CS 133: Databases

Fall 2019
Lec 17 – 11/07
Transactions
Prof. Beth Trushkowsky

Warm-up Exercise

(See exercise sheet. You can start before class.)

S1 and S2 are *not* conflict-equivalent.

S1 is conflict equivalent to the serial schedule T1;T2 and is thus conflict serializable.

Goals for Today

- Discuss how to achieve conflict serializable schedules using *locks*
- Understand how to manage locks and *deadlock* when implementing 2PL or Strict 2PL
- ~~Reason about issues that can arise when data is inserted or deleted~~

Locks

- We use **locks** to control access to objects
- Shared (S) locks – multiple transactions can hold these on a particular object at the same time.
- Exclusive (X) locks – only one of these and no other locks, can be held on a particular object at a time.

Lock Compatibility Matrix

	S	**X**
S	√	–
X	–	–

Basic Locking: Attempt

A= 1000, B=2000, Output from T_2's print =?

T_1	T_2
Lock_X(A) <granted>	Lock_S(A)
Read(A)	
A = A-50	
Write(A)	
Unlock(A)	<granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
	Read(B)
<granted>	Unlock(B)
	PRINT(A+B)
Read(B)	
B = B +50	
Write(B)	
Unlock(B)	

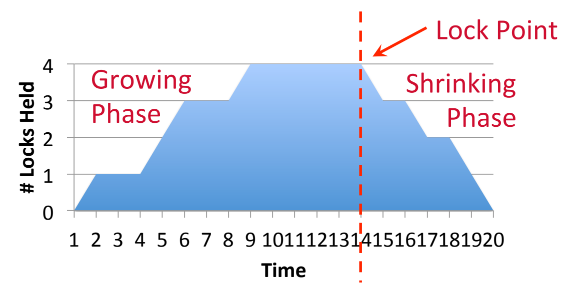
Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - a S (*shared*) or an X (*exclusive*) lock on object before reading
   - an X (*exclusive*) lock on object before writing

Can *upgrade* a Shared lock to an eXclusive lock! (*when okay*?)

2) **All lock requests must precede all unlock requests!**
   → *a xact cannot request additional locks once it releases any*

Each transaction has a "growing phase" followed by a "shrinking phase"



Basic Locking: Take 2

A= 1000, B=2000, Output =?

T_1	T_2
Lock_X(A) <granted>	Lock_S(A)
Read(A)	
A = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B = B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Basic Locking: Take 2 (with abort)

A= 1000, B=2000, Output =?

T_1	T_2
Lock_X(A) <granted>	Lock_S(A)
Read(A)	
A = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B = B +50	
Write(B)	
Unlock(B)	<granted>
ABORT!!	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

T_2 has read uncommitted changes! *It must also abort.*

Avoiding Cascading Aborts: Strict 2PL

- Problem with 2PL: cascading aborts
- Another example:
  rollback of T1 requires **rollback** of T2

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A)	

- Solution: **Strict Two-phase Locking** (Strict 2PL):
  - Same as 2PL, except for when locks can be released:
  - *All locks held by a transaction are released **only** when the transaction completes*

Consequence: a writer will block all other readers until the writer commits or aborts

Exercise 2

a) Yes 2PL, No Strict 2PL

b) Neither (schedule not conflict-serializable)

View Serializability

Checking for this is NP-complete!

- Schedules S1 and S2 are view equivalent if:
  - If T1 **reads initial value of A** in S1, then T1 also reads initial value of A in S2
  - If T1 **reads value of A written** by T2 in S1, then T1 also reads value of A written by T2 in S2
  - If T1 **writes final value of A** in S1, then **T1** also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

→

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Lock Management

- Lock/unlock requests are handled by the Lock Manager
  - Have table with entry for each **currently held lock**
- What object is being locked?
  - Possibilities: table(s), row(s), page(s)…
  - Too coarse-grained limits concurrency!
- **Lock table entry**
  - Object id of object being locked (e.g., table, row, page)
  - (Pointer to) list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - (Pointer to) **queue of lock requests**

Lock Management (cntd)

- When a lock request arrives
  - Check if any xact currently holds a conflicting lock on the object
  - If not, create an entry and grant the lock
  - Else, put the requesting xact on the wait queue

Locking and unlocking have to be atomic operations!

Try Exercise 3

ObjectID	LockType	Xacts	XactsWaiting
A	S	T1	
D	S	T1, T3	
B	X	T2	T1, T4
C	S	T3	T2

Basic Locking: Example (Take 3)

Lock_X(A) <granted>	
	Lock_S(B) <granted>
	Read(B)
	Lock_S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	

Deadlocks

- ***Deadlock***: Cycle of transactions waiting for locks to be released by each other.
- Can see cycle in a **waits-for graph**:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Two main ways of dealing with deadlocks in DBMS:
  - Deadlock **prevention**
  - Deadlock **detection**

Deadlock Prevention

- Assign priorities based on *timestamps*
- Suppose Ti wants a lock that Tj holds
  Two possible policies:
  - **Wait-Die**: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti is older, Tj aborts (gets “wounded”); otherwise Ti waits

In both, the **older** xact never aborts

- If a transaction re-starts, make sure it gets its original timestamp

Why?

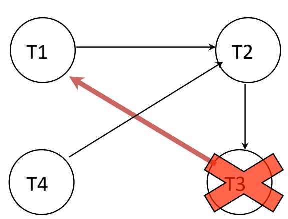
Deadlock Detection

- Alternative is to allow deadlocks to happen but to check for them and fix them if found.
- Periodically **check for cycles** in the waits-for graph
- If cycle detected – find a transaction whose removal will break the cycle and kill it

Deadlock Detection (Cntd)

Example:

T1: S(**A**), S(**D**), S(**B**)
T2: X(**B**) X(**C**)
T3: S(**D**), S(**C**), X(**A**)
T4: X(**B**)



Deadlock Exercise: 4

Start with sequence 1

Sequence 1:
T2 blocks on T1 on object A
T1 blocks on T3 on object B
When T3 finishes, T1 resumes and gets B
When T1 finishes, T2 resumes and gets A (and then B)

Sequence 2:
T2 blocks on T1 on object A
T3 blocks on T2 on object B
T1 blocks on T2 on object B
DEADLOCK! Waits-for-graph has cycle between T1 and T2

Lab 4: Lock-based Concurrency Control

- Goal of Lab 4: add page-level locking to SimpleDB
  - Strict 2PL
  - Shared and Exclusive locks
  - Dealing with deadlock
  - Dealing with BufferPool eviction (more in Recovery lecture)

Permissions.READ_ONLY vs READ_WRITE

Concurrency: How does it Happen?

- *Process*: executing instance of an program
- *Thread*: a path of execution ("control flow") within a process
  - Can be many threads within a process!
  - Threads have **shared access to data structures** within the process

Such as, say, a data structure managing Lock requests

Java: Thread Synchronization

- Thread synchronization in Java
  - Uses keyword *synchronized*
  - Synchronize specific block of code:
    synchronized(this) { // some code }
  - Synchronize entire method:

```
private synchronized void flushPage(PageId pid) {
      // some code
}
```

Skeleton code for Lock Manager and Buffer Pool already has these in place

Lab 4: Skeleton Code

- In `BufferPool.java`
  - Can create instance of *Lock Manager* class
  - Your choice to use skeleton `LockManager.java`
  - Example: `BufferPool.getPage()` will require that the transaction acquires a lock first!
- Lock table data structure(s), should be able to:
  - Given transactionId, which pages does it have locked?
  - Given a page Id, which xacts hold a lock on the page?
  - Given a page, which Permissions is it locked with?