DBMS Unit 5

(*) Transaction:

→ Transaction is a set of logically Related operations.
→ Each transaction operates on a common database to produce desired results.
→ Since multiple transactions operate on the database at the same time, the need for concurrent operations and interleaving is brought.
→ Concurrency leads to several data integrity problems.
→ There are certain properties to solve these problems.
→ A schedule is a collection of currently operating transactions.
→ Serializibility of the scheduler is the key to control errors due to concurrent transactions.
→ for example, You are transferring money from your account to your friends bank account. The set of operations would include:
(a). Read your account balance.
(b). Deduct amount from your balance.
(c). Write remaining balance to your account.
(d). Read your friend's account balance.
(e). Add the amount to his account balance.
(f). Write the new updated balance to his account.
→ This whole set of operations can be called a transaction.
→ A transaction generally represents a change in database.

→ Two operations of transaction: ①. Read ②. Write.
Read(A) ⇒ Accessing the data item from database to main memory.
Write(A) ⇒ Updation of data item to database.

⇒ R(A)
A = A - 500
W(A)
R(B)
B = B + 500
W(B)
Commit → after commit, changes in database happen

A = 1000 → 500
B = 2000 → 2500

(*). Properties of transactions:

for effective and smooth database operations, transactions should possess several properties. These properties are:

(a). Atomicity: A transaction is an atomic unit. It cannot be broken down further into a combination of transactions. A given transaction will either get executed or is not performed at all. There is no possibility of a transaction getting partially executed.

(b). Consistency preservation: A transaction is said to be consistency preserving if it's complete execution takes the database from one consistent state to another.

Example, If a database ~~contains~~ stores n values and their sum, then the database is said to be consistent if the addition of these N values actually lead to their sum.

The values should be made consistent once the transaction starts operating.

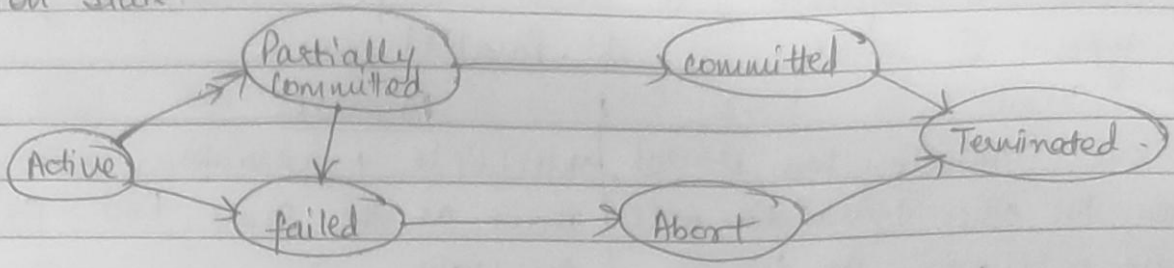
(c). Isolation: Since a large no. of transactions keep executing concurrently, no transaction should get affected by the operation of other transactions.

(d). Durability: The changes effected to database should be permanent. They should not vanish once the transaction is removed. These changes should not be lost due to any failures at later stages.

Isolation: A = 5000

T_1	T_2
r(A)	
A = A - 1000	
	r(A)
w(A)	A = A + 5000
	w(A)
4000	5500

Transaction State:



ⓐ. Active: The transaction is in execution.
ⓑ. Partially Committed: After the execution of last operation, transaction will be in partially committed state.
ⓒ. Committed: When all the modification is done after execution of final instruction.
ⓓ. Failed → After discovery that normal execution can no longer be proceeded.
ⓔ. Aborted → After transaction has been rolled back & database is restored to its previous state. 2 options after abort: ⓐ. Restart
ⓑ. Kill.

(*). Concurrency Control:
→ Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transaction.
→ Purpose ⇒ ⓐ. To ensure isolation, to preserve database consistency to resolve read-write and write-write conflicts.

Concurrency control techniques:

(a). Lock-Based Protocol: There are two modes of a lock:
(i). Shared lock (ii). Exclusive lock.

(*) Schedule: A sequence in which multiple transactions are scheduled.

2 types → (a). Serial (b). Parallel.

Serial

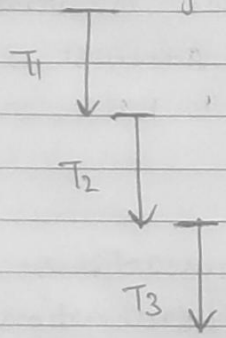
→ If one Transaction has started, then the other transaction will start only when 1st is completed.

→ No other transaction can interfere.

→ Concistent. There is no problem in Database.

→ Problem is waiting time.

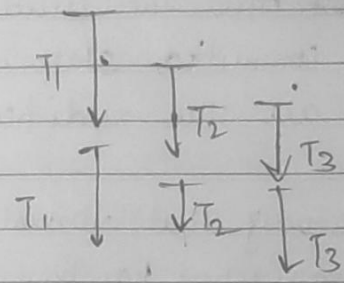
→ Advantage → Security.



Ex → ATM machines.

Parallel

→ Multiple transactions can come and start at the same time. Execute parallelly.



Ex → Banking Apps. It may be possible that multiple users are logging in and transferring at the same time.

→ System performance High.

→ Throughput is high.

→ less time required.

Serial Schedule

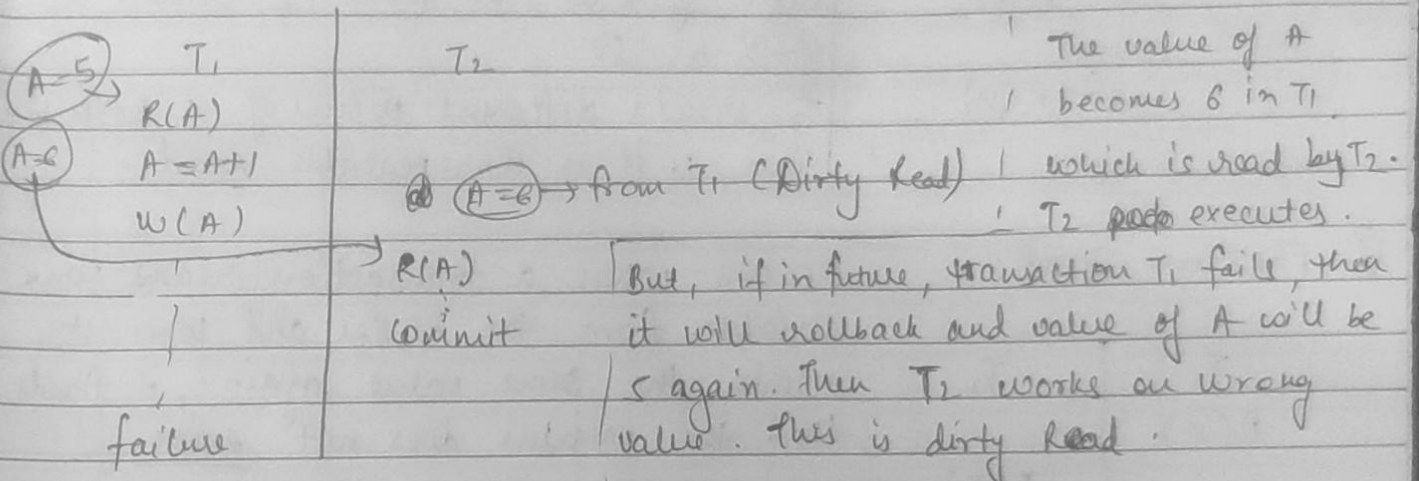
T_1	T_2
read(A)	
A = A - 1000	
write(A)	
read(B)	
B = B + 1000	
write(B)	
	read(A)
	temp = A * 10 / 100
	A = A - temp,
	write(A)

Concurrent Schedule.

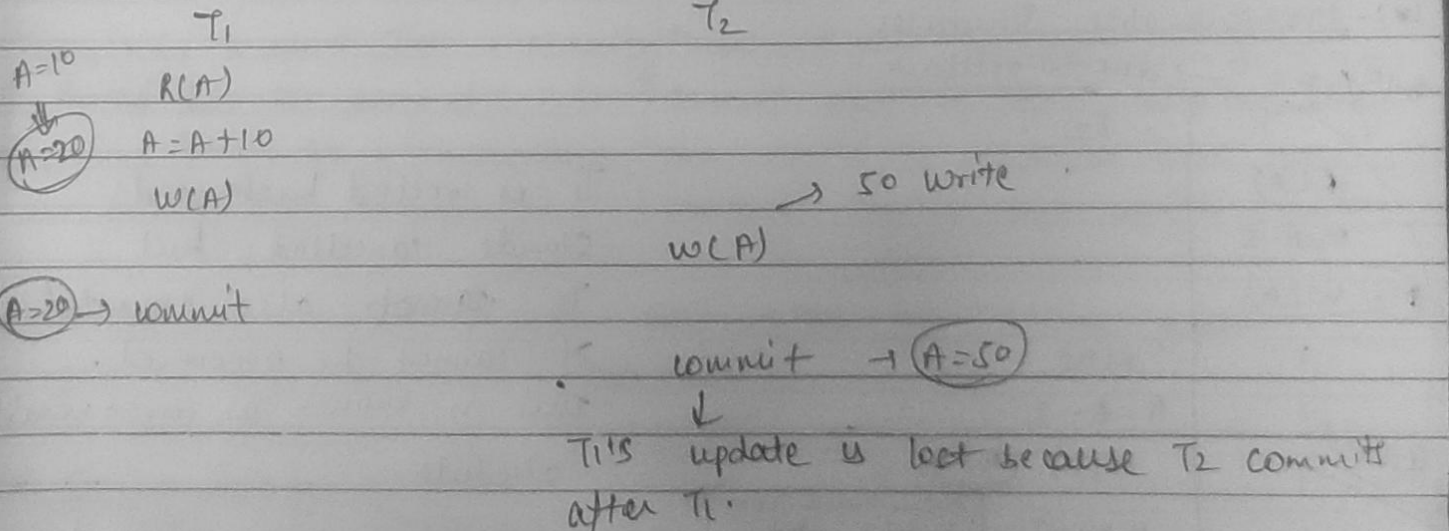
T_1	T_2
read(A)	
A = A - 1000	
write(A)	
	read(A)
	temp = A * 10 / 100
	A = A - temp
	write(A)
read(B)	
B = B + 1000	
write(B)	

(*). Types of Problems in concurrency:

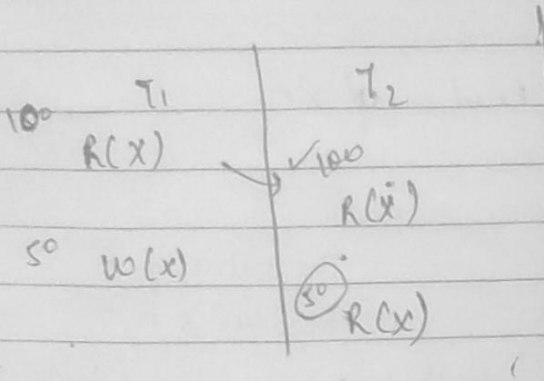
(a). Dirty Read: Also called Uncommited Read of RAW.
Temporary Update.



(b). ~~Deadlock~~ Lost Update / Write-Write conflict Problem / Blind Write.

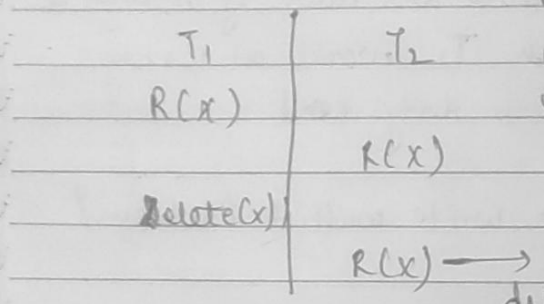


© · Unrepeatable Read / Read write conflict:



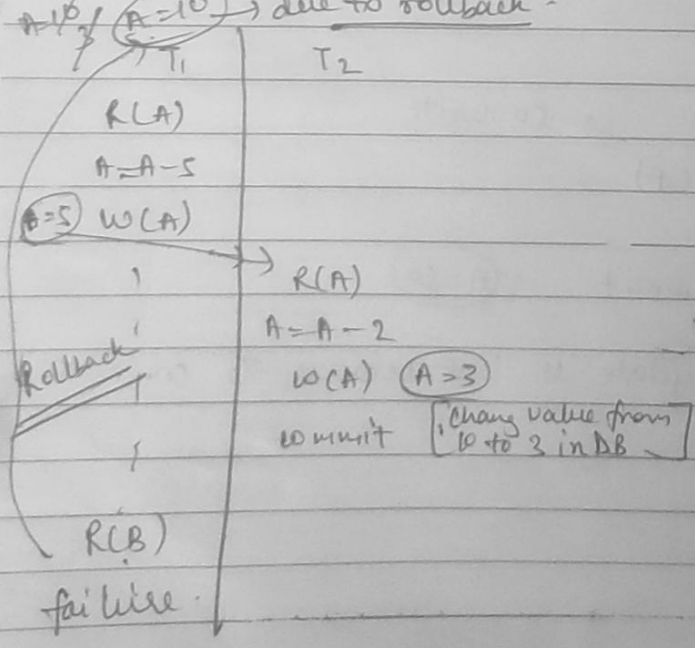
T_1 reads value of X as 100, T_2 would read value as 100. Then T_1 writes value of X as 50, Now T_2 reads value of X as 50.
T_2 reads different values of X both the time, ∴ it is Unrepeatable Read.

(d) · Phantom Read:



occurs when a transaction reads some variable from the buffer and when it reads the same value again, it finds that the variable does not exists.

(e) · Irrecoverable Schedules:

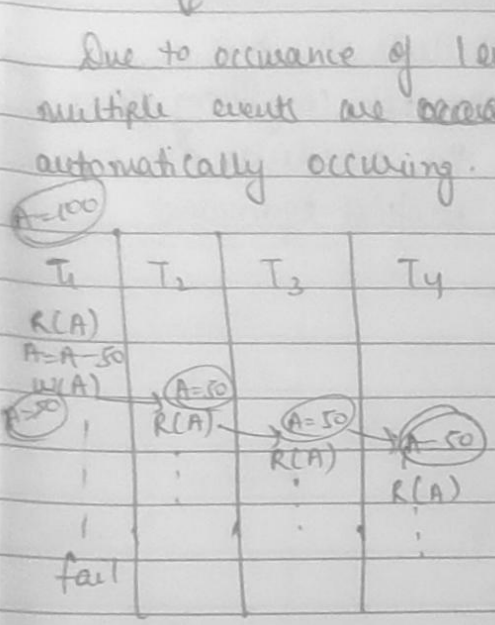


T_1 is rolled back and stands cancelled, but T_2 also executed. It cannot be recovered. This is known as irrecoverable schedule.

(*) Cascading Schedule Vs Cascadeless Schedule

↓

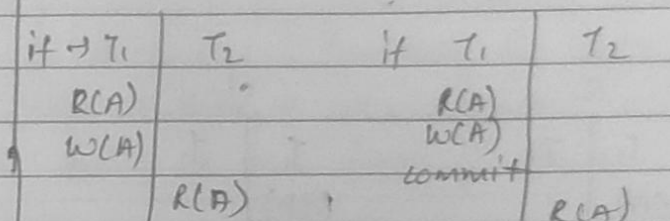
Due to occurance of 1 event, multiple events are automatically occuring.



T_1 will be rolled back, A will become 100 in DB, Now T_2, T_3 & T_4 are working on dirty data, to remove this problem we will tell T_2, T_3 & T_4 to abort. This is cascading T_1 failed but we aborted T_2, T_3 & T_4 forcefully. All 4 transactions are failed. Degrades performance.

↓

To remove the problem of cascading We will not allow T_2, T_3, T_4 . . . schedules to read values used by T_1 until Transaction T_1 successfully commits or Aborts. T_2, T_3, T_4 . . . can work on other values but not on the value that is in T_1.



Cascading Schedule

Cascadeless Schedule

There is one problem, T_2, T_3, T_4 cannot read the same value but they can write the value in their local memory. This is lost update problem.

(*) Strict Recoverable Problem: This solves the Write read problem of cascadeless schedule.

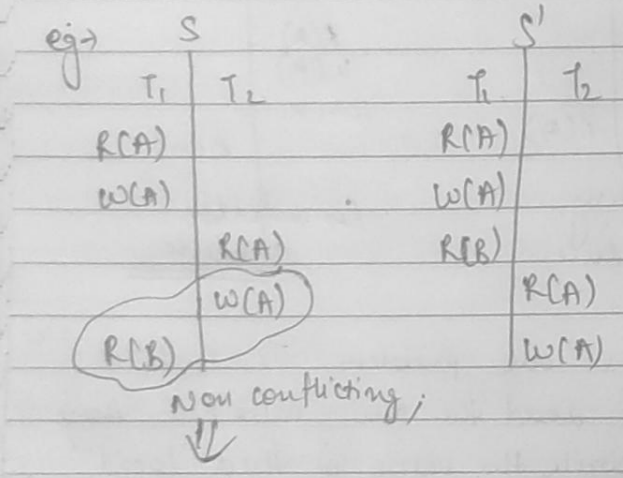
(*) Serializability: → Conflict
→ View

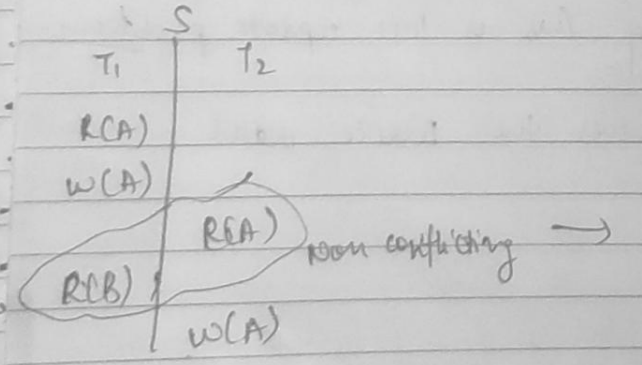
(*). Conflict Equivalent:

R(A)	R(A)	Non-Conflict Pair
R(A)	W(A)	Conflict Pairs
W(A)	R(A)	
W(A)	W(A)	
R(B)	R(A)	Non conflict Pairs.
W(B)	R(A)	
R(B)	W(A)	
W(A)	W(B)	

If there are two adjacent non-conflicting pairs, then we switch them. If a schedule S = a schedule S' by switching the non-conflicting pairs, then both the schedules are said to be conflict equivalent schedules.

eg→





We see that S = S', therefore, both the schedules are conflict equivalent.

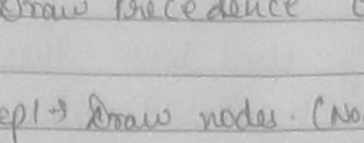
(*) Conflict Serializability:

Draw Precedence Graph.

Step 1 → Draw nodes. (Nodes = No. of transactions).
Eg → 3 transactions, 3 nodes.

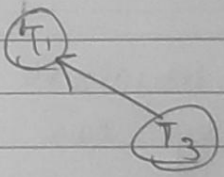
S

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)		



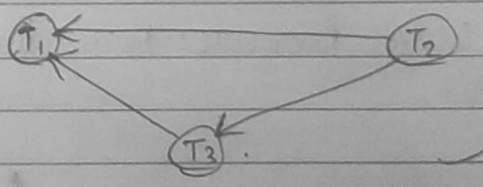
Step 2 → Check conflicting pairs from up to down.
1st R(x) in T_1, conflicting pair will be W(x)
we check if there is W(x) in T_2 & T_3, There is no W(x) in T_2 & T_3, therefore there is no problem. It is checked. Similarly for R(Y).

③ Now R(x), conflict → W(x) present in T_1
Therefore draw a line from T_1 to T_3.



④ Check for others too.

⑤ Complete Graph:



Now, check if loop / cycle exists in a graph, In the above graph, there is no loop / cycle, therefore it is conflict serializable. It is also serializable & consistent.

To check the sequence, we check the indegree in precedence graph.

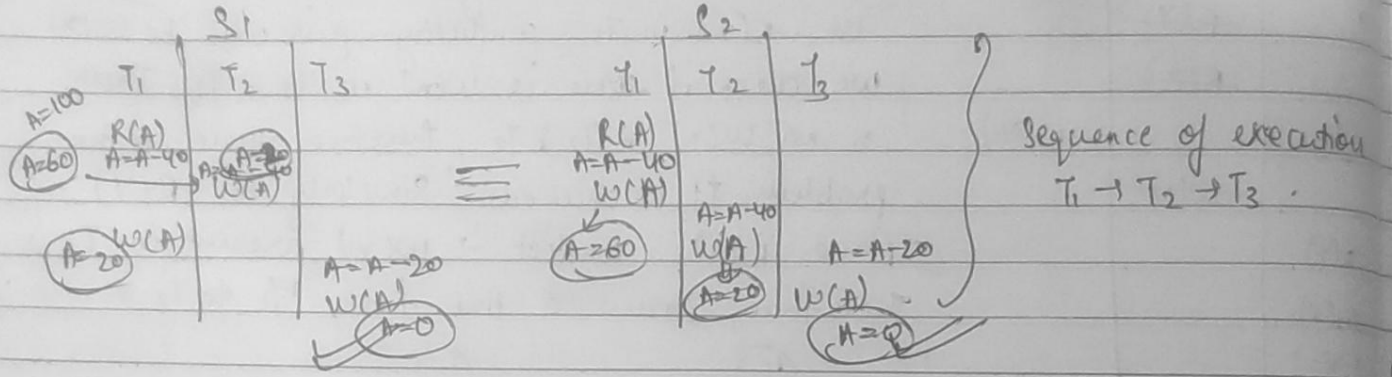
$T_2 \rightarrow 0$
$T_3 \rightarrow 1$
$T_1 \rightarrow 2$

Therefore, Sequence will be $T_2 \rightarrow T_3 \rightarrow T_1$

(*). View Serializability: Used when there is a loop in precedence graph. A schedule is called view serializable if it is view equivalent to another schedule.

→ Checking view serializability of a schedule is NP hard Problem.

→ All conflict serializable schedules are view serializable but not vice versa.



S_1 and S_2 both are view serializable. Condition: final write operation must be done by same transaction.

(*). Concurrency Control Protocols: Main aim is to achieve serializability and recoverability.

(a) Shared Exclusive locking:

→ Shared lock(S) ⇒ If transaction has locked data item in shared mode, then it is allowed to read only.

→ Exclusive lock(X) ⇒ If transaction has locked data item in exclusive mode, it it is allowed to read and write both.

→ Main objective of lock based protocol is to make a schedule consistent.

→ One transaction will work on a data item, we will lock that data item and if another transaction needs to work on the same data item, then it will have to wait until the item is released/unlocked.

→ In this case, we are not able to achieve concurrency.

Shared Mode: T_1 | Exclusive Mode: T_2

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

Compatibility table:

	Shared	Exclusive
Shared	True	false
Exclusive	false	false.

Problems in Shared / Exclusive locking:
→ May not be sufficient to produce only serializable schedule.
→ May not free from Irrecoverability.
→ May not free from deadlock (wait for infinite time).
→ May not free from starvation (waits for a particular time).

(b). Two-Phase Locking Protocol (2PL): Extension of simple shared and Exclusive locking.

Two phases: → Growing phase - locks are acquired and no locks are released.
→ Shrinking phase - locks are released and no locks are acquired.

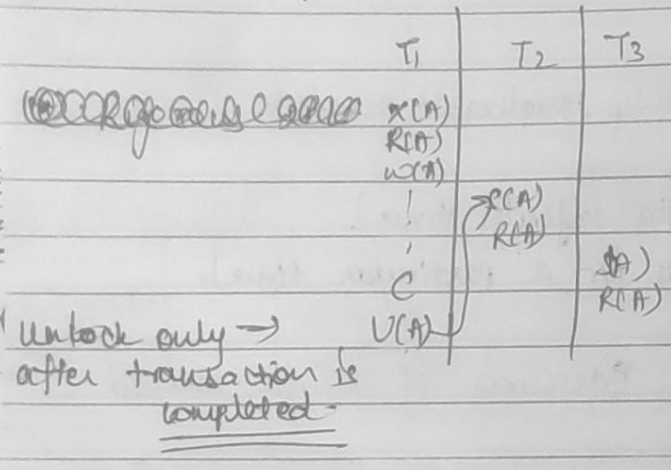
Example:

T_1	
X(A)	growing phase.
S(B)	
R(A)	
W(A)	
R(B)	
S(A)	
R(C)	
S(D)	
R(D)	
U(A) ←	shrinking Phase. As soon as one lock is released, Phase is shrinking.

→ used to achieve serializability.
→ Any transaction that follows 2PL will be serializable.

Problems in 2PL:
→ May not free from irrecoverability
→ Not free from deadlocks
→ not free from starvation
→ not free from cascading Rollback

(*). Strict 2PL: It should satisfy the basic 2PL and all exclusive locks should hold until commit / Abort.
→ solves the problem of cascading Rollback. (on same data)
→ After the transaction is committed, then another transaction will read the value from the database that will solve the problem of cascading rollback and irrecoverability.



(*) Rigorous 2PL: We can release shared locks in Strict 2PL. But we cannot release even shared locks in Rigorous 2PL. Rigorous 2PL should satisfy the basic 2PL and all the shared and exclusive locks both should hold until commit / Rollback.
It is more restrictive than Strict 2PL.
⇒ Deadlock and Starvation Problem still remains.

(*). Conservative 2PL: Let the transaction acquire all the locks before it has started. If all the locks are not available then the transaction must wait.
→ Here, we must have the knowledge of all data items that will be required during execution. It helps in deadlock prevention.
→ There is a possibility of irrecoverable schedules & cascading rollbacks.

© Timestamp Ordering Protocol:

→ Unique value is assigned to every transaction once it enters the system. It tells the order when they enter into the system.

10:00	10:10
T_1	T_2
Timestamp → 100	200

We can conclude that T_1 is older and T_2 is younger.

Larger timestamp is provided to the transaction that comes afterwards.

→ Time stamp of a transaction is denoted by $TS(T_i)$

→ Read time stamp (RTS): latest transaction no. that has performed Read successfully. Read_TS

→ Write time stamp (WTS): latest transaction no. that has performed write successfully. Write_TS.

eg →

T_1 (10)	T_2 (20)	T_3 (30)
W(A)		
	W(A)	
		W(A)

↑ latest write → WTS = 20

→ Timestamp

T_1 (10)	T_2 (20)	T_3 (30)
R(A)		
	R(A)	
		R(A)

↑ latest Read (RTS = 30)

→ The transaction that comes first will be completed first.

Accepted cases:

(100) T_1	(200) T_2
R(A)	
	W(A)

(100) T_1	(200) T_2
W(A)	
	R(A)

(100) T_1	(200) T_2
W(A)	
	W(A)

Not accepted cases:

(100) T_1	(200) T_2
	R(A)
W(A)	

(100) T_1	(200) T_2
	W(A)
R(A)	

(100) T_1	(200) T_2
	W(A)
W(A)	

Rules:

1. Transaction T_i issues a Read 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 a write 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 $WTS(A) = TS(T_i)$

Example: To find which transaction will rollback using Time Stamping

(100) T_1	(200) T_2	(300) T_3
R(A)		
	R(B)	
W(C)	Rollback	
		R(B)
R(C)		
	W(B)	
		W(A)

	A	B	C
RTS	100 → 300	200 → 300	100 → 100
WTS	300	0	100

① $WTS(A) > TS(T_1)$? ⇒ $0 > 100$ False
Set $RTS(A) = Max\{0, 100\}$.

② $WTS(B) > TS(T_2)$? ⇒ $0 > 200$? False
Set $RTS(B) = Max(0, 200)$

③ $RTS(C) > TS(T_1)$? ⇒ $0 > 100$? False
$WTS(C) > TS(T_1)$? ⇒ $0 > 100$? False
Set $WTS(C) = TS(T_1)$

④ $WTS(B) > TS(T_3)$? ⇒ $0 > 300$? False
Set $RTS(B) = Max(0, 300)$

⑤ $WTS(C) > TS(T_1)$? ⇒ $100 > 100$? False
Set $RTS(C) = max(0, 100)$

⑥ $RTS(B) > TS(T_2)$? ⇒ $300 > 200$ True
Rollback T_2

⑦ $RTS(A) > TS(T_3)$?
⇒ $100 > 300$ False
$WTS(A) > TS(T_3)$
$0 > 300$ False
Set $WTS(A) = TS(T_3)$

∴ Transaction T_2 Rollback first T_1 & T_3 get completed then T_2 starts again with a new timestamp.

(*) Recovery from Transaction failures: Purpose of recovery is to bring the database into last consistent state, which existed before the failure.

→ To recover from Transaction failure, atomicity of transactions as a whole must be maintained.

→ There are techniques that help in Recovery and maintaining the atomicity of transactions:

(a) Log - Based Recovery
(b) Shadow Paging.
(c) Check points
(d) Backup Mechanism.

→ Reasons for transaction failure:

(a) Logical errors: due to error in code or any internal condition.
(b) System errors: where the database itself terminates the transaction b/c of some system failure or in case of deadlock / resource unavailibility.

(*) LOG BASED Recovery: → log is a sequence of records which maintains the records of actions performed by a transaction.

→ It is important that these logs are written prior to modification and stored on a stable storage media.

→ Start → Modify → Complete (Commit).

→ The database can be modified using 2 approaches:

(a) Deferred DB modification: All the logs are written on to a stable storage and database is updated when a transaction commits.

(b) Immediate Database Modification: Each log follows an actual database modification and the database is modified immediately after the operation is performed

(*) Checkpoints to Recover from Transaction failures: In parallel execution of transactions, the logs are interleaved. At the time of recovery, it becomes hard for the recovery system to backtrack all the logs. To ease this situation, modern DBMS uses the concept of checkpoints.

→ Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.

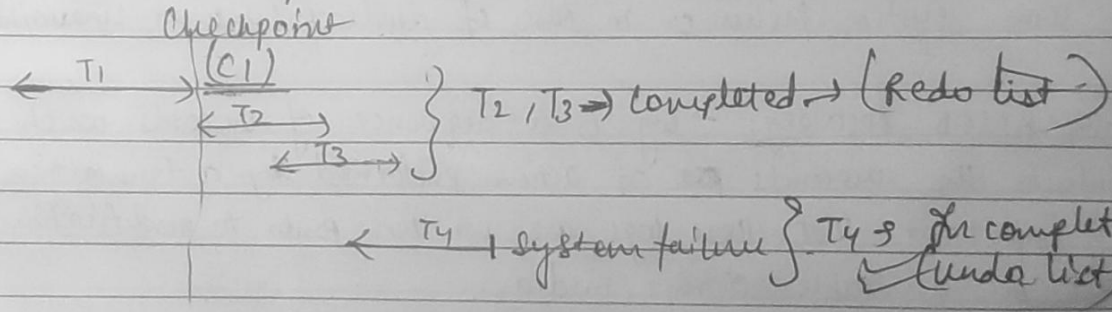
→ Checkpoint declares a point before which the DB was in consistent state and all the transactions were committed.

→ Maintains two lists: ⓐ Redo ⓑ Undo.

→ It saves the extra time in processing all the log files and saves Redoing of unnecessary transactions.

→ Redo list → Completed transactions before checkpoint.

→ Undo list → Incomplete transactions before checkpoint.



(*) Backup Mechanism to recover from transaction failure:

→ Creates backup copies of DB. This backup copy helps to recover the database in case of any damage.

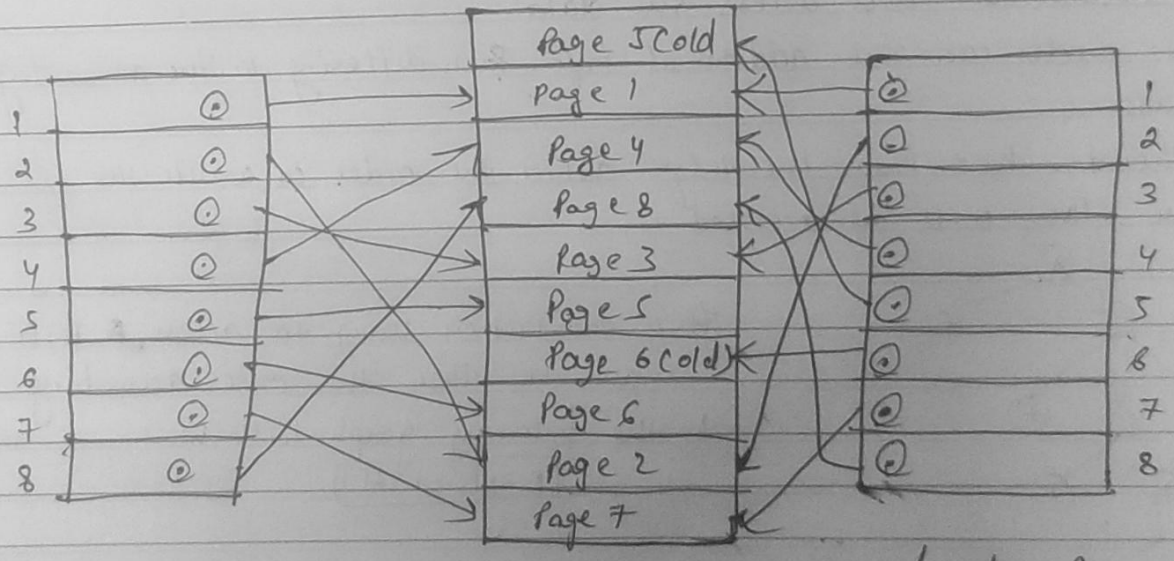
→ Backup can be complete copy of database or an incremental copy.

→ Incremental Copy → Consists of modification only since the last complete/Incremental Backup.

→ The backups are usually stored on an offline storage like magnetic tape.

(*) Shadow Paging:

→ Alternative to log-Based Recovery.
→ It requires fewer disk accesses than other methods.
→ The database is partitioned into some number of fixed length blocks that are referred to as pages.
→ Assume there are n pages numbered from 1 to n. These pages are not located in any particular order on the disk.
→ We create a page table for finding any particular page.
→ Each entry contains a pointer that points to a particular page on the disk.
→ Two tables are maintained, → Current page table &
→ Shadow Page table.
→ Shadow page table is never changed, the current page table makes all the changes.



Current Page table.

Shadow Page table.

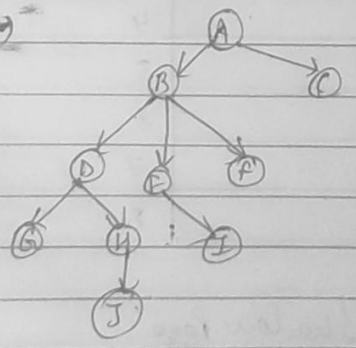
→ When a transaction begins executing, the current page table is copied into a shadow page table and then all the changes are performed in the current page table.

→ The memory contains both, the old and the new pages. In case of any failure, the old pages remain intact pointed to by the shadow table. If the transaction completes successfully & the changes have been written on a stable storage, then the contents of the shadow table are overwritten with the current table. The old pages are removed from the memory and only the new pages are retained.

Concurrency Control Protocols (continued)

Graph Based Protocols:

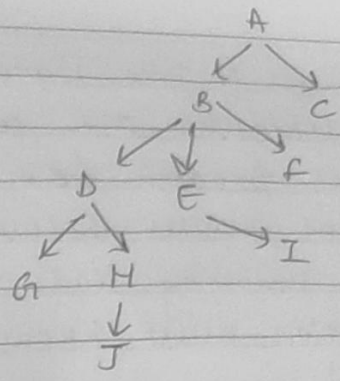
→ Used when we wish to develop a lock-based protocol that is not 2PL.
→ In this case, we will need additional information that how each transaction will access the data.
→ Various models can give additional info. Each differing in the amount of information.
→ We should have prior knowledge about the order in which the database items will be accessed.



If a transaction wants to access B, D and H, then the order according to the following graph will be
B → D → H.

→ we want to achieve serializability without using 2PL Protocol.

(*) · Tree Protocol:



→ Rule 1: You can lock any data item.
→ Rule 2: Now, a data item can ~~only~~ be locked further only if the parent of that data item is already locked.

Example, if we lock B in the first Rule, then according to 2nd rule, we can only lock D, E and F.

→ Rule 3: Data item can be unlocked at any time unlike the 2PL Protocol where you can unlock only after all the locks have been acquired. This helps in concurrency.

→ Rule 4: If one item is locked, used and then unlocked, then you cannot relock the item again by a particular Transaction.

→ Try to unlock as early as possible if the data items are not needed.

eg:

T_1	T_2
lock x (B)	lock (A)
lock x (D)	lock (B)
lock x (C)	lock (C)
Unlock x (B)	Unlock (A)
Unlock x (D)	lock (D)
Unlock x (E)	Unlock (B)
	lock (G)
	Unlock (D)
	Unlock (G)
	Unlock (C)

Here, T_2 wants to work only on G and C, but in order to reach G and C, it is necessary to lock all the other nodes that are being locked here.

(*) Properties of Tree Protocol:

→ All schedules are conflict serializable and view serializable.
→ Ensures freedom from deadlock.
→ Does not ensures recoverability and cascadelessness.
→ Early unlocking leads to shortest waiting time.
→ Transaction must know exactly which data item needs to be accessed.
→ Recoverability and cascadelessness can be ensured by not unlocking before commit.

(*) Deadlock Handling:

→ A system is in deadlock state if there exists a transaction such that every transaction is waiting for another transaction in the set.
→ If T_1 is waiting for data items that are provided to T_2, T_2 is waiting for data items provided to T_3 and so on, then the system is in deadlock state.
→ System must have proper methods to deal with deadlock.
→ Deadlock reduces resource utilization and increases inefficiency.

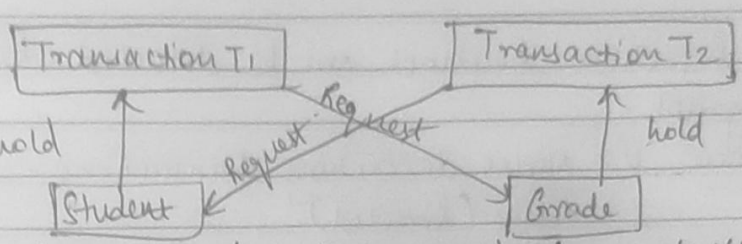
Two ways to deal with deadlock:

ⓐ Deadlock Prevention: ensures that the system will never enter a deadlock state. Used when Risk is High.
ⓑ Deadlock detection & Recovery: Allows the system to enter in deadlock state and then recovers. Used when risk is low.

→ Graph Based protocol, Conservative 2PL Protocol, Time Stamping Protocol and tree based protocol, there is no possibility of deadlock.

· Conditions for Deadlock: ① Mutual Exclusion ② Hold and wait
③ No preemption ④ Circular wait

Example:



Transaction T_1 Holds student but is requesting for grade that is held by T_2.

Deadlock State.

Similarly T_2 Holds grade but is requesting for student that is hold by T_1.

(*) · Deadlock Prevention:

→ Suitable for large databases.

→ A deadlock can be prevented if the resources are allocated in such a way that deadlock does not occur.

→ DBMS analyzes the operations whether they can create a deadlock or not. If there are chances of deadlock, then the transaction is not allowed to execute.

→ Deadlock prevention schemas: They use Timestamps in order to ensure that deadlock does not occur. They are.

(a): wait-die Schema: If T_1 requests for a resource that is held by T_2, then, 2 conditions:

(i) If T_1 is older than T_2, (if T_1 came earlier in the system), then it is allowed to wait for the resource.

(ii) If T_1 is younger than T_2, then it is killed and restarted later with the same timestamp.

(b) wound-wait Schema: 2 possibilities:

(i) If T_1 is older than T_2, then it is allowed to rollback T_2. Then T_1 takes the resource and completes its execution and T_2 is restarted later with same timestamp.

(ii) If T_i is younger than T_2, then it is allowed to wait for the resources.

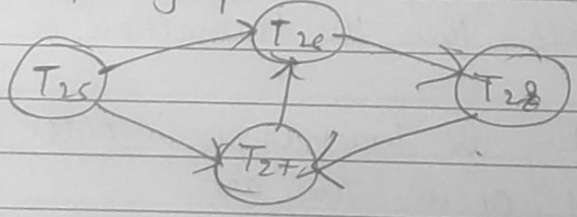
(*) Deadlock Detection: Deadlock can be detected by a wait-for graph.

$G(V, E)$ where V = Nodes describing transactions.
E = Edges (Directed)

⇒ $T_i \rightarrow T_j$, when T_i is requesting a data item that is being held by T_j. In the wait for graph, we draw a directed edge from T_i to T_j.

→ Deadlock exists when there is a cycle in the wait-for graph.
→ Each transaction involved in the cycle is said to be deadlocked.
→ To detect deadlock, the system needs to maintain the wait-for graph.

⇒

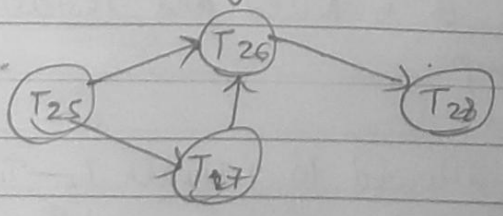


wait for graph (with a cycle).

Deadlock State

① T_{25} is waiting for T_{26} & T_{27}.
② T_{26} is waiting for T_{28}
③ T_{27} is waiting for T_{26}
④ T_{28} is waiting for T_{27}

⇒



wait for graph (with no cycle)
Not in a Deadlock state