CSE 444: Database Internals

Section 5: Transactions

Today

- Serializability and Conflict Serializability
  - Precedence graph
- Two-Phase Locking
  - Strict two phase locking
- Lab 3 – Transactions

Problem 1: Serializability and Locking

- Is this schedule conflict seriali

What is
- Serializability
- Conflict Serializability?

T_0	T_1
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	C_1
$R_0(B)$	
$W_0(B)$	
C_0	

Review: (Conflict) Serializable Schedule

- A schedule is ***serializable*** if it is equivalent to a serial schedule
- A schedule is ***conflict serializable*** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

Review: (Conflict) Serializable Schedule

- A schedule is ***serializable*** if it is equivalent to a serial schedule

- A schedule is ***conflict serializable*** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

Example:

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

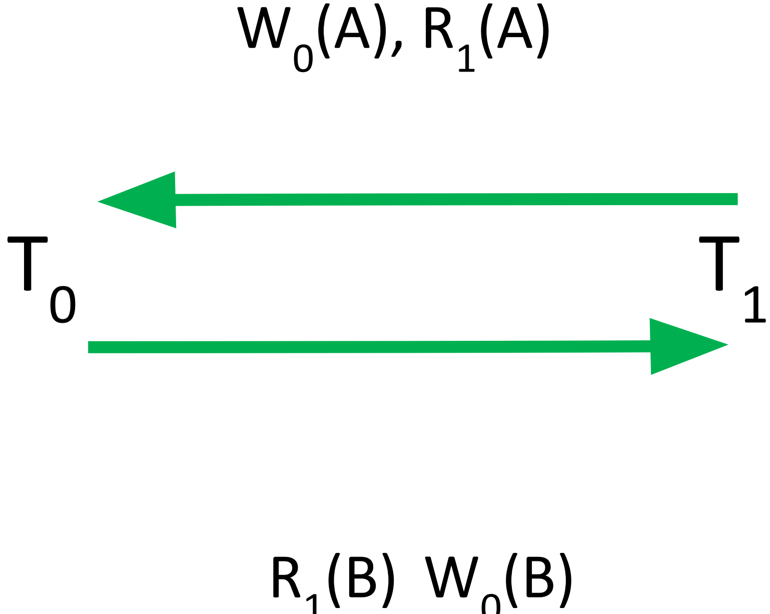
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Problem 1: Serializability and Locking

- Is this schedule conflict serializable?

T_0	T_1
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	C_1
$R_0(B)$	
$W_0(B)$	
C_0	

- No.
- The **precedence graph** contains a cycle



- So, use 2PL ...
  - Original schedule below

T_0	T_1
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	C_1
$R_0(B)$	
$W_0(B)$	
C_0	

- So, use 2PL …
  - Original schedule below

What is
- Two Phase Locking
- Strict Two Phase Locking?

T_0	T_1
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	C_1
$R_0(B)$	
$W_0(B)$	
C_0	

Review: (Strict) Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

Strict 2PL:

All locks held by a transaction are released when the transaction is completed

- Ensures that schedules are **recoverable**
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids **cascading rollbacks**

- How can 2PL can ensure a conflict-serializable schedule?
  - Original schedule below

T_0	T_1
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	C_1
$R_0(B)$	
$W_0(B)$	
C_0	

T_0	T_1
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	

T_0	T_1
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	$L_1(A)$: Block

T_0	T_1
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	$L_1(A)$: Block
$L_0(B)$	
$R_0(B)$	
$W_0(B)$	
$U_0(A)$	
$U_0(B)$	
C_0	

T_0	T_1
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	$L_1(A)$: Block
$L_0(B)$	
$R_0(B)$	
$W_0(B)$	
$U_0(A)$	
$U_0(B)$	
C_0	
	$L_1(A)$: Granted
	$R_1(A)$
	$L_1(B)$
	$R_1(B)$
	$U_1(A)$
	$U_1(B)$
	C_1

T_0	T_1
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	$L_1(A)$: Block
$L_0(B)$	
$R_0(B)$	
$W_0(B)$	
$U_0(A)$	
$U_0(B)$	
C_0	
	$L_1(A)$: Granted
	$R_1(A)$
	$L_1(B)$
	$R_1(B)$
	$U_1(A)$
	$U_1(B)$
	C_1

Is this strict 2PL?

No, release locks after commit

Lab 3 - Transactions

- NO STEAL / FORCE buffer management policy
  - you shouldn't evict dirty(updated) pages from the buffer pool if they are locked by an uncommitted transaction. (this is NO STEAL)
  - on transaction commit, you should force dirty pages to disk (e.g., write the pages out) (this is FORCE)
- Recommend - locking at page level
  - you can acquire and release locks in BufferPool.getPage(), instead of adding calls to each of your operators
  - Might have to change previous implementations to access pages using BufferPool.getPage()

Lab 3 - Transactions (contd.)

- You need to implement **shared** and **exclusive** locks
  - Before read, it must have a shared lock
  - Before write, it must have an exclusive lock
  - Multiple transactions can have a shared lock
  - Only one transaction may have an exclusive lock on an object
  - If transaction *t* is the only transaction holding a shared lock on an object *o*, *t* may upgrade its lock on *o* to an exclusive lock
- You need to implement **strict two-phase locking**
  - transactions should acquire the appropriate type of lock on any object before accessing that object
  - transaction shouldn't release any locks until after the transaction commits.

Lab 3 - Transactions (contd.)

- You will need to implement a LockManager class that will hold data structures to keep track of which locks each transaction holds and that check to see if a lock should be granted to a transaction when it is requested.
- Read about Synchronization in Java, and use the synchronized keyword in appropriate places in LockManager
- You will have to also throw appropriate exceptions like TransactionAbortedException

Lab 3 - Transactions (contd.)

- Handling deadlocks
  - implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time
  - implement a cycle-detection in a dependency graph data structure, if cycle exists when granting a new lock abort something.
- Design Choices:
  - Locking Granularity: page-level vs tuple-level (our tests assume page-level)
  - Deadlock Detection: timeout vs dependency graphs
  - Deadlock Resolution: aborting yourself vs aborting others
- Read the spec carefully for more details about various methods and edge cases.