CMPT 321
FALL 2017

Concurrent DB operations: short version

By Marina Barsky

Serializable schedules

- **Serial Schedule** = All actions for each transaction are consecutive.

  r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B); …

- **Serializable Schedule**: A schedule whose "**effect**" is equivalent to that of some serial schedule.

Conflicts: summary

There is a conflict if one of these two conditions hold:

1. A read and a write of the same X, or
2. Two writes of the same X

- Such actions conflict in general and *may **not** be swapped in order*.
- All other events (reads/writes) of 2 different transactions may be swapped without changing the **effect** of the schedule.

A schedule is ***conflict-serializable*** if it can be converted into a **serial** schedule by a series of **non-conflicting swaps** of adjacent elements

Example 1

Transaction 1	
r(A)	A-100
w(A)	
r(B)	B+100
w(B)	

Transaction 2	
r(A)	A*1.01
w(A)	
r(B)	B*1.01
w(B)	

Example 1: swapping non-conflicting actions

Transaction 1	Transaction 2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

Example 1: End up with a serial schedule

Transaction 1	Transaction 2
r(A)	
w(A)	
r(B)	
w(B)	
	r(A)
	w(A)
	r(B)
	w(B)

Example 1 conclusion: can use original schedule

Transaction 1		Transaction 2	
r(A)	A-100		
w(A)			
		r(A)	A*1.01
		w(A)	
r(B)	B+100		
w(B)			
		r(B)	B*1.01
		w(B)	

Example 2

Transaction 1	
r(A)	A-100
w(A)	
r(B)	B+100
w(B)	

Transaction 2	
r(A)	A*1.01
w(A)	
r(B)	B*1.01
w(B)	

Example 2: cannot swap conflicting actions

Transaction 1	Transaction 2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
r(B)	
w(B)	

Example 2 conclusion: this schedule should be rejected

Transaction 1		Transaction 2	
r(A)	A-100		
w(A)			
		r(A)	A*1.01
		w(A)	
		r(B)	B*1.01
		w(B)	
r(B)	B+100		
w(B)			

Testing schedule for serializability

- Non-swappable pairs of actions represent potential conflicts between transactions.
- The existence of non-swappable actions enforces an **ordering** on the transactions that include these actions.

We can represent this order by a **precedence graph**

- **Nodes**: transactions $\{T_1,\ldots,T_k\}$
- **Arcs**: There is a directed edge from T_i to T_j if they have conflicting access to the same database element X and T_i is first:

    written $\mathbf{T_i <_S T_j}$.

Precedence graphs: example 1

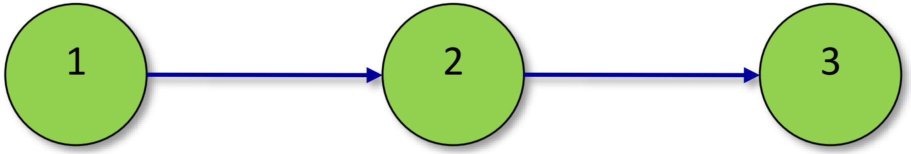
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Note the following:

- $w_1(B) <_S r_2(B)$
- $r_2(A) <_S w_3(A)$

➢ These are conflicts since they contain a read/write on the same element

➢ They cannot be swapped. Therefore $T_1 < T_2 < T_3$



Conflict-serializable

Schedule accepted

Precedence graphs: example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Note the following:

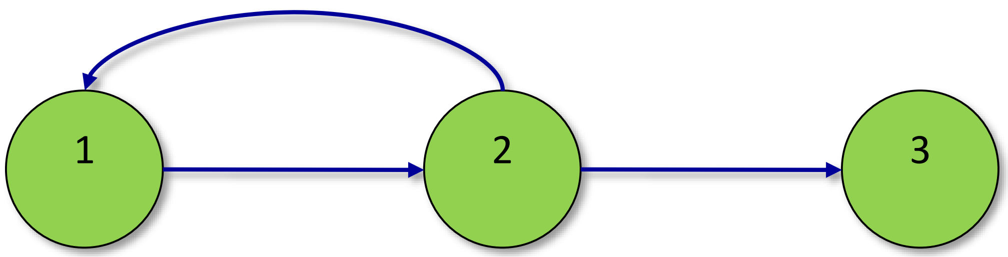
- $r_1(B) <_S w_2(B)$
- $w_2(A) <_S w_3(A)$
- $r_2(B) <_S w_1(B)$

➢Here, we have

$T_1 < T_2 < T_3$,

but we also have

$T_2 < T_1$



Not conflict-serializable

Schedule rejected

Enforcing serializability by locks

- To prevent non-serializable schedules to be submitted (which will result in **rejecting all transactions**) we can enforce serializability by **locks**
- Before reading or writing an element X, a transaction T_i requests a lock on X from the scheduler
- The scheduler can either grant the lock to T_i or make T_i wait for the lock
- If granted, T_i should eventually unlock (release) the lock on X.

- Notations:

  $L_i(X)$ = "transaction T_i requests a lock on X"

  $u_i(X)$ (or **$uL_i(X)$**)= "T_i unlocks/releases the lock on X"

T_1	T_2	A	B
		25	25
$L_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A)$;**$u_1(A)$**		125	
	$L_2(A)$;$r_2(A)$		
	A = A * 2		
	$w_2(A)$;**$u_2(A)$**	250	
	$L_2(B)$;$r_2(B)$		
	B = B * 2		
	$w_2(B)$;**$u_2(B)$**		50
$L_1(B)$;$r_1(B)$			
B = B + 100			
$w_1(B)$;**$u_1(B)$**			150

T1 unlocks A so T2 is free to lock it

Legal schedule with locks

- T1 adds 100 to both A and B
- T2 doubles both A and B

T_1	T_2	A	B
		25	25
$L_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A)$;$u_1(A)$		125	
	$L_2(A)$;$r_2(A)$		
	A = A * 2		
	$w_2(A)$;$u_2(A)$	250	
	$L_2(B)$;$r_2(B)$		
	B = B * 2		
	$w_2(B)$;$u_2(B)$		50
$L_1(B)$;$r_1(B)$			
B = B + 100			
$w_1(B)$;$u_1(B)$			150

Using locks does not necessarily make schedule serializable!

- T1 adds 100 to both A and B
- T2 doubles both A and B
- Expected result: A=B, and should be 250 for both by the end!

Two-Phase Locking

There is a simple condition, which guarantees conflict-serializability:

In every transaction, all lock requests (phase 1) precede all unlock requests (phase 2).

T_1	T_2	A	B
		25	25
$L_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A)$; **$L_1(B)$**; **$u_1(A)$**		125	
	$L_2(A)$;$r_2(A)$		
	A = A * 2		
	$w_2(A)$	250	
	$L_2(B)$ Denied		
$r_1(B)$			
B = B + 100			125
$w_1(B)$;**$u_1(B)$**			
	$L_2(B)$;**$u_2(A)$**;$r_2(B)$		
	B = B * 2		
	$w_2(B)$;**$u_2(B)$**		250

Simple locks are too restrictive

- While simple locks + 2PL guarantee conflict-serializability, **they do not allow two readers of DB element X at the same time.**
- **But having multiple readers is not a problem for conflict-serializability (since read actions are non-conflicting)!**

Solution: Two types of locks:

- **Shared lock $sL_i(X)$**
- **Exclusive lock $xL_i(X)$**

	S	X
S	yes	no
X	no	no

Deadlocks (on a single element)

Example:T1 and T2 each reads X and later writes X.

T1	T2
sL1(X)	
	sL2(X)
xL1(X) denied	
	xL2(X) denied

Problem: when we allow 2 types of locks, it is easy to get into a deadlock situation.

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Possible solution: Update Locks

- Only an **update lock** (**not shared lock**) can be upgraded to **exclusive lock** (if there are no shared locks anymore).
- A transaction that plans to write some element X, **asks initially for an update** lock on X, waits until all shared locks (if any) are released, and then asks for an exclusive lock on X.
- **No new locks** on X by other transactions are permitted while X is in an update lock mode

Notation: **Update lock $udL_i(X)$**

Schedule with update locks: example

T1	T2	T3
sL(A); r(A)		
	udL(A); r(A)	
		sL(A) **Denied**
	xL(A) **Denied**	
u(A)		
	xL(A); w(A)	
	u(A)	
		sL(A); r(A)
		u(A)

(No) Deadlock Example

T_1 and T_2 each read X and later write X.

T1	T2
sL1(X);	
	sL2(X);
xL1(X); denied	
	xL2(X); denied

Deadlock when using sL and xL locks only.

T1	T2
udl1(X); r(X);	
	udL2(X); denied
xL1(X); w(X); u(X);	
	udl2(X); r2(X); xl2(X); w2(X); u2(X)

Fine when using update locks.

Benefits of Update Locks

sl – shared lock
udl – update lock
xl – exclusive lock
u - unlock

T1	T2	T3	T4	T5	T6	T7	T8	T9
sl(A);r(A)								
	sl(A);r(A)							
		sl(A);r(A)						
			sl(A);r(A)					
				udl(A);r(A)				
					sl(A);denied			
						sl(A);denied		
							sl(A);denied	
								sl(A);denied
u(A)								
	u(A)							
		u(A)						
			u(A)					
				xl(A);w(A)				
				u(A)				
					s(A);r(A)			
						s(A);r(A)		
							s(A);r(A)	
								s(A);r(A)

Note how transactions T1-T4 were able to read A until T5 declared its intention to write with update lock and waited for them to release shared lock to get an exclusive lock on A