Ch-3 - Concurrency Control.

Lock-based Protocol:-

↳ Locks → Two modes in which a data item may be locked:

1) shared: If a transaction T_i has obtained a shared-mode lock (denoted by S) on item Q, then T_i can read, but cannot write, Q.

2) Exclusive - If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on item Q, then T_i can both read and write, Q.

→ The x'actions makes the request to the concurrency control manager. The x'actions can proceed with the operation only after the concurrency-control manager grants the lock.

→ Using these two lock modes, multiple x'actions can read a data item at a time, but cannot write at same time.

Aim of lock:- To ensure isolation i.e. to require that data items be accessed in a mutually exclusive manner; i.e. while one x'action is accessing a data item, no other can modify. The most common method used to implement this, by holding a lock on that item.

a)

Compatiblity function & Compatibility Matrix :-

↳ Definition: Let A and B represent arbitary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which Transaction Tj (Ti≠Tj) currently holds a lock of mode B. If x'action Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we can say mode A is compatible with mode B.

↳ Such a function can be represented conveniently by a matrix.

	S	X
S	True	false
X	false	false

Lock-compatibility matrix comp.

Here in above matrix we can say that by using S & X locks as discussed previously we can get compatibility true only when we use shared locks on transactions. otherwise exclusive lock on Ti can't permit other lock to be on that same transaction Ti.

Transactions schedule with lock:-

Example:-

schedule 1

T_1	T_2	concurrency control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
B = B - 50		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display(A+B)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
A = A - 50		
write(A)		
unlock(A)		

In above given schedule, we first lock B by using exclusive lock in T_1, i.e. we can read & write on B.

- After that we lock A by shared lock in T_2 that means we can only read A but can't write, same way T_2 uses B in shared mode and then displays result of A+B.
- After then T_1 lock A by using exclusive lock & made changes in it. So by using locks these way we can see that there's less chances of system to come into inconsistant state which prevent to violation of isolation property.
- Now for ex:-

T_1	T_2	
Lock-X(A)		
	Lock-S(A)	
R(A)	:	
A = A+50	:	
W(A)	:	wait
unlock-(A)	:	
	Schedule 2	

In this, case T_2 will wait for T_1 to complete it's task as it locked A by using exclusive lock. So no other action can use data item A, until T_1 unlocks it.

~~Deadlock:-~~ Deadlock Condition:

Example: Schedule 3

T_1	T_2	Concurrency control manager
lock-X(A)		Grant X(A)
	lock-X(B)	Grant X(B)
read(A)		
A=A+50		
~~write(A)~~		
	read(B)	
	B=B+100	
lock-X(B)	~~write(B)~~ →	requesting for X(B) in waiting state
:	lock-~~X~~(A) →	requesting for X(A)
wait	:	in waiting state.
	wait	

As shown in above example, when these type of ~~cond~~ situation arises in which transactions are wait for other transactions to complete & release data item forever such condition is called as deadlock.

Definition:-

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another

transaction in a set.

↳ There are two principle methods for dealing with deadlock problem:

1) deadlock prevention: protocol to ensure that the sys. will never enter a deadlock state.

2) deadlock detection & recovery: Allow system to enter a deadlock state & then try to recover by using this scheme.

1) Deadlock Prevention:-

↳ first approach → before begin of transaction execution, each transaction locks all its data items.

↳ disadvantages: i) hard to predict, before begining of x'action, that what data item need to be locked.

ii) data item utilization may be very low, as many of the data items may be locked but unused for a long time.

- Second Approach:- To impose an ordering of all data items, and to require that a x'action lock data items only in a sequence consistent with the ordering.
- Once a x'action has locked a particular item, can't request locks on items that precede that item in the ordering.
- for ex: data items we are having, A, B, C, D, E, which having ordering sequence 1, 2, 3, 4, and 5. So now if T_i Transaction locks item C then it can use only D, E nowonwards. It is unable to use items A & B.

- Next Approach:- By using preemption & rollbacks:
  - In preemption, when a x'action T_j requeast a lock on ~~transactio~~ data item hold by T_i, then lock granted to T_i may be preempted by rolling back of T_j & granting of lock to T_j. T_i [take lock from T_i & assign to T_j]

↳ To perform these tasks it using timestamp mechanism. It provides timestamp to every transaction.
For ex: T_1 starts at 5 ms & T_2 starts at 10 ms then it assign timestamp as $T_{1(5)}$, $T_{2(10)}$. This means T_1 is older then T_2 & or T_2 is younger then T_1.

↳ Two different deadlock-prevention schemes using time-stamps have been proposed!
[Also known as deadlock avoidance Techniques]
1] wait-die: It's nonpreemptive technique. When T_i request data item hold by T_j, then T_i will allowed to wait if it has smaller timestamp than T_j (T_i is older than T_j) other wise, T_i will be rolled back (dies).
ex: T_1 & T_2 & there timestamps 5 & 10 i.e. $T_{1(5)}$ & $T_{2(10)}$. Now T_1 request data item A from T_2, then T_1 will wait until T_2 can't release it as T_1 is older than T_2. If T_2 request for item B which is hold by T_2, then T_2 will rollback as T_2 is younger then T_1.

2] Wound-wait scheme: It's preemptive technique. When x'action T_i request for data item held by T_j, T_i is allowed to wait only if it's timestamp is larger than T_j (if T_i is younger than T_j) otherwise T_j will rolled back (T_i will kill T_j)

for ex:- T_1, T_2 : T_1 → older

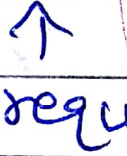
Timestamp 5, 10 : T_2 → younger

T_2 → holds → B

T_1 request

As T_1 is older it will kill T_2 [rollback T_2] & lock B.

T_1 → holds → A



T_2 request.

As T_2 is younger then T_1, T_2 will wait.

Deadlock detection & Recovery:-

4 Deadlock detection:- It can be described precisely in terms of a directed graph called a wait ~~graph~~ for graph.

→ This graph consist of a pair $G=(V,E)$
V is set of vertices &
E is set of edges.

→ Set E contains ordered pair as, $T_i \rightarrow T_j$.
i.e. T_i is waiting for T_j to release a data item. When T_j release data item this edge is also remove from edge set.

→ Based on this edge set, wait for graph is created.

→ If there exist any cycle in wait for graph then there's deadlock exist in a system.

For ex:-

4 We are 4 x'actions - T_1, T_2, T_3, T_4

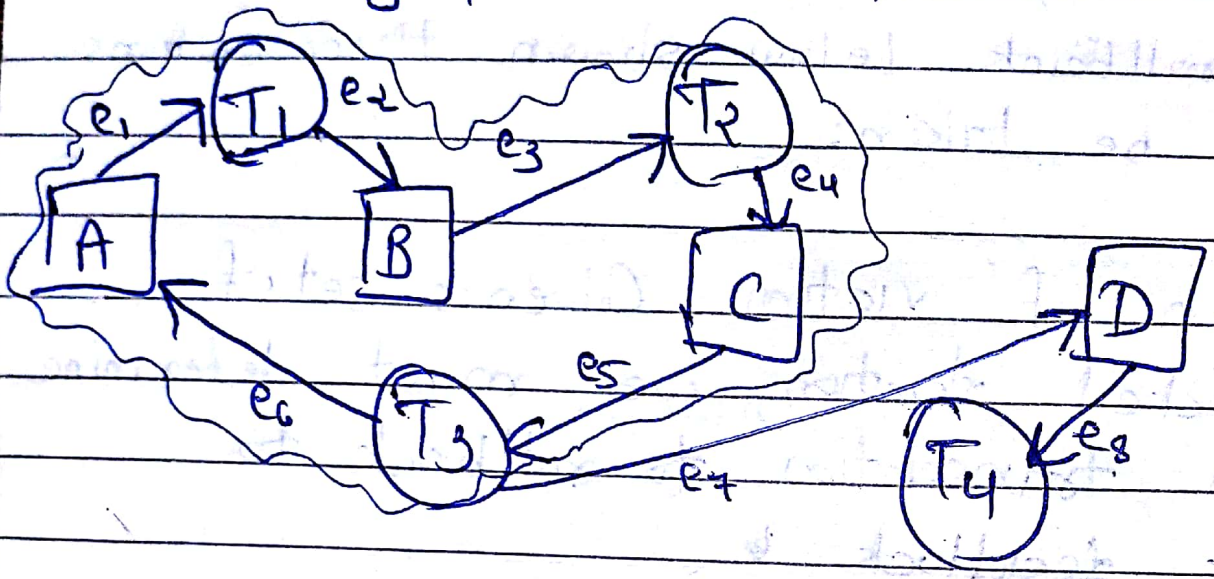
Now T_1, holding A & request for B.

T_2, holding B & request for C.

T_3, holding C & request for A & D.

T_4, holding D.

Based on above description we can create graph as below,



e_1 - shows A is holds by T_1

e_2 - shows T_1 request for B.

e_3 - shows B is hold by T_2

e_4 - shows T_2 request for C.

e_5 - shows C is hold by T_3

e_6 - shows T_3 request for A.

e_7 - shows T_3 request for D.

e_8 → shows D is hold by T_4.

↳ As in fig we can see that there exist cycle $\rightarrow A \rightarrow T_1 \rightarrow B \rightarrow T_2 \rightarrow C \rightarrow T_3 \rightarrow$

So this system contains deadlock.

Deadlock Recovery: After detection of deadlock in a system, we need to recover from it.

→ for rollback below shown three actions need to be taken:

i) selection of victim:- Given a set of deadlocked x'action, we must determine which transaction to rollback to break deadlock.

ii) Rollback:- After detection that which x'action must rollback, we need to decide upto what point particular x'action must be rollback to remove deadlock.
Either we can perform total rollback or partial rollback.

- Total rollback:- Abort the x'action & restart it.
- Partial rollback:- To get effective output must be rollback upto the point it is necessary. It requires the system to maintain addition information about the state of all running transactions.

iii) Starvation:- In a system where selection of victim is based on primarily on cost factors, it may happen that the same x'action is always picked as a victim. As a result this transaction never completes its assigned task, thus there is starvation.

↳ for this we must ensure that a x'action can be picked as a victim only a (small) finite number of times.

⇒ Two-Phase Locking Protocol:-

↳ To ensure serializability we use this protocol. This protocol works in two phase as explained below

i] Growing phase- A transaction may obtain locks, but may not release any lock.

ii] Shrinking phase- A transaction may release locks, but may not obtain any new locks.

→ Initially x'action is in Growing phase. It acquires as many locks as needed. Once it release any lock it enters in shrinking phase, in which it can't obtain any lock.

U) Before continuing in Two-phase locking protocol need to understand some terms:-

1] Cascading rollback:- In a schedule due to one transaction T_i, ~~needs to~~ we ~~rollback~~ needs to rollback other transactions which are dependent on that T_i transaction, it this phenomenon, is called as cascading rollbacks.

For ex:-

T_1	T_2	T_3
R(A)		
A = A + 50		
W(A)		
	R(A)	
	A = A + 10	
	W(A)	
		R(A)
		A = A - 5
		W(A)
abort		

Now, if T_1 Rollbacks, we need to perform rollback of T_2 & T_3 also, as they are dependent on T_1, so this ~~set~~ is called as cascading rollback.

→ Undoing these much amount of work leads is not feasible. So it is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called as, Cascadeless schedules-

It is one where, for each pair of transaction T_i & T_j such that T_j reads a data item previously written by T_i, after the commit operation of T_i.

for ex:-

T_1	T_2
R(A)	
A = A-50	
W(A)	
Commit	
	R(A)
	W(A)
	⋮

Above schedule is a cascadeless schedule.

Continue with, Two phase locking protocol:-

Example of it:-

T_1	T_2
lock X(A)	~~lock X(B)~~
R(A)	lock X(A)
A = A + 50	R(A)
W(A)	A = A + 50
unlock(A)	W(A) → (lock X(B))
lock X(B)	R(B)
R(B)	B = B - 100
B = B - 100	W(B)
W(B)	unlock(B)
unlock(B)	unlock(A)

for above x'actions, T_1 is not take care of two-phase locking protocol as, it first lock item & release it & then again lock & release which violates two-phase locking protocol properties.

↳ T_2 is fulfills two-phase locking protocol properties as, first it comes in growing phase by locking all items & then comes in shrinking phase by unlocking all the items.

→ The point in the schedule where the x'action has obtained its final lock is called as lock point of transaction.

→ Cascading rollback may occur under two-phase locking. As given in below ex: schedule

T_1	T_2	T_3
lock X(A)		
R(A)		
A=A+50		
W(A)		
unlock(A)		
	lock X(A)	
	R(A)	
	A=A-50	
	W(A)	
	unlock(A)	
		lock S(A)
		R(A)

This schedule is a partial schedule which contains properties of two phase locking protocol. But what if T_1 fails after T_3 R(A), then we must rollback T_2 & T_3 so it leads to cascading rollback.

- Cascading rollback can be avoided by modification of two phase locking called Strict two-phase locking protocol.
In this protocol we require two phase of locking protocol i.e. Growing & Shrinking as well as we requires that all exclusive mode locks taken by a transaction be held until that transaction commits.

- Another variant of two-phase locking is Rigorous two-phase locking protocol, which requires that all locks [shared & exclusive] be held until the transaction commits.

Advantages of Two-phase locking protocol:-
- Ensure conflict serializability.

Disadvantage:-
- Doesn't ensure deadlock in a system.