DBMS PART-II !

Starting Date :-
15th Sept 2022

Ending Date :-

15/09/2022

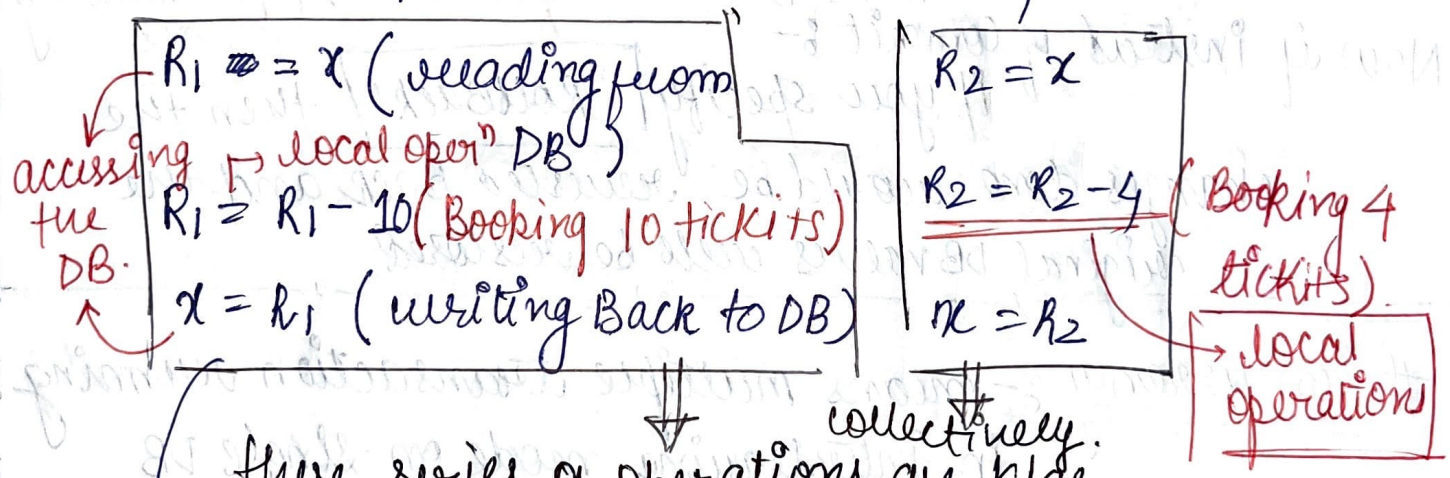
#lesson 01#

#Transactions#

→ Now consider you are booking a ticket on IRCTC and some other person also booking same ticket.

So after booking successfully :- the DB values are updated on IRCTC portal.

So we are performing some series of operations on database! So that the DB is updated after performing those series of operations.



these series of operations, are known collectively as "transactions"

This complete is one transaction

↳ defⁿ :- Logic unit of DB which includes one or more DB access operations

Here we don't care about the local operations.

→ Schedule :- collection of all transactions running on a database.

→ Now, when you have changed the DB :- those changes are done on temporary Basis

So to make the changes = permanent

↓

after the transaction :- you have to use the keyword :-

"Commit"

Ex:- $x = 5 \not{7} \to 5$

Transact^n T_1 :- $x = Read(x)$

$x = x + 2$

write (x) → until here 'x' will be updated to 7 on temporary Basis only

Commit → after this :- x will be updated to 7, permanently

Rollback

Rollback will undo only Non committed values only.

Now if instead of commit :-

If you specify "RollBack" then the changes done would be reverted Back and the original DB values will be restored

---

Concurrency :- means multiple transaction running in interleaving mode on single DB

consider two transaction T_1 and T_2 on same DB :-

$T_1 = R(x), R(y), W(x)$

T_2 :- $R(y), R(z), R(x)$

Now when No concurrency allowed :- then either all stmts of T_1 should run & then T_2

or

all stmts of T_2 should first run and then of T_1

T_1	T_2
R(x)	
R(y)	
W(x)	
	R(y)
	R(z)
	R(x)

So one after the another transactions will run on a single DB → which will create problems for users.

Now when concurrency is allowed :-

at a time we are running only 1 stmt of T_1 or T_2 and then next or another stmt of either T_1 or T_2 :-

T_1	T_2
R(x)	
R(y)	
	R(y)
W(x)	
	R(x)
	R(x)

concurrent running, one stmt after another and not parallel.

Now why Concurrency :-

we have following 3 reasons :-

1. Improved throughput :- many transactions done concurrently.
   - ↳ more transactions complete for a certain period of time.
2. Resource utilization → becomes better.
3. Reduced waiting time.

⇒ Problems with concurrency :-

1. Recoverability problems
2. deadlock.
3. serializability Issues.

Now the transactions running on a single DB, must follow these four properties ⟹

#ACID property :-

A → atomicity → either transaction completed fully or rolled back from where it was stuck → all or none !

C → consistency → multiple transactions running on a single DB, then all the transactions should give you the consistent result
→ values updated in DB should be consistent

I → Isolation

D → Durability.

if transaction could not complete then it should be rollbacked to old values

I → Isolation

One transaction should not affect the result of other transaction

↳ when all the transactions running concurrently.
↳ transactions must be independent of other transactions

→ D :- Durability :- whatever DB changes have been done : those should be durable/long lasting.
→ those changes should be visible for very long time !

#Now problems with concurrency

01: Dirty read or temporary update problem :-

↳ one transaction reading the dirty/or changed value

Ex :- consider following two transactions :-

T1	T2
R(x) = 10	
x = x+2	
W(x)	
	R(x) = 12 → dirty read.

x = 12 ⇦

$x = 10 \rightarrow 12 \rightarrow 10$

But this has read (12) which is not the actual value!

failed.
↳ rollback happend ↑

02. phantom read :- reading the DB value that donot exists

T1	T2
100 ← R(x)	R(x) → 100
delete(x)	R(x) → Phantom read.
↳ deleted x	↳ trying to read x = but it DNE.

$x = 100$ gone

03. Unrepetable read :- here for 2nd time :- reading the updated value.
problem

T1	T2
10 ← R(x)	R(x) → 10
x=5 ← W(x)	R(x) → 5

$x = 10 \rightarrow 5$

04. Lost update problem :- Some transactions are trying to update the DB, But that updati is lost

T1	T2
10 ← R(x)	
12 ← x = x+2	
12 ← W(x)	W(x) = 5
	commit → makes x = 5 permanent
commit →	update of x = 12 is lost

$x = 10 \rightarrow 12 \rightarrow 5$

→ so final value = 5

→ Incorrect Summary problem :-

T_1	T_2
R(x)	
x = x+10	
w(x)	
	R(x)
	R(y)
	sum = x+y
	w(sum)
R(y)	
y = y+10	
w(y)	

x	y
15	17

when $T_1 \rightarrow T_2$

x = 25
y = 27
Sum = 52

and when $T_2 \rightarrow T_1$

Sum = 32
x = 25
y = 27

} when No con-currency

when concurrency :

x = 25
Sum = 42
y = 27

as we can see the sum value is different in all three cases

Good v/s Bad Schedules.

Bad Schedules → having problems → having the inconsistent result.

Good Schedules → producing the correct or consistent result
→ concurrent schedule but generating result like all transactions are running in isolation.

Serial V/s Non Serial Schedule
↳ Concurrent transactions.

↓
one transaction completed then only other transaction will run.

Serializable Schedule :- Schedule which is Non serial but produces result exactly like it is a Serial Schedule

Ex:

T_1	T_2
R(A)	
A=A+5	
W(A)	
R(B)	
B=B+3	
W(B)	
	R(A)
	A=A+4
	W(A)
	R(B)
	B=B+2
	W(B)

A = ~~20~~ → ~~25~~ → 29
B = ~~20~~ → ~~23~~ → 25.

} this was the Serial schedule

Now Non Serial Schedule :-

T_1	T_2
R(A)	
A=A+5	
W(A)	
	R(A)
	A=A+4
	~~W(A)~~
R(B)	
B=B+3	
W(B)	
	W(A)
	R(B)
	B=B+2
	W(B)

A = ~~20~~ → ~~25~~ → 29
B = ~~20~~ → ~~23~~ → 25

→ So a concurrent schedule, which produces result like a serial schedule

Hence this schedule is a Serializable Schedule.

→ Serializability :- method to check if a concurrent schedule is Serializable or not?

we have its two types :-

1. Conflict 2. View

17/09/2022 # lesson 02 #

Note :- if there are n transactions in a schedule, then there are n! possible serial schedules

Now How to verify that a schedule is Serializable or not?

↳ one method is to run the concurrent transactions of the schedule :- and check the result with that of n! possible serial schedules.

if one of the results of serial schedule matches with that of concurrent transaction's result then the schedule is serializable

S

T_1	T_2	T_3
—		—
	—	
—		—
		—

concurrent transactions producing result R

Serial schedules

T_1	T_2	T_3	→ R_1
T_1	T_3	T_2	→ R_2
T_2	T_1	T_3	→ R_3
T_2	T_3	T_1	→ R_4
T_3	T_1	T_2	→ R_5
T_3	T_2	T_1	→ R_6

Now if 'R' matches with anyone of R_1, R_2, R_3, R_4, R_5 or R_6 then our schedule 'S' is serializable.

So we have following two ways to check given schedule is serializable or not?

1. Conflict serializability
2. View ——— 11 ———.

But first :- one imp. term :-

Equivalence :- if two schedules (S_1 and S_2) are producing similar results :- then they are said to be equivalent :- $S_1 \cong S_2$

S_1 and S_2 = might not be equal.
→ but they can be equivalent!

Now the defn of serializability :-

A given Non Serial Schedule 'S' is serializable if it is equivalent to any of the "n!" possible Serial schedules. (where n = no. of transactions in the Schedule).

Conflict Serializability :-

* conflict :- if two transaction (T_1 & T_2) are having few stmts which are independ to each other :- then moving / shifting those stmts (within that transaction) will not effect the other transaction → then T_1 and T_2 don't have any conflict.

and if stmts of T_1 and T_2 are dependent :- then T_1 and T_2 are in conflict state

A = 5

Ex:

T_1	T_2
R(A)	
	R(A) = 5

No conflict here

A = 5

we can move or shift stmts of T_1 and T_2 anywhere

T_1	T_2
R(A)	
5	W(A) → (7)

if we write R(A) in T_1 after W(A) of T_2 :- Then it will read (7) instead of (5)

hence conflict is present here.

→ if T_1 and T_2 = working on two different values ($T_1 \to A$ and $T_2 \to B$) then No any conflict would be there!

So Conflict :- 2 database access stmts are conflicting if and only if :-

(i) Both should be in 2 different transactions
(ii) Both should be accessing same DB value.
(iii) one of them should be a write access.

Ex. Consider S_1 and S_2 :-

S_1

T_1	T_2
R(x)	①
	W(x)
	R(y)
R(z)	②
	W(z)
R(y)	

$T_1 \xrightarrow{x} T_2$
$T_1 \xrightarrow{y} T_2$
} these conflicts we have.

S_2

T_1	T_2
R(x)	①
	W(x)
R(z)	
	R(y)
R(y)	②
	W(z)

$T_1 \xrightarrow{x} T_2$
$T_1 \xrightarrow{z} T_2$

we have same conflicts in S_1 and S_2

Hence S_1 and S_2 are conflict equivalent schedules.

Now consider S_3 :-

T_1	T_2
R(x)	
	w(x)
	R(y)
	w(z)
R(z)	
R(y)	

Here $T_1 \xrightarrow{x} T_2$ ✓

But in case of z

$T_1 \xleftarrow{z} T_2$

↳ conflict is in opposite direct^n

So S_3 :- not equivalent to either S_1 or S_2

So Conflict Stmts. should be same to same in two schedules for being conflict-equivalent.

→ Now Conflict Serializability :- for a given Schedule S is it equivalent to S' or not.

S' = one of the n! possible serial schedule

Ex : consider a Schedule :-

T_1	T_2	T_3
R(x)		
	w(x)	
		R(y)
	w(y)	
R(y)		

→ using conflict precedence graph we will find this Schedule S is serializable or not ?

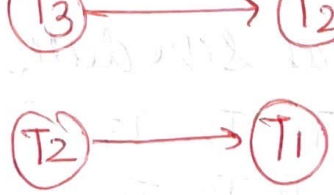
$T_1 \xrightarrow{x} T_2$ = conflict ✓

↳ because of this conflict :- T_1 should run first and then T_2 we cannot change the sequence of Stmts of transac^{n}s

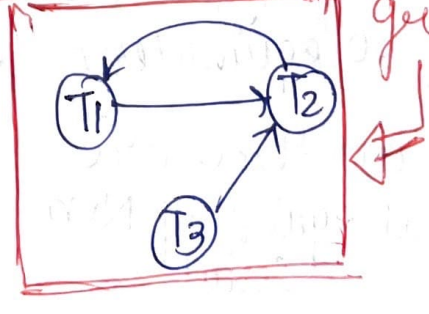
Other conflict :-

$T_3 \xrightarrow{y} T_2$

$T_2 \xrightarrow{y} T_1$



Combining :- precedence graph



Now there is a cycle b/w T_1 and T_2 in graph :-
means which one to run first?
↳ we are in dilemma

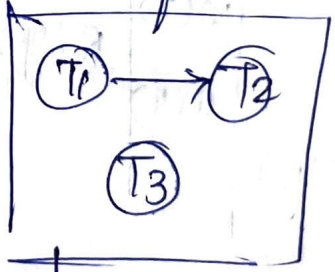
So whenever a cycle is present :- the given Non serial Schedule 'S' will not be equivalent to any other serial schedule at all.

So given Schedule :- Not conflict Serializable :
↳ there different conflicts saying :-
one saying :- T_1 should run first } dilemma
and
other saying :- T_2 should — " — ↓
hence not serializable.

Now Consider :- S

T_1	T_2	T_3
R(x)		
	W(x)	
		R(y)
	R(y)	
R(y)		

here we have only one conflict
$T_1 \xrightarrow{x} T_2$



So the graph :-

This says :- one cond^n only T_1 should run first than T_2

and no any restrictions on T_3.

So equivalent serial schedules :-

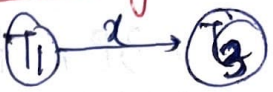
all three are serial equiv. of Non serial S.

$\left\{ \begin{matrix} T_1 & T_2 & T_3 \\ T_1 & T_3 & T_2 \\ T_3 & T_1 & T_2 \end{matrix} \right\}$ ✓

Q.11. Consider the Non Serial Schedule S :-

T_1	T_2	T_3
R(x)		
	R(y)	
	w(y)	R(y)
w(x)	R(x)	w(x).
	R(x)	

following are the conflicts :-

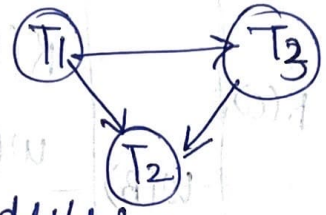


$T_3 \xrightarrow{y} T_2$

$T_1 \xrightarrow{x} T_2$

$T_3 \xrightarrow{x} T_2$

So the conflict precedence graph :-



No any cycle = So this is conflict serializable!

So equivalent Serial Schedule :-

$S' = T_1 \rightarrow T_3 \rightarrow T_2$ = This should be the sequence K|as conflict serializability Seq.

Now for confusing :- they will give you the schedule as follows :-

S_1 : 2RA, 1WB, 1RA, 1WA, 3RB, 3WB, 2WA, 3WA.

2 R A

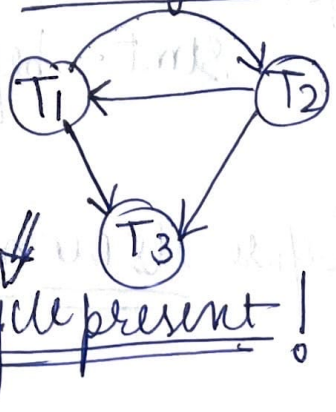
Transactn No. ← 2 ; R → Operatn ; A → data value

So the table will be like :-

T_1	T_2	T_3
	R(A)	
w(B)		
R(A)		
w(A)		
		R(B)
		w(B)
	w(A)	
		w(A)

So not Serializable.

and the graph :-



cycle present!

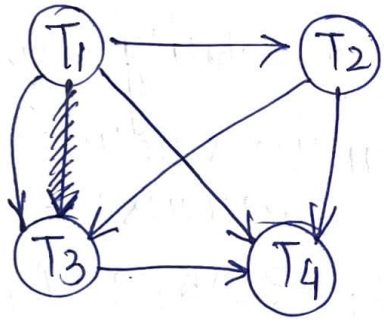
Qu. following schedule is serializable or not?

S: R1A, W1B, R2B, R3C, W1A, R4A, R2C, W4A, W3B, R4B, W4C.

Now the table:-

T_1	T_2	T_3	T_4
R(A)			
W(B)			
	R(B)		
	R(C)		
W(A)			
			R(A)
	R(C)		
			W(A)
		W(B)	
			R(B)
			W(C)

and the graph:-



No any cycle present then equivalent serial schedule:-

T_1 T_2 T_3 T_4

Now, when commit Keyword is used :-

So if in a transactn T_1 = commit is used then you don't need to check for conflict after its commit stmt. before commit only you need to check for all the conflicts possible.

before its own ~~conf~~ commit only.

Ex

	T_1	T_2	T_3
1.	R(x)		
2.		W(x)	
3.		commit	
4.			W(x)
5.			W(y)
6.			commit.
7.	R(y)		
8.	commit		

So here we will search for conflic of T_2 until line NO. 3 only. after that we don't have to look for any conflict.

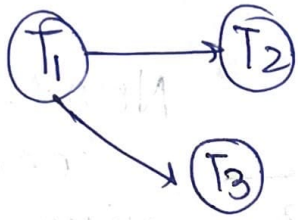
So here:- we have conflicts:-

$T_1 \xrightarrow{x} T_2$

$T_1 \xrightarrow{x} T_3$

No conflicts for T_2.

So graph :-



we will check for conflict till my own transaction is committed after that: no any checking!

* Some Conflict Serializable Schedules are Bad schedules.

conflict serializable schedules.

19/09/2022.

Lesson 03

→ Schedules

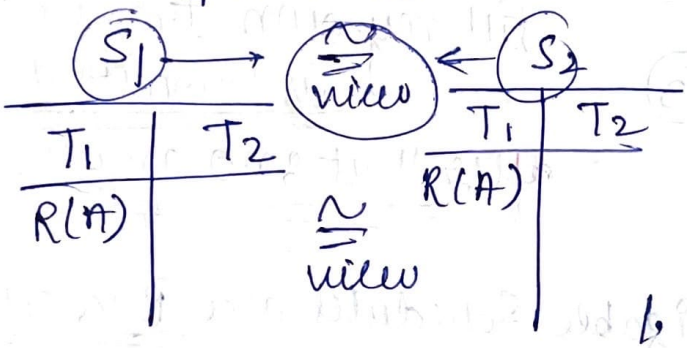
good.

bad.

Some are covered by conflict serializability and the remaining are detected by view seria.

View Equivalence :-

Consider following two schedules :-
S_1 and S_2.



Now the first condition for view equivalence :-

~~If S1T1 is reading~~

1. If T_0 is reading first directly from DB

if S_1T_1 reading R(A) directly from DB first, even before S_1T_2 then S_2T_1 should also read 'A' directly from DB first, before S_2T_2. and they or anyone of S_1T_1 and S_2T_1 should not read updated value of A. :-

S_1			S_2	
T_1	T_2	X	T_1	T_2
R(A)		X		R(A)
W(A)	R(A)	X	R(A)	

} Not view equivalent

Also :-

S_1	
T_1	T_2
R(A)	
W(A)	
	R(B)
	R(A)
W(B)	

here S_1T_2 reading B first directly from DB

S_2	
T_1	T_2
	R(A)
R(A)	
W(A)	
W(B)	
	R(B)

here S_2T_2 reading B's updated value

updates B

⇒ Not view equivalent

So this was 1st Cond^n :- Who is Reading first directly from DB?

Now 2nd Cond^n :- Who is reading from other?

↳ which means :-
Who is reading the updated value of any variable?

↳ if S_1T_2 = reading updated value of A → which was updated in S_1T_1 :- then Simultaneously S_2T_2 should also read the updated value of A which is updated by S_2T_1.

Ex :-

S_1	
T_1	T_2
R(A)	
W(A)	
	R(B)
	R(A)
W(B)	

S_2	
T_1	T_2
	R(B)
R(A)	
W(A)	
	R(A)
W(B)	

view

both read A first

both read B first

both read updated A

here both points satisfied here.

Now 3rd :- who is writing last?

↳ if one transaction (let's say S_1T_i) is writing any value at last then corresponding transaction of other schedule (S_2T_i) should also write the same variable's value at last!

Ex:- following two schedules are view equivalent:-

S_1	
T_1	T_2
R(A)	
	R(A)

$\simeq$ view

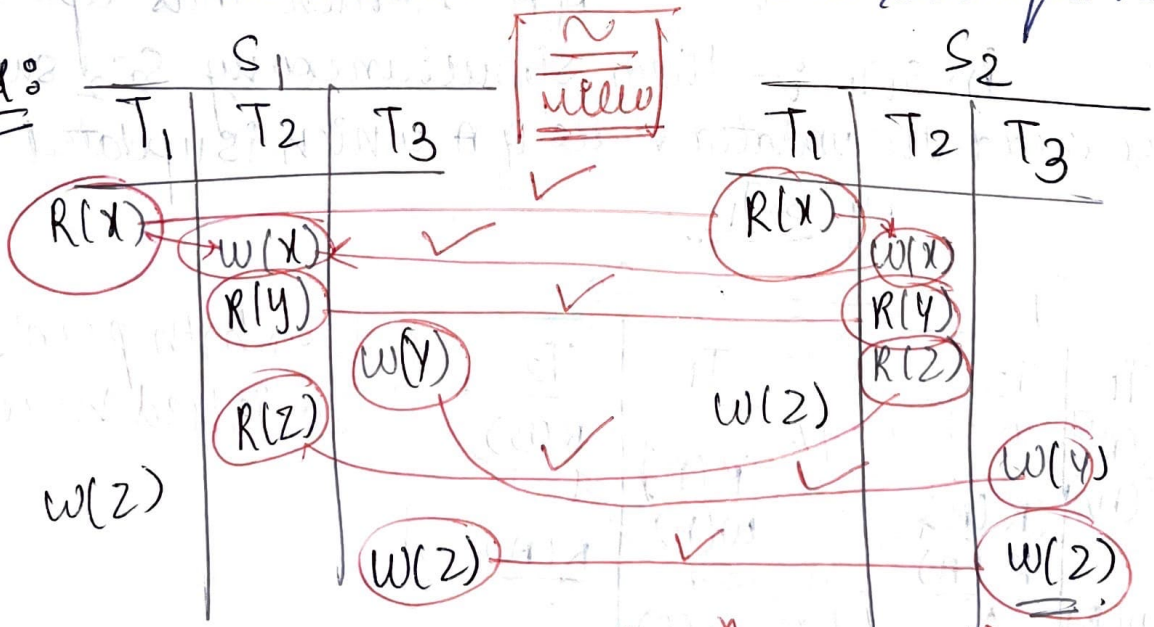
S_2	
T_1	T_2
	R(A)
R(A)	

} now here both are reading value of A from DB directly w/o any updation

↳ if any updation

* all three cond^n's must be true then only two schedules will be view ~~equivalent~~ equivalent.

Ex:



So here all three cond^n's are satisfied.
Here these both are view equivalent

#View Serializability :-

A Schedule is called view serializable. (Non Serial schedule) if it is view equivalent to any of the n! possible serial schedule.

↳ same as we done in conflict seriali.

↳ here also we will draw the graph of precedence.

by drawing the graph in conflic, we selected the suitable serial schedule way.

But here it doesn't work like that

Ex :- T_1 T_2 T_3 T_4

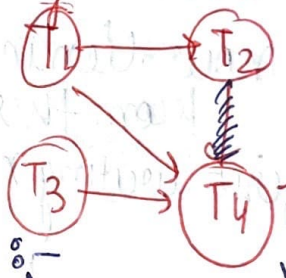
$W(x)$ → $R(x)$ ⇒ → View $T_1 \rightarrow T_2$

T_2 reading x which is updated by T_1 befor it.

$W(x)$ → writing x at last.

no relⁿ b/w T_1 and T_3

So T_1 and T_3 should be completed before T_4.



Should be done at last.

So Serial schedules possible :-

T_1	T_2	T_3	T_4	✓
T_3	T_1	T_2	T_4	✓

Only these two correct!

T_1	T_3	T_4	T_2	✗
T_3	T_1	T_4	T_2	✗

T_2 cannot read after T_4.

T_1 T_3 T_2 T_4 ✗ = T_2 cannot read x after T_3 has updated it.

T_2 has to read x → only after T_1 has updated it.

→ So in view serializibility :- if we have one Schedule, we need to check almost all possible Serial schedules for view serializability.

* we have to check all serial schedules which came from the graph

↳ we have to eliminate among them also.

* So the precedence graph will not give final answer

Ex:

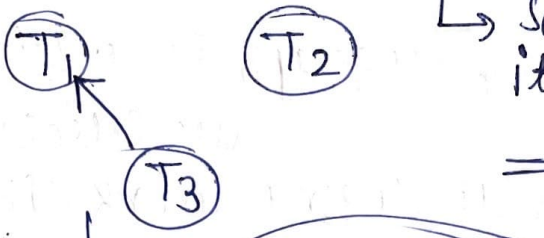
T_1	T_2	T_3
R(X)		
	R(Y)	W(Y)
W(Y)		
	W(X)	

↳ So T_2 can write at any time

T_2 = done at last because he only writing x at last.

T_1 = writing Y at last.
↳ so after T_3 it should come.

no any edge in graph becoz of 2nd point.



$T_2\ T_3\ T_1$
$T_3\ T_2\ T_1$
$T_3\ T_1\ T_2$
} only 3 possibility from graph

now eliminate from these three.

here T_2 first writing x and then T_1 = reading the updated value.
↳ So rejected

Similarly :- 2nd = $T_3\ T_2\ T_1$ = also rejected.

Now 3rd :- $T_3\ T_1\ T_2$:- T_2 reading Y value after T_3 and T_1 has written it.
↳ this also rejected.

→ So no any serial schedule is equivalent to it.

Ex.

T_1	T_2	T_3
R(X)		
	W(X)	
W(X)		

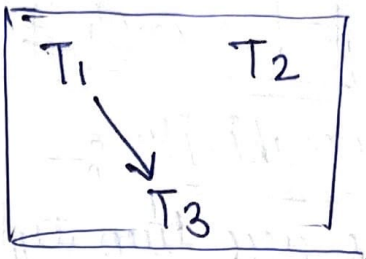
$T_3 \leftarrow T_2$
↑
T_1

W(X)
T_1 reading x after T_2 writing it.

$T_2\ T_1\ T_3$ XX
$T_1\ T_2\ T_3$

Ex:

T_1	T_2	T_3
R(x)		
	R(y)	
w(z)		w(z)
		R(y)
	w(y)	



and T_2 = should be anywhere.

→ possible :-

$T_1\ T_2\ T_3$ ✗

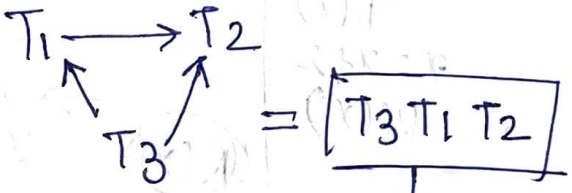
$T_1\ T_3\ T_2$ ✓

$T_2\ T_1\ T_3$ ✗

T_3 reading y after T_2 has updated it

Ex:

T_1	T_2	T_3
w(x)		
	R(x)	
		w(x)
		w(y)
	R(y)	
w(y)		
	w(x)	



This also wrong because T_2 reading y ⇒ it is updated by T_1. ✗

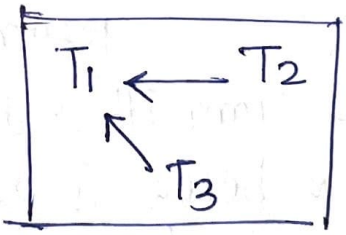
↓

So not view serializable!

Ex:

S1

T_1	T_2	T_3
R(y)		
	w(x)	
w(x)		
		R(y)
	R(y)	
		w(x)
w(x)		



→ $T_3\ T_2\ T_1$ ✓

→ $T_2\ T_3\ T_1$ ✓

} both sequences are correct.

20/9/2022

Lesson04

Recoverability :-

there are two types of schedules

recoverable Not recoverable.

→ recoverable schedules are there :- when no any committed transaction should be rolled Back.

Ex:

S_1

T_1	T_2
R(x)	
x = x+2	
W(x)	
	R(x) → now here T_2 reading the updated values of x, done by T_1 = So this is dirty read
	x = x+3
	W(x)
	commit.
failed	

$x = 5 \to 7 \to 10$

Now when T_1 is rolled back / failed then T_2 also have to rollback because T_2 did dirty read from T_1.

But T_2 = committed transaction :- still T_2 have to rollback becoz T_1 failed.

So here :- T_2 :- committed transaction = have to rollback

So This schedule S_1 = Not recoverable

→ So when commit happens :- the original values are permanently updated to new values so when rolling back the committed transactions, we lost the we will not get the original values!

Hence that schedule is not recoverable.
↳ because of the committed transactions is rolled Back.

Ex: Now when T_2's commit come after T_1's commit :- then our schedule is recoverable !

↳

S_1

T_1	T_2
R(x)	
x = x+2	
W(x)	
	R(x)
	x = x+3
	W(x)
commit	
	commit.

Here T_2 is dirty reading the updated x by T_1
So here first commit should be of T_1 and Then of T_2 = && then our schedule is recoverable.

↳ The Sequence of commit should be in the order in which the values are read from each other.
↳ then our schedule is recoverable !

Now we have following two types of recoverable schedules :-

01. Cascading Recoverable rollback :

T_1	T_2	T_3
R(x)		
x = x+2		
W(x)		
	R(x)	
	x = x+3	
	W(x)	
		R(x)
		x = x+4
		W(x)
* failed		
commit		
	commit	
		commit.

Now if at * :- failed stmt is there :- then t_1 has to roll back :- Now T_2 = reading updated x from T_1 :- So T_2 has to roll back too. Similarly T_3 also have to rollback

↳ So that the original value of x we should get

→ So here T_1 = triggering the rollback of T_2 and of T_3 too.

↳ So this is cascading reco. rollback.

Cascading means :- multiple rollback happening becoz of one failure which are dependent on the failed transaction.

02. Cascadless Recoverable Rollback.

↳ here we will not have such kind of triggering.

T_1	T_2	T_3
R(x)		
x=x+2		
W(x)		
commit		
	R(x)	
	x=x+3	
	W(x)	
	commit	
		R(x)
		x=x+4
		W(x)
		commit

⇒ This is recoverable schedule

now in T_1 :- before commit if it is failed then only T_1 has to rollback and no other transaction.

Similarly if in T_2 → before commit happens :- T_2 is failed then, only T_2 have to rollback and no other transaction

This is cascadless rollback.

↳ only self rollback happens here.

Que.

T_1	T_2	T_3
W(x)		
	W(y)	
	R(x)	
		R(y)
commit		
	commit	
		commit

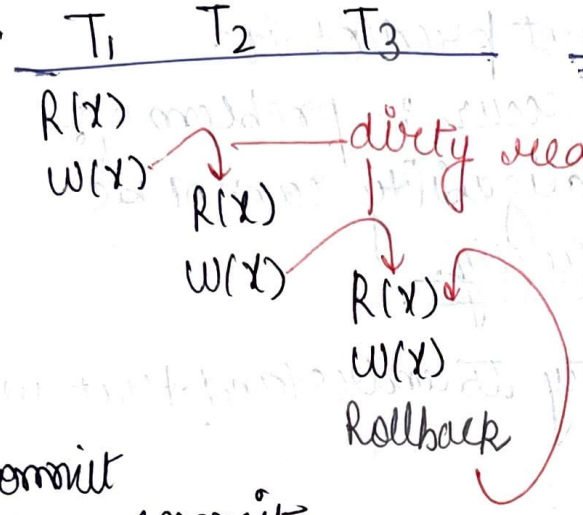
⇒ This schedule is recoverable or not?

⇒ Not recoverable

because if in T_2 rollback happens :- then T_3 have to rollback too :- but it is committed before only.

↳ hence not recoverable!

Ques.



⇒ Now this is recoverable. because no any commit stmt is there which is rolled back.

⇒ and also if T_1 first rolledback before commit happens :- then T_2 also have to rollback because of dirty read.

→ when conflict serializability :-

if precedence graph = given

↓

then to detect if cycle is present or Not

↓

then $O(n^2)$ time

or $O(n+|E|)$ ←

→ view serializability :- if precedence graph = given

↓

find all possible sequences ($n!$) ⇐

for each seq. try to check view equivalence

↳ $O(k \cdot n!)$ time

where k = time to check view equivalence for each seq.

NP-Hard problem

21/09/2022

Lesson 5

In Practical scenario ⇒ actual schedule cannot be predictable → about the sequence & the execution of transactions in concurrent manner.

↳ we cannot predict it.

So this problem will occur :- problem of:- conflict and view serializability cannot be implemented practically.

↳ So how we will try to understand that which schedule is good or bad?

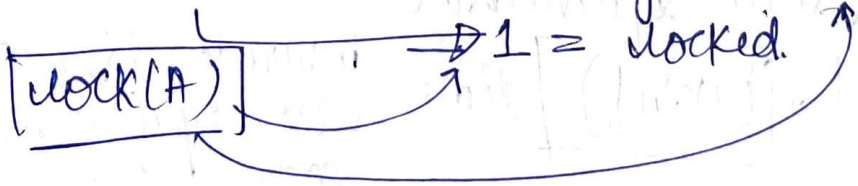
↳ for that we are going to have some practical implementations of the Schedules through the locking protocol.

Locking protocols.

Now what is a lock? → This is exactly like Mutual Exclusion.

↳ So whenever a transaction (T_i) wants to access a data value (A, B, C, ---) then that transacth need to have the lock on that data value before using it. and simultaneously another transaction ($T_2, T_3, ---$) cannot access that particular data value (A) until and unless first transacth has released its lock.

→ lock (data value) = can be binary = 0 → means free
→ 1 = locked.



Ex:

T_1	T_2
lock(A)	
R(A)	lock(A)
W(A)	R(A)
:	
Unlock(A	
	lock(A)
	R(A)

Here T_1 has control on A

this will be failed or not executed because at this time T_1 is accessing A and has made it locked!

So this has to be shifted here.

→ Now if we have three data items = A, B & C :- then for each data item :- a separate lock will be there!
↳ lock(A), lock(B) and lock(C).

→ for all the accesses → we will have to take the lock on that data item.
↳ whether read or write

Now we have following two types of locks :- locks

1. Shared lock :- when a transaction only reads.
2. Exclusive lock :- when a transaction wants to write → lock X

Now consider two transactions :-

T_1	T_2
locks(A)	
R(A)	
	locks(A)
	R(A)

This is allowed!

T_1 = only reads A
↳ So it will try to read A with shared lock.

So that other transaction simultaneously can have the shared lock on A such that Read-Read actions cannot affect the data item

* Just like the readers-writers problem.

Now if another transaction (T_3) comes, then for only reading A purpose → it can have the shared lock on A.

→ now the compatibility :-

	Shared	Exc.
Shared	✓	X
Exc.	X	X

Ex :-

T_1	T_2
lockS(A)	
R(A)	
	W(A) → now this is not possible.
unlockS(A)	
	So this has to be shifted below!
	lockX(A)
	W(A)
	unlockX(A)

→ Now the background picture of lock :- Busywaiting :-
for understand that a particular data item is locked under which mode ? = ~~data~~ Shared or exclusive ?

T_1	T_2
lockS(x)	
R(x)	
	lockX(x)
	W(x)
unlock(x)	
↳ Now locks(x) = 0	
this will take the shared lock	shifted downwards
	this is not possible. as lock on A already acquired by T_1

lock(x) → for each variable, lock will have such kind of information

available	mode
0 → 1 → 0	shared

Now in previous :- lockX(x) = cannot run successfully when T_1 has shared lock on it.

↳ so it will try and try in repetative mode

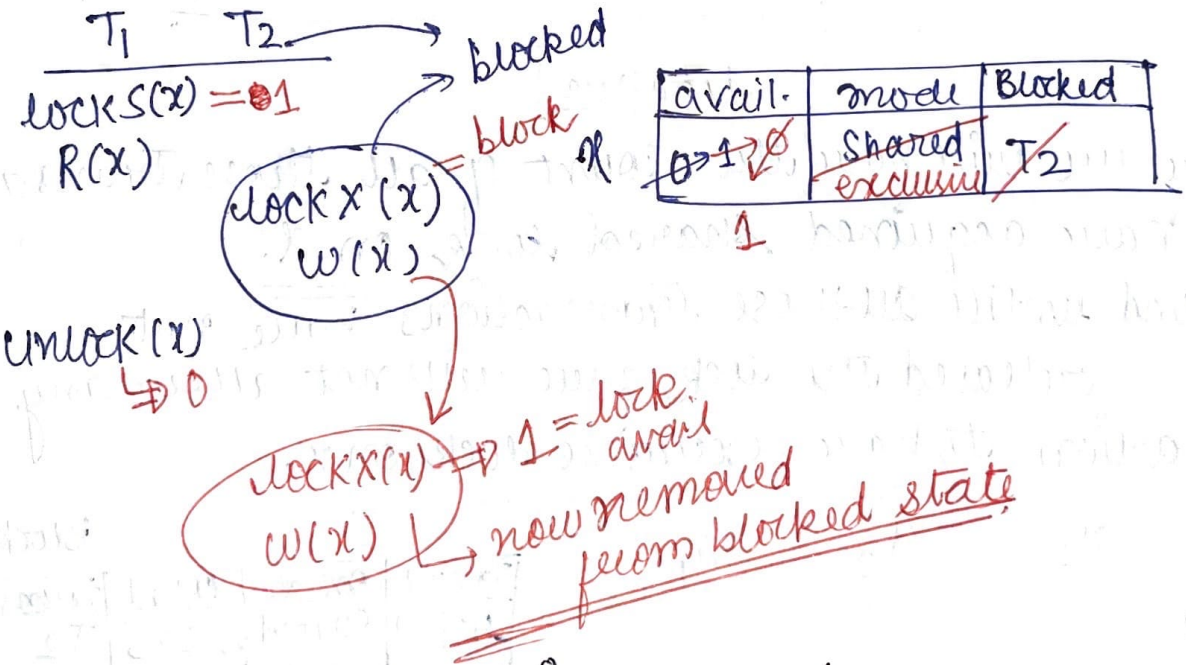
↳ this is called as Busy waiting.

↳ it will keep running.

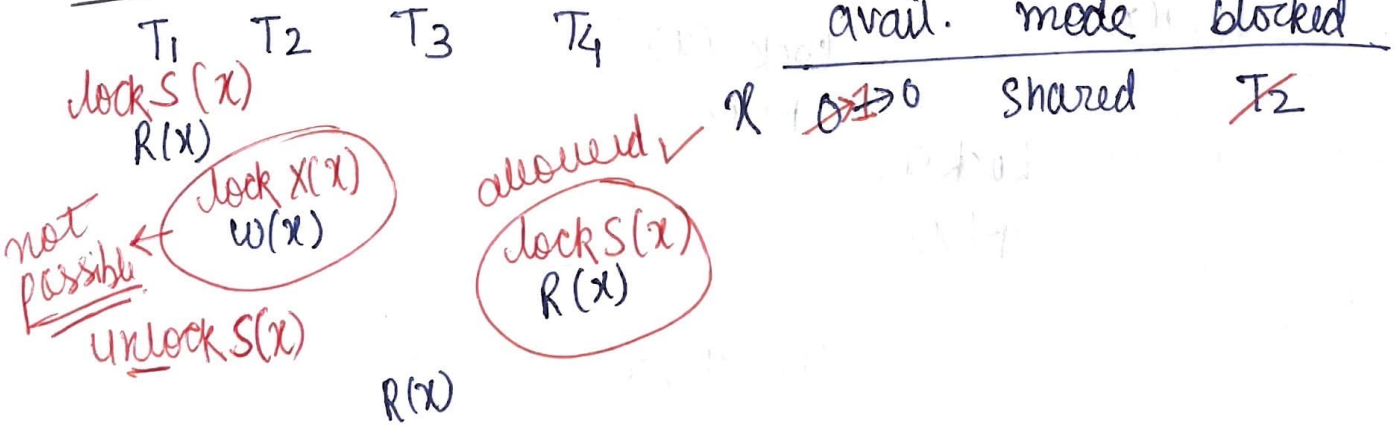
↳ Unecessary taking CPU cycles

then to solve this issue of busy wait :- scientist have thought to block that transaction which is going to Busy wait condⁿ :-

→ Lock : Blocked transaction #



Now when we have multiple shared locks :-



when the unlock S(x) comes in T_1 :- lock becomes again available (0) but still now T_4 is reading in shared mode and now since (lock=0) then T_2 will be no more in blocked state.

and so T_2 will have lockX on x

↳ but it should not be allowed because T_4 is still reading it

↳ but this implementation is allowing it to do so.

↳ you are allowing T_4 = to have shared lock & simultaneously T_2 = —"— excl. lock on x together!

So here we will have the count of all those transactions which have acquired shared lock on x.

and until all those transactions have not released the lock : we will not allow any transaction to have exclusive lock on x.

T_1 T_2 T_3 T_4

LockS(x)
R(x)

LockX(x)
W(x)

LockS(x)
R(x)

LockS(x)
R(x)

unlock S(x)

unlock S(x)

unlock S(x)

Blocked			
avail	mode	count	transac
~~0~~ 1 ~~0~~	Shared	~~1~~ 2 ~~2~~ 3	T_2

~~0~~ ~~1~~ ~~2~~

22/09/2022

#lesson 06#

* Here in our previous example of multiple Shared locks :- T_2 is in Blocked state until all other Transactions have released their locks. and if other three transactions - T_1, T_3 or T_4 acquire the shared lock for read(x) again and again then it might be possible that [T_2 may starve]

So :- Solution :

#Multiple Shared locks w/o Starvation.

T_1	T_2	T_3	T_4
lockS(x)			
R(x)			
	LX(x)		
	W(x)		
		LS(x)	
		R(x)	
			LS(x)
			R(x)

	Lock var.	mode	count	Blocked tran.
x	~~0~~ 1	Shared	1	T_2 T_4 T_3

→ Now, when T_1 runs, it acquires lock on x and then T_2 wants to write → so blocked

now comes T_4 :- T_4 will be allowed to acquire the lock only when no any transaction is blocked. But now T_2 is blocked → So T_4 will not be allowed to have the shared lock on x.

So T_4 = also blocked because we don't want T_2 to starve.

→ if T_2 = not arrived → So No any transaction in Blocked State → So remaining transactions who are only reading x → will be allowed to have the Shared lock on x.

→ now when T_1 unlocks x :- then all the Blocked transactions will be ~~unlock~~ UnBlocked !

Lock Upgrade :-

T_1

LS(x)
R(x)

LX(x)
W(x)

	lock	mode	count
X	~~0~~ 1	Shared	1

Exclusive

→ when only single transaction has acquired the lock, then if first (count = 1) that transaction reads & that variable and then wants to write on the same variable, so here w/o unlocking → lock can be upgraded from Shared to Exclusive.

lock downgrade :- from exclusive to shared lock on same variable by single transaction only.

T_1

lockX(x)
W(x)

lockS(x)
R(x)

	lock	Mode	count
X	0→1	Exclusive	1

Shared

→ here no any checking on count is req.d

Problem with locking mechanism :-

T_1	T_2
LS(x)	
R(x)	
unlock(x)	
	LX(x)
	W(x)
	unlock(x)
LS(x)	
R(x)	
unlock(x)	

→ now the problem is : locking is allowing : unrepeatable read.

→ Locking is allowing the Bad Schedules also because here systematically we are not doing it.

So we have :-

two phase locking protocol (2PL protocols)

↳ Systematic locking mechanism.

↳ very restrictive manner!

→ Basic 2PL
→ Strict 2PL
→ Rigorous 2PL

} three types of 2PL.

Basic 2PL protocols :- So here when a transaction did the Unlock for one (1st time) :- then that transaction is not allowed to lock any database item.

means :- only lock, lock, lock ---- of multiple data items should be done first.

then once unlock happend --- then only unlock, unlock, ---- will happen.

within a single transaction only.

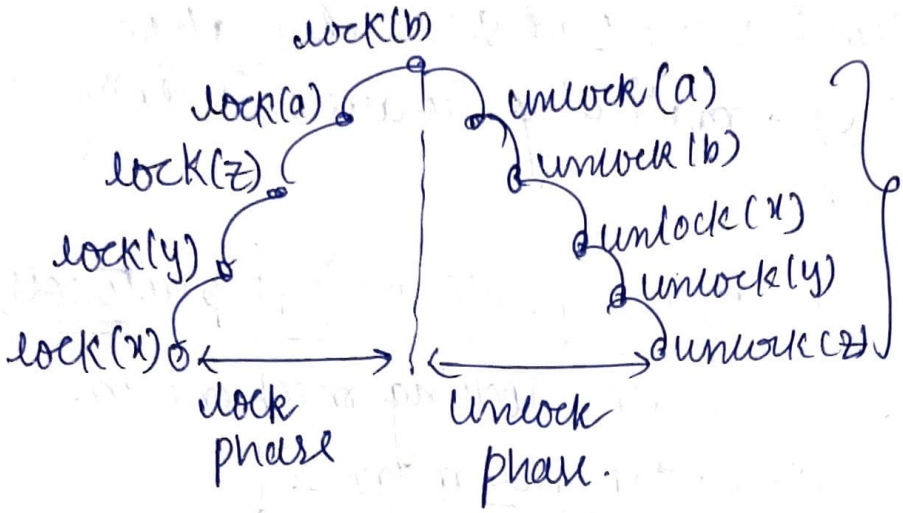
→ So that's why it is two phase : lock phase and unlock phase

Ex :

T_1	T_2
lock S(x)	
R(x)	
	lock X(x)
	W(x)
R(x)	
unlock S(x)	

will not be ~~not~~ allowed.

⇒ postponed here



here sequence don't matter.

Ex:

T_1	T_2
Lock(x)	
R(x)	
Lock S(y)	
unlock(x)	
	lock X(x)
	W(x)
R(y)	
unlock(y)	
	unlock(x).

→ here a better implementation we tried

first acquired all necessary locks then only unlock.

can happen anytime

now another Ex:-

T_1	T_2
lock S(x)	
R(x)	
	LX(x)
	W(x)
	LS(Y)
	R(Y)
LX(y)	
unlock(x)	
W(y)	
unlock(y)	

not allowed → allowed after unlock(x) in T_1

	avail	mode	count	blocked
X	~~0~~ → ~~1~~ → 0	shared	~~1~~ → 0	T_2
Y	~~0~~ → 1	excl		T_2

unlock(y)

Now one imp. statement :-

"Every schedule which is allowed Under basic 2PL is conflict serializable also."

Ex. When we have following Schedule :-

T_1	T_2
lockS(x)	
R(x)	
	lockX(y)
	W(y)
	LX(x) not allowed now
	W(x)
LS(y)	
R(y)	

	Avail.	Mode	count	blocked
X	0→1	shared	1	T_2
Y	0→1	exclu.	0	T_1

Now here both transactions T_1 and T_2 are waiting for each other to Compute!

↳ So this situation is of Deadlock.

Now the Conclusion :-

#Basic 2PL protocol Suffers from Deadlock

→ Now Can we acquire lock with Single instⁿ?

The lockX(A) ⇒ dataBase statement :- might run on CPU in Multiple instⁿs.

But if we have that suitable H/w or architecture of the system then it is possible to execute lockX(A) atomatically.

Here In 2PL Basic protocol :- it is also possible that Small transactions starve due to large transactions.

AND :- vice versa also true : largeer one can also starve due to small-small transactions

Strict Schedule :- it says :-

Every write operation should be ended with commit before other transaction performs read or write on the same DB item.

Ex :-

T_1	T_2	T_3
lockX(x)		
W(x)		
commit		
	R(x)	
		W(x)

Then this should read only the committed value of x in T_1

Strict 2PL protocol :-

It says that :- if we have taken the exclusive lock on a data item then it should not be released until the commit happens on that data item.

Ex :-

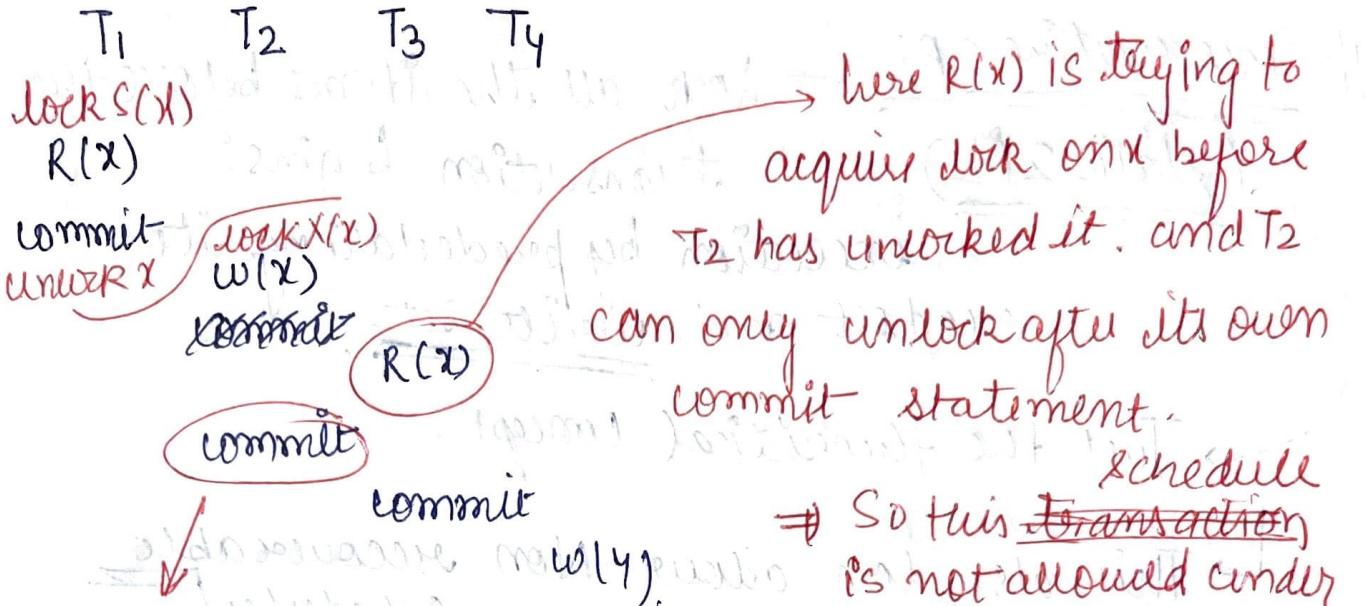
T_1
lockX(A)
W(A)
commit
unlockX(A)

Ex :-

T_1	T_2
LX(x)	
W(x)	
LS(y)	
	LS(x) → will not happen until 'x' is unlocked in T_1 ↳ so T_2 → blocked.
	R(x)
R(y)	
unlock(y)	
commit	
unlockX(x)	
	LockX(y) → here T_2 = unblocked and allowed to execute
	W(y)
	commit

* Strict 2PL allows only strict schedules.

Rigrous 2PL :- Here: Every lock should be released after Commit.



here R(x) is trying to acquire lock on x before T_2 has unlocked it. and T_2 can only unlock after its own commit statement.

⇒ So this schedule is not allowed under rigrous 2PL protocol

if this commit stmt is shifted just after W(x) of T_2 and before R(x) of T_3 then this schedule is allowed under rigrous 2PL.

* Strict and Rigrous 2PL protocol don't have dirty read.

* Strict and rigrous 2PL have only recoverable schedules (cascadeless)

26/09/2022. # Lesson 07 #

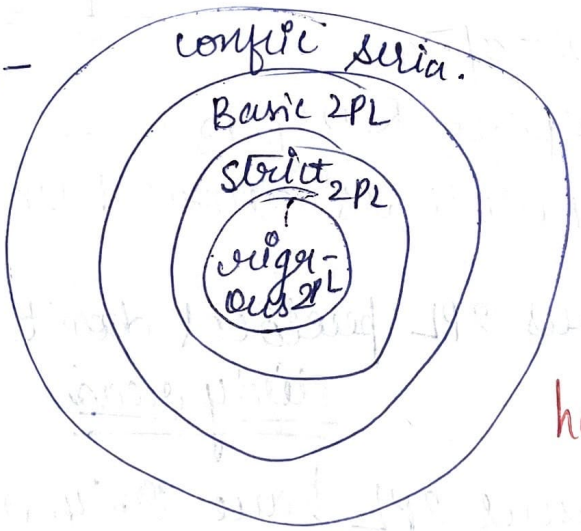
Note :- Basic 2PL may allow non-recoverable schedules.

Conservative 2PL (static 2PL) :- lock all the items before the transaction begins execution by predeclaring its read set and write set.

↳ Just the theoritical concept.

↳ This also can allow Non recoverable schedules.

2PL :-



* Rigorous 2PL may have a deadlock.

Infact all 2PL protocols may have deadlock.

Ex :- following schedules allowed by ? 2PL or not.

T_1	T_2	T_3
	LS(A) R(A)	
		W(A)
W(B)		
	R(B)	
	W(C)	

In T_2 :- whenever we take lock on A then we can unlock it only after its last stmt = W(C) and similarly unlock on B also after last stmt.

and T_1 have to be shifted after wards.

So the given schedule is not allowed in 2PL.
and the given schedule is conflict serializable! (Basic)

Time stamp # (TS)

TS = 2 | TS = T4.

T_1	T_2

T_1 or T_2 = whichever comes first is older transaction and whichever comes last (recent) is younger one.

Here T_1's TS = 2 = older
T_2 TS = 4 = younger.

Now if TS(T_1) = 2
TS(T_2) = 6
TS(T_3) = 5

then younger to older :- [T_2, T_3, T_1]

So timestamp (TS) = time at which the transaction started.

deadlock prevention :- we have following two methods :-

1. wait die 2. wait-wound.

→ assume there are 2 transactions :- Ti and Tj.
Ti tries to acquire lock on a DB item x, which is already locked by Tj.

Now according to wait-die :- An older transaction is allowed to wait for a younger transact^n whereas a younger transact^n requesting an item held by an older transaction is aborted and restarted with same timestamp.

So we have two options :-

$TS(T_i) < TS(T_j) \Rightarrow$ T_i = older, T_j = younger → T_i waits for lock held by T_j

or

$TS(T_i) > TS(T_j) \Rightarrow$ T_i = younger, T_j = older.

↳ abort T_i and restart T_j with same timestamp.

→ Now according to wait-wound :-

A younger transact^n is allowed to wait for an older one, whereas if an older transact^n req. an item held by the younger transac^n, we preempt the younger transaction by aborting it. Restart younger with same timestamp.

So here also two options :-

$TS(T_i) < TS(T_j) \Rightarrow$ T_i = older, T_j = younger } Preempt T_j lock held by T_j and abort T_j.

or

$TS(T_i) > TS(T_j) \Rightarrow$ T_i = younger, T_j = older. } T_i will wait here

→ whenever the older transaction is waiting then Starvation also possible here!

Now if FIFO order is maintained :- then
No Starvation in wait-die and wait-wound.

Que. assume that T_1 request a lock held by T_2.
Consider :- the table which shows the actions taken for wait-die and wait_wound schemes :-

	wait-Die	wait-wound
T_1 = younger than T_2	W	X
T_1 = older — " —	Y	Z

then what will be the correct status of T_1 and T_2 at W, X, Y, Z ?

Soln. lock is held by T_2 and T_1 req. for the lock.
then: W = abort T_1 X = T_1 waits
Y = T_1 waits Z = T_2 aborted and restarted with same timestamp.

→ Read TS(A) :- youngest transaction who reads A.
→ write TS(A) :- " " writes A.

Ex: ① TS=1 T_1 | TS=2 T_2 | T_3 TS=3

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

Initialization :-
R_TS(i) = 0
W_TS(i) = 0

So here :-

R_TS(A) = 0
W_TS(A) = 0

Now when 1st stmt of T_1 executes :-

then $R\_TS(A) = 1$

$W\_TS(A) = 0$

then 1st stmt of T_2 :-

$W\_TS(A) = 2$

when 2nd stmt of T_2 :- then $R\_TS(A) = 1 \rightarrow 2$

because T_2 = younger than T_1

Similarly $W(A)$ of T_3 runs then

$W\_TS(A) = 2 \rightarrow 3$

Now when $W(A)$ of T_2 runs :-

final value $\Leftarrow$ $W\_TS(A) = 3$ → will not change

because T_3 = youngest

Similarly :- youngest transaction to read A is

not T_1 it's T_2

hence final value of $R\_TS(A) = 2$.

Basic timestamp algorithm

for concurrency control.

↳ concurrency control says :-

$T_1 \rightarrow T_2 \rightarrow T_3 \Rightarrow$ execution order

where T_1 = oldest and

T_3 = youngest.

now here if T_2 or T_3 has done write on any DB item :-

then T_1 should not read it.

↳ then ordering will be lower a dictory purpose

So :- if $TS(T_1) < TS(T_2) < TS(T_3)$ - - - -

T_1 T_2 T_3
R(x) or W(x)

W(x).

→ So here T_1 = doing write after the younger trans. has read/write it = So order violated here

→ So here T_1 = rolled back. and restarted with younger timestamp.

→ So we don't want to perform a write opr^n after a younger trans has done read/write.

T_1 T_2 T_3 - - - -
W(x)

R(x)

→ Here younger transactions T_2, T_3 - - - has written on 'x' and then T_1 is reading that x :- then abort T_1 and rollback it and restart T_1

Now consider :-

T_1 T_2 / T_3 - - - -
R(x) or W(x)

W(x).

So here the W(x) opr^n of younger transaction is allowed and set :-

$W\_TS(x) = TS(T_2 \text{ or } T_3 \text{ or } \text{---})$

↓

whoever will be youngest to write on x.

T_1 T_2
W(x)
R(x)

Now here R(x) of T_2 also allowed since T_2 is younger one to read x and

Set $R\_TS(x) = TS(T_2)$ ⇒ time stamp of youngest

* Restarted ~~timestamp~~ transaction gets a younger timestamp.
↳ whichever trans. we have aborted.

* Concurrency problem arrives only when atleast one write operation done on shared / common data Base item.

Ex: T_1 T_2 T_3 Now R_TS(x) = 0, W_TS(x) = 0 } initially

T_3: R(x) ↳ no read/write before this then R_TS(x) = 3

T_2: R(x) ↳ This also allowed → as no one has written.

T_1: R(x), W(x) ↳ now this has not allowed.

because for write(x) = we have to check whether a younger one has read or written on it?

↳ yes here → So T_1 = aborted and restarted

Ex:- now check algo on :-

T_1 T_2 T_3 T_4

Initially :-
R_TS(x) = 0
W_TS(x) = 0

T_4: R(x) ↳ no read/write before this so :- allowed
R_TS(x) = 4
W_TS(x) = 4

T_3: W(x) → R_TS(x) = 4 > 3 → T_3 also aborted

T_2: R(x), W(x) → R_TS(x) = 4 > 2 → So T_2 = aborted.

T_1: also aborted not allowed ← XX R(x), W(x) → R_TS(x) = 4 > 1

27/09/2022 # Lesson 08#

Ex: use Basic timestamp algo:-

T_1 T_2 T_3

$R(x)$ ↙

$W(x)$ ↙

$W(x)$

↓

not done because $W\text{-}TS(x) > 1$

↓

2 → So T_1 = aborted.

initially:-

$R\text{-}TS(x) = 0 \rightarrow 1$

$W\text{-}TS(x) = 0 \rightarrow 2 \rightarrow 3.$

$W(x)$ ↙

→ done:- $W\text{-}TS(x) = 2 < 3$

→ allowed. 2 ↙

Now Consider this:-

T_1 T_2 T_3

$R(x)$ → reads 5.

$W(x) = 10$

$W_1(x) = 15$

$W(x) = 25$

now according to Basic TS algo:- the T_1's write operation will be done first then T_2 and then T_3.

Timestamp ordering:- $T_1 \rightarrow T_2 \rightarrow T_3$

then value of x after:-

$T_1 \rightarrow T_2$:-

$x = 5$ (initially)

after T_1 = $x = 15$

after T_2 = $x = 10$ → So here the T_1 ka contribution in x is neglected.

we have completed T_1:- but its changes are overwritten.

- T_1's write operation will not have any effect on DB value → still T_1 performed & completed.

and then T_3 will write on x according to timestamp ordering :- T_3 will write 25 on x → final DB value

So when $T_1 \rightarrow T_2$:- our expected result = $\boxed{x=10}$

So why are you doing the write oper^n in T_1 = just skip it → according to Thomas rules

→ So thomas ~~write~~ write rule :-

→ Read is same as Basic timestamp rule

→ write (A) in transaction T :

- if R_TS(A) > TS(T) then abort T, rollback and restart T
- Else if W_TS(A) > TS(T) then skip the write operation.
- Else perform write(A) of T and update $\boxed{W\_TS(A) = TS(T)}$

Qu. T_1 T_2 T_3 T_4 — Initially R_TS(x) = 0, W_TS(x) = 0

R(x) ✓

R(x) ✓

W(x) ✓

R_TS(x) = 2 → 3

W_TS(x) = 0 → 4

W(x)

↳ for this W_TS(x) = 4 > 3

↳ skip this write oper^n.

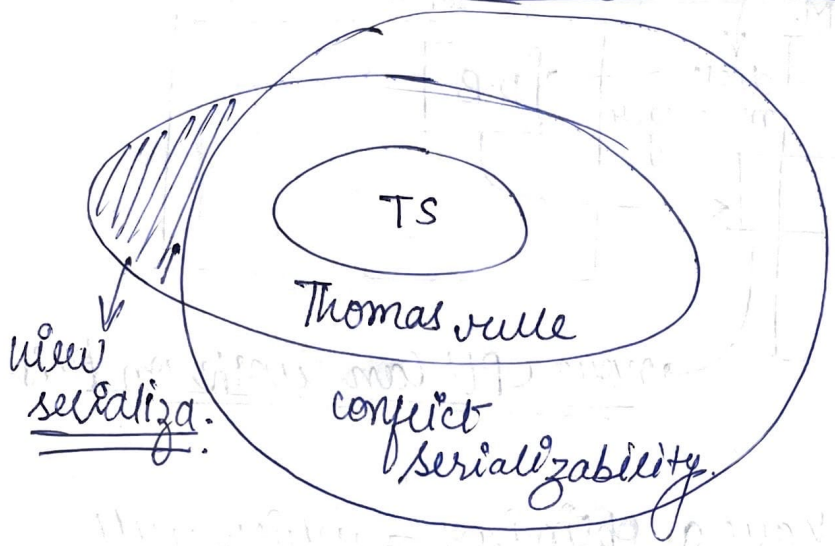
W(x)

↓

R_TS(x) = 3 > 1

↳ abort T_1

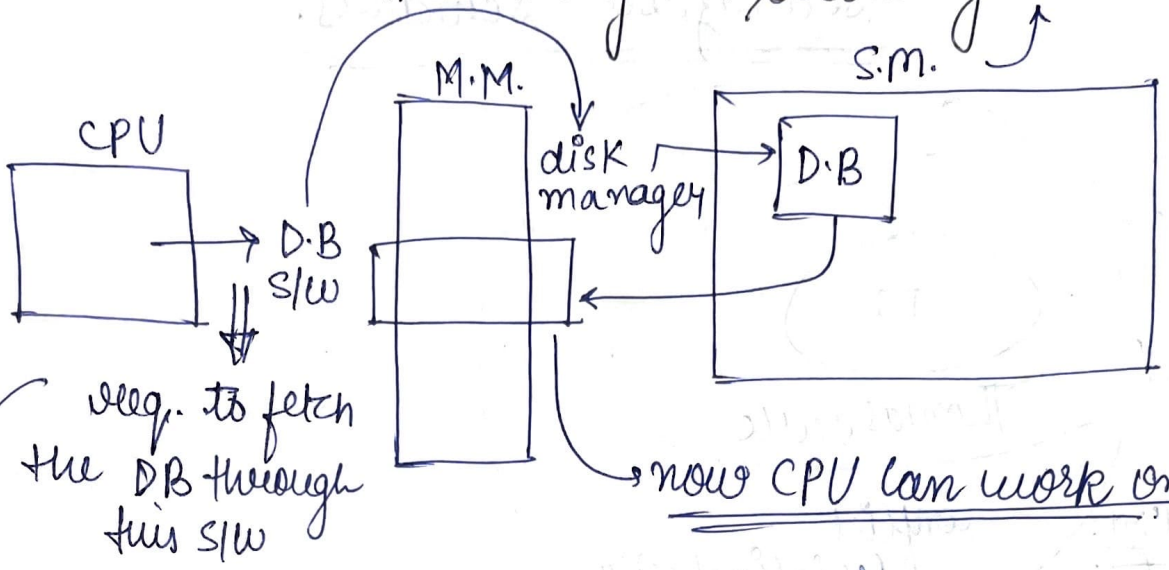
* Basic TS allows conflic serializable schedules
* Thomas rule allows more than conflict serializable schedules.



28/09/2022

Memory structure

↳ main memory ↳ secondary memory.



↳ here in CPU we have a optimizer = which will analyse the i/p SQL query through which how fastly & (efficiently) secondary memory because the access time of secondary memory = is v. high

↳ optimizer will efficiently analyse & run the SQL query such that min^m times secondary memory have to access.

Disk blocks and record storages

↳ OS divides the secondry storage (disk) into several sized blocks

Now assume we have created blocks = each of 128 bytes and consider a file of DB(Table) = 256 records

record size = 16 B

then no. of records to be stored per block = $\frac{128\,B}{16\,B}$

= 8 records

and no. of blocks needed to store D.B file = $\frac{256}{8} = 32$.

Now assume those 256 records belong to students table with R.NO as a col.

and we are running the following query:-

```
Select * from Students
where R.NO. > 5 and R.NO. < 50
```

→ So as of now :- the query processor don't know exactly that which record is present on which block? ⟹ So it will go through all the blocks. and have to spend a lot of time.

So here the file should be stored in an organised way so that time for searching should be min^m and the performance should be improved.

↳ The storage of the DB file should follow some pattern and it should be accessible directly otherwise we have to search for a single record linearly :- linear search.

* we can have records of :- fixed length or variable length.

These are v. easy to maintain

not easy to maintain

little difficult technique.

saves a lot of space.