DBMS - V

TRANSACTION

* It is an unit of program execution that that accesses and updates various data items.

* Main issues:
  (i) Hardware failure (or) System crashes
  (ii) Concurrent execution of multiple transaction.

Properties of transaction — To maintain data integrity

↳ A C I D

Atomicity / Consistency / Isolation / Durability

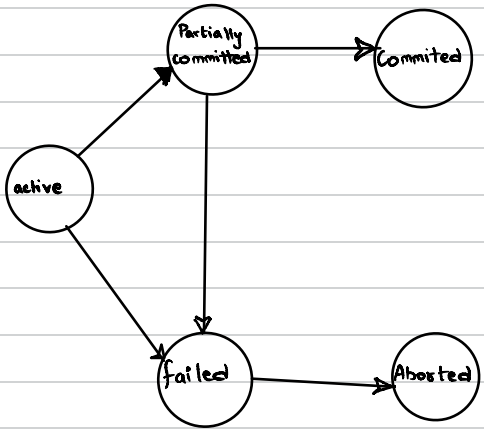
#ATOMICITY: Either all trasactions are properly displayed in Database or none.

#CONSISTENCY: Execution of Transaction in isolation preserves consistency of Database.

ISOLATION: Although multiple transaction execute concurrently, Each transaction MUST be unaware of the other.

DURABILITY: After transaction completes succesfully, it remains in Database (the change), even after system failures.

TRANSACTION STATE



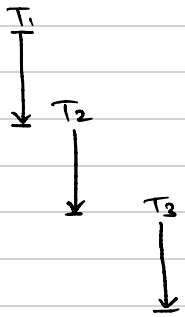
#ACTIVE: Initial state, transaction stays in this state while executing.
#PARTIALLY COMMITTED: After final statement has been executed.
FAILED: After discovery - Normal execution can no longer proceed.
ABORTED: Transaction rolled back and database restored to prior state.
COMMITTED: Succesful Completion

Schedule

* It is a Sequence of instructions that specifies the chronological order in which the instructions of concurrent transactions are executed
  ↳ Multiple transactions allowed to run concurrently in system.

Serial Schedule

* Once a Transaction starts (whichever it may be), it must complete that transaction before going to another.



Example :

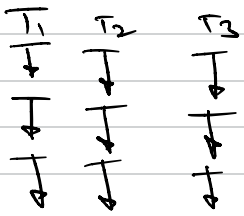
T_1	T_2
read(A)	
A := A − 50	
write (A)	
read (B)	
B := B + 50	
Write (B)	
Commit	
	read(A)
	A := A * 1.5
	write (A)
	read (B)
	B := B ÷ 1.5
	Write (B)
	Commit

* Advantage :
  - Consistent

* Disadvantage :
  - Waiting-time

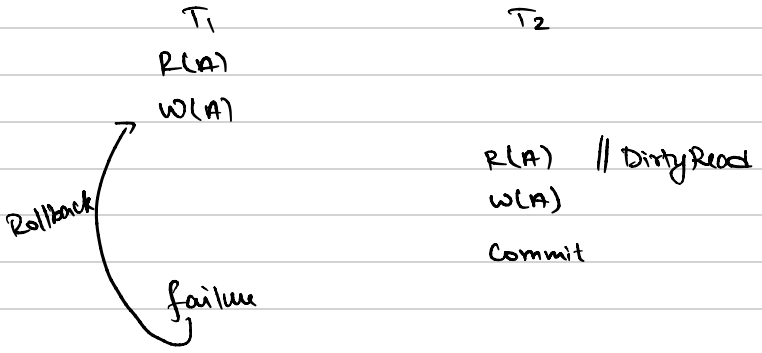
Parallel Transaction

* Multiple Transactions are performed at Single time.



Problems in Concurrent Execution.

#1 - **DIRTY READ**:



#2 **INCORRECT SUMMARY**:

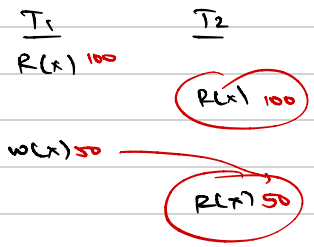
* When One transaction updates a Record
* While another transaction performs aggregate function on that record.

#3 **LOST UPDATE**:

* Update done by One transaction is lost due to overwritting of update done by another transaction.

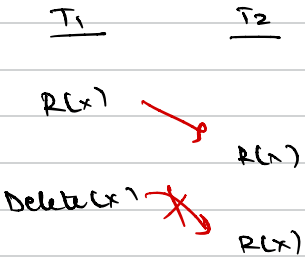
#4 **UNREPETABLE READ**:

* Two or more read operation in same transactions reads different values.



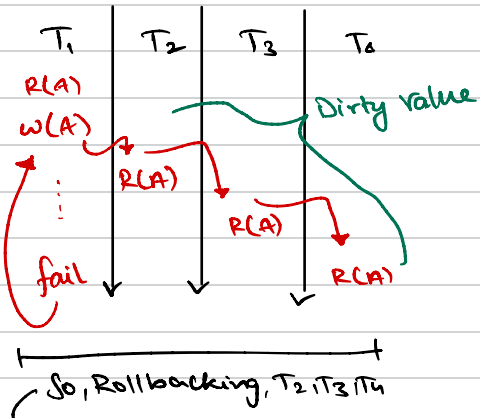
PHANTOM READ:

* Reading of same Variable again but Variable not present



Cascading vs Cascadeless Scheduler

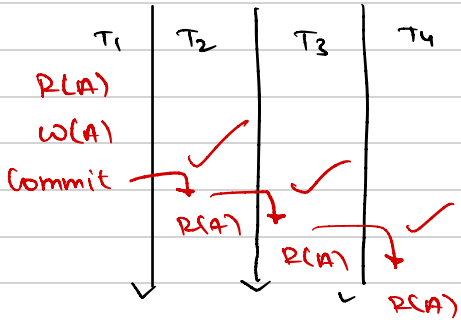
→ Due to Occurence of One Event, multiple events occur



So, Rollbacking, T2, T3, T4

→ Cascading.

→ Doesn't allow Read if it is not commited.



Serializability → Ability to make Serial execution of Schedule / Set of transactions.

S

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

Precedence graph

S

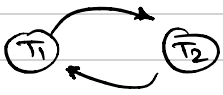
T_1	T_2
	R(A)
	W(A)
R(A)	
W(A)	

Precedence Graph

Already Serial Schedule.

S_1

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



Parallel Exec includes LOOP

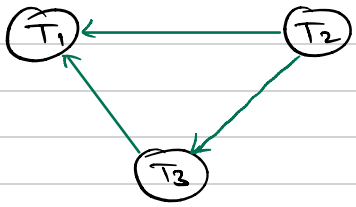
$T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$

Ways / forms of Serializability

- Conflict Serializable
- View Serializable

Conflict Serializability → Check Conflict pairs in other transaction and Draw Edge? (R-W, W-R, W-W)

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)~~		



If loop / cycle exist:

↓

No loop, so – CONFLICT SERIALIZABLE

$T_1 \rightarrow T_2 \rightarrow T_3$

$T_1 \rightarrow T_3 \rightarrow T_2$

$T_2 \rightarrow T_1 \rightarrow T_3$

$T_2 \rightarrow T_3 \rightarrow T_1$

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

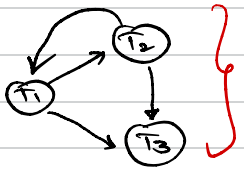
(no incoming Edge to T_2)

$T_2 \rightarrow T_3 \rightarrow T_1$

View Serializability

S

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



Not - Conflict Serializable because loop.

↓

Use View Serializable

→ Rearranging table to make it Serial → Equivalent.

EQUIVALENT

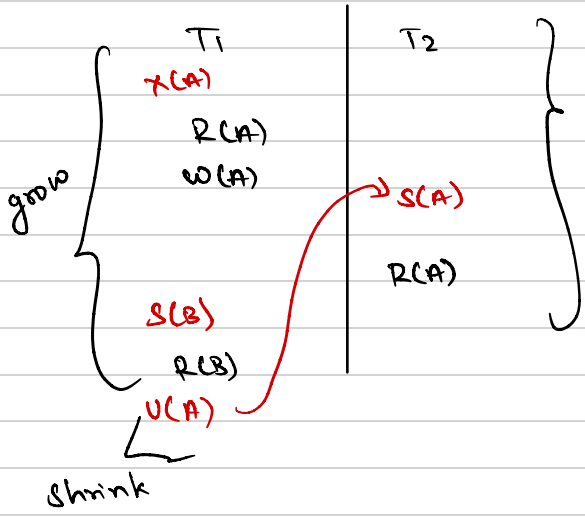
T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

2 Phase - Locking Protocol (2PL) → Extension of Shared-locking protocol.

#1: GROWING PHASE: Locks are acquired, No locks are released

#2: SHRINKING PHASE: Locks are released, No locks are Acquired.

* 2PL is Implemented to acheive Serializability.



$T_1 \longrightarrow T_2$ (Serializability Achieved)

Lockpoint(*): point where transaction attained it's final lock.

$(T_1 \longrightarrow T_2)$

Deadlocks.

* It is a situation where 2/more transactions are waiting for one-another to release locks.

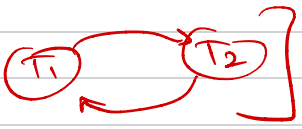
#Ex:

T_1	T_2
lock-X(x)	
R(x)	
W(x)	
	lock-X(y)
	R(y)
	W(y)
Lock-X(y)	
R(y)	
W(y)	
	Lock-X(x)
	R(x)
	y(x)

Works fine till here

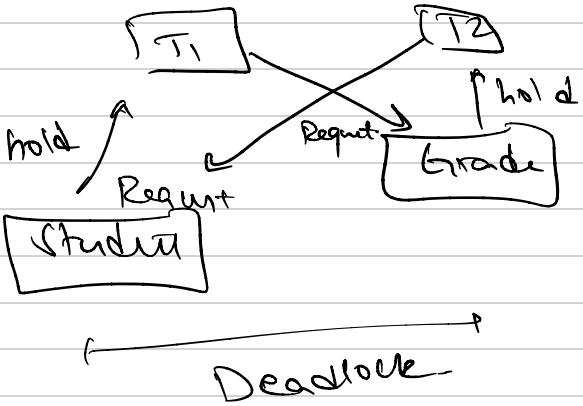
Wait for T_2 to release

Wait for T_1 to Release



wait-for graph — cycle ↓ Deadlock

Detection



Prevention / Handling

#1 - Wait-die scheme (non-preemptive):

* Older transaction wait for Younger transaction to Release Data item.
* Younger transactions never wait for older ones.
  ↳ Rolled back
* Transaction may die several times before acquiring Data item.

Starvation is avoided since older transaction has precedence over younger

#2- Wound-wait scheme (pre-emptive):

* Older Transactions forces (wounds) younger transaction to rollback instead of waiting.
* Younger Transactions may wait for older one.
* fewer Rollbacks than wait-die.

#3- Timeout-Based scheme:

* Transaction waits for lock only for specific amount of time. else, rolled back & restarted
* Simple to implement but Starvation is possible.
* Difficult to Determine good value of timeout interval.

Recovery

#1: Kill process, check Deadlock removed until Success

#2: Concurrency control

#3: Resource pre-emption. - forcibly removing resource & giving to other process to prevent Deadlock. Could cause starvation