction reads a data
Lets say $T_1 \rightarrow R(A)$ then no other transaction is allowed to read A. This prevents cascaded Aborts but is still susceptible to writer writer problem (Update recovery problem)

Writer writer problem (Update recovery).

T_1	T_2	
		A = 20
20 R(A)		
	R(A) 20	
10 W(A = A-10)		
	W(A) = A-5 15	
	commit 15.	
fails		
20 rollback.		

Thus the update done by T_2 is non recoverable due to the roll back of T_1 which sets A back to 20.

SERIALIZABILITY

It ensures that the execution of concurrent transaction is equivalent to some serial execution.

Conflict serializable:
Achieved by reordering non conflicting operations.

View serializable:
Ensures that the final database state matches a serial schedule when considering reads and writes.

CONFLICT EQUIVALENT

R(A) R(A) – Non conflict pairs
R(A) W(A)
W(A) R(A) } Conflict pairs.
W(A) W(A)

R(A) R(B)
R(A) W(B)
W(A) R(B) } non conflict pairs
W(A) W(B)

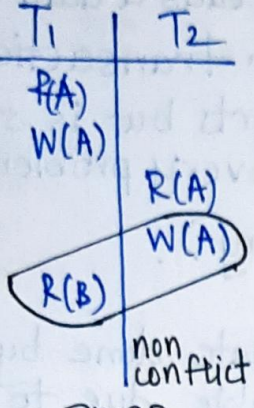
step 1: swap adjacent non conflict pair
step 2: dont swap adjacent conflict pair.

Thus if $S \equiv S'$ then S' is serializable.

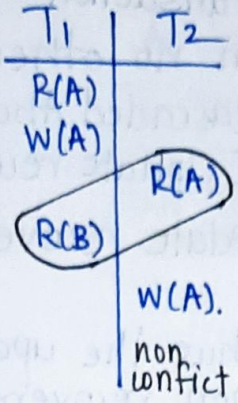
example:

S

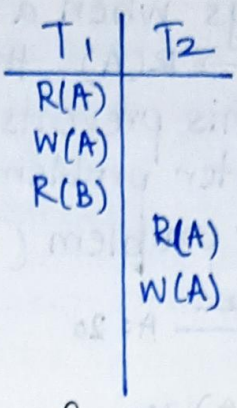
T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	



swap.



swap.



final.

S'

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

Thus we can convert S to be S'

$\therefore S \equiv S'$

Hence S' is serializable.

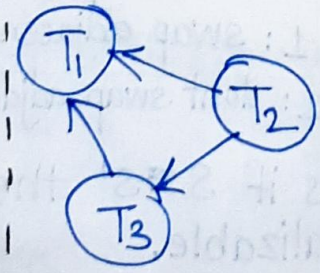
CONFLICT SERIALIZABILITY

- check conflict pair in other transaction and draw edges.
- check if loop / cycles are present in the precedence graph.
- If no loop then conflict serializable → serializable → consistent
- check in degree, least indegree = 1st execution.

example

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)		

Graph. (no loop)



conflict serializable

T_2 - indegree = 0.
remove T_2

T_3 - indegree = 0
remove T_3

T_1 - indegree = 0
remove T_1

$T_2 \rightarrow T_3 \rightarrow T_1$

VIEW SERIALIZABILITY:

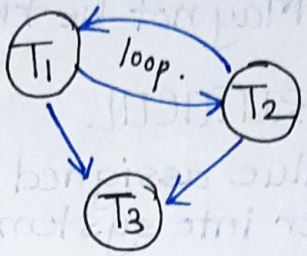
when a transaction $S' \equiv S$ and final DB state matches.

example:

T_1	T_2	T_3
R(A)	-40	
	W(A)	
-40		
W(A)		
		-20
		W(A)

S

Precidence graph



thus with conflict serializability it is unanswerable.

T_1	T_2	T_3
R(A)		
W(A)		
-40		
	W(A)	
	-40	
		W(A)
		-20.

S'

So when A=100.

~~T_1 R(A) 100 → S.W)~~

S : $T_1 \cdot R = 100$, $T_2 \cdot W = 60$, $T_1 \cdot W = 20$, $T_3 \cdot W = 0$

S' : $T_1 \cdot R = 100$, $T_1 \cdot W = 60$, $T_2 \cdot W = 20$, $T_3 \cdot W = 0$

As final state is 0

$\therefore S \equiv S'$

Thus it is view serializable.

$T_1 \rightarrow T_2 \rightarrow T_3$

CONCURRENCY CONTROL

1. Shared Exclusive Locking:

- Shared lock (S): If transaction locks data item then read only allowed.
- Exclusive lock (X): If transaction locks data item then read & write allowed.
- Unlock (U).

example

T_1	T_1	T_1
S(A)	S(A)	X(A)
R(A)	R(A)	R(A)
U(A)	W(A)	W(A)
✓	U(A) ✗	U(A)

shared lock doesnt allow write operation on that data item.

Compatibility table

for A	Req S	Req X
Grant S	Yes	No
Grant X	No	No

Problems with locking.

- May not be sufficient for serializable schedule.
- May not be free from irrecoverability
- May not be free from deadlock
- May not be free from starvation.

TIMESTAMP ORDERING PROTOCOL

Timestamp is a unique value assigned to a transaction.
Tells order when they enter into system.
Read-TS (RTS): last transaction no which performed read successfully.
Write-TS (WTS): last transaction no which performed write successfully.

(oldest transaction executes first / on priority)

Rules:

Transaction Ti issues Read (A)

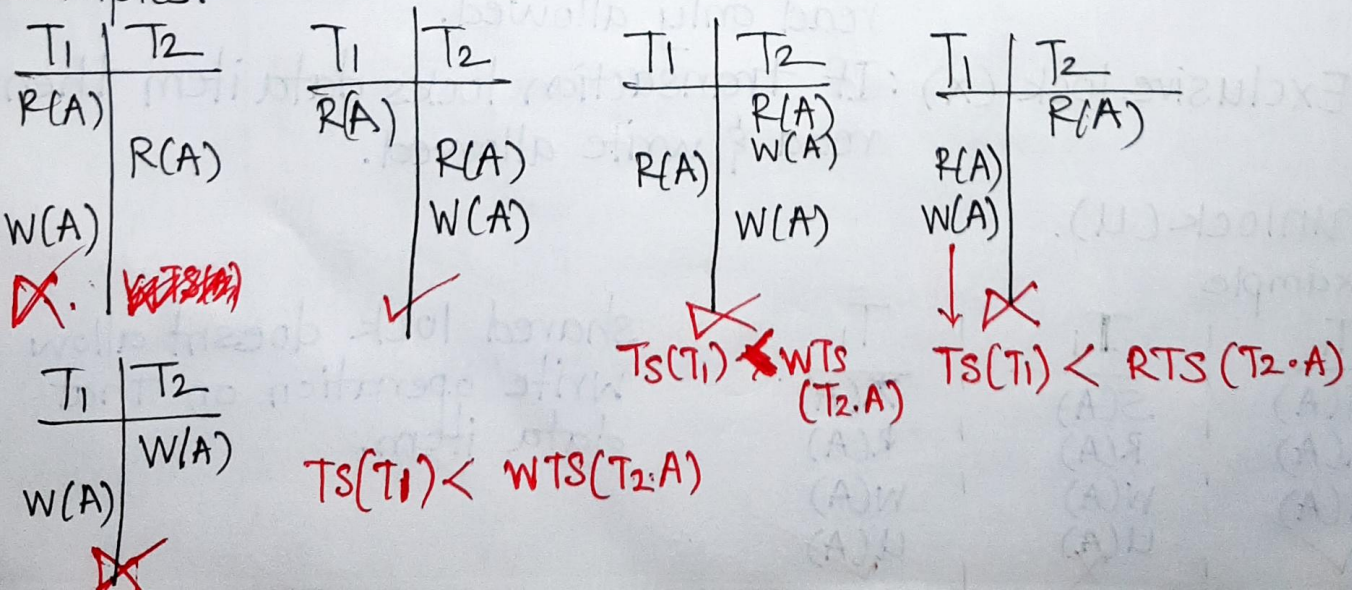
a) If TS(Ti) < WTS(A), Rollback Ti
b) else Set RTS(A) = Max {RTS(A), TS(Ti)}

Transaction Ti issues write(A)

a) If TS(Ti) < WTS(A), Rollback Ti
b) If TS(Ti) < RTS(A), Rollback Ti
c) else set WTS(A) = TS(Ti)

This protocol assumes $T_1 \rightarrow T_2 \rightarrow T_3 \cdots$ serializability

examples.



LOG BASED RECOVERY:

The most widely used data structure for recovering DB modification.

Log is a sequence of log records, recording all the update activities in DB

Structure:

Transaction identifier	Data item identifier	Old values	New values

We denote various types of log records as

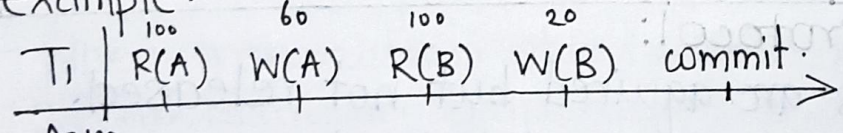
$\langle T_i, start \rangle$ T_i has started

$\langle T_i, data, V_1, V_2 \rangle$ T_i performed operation on data with old value V_1 to new value V_2

$\langle T_i, commit \rangle$ T_i has comitted

$\langle T_i, abort \rangle$ T_i has aborted.

example:



log: $\langle T_1, start \rangle$

$\langle T_1, A, 100, 60 \rangle$

$\langle T_1, B, 100, 20 \rangle$

$\langle T_1, commit \rangle$

whenever write is performed log must be created before commit we also have the abilities to undo modification done to DB.

Immediate Modification.

modification in active state
(modifications are done before commit)

<To, start>
<To, a, 10, 5>
<To, b, 10, 15>
---------- DB {a=5, b=15}
<To, a, 5, 0>
---------- DB {a=0}
<To, commit>

Deffered Modification.

modification in commit state
(modification are done after commit)

<To, start>
<To, a, 10, 5>
<To, b, 10, 15>
<To, a, 5, 0>
<To, commit>
---------- DB {a=0, b=15}

<To, start>
<To, a, 10, 5>
<To, b, 10, 15>
<To, a, 5, 0>
~~<To, commit>~~
<To, abort>
- - - - - - - DB {a=10, b=10}

DB {a=5, b=15}
DB {a=0}

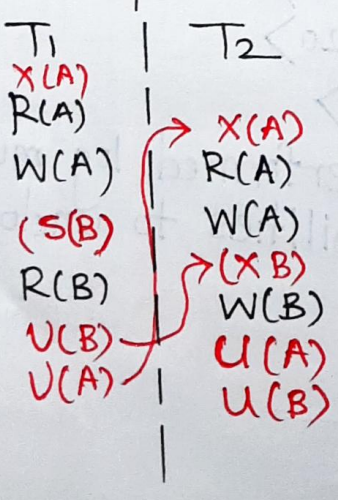
<T1, start
<T1, a, 10, 5>
<T1, b, 10, 15>
<T1, a, 5, 0>
<T1 abort>
- - - - - - - - DB {a=10, b=10}

2 Phase Locking Protocol:

growing phase : locks are aquired but not released.
shrinking phase : locks are released but not acquired.

Problems :
May not be free from :
Irrecoverable
Deaadlock
Starvation
Cascaded abort.

Example.



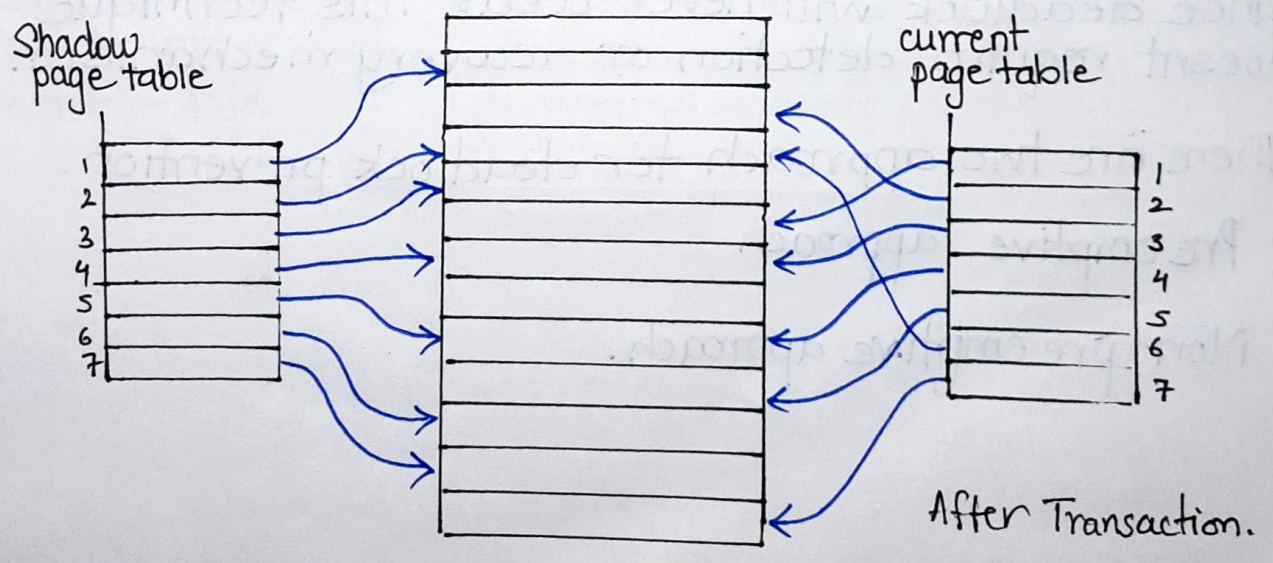
Strict 2PL: Hold all exclusive lock until commit/abort

rigorous 2PL: Hold all shared & exclusive lock until commit/abort.

Shadow Paging Technique.

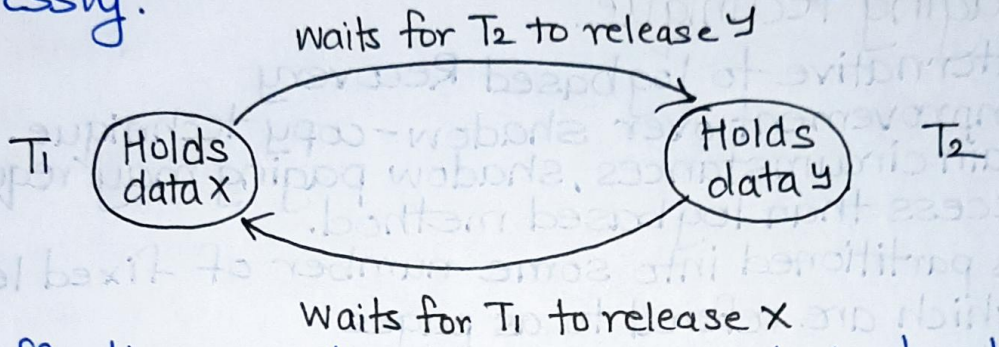
It is an alternative to log based Recovery
It is an improvement over shadow-copy technique.
Under certain circumstances, shadow paging may require few disk access than log based method.

- The DB is partitioned into some number of fixed length blocks, which are refered to as pages
- Assume there are n pages, numbered from 1 to n These are not stored in any particular order on the disk
- For finding ith page of database for any given i for this we use page table.
- Each entry contains a pointer to a page on disk.
- The key idea is to maintain 2 page tables.
  - Shadow page table (as backup)
  - current page table (for modification)
- When transaction starts both tables are identical
- If no issue in transaction then current page table becomes new shadow page table

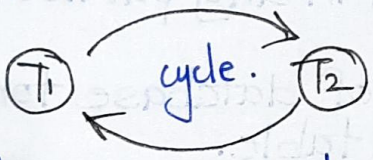


DEADLOCK

In a database environment, deadlock is said to be a condition where two or more transaction wait for one another forever, thus waiting in a cycle fashion endlessly.



An effective way to detect a deadlock is to draw a wait for graph



when a cycle/loop means deadlock.

DEADLOCK PREVENTION.

This technique ensure no deadlock would occur by making it structurally impossible.

If transaction manage senses a deadlock it aborts the transaction and restarts.

It is a cautious approach.

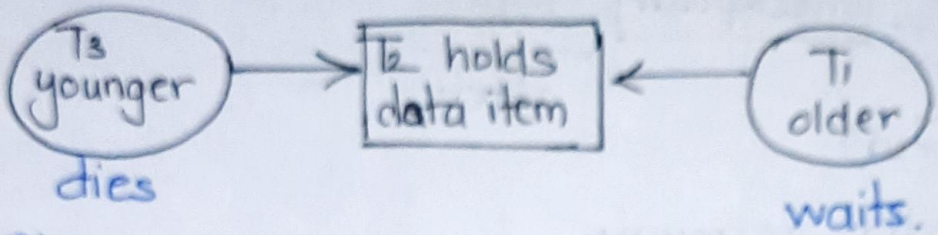
Since deadlock will never occur this technique doesnt require detection or recovery mechanism.

There are two approach for deadlock prevention.

- Pre emptive approach
- Non-pre emptive approach.

Non pre-emptive approach:

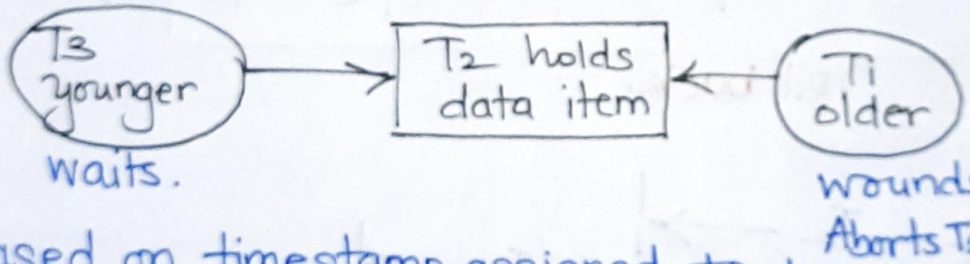
Logic used is that its better to restart a young transaction than an older one as more resources have been invested in older transaction.



- Since older transaction waits for younger in a way it looses priority.
- It may lead to numerous restart of younger transaction.

Pre emptive approach:

The logic used here is also same the only difference is if older transaction requests the current one dies and younger one waits.

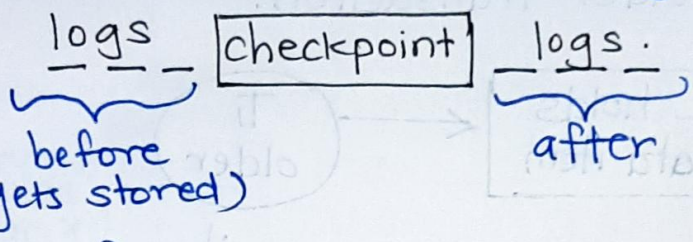


Based on timestamp assigned to transaction, two main deadlock prevention techniques are identified.

- Wait – Die
- Wound – Wait.

CHECKPOINTS :

It is a mechanism in which all the previous logs are permanently stored in storage disk.



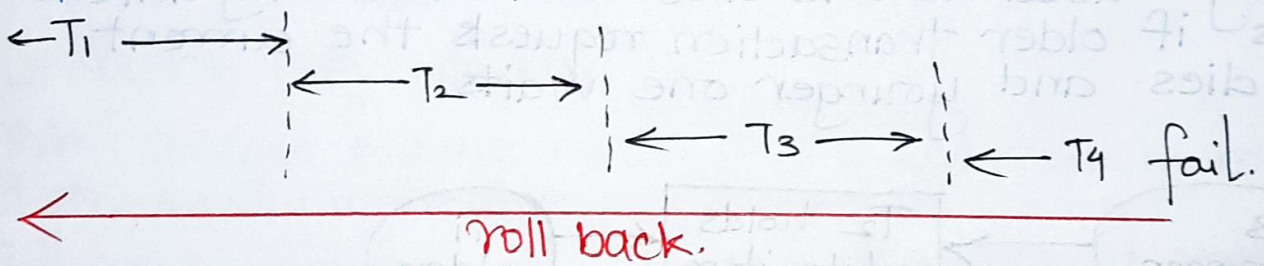
This helps in faster recovery.

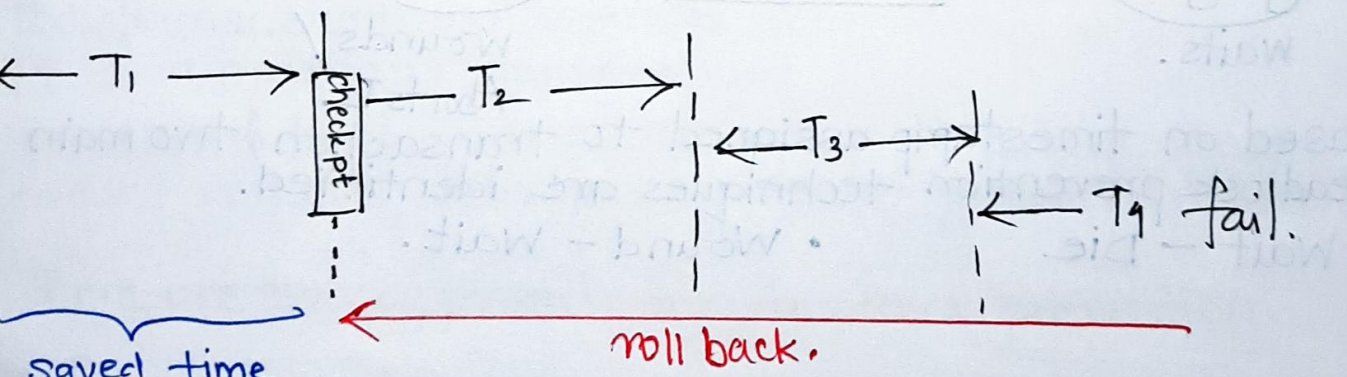
example : if T_1, T_2, T_3, T_4 are running. T_1 commits & leaves but the system fails.

∴ The system rolls back all transactions

∴ T_1 is redone which was not necessary.

But if we add a checkpoint then it only rolls back upto latest checkpoint.





saved time & resources.