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.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>√</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Locking: Attempt
A= 1000, B=2000, Output from T₂’s print =?

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)
<granted>
Read(B)
Unlock(B)
PRINT(A+B)

Basic Locking: Take 2
A= 1000, B=2000, Output =?

Lock_X(A) <granted>   Lock_S(A)
Read(A)
A = A-50
Write(A)
Lock_X(B) <granted>
Unlock(A) <granted>
Read(A)
Unlock(S)
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 =?

Lock_X(A) <granted>   Lock_S(A)
Read(A)
A = A-50
Write(A)
Lock_X(B) <granted>
Unlock(A) <granted>
Read(A)
Unlock(S)
Read(B)
B = B +50
Write(B)
Unlock(B) <granted>
Unlock(A)
Read(B)
Unlock(B)
PRINT(A+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:
A=	1000,	B=2000,	Output	from	T₂’s	print	=?

Is	it	a	2PL	schedule?
Yes:	so	it	is	serializable.

Basic Locking:
A=	1000,	B=2000,	Output	=?

Basic Locking:
A=	1000,	B=2000,	Output	=?

T₂ 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

  \[
  \begin{align*}
  T1: & \quad R(A), W(A), R(B), W(B), \text{Abort} \\
  T2: & \quad R(A), W(A)
  \end{align*}
  \]

- 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

View Serializability

- 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

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

Exercise 2

a) Yes 2PL, No Strict 2PL

b) Neither (schedule not conflict-serializable)
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!*

<table>
<thead>
<tr>
<th>ObjectID</th>
<th>LockType</th>
<th>Xacts</th>
<th>XactsWaiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>S</td>
<td>T1, T3</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>T2</td>
<td>T1, T4</td>
</tr>
<tr>
<td>C</td>
<td>S</td>
<td>T3</td>
<td>T2</td>
</tr>
</tbody>
</table>

Try Exercise 3

Basic Locking: Example (Take 3)

```
<table>
<thead>
<tr>
<th>Lock_X(A)  &lt;granted&gt;</th>
<th>Lock_S(B)  &lt;granted&gt;</th>
<th>Read(B)</th>
<th>Lock_S(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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)

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

Java: Thread Synchronization

- Thread synchronization in Java
  - Uses keyword `synchronized`
  - Synchronize specific block of code:
    ```java
    synchronized(this) { // some code }
    ```
  - Synchronize entire method:
    ```java
    private synchronized void flushPage(PageId pid) {
        // some code
    }
    ```

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?

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