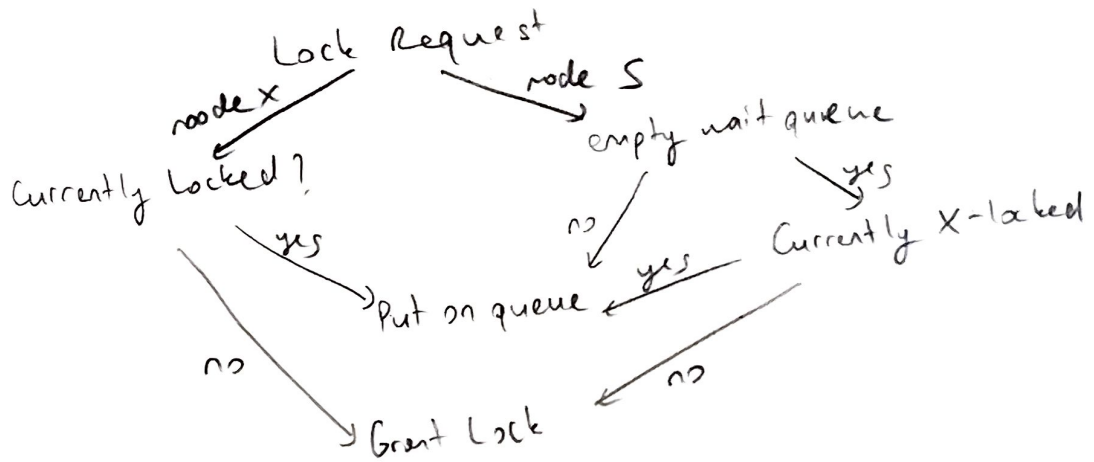
week 7

Handling lock request

Lock entry has

oid	mode	List (holding lock)	List (Queue)
id	S, X	T_1, T_2 --	T_3, T_4 --



Deadlock → detection or prevention

Deadlock Detection

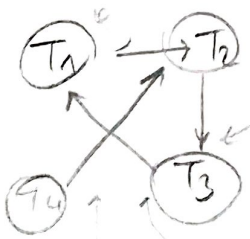
- Waits for graph

  $T_i \rightarrow T_j$, T_i is waiting for T_j

- Periodically check for deadlock (simple hack → timeout)
- Shoot some txn to break the cycle.

ex:

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



deadlock

Shoot something to break the cycle

Deadlock Prevention

- Instead of detecting deadlocks after happening, avoid any deadlock from happening.

Wait-Die

- Assign timestamp to each txn.
- Older one has priority.

ex: T_J has lock, T_i request the lock

case 1: T_i is older, it waits.

case 2: T_i is newer, it dies.

Older	- newer
waits	- dies

After abort, restart with original timestamp. So that it can have the turn.

Wound-Wait

- Older has priority

ex: T_J has lock, T_i request

case 1: T_i is older, abort T_J

case 2: T_i is newer, waits.

✓ (same)

Prevention vs Detection

- Prevention is costly.
- Deadlock is not happening too frequently. If so, rethink your design (txns)

Phantom Problem

- So far db was static (no insert/delete).

ex.

T1	T2
S* where blue	
	insert into blue
S* where blue	

either $T_1 - T_2$ or $T_2 - T_1$. Both are possible schedule.

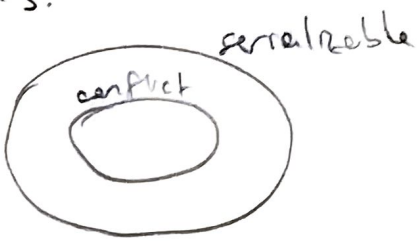
This (schedule) is not equal to them

It is not serializable

- suppose initially there are 2 blue products X_1, X_2.

$R_1(X_1)\ R_1(X_2)\ W_2(X_3)\ R_1(X_1)\ R_2(X_2)\ R_1(X_3)$

This is conflict serializable. No conflicting actions.

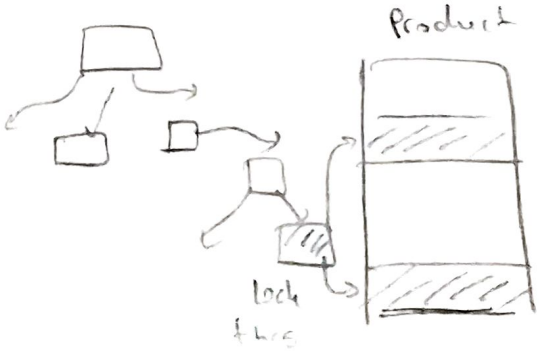


- Conflict serializable
- Not serializable (due to phantom)

- Strict 2PL is not enough for serializable.

* In static db (no insert/delete)
  conflict seri $\Longrightarrow$ serializable
* In dynamic db (insert/delete ✓)
  conflict s $\not\Longrightarrow$ serializable

Solutions
- Lock entire table
- Lock index of blue, if index available.
- Use predicate lock (color="blue")
  ↳ hard to implement, arbitrary predicates



- Dealing with phantoms is expensive

Set txn [read only] → This will not acquire any locks.
↳ Good for performance if you don't care the exact result.

— — count*
— —
— —

Commit

Isolation Levels

1) Dirty Reads

- set ~~txn~~ iso. level Read uncommitted
- I'm fine with reading half-completed.
- Long durating Write locks (Strict 2PL)
- No read locks → dirty / incensistent reads

2) Read Committed

→ _____ read committed

- write Long lock (strict 2PL)
- read short lock (not 2PL) → only acqure while reading.
  ↳ when reading same element twice, (unrepeatable reads) may get 2 diff. values.

ex: Tx1
db.setIsoLevel(read-committed);
db.autocommit(false)
read Acc
sleep
readAcc → might get diff values.

3) Repeatable Read

- I want to read as the same
  - iso level repeatable read.

- Long write (strict 2PL)
- Long read (strict 2PL)

} → ☆ Not serializable

ex

TX1

db.setTxnIso(repeatable read)
autocommit(false)
read
sleep
read

→ will read the same

ex: TX1

TX2

countAcc()
sleep
countAcc()

insertAcc()

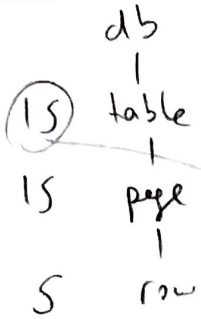
→ can see a different count.

4) Serializable

Long write (strict 2PL)
Long read (//)
- deals with phantom
- It locks entire table

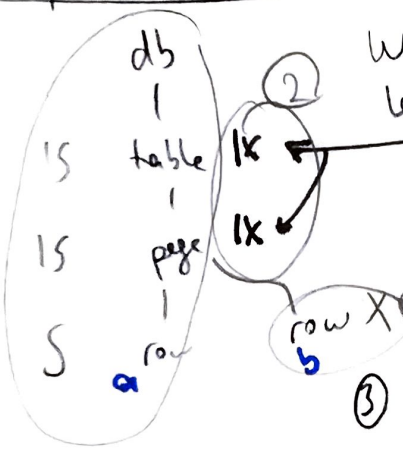
★ Difference between serializable and repeatable read

Serializable



When a txn wants to insert, it should put X lock to the table.
However, the table has IS lock. It will wait.

Repeatable read



When a txn wants to insert, it will put IX to upper levels

A page have different pages

1 Since TX2 acquired the lock (3), it will write.
When Tx1 reads the entire table (the page has both (a, b) rows now)
it will find both entities.

Remember

Hierarchical Locking

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	no
IX	✓	✓	no	no	no
S	✓	no	✓	no	no
SIX	✓	no	no	no	no
X	no	no	no	no	no

When the element (page, row, db, table) has IS lock, the txn that request the lock will wait if it wants X lock on the element

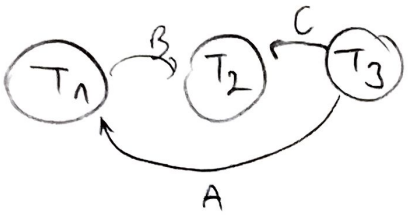
When a tx1 has S lock on the element, other tx2 can also use S lock.

EXAMPLES

EX1: Find all conflicts

$w_3(A)\, r_1(A)\, w_1(B)\, r_2(B)\, w_3(C)\, r_2(C)$

Is it conflict serializable?



Yes, conflict serializable, no cycle.

Serial equivalent?

- Try to swap nonconflicting s

$w_3(A)\, w_3(C)\, r_1(A)\, w_1(B)\, r_2(B)\, r_2(C)$

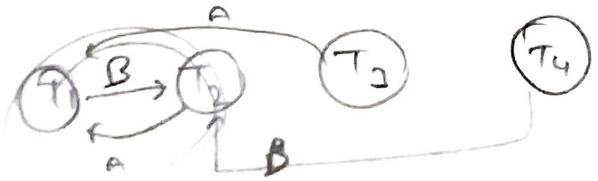
only serial eq. $\boxed{T_3 - T_1 - T_2}$

ex2: Find all conflicts

not necessary 2 to 2.

$r_1(A)\, r_2(A)\, r_1(B)\, r_2(B)\, r_3(A)\, r_4(B)\, w_1(A)\, w_2(B)$

Conflict serializable?



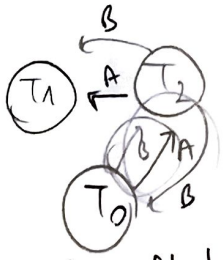
cycle, not conflict serializable

ex3: Conflict serializable?

T_0	T_1	T_2
$r_0(A)$		
$w_0(A)$		
		$r_2(A)$
		$w_2(A)$
	$r_1(A)$	
$r_0(B)$		
		$r_2(B)$
$w_0(B)$		
		$w_2(B)$
	$r_1(B)$	
C		
C		C

use 2PL

T_0	T_1	T_2
$L_0(A)$		
$L_0(B)$	it can acquire B later, but it cannot release A before acquiring B. (2PL)	
$R_0(A)$		
$W_0(A)$		
		$L_2(A)$ //denied, waits
$U_0(A)$		
		//will acquire lock on A
		$r_2(A)$
		$w_2(A)$
	$L_1(A)$ //denied, waits	
$r_0(B)$		
$w_0(B)$		
$U_0(B)$		
		$L_2(B)$ ✓ granted
		$R_2(B)$
		$W_2(B)$
		$U_2(A)$
		$U_2(B)$
	//will acquire lock on A	
	$L_1(B)$	
	$R_1(A)$	
	$R_1(B)$	



cycle, not conflict ser.

another solution with different schedule

T_0	T_1	T_2
$L_0(A)$		
$L_0(B)$		
$r_0(A)$		
$W_0(A)$		
		$L_2(A)$ //denied
	$L_1(A)$ //denied	
$U_0(A)$		
		//it will have A lock
		$L_2(B)$ //denied
$r_0(B)$		
$w_0(B)$		
$U_0(B)$		
Commit		
		//will have B lock
		$r_2(A)$
		$w_2(A)$
		$U_2(A)$
	//will get A lock	
	$L_1(B)$ denied	
		$r_2(B)$
		$w_2(B)$
		$U_2(B)$
		Commit

Continues

EX4: Strict 2PL

wait for graphs

T_1	T_2	T_3
r(A)		
	r(A)	
		r(D)
		w(D)
		w(A)
	r(C)	
w(B)		
	w(B)	

deadlock detection

Locks

op	A	B	C	D
$r_1(A)$	S_1			
$r_2(A)$	S_1, S_2			
$r_3(D)$	S_1, S_2			S_3
$w_3(D)$	S_1, S_2			X_3
$w_3(A)$	S_1, S_2			X_3
$r_2(C)$	S_1, S_2		S_2	X_3
$w_1(B)$	S_1, S_2	X_1	S_2	X_3
$w_2(B)$	S_1, S_2	X_1	S_2	X_3
Commit T_1	S_2	X_2	S_2	X_3
Commit T_2	X_3	–	–	X_3
Commit T_3	–	–	–	–

no wait queue, it can upgrade

$T_3 \xrightarrow{A} T_1$, $T_3 \xrightarrow{A} T_2$ — T_3 will wait for T_1 and T_2 on A

T_3 is delayed

same graph element in B (X_1).

T_2 will be put in queue.

T_2 is delayed

$T_3 \xrightarrow{A} T_1$, $T_3 \xrightarrow{A} T_2$, $T_2 \xrightarrow{B} T_1$

will release locks

T_2 can enter

$T_1 - T_2 - T_3$ equivalent

wait-die
old young

TimeStamp $T_1 = 1$ → oldest
$T_2 = 2$
$T_3 = 3$ → youngest

Locks

op	A	B	C	D	comment
$r_1(A)$	S_1				
$r_2(A)$	S_1, S_2				
$r_3(D)$	S_1, S_2			S_3	
$w_3(D)$	S_1, S_2			X_3	
$w_3(A)$	S_1, S_2			—	T_3 is young. T_1 has the lock. T_1 is old. T_3 dies
$r_2(C)$	S_1, S_2		S_2		
$w_1(B)$	S_1, S_2	X_1	S_2		
$w_2(B)$	S_1	X_1			T_2 wants X lock on B. T_1 has the lock. T_1 is old. T_2 dies

T_1 commits.

Now, T_3 or T_2 will be started again.

We dont know when.

Since we are starting with old TS (timestamp), they wont starve

Optimistic

Until now, we deal with pessimistic (locks).

optimistic → assumes there won't be many conflicts

↳ abort when conflict is detected

Timestamps

- Each txn have unique ts(t)
- Timestamp order defines serialization order of the txn.

with each element X

RT(X) → highest ts of read X.

WT(X) → // any(read/write) X

C(X) → commit bit. highest ts that wrote X committed

☆ when $R_T(X)$ requested

$W_u(X)$ ---- $R_T(X)$

check $TS(U) \leq TS(T)$

ex: $TS(U)=1$
$TS(T)=5$ → newer

$W_u(X)$ --- $R_T(X)$ → check $\underbrace{WT(X)}_{1} > \underbrace{TS(T)}_{5}$

↳ $wt(X)=1$

daha sonraki txn yazmış bitirmiş. Ben yeniyim. Şimdi okuyorum OK ✓

ex: $TS(U)=5$ → new
$TS(T)=1$ → old

$W_u(X)$ --- $r_T(X)$ → check $\underbrace{WT(X)}_{5} > \underbrace{TS(T)}_{1}$

↳ $wt(x)=5$

- Benden sonra gelen txn yazmış.
- roll back T. (Read too late) ☆

- T's timestamp is $\boxed{\text{new}}$, when it is restarted.

★ when $W_T(X)$ requested

$r_U(X) \dots w_T(X)$

ex. $TS(T) = 1 \rightarrow$ old
$TS(U) = 5 \rightarrow$ new

$r_U(X) \dots w_T(X) \rightarrow$ check $\underbrace{RT(X)}_{5} > \underbrace{TS(T)}_{1}$

$\rightarrow rt(x) = 5$

Benden sonra gelen adam okumuş.
Yazmak için geç kaldım.
Write too late ★

T needs to rollback

ex: $TS(T) = 5 \rightarrow$ new
$TS(U) = 1 \rightarrow$ old

$r_U(X) \dots w_T(X) \rightarrow$ check $RT(X) > TS(T)$
$1 > 5$

$\rightarrow rt(X) = 1$

no. It is ok. Benden sonra gelen adam okumuş. Problem yok.

Thomas' Rule

★ when $W_T(X)$ requested

$W_U(X) \dots W_T(X)$

case 1: $TS(T) = 5 \rightarrow$ new
$TS(U) = 1 \rightarrow$ old

start(U) start(T) ... $W_U(X) \dots W_T(X)$

$WT(X) = 1$

check $WT(X) > TS(T)$
$1 > 5$

Yaz. Ben daha sonra yazan lazım ✓
write and update $\underline{WT(X) = TS(T)}$

case 2. $TS(T) = 1 \rightarrow$ old
$TS(U) = 5$ new

$W_U(X) \dots W_T(X) \rightarrow$ check $W_T(X) > TS(T)$
$5 > 1$

$\rightarrow WT(X) = 5$

Benden sonra gelen adam yazmış. Zaten böyle olmalı. Ignore write & continue.

ex5: Basic timestamping

T1	T2	T3
r(A)		
	r(A)	
		r(D)
		w(D)
		w(A)
	r(C)	
w(B)		
	w(B)	

initially $T_1 = 1$, $T_2 = 2$, $T_3 = 3$

	Op	A rt	A wt	B rt	B wt	C rt	C wt	D rt	D wt
	$r_1(A)$	1							
1	$r_2(A)$	2							
	$r_3(D)$							3	
2	$w_3(D)$							3	3
3	$w_3(A)$		3						
	$r_2(C)$					2			
	$w_1(B)$				1				
4	$w_2(B)$				2				
☆									

1 $r_T(X)$ is requested. Look at prev write time $WT(A) = 0$

$W_T(X) > TS(T)$

$0 > 1$ no. Benden sonra gelen eden yazmış. OK.

2 $w_T(X)$, $w_3(D)$ is requested

$\underbrace{W_T(X)}_{3} > \underbrace{TS(T)}_{3}$ Benim zaten. ✓

3 $w_3(A)$

$WT(X) > TS(T)$

$0 > 3$ ✗ ok

4 $w_2(B)$

$WT(X) > TS(T)$ → Benden sonra gelen eden yazmış OK.

$1 > 2$

☆ In this position, what if I have $W_2(A)$?

$\underbrace{W_T(X)}_{3} > TS(T)$

$3 > 2$ → Benden sonra gelen eden yazmış bile ignore $W_2(A)$.

☆ What if I have $W_1(C)$?

$WT(X) > TS(T)$ → no write before

$RT(X) > TS(T)$ → Benden sonra gelen eden, ben yazmadan okumuş. Abort T_1.

$2 > 1$

Algorithm so far

T wants to read X

If $WT(X) > TS(T)$ roll back T → benden sonra gelen adam yazmış.

Else → benden sonraki yazmış, bitmemiş ✓

read & update $rt(X)$ to larger of $TS(T)$ or $RT(X)$

T wants to write X

If $RT(X) > TS(T)$ → benden sonra gelen adam okumuş bile

Rollback T.

Else if $WT(X) > TS(T)$ → benden sonra gelen adam yazmış zaten üstüne yazacaktı.

Ignore write & continue

else write and update $WT(X) = TS(T)$

Commit Bit

Prev. schedule does not work correctly if abort happens.

$T(U)=1$
$T(T)=2$
$W_U(X)$ --- $W_T(X)$ → $WT(X) > TS(T)$
↳ $wt(1)$ $\quad 1 > 2$

normally it is ok.

However, if the $C(X)$ bit is false, T needs to wait until it becomes true.

lost item that write on X did not finish yet.

$T(T)=1$
$T(U)=2$
$W_U(X)$ ---- $W_T(X)$ -- abort U
↳ $WT(X)=2$ $\quad$ ↳ $RT(X) > TS(T)$
$WT(X) > TS(T)$
$2 > 1$

In thomas rule, it is skipped. (Benden sonra gelen adam yazdı zaten) Benimkisini üzerine yazacaktı

Ama, skip yapılırsa info is lost.

Algorithm with Commit Bit

T wants to read X.
if $WT(X) > TS(T)$ → rollback. Benden sonra gelen yazmış.
else if $C(X)$ = false, wait
else read & update $rt(x)$ larger of $TS(T)$ or $RT(X)$

T wants to write X
if $RT(X) > TS(T)$ → rollback Benden sonra gelen okudu
else if $WT(X) > TS(T)$ → benden sonra gelen adam yazmış.
if $C(X)$ false → wait → abort yapersa info kaybolmasın
else ignore write → benimkisini üstüne yazacaktı zaten
else write & update $WT(X) = TS(T)$, $C(X)$ = false

ex Timestamp with commit bit.

T_1	T_2	T_3	T_4	A
1	2	3	4	RT=0 WT=0 C=true
	$w_2(A)$			WT=2 ★ RT=0 C=false ★
$r_1(A)$ aborts Because WT(X)>TS(T) 2 > 1 newer trn already written				
		r3(A) delay WT(X)>TS(T) 2 > 3 X C=false		
	C			WT=2 RT=0 C=true ★
		r3(A) WT(X)>TS(T) 2 > 3 X C=true read ✓		WT=2 RT=3 ★ C=true
			$W_4(A)$ RT(X)>TS(T) 3 > 4 X WT(X)>TS(T) 2 > 4 X write ✓	WT=4 ★ RT=3 C=false ★
		$W_3(A)$ delay, commit bit is false		
			abort → callbacks	WT=2 RT=3 C=true
		$W_3(A)$ ✓		WT=3 ★ RT=3 C=false ★