Transactions: Concurrency Control

Amol Deshpande
CMSC424
Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
- Week 5: Transactions 2 (Homework Due May 1)
  - Transactions: Serializability, Recoverability
  - Transactions: Concurrency 1
  - Transactions: Concurrency 2: Other Concurrency Schemes
  - Transactions: Recovery
- Week 6: Distributed Transactions; Miscellaneous Topics (Homework Due May 8)
Transactions: Concurrency 1

- Book Chapters
  - 15.1, 15.2, 15.3

- Key topics:
  - Using locking to guarantee concurrency
  - 2-Phase Locking (2PL)
  - How “deadlocks” can happen and how to avoid them or recover from them
  - Multi-granularity locking and its benefits
Approach, Assumptions etc..

- **Approach**
  - Guarantee conflict-serializability by allowing certain types of concurrency
    - Lock-based

- **Assumptions:**
  - Durability is not a problem
    - So no crashes
    - Though transactions may still abort

- **Goal:**
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)
Lock-based Protocols

- A transaction must get a lock before operating on the data

- Two types of locks:
  - Shared (S) locks (also called read locks)
    - Obtained if we want to only read an item – lock-S() instruction
  - Exclusive (X) locks (also called write locks)
    - Obtained for updating a data item – lock-X() instruction

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>display(A+B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A ← A + 50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>display(A+B)</td>
</tr>
<tr>
<td>A ← A + 50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
</tr>
</tbody>
</table>
Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  
  ★ It decides whether to *grant* a lock request

- T1 asks for a lock on data item A, and T2 currently has a lock on it?
  
  ★ Depends

<table>
<thead>
<tr>
<th>T2 lock type</th>
<th>T1 lock type</th>
<th>Should allow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Shared</td>
<td>YES</td>
</tr>
<tr>
<td>Shared</td>
<td>Exclusive</td>
<td>NO</td>
</tr>
<tr>
<td>Exclusive</td>
<td>-</td>
<td>NO</td>
</tr>
</tbody>
</table>

- If *compatible*, grant the lock, otherwise T1 waits in a *queue*. 
Lock-based Protocols

How do we actually use this to guarantee serializability/recoverability?

- Not enough just to take locks when you need to read/write something

```
T1
lock-X(B)  # Lock B
read(B)    # Read B
B ← B-50   # Update B
write(B)   # Write B
unlock(B)  # Unlock B

lock-X(A)  # Lock A
read(A)    # Read A
A ← A+50   # Update A
write(A)   # Write A
unlock(A)  # Unlock A

lock-X(A), lock-X(B)  # Lock both
TMP = (A + B) * 0.1  # Calculate TMP
A = A - TMP  # Update A
B = B + TMP  # Update B
unlock(A), unlock(B) # Unlock both
```

NOT SERIALIZABLE
2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them

- Phase 2: Shrinking phase
  - Transaction may only release locks

Can be shown that this achieves \textit{conflict-serializability}

- \textit{lock-point}: the time at which a transaction acquired last lock
- if \( \text{lock-point}(T_1) < \text{lock-point}(T_2) \), there can’t be an edge from \( T_2 \) to \( T_1 \) in the \textit{precedence graph}

Example:

\begin{align*}
T_1 & \\
lock-X(B) & \\
read(B) & \\
B & \leftarrow B-50 \\
write(B) & \\
unlock(B) & \\
lock-X(A) & \\
read(A) & \\
A & \leftarrow A + 50 \\
write(A) & \\
unlock(A) &
\end{align*}
Example: T1 in 2PL

Growing phase:
- lock-X(B)
- read(B)
- B ← B - 50
- write(B)
- lock-X(A)
- read(A)
- A ← A - 50
- write(A)

Shrinking phase:
- unlock(B)
- unlock(A)

Growing phase
2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td></td>
<td>unlock(A)</td>
</tr>
<tr>
<td>&lt;xction fails&gt;</td>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock-S(A)</td>
</tr>
</tbody>
</table>
2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

- Guaranteeing just recoverability:
  - ★ If T2 reads a dirty data of T1 (ie, T1 has not committed), then T2 can’t commit unless T1 either commits or aborts
  - ★ If T1 commits, T2 can proceed with committing
  - ★ If T1 aborts, T2 must abort
    - ➢ So cascades still happen
Strict 2PL

Release *exclusive* locks only at the very end, just before commit or abort.

Works. Guarantees cascade-less and recoverable schedules.
Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are not important

- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
  - The serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system

- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn’t change any of the other properties of the protocol
Implementation of Locking

- A separate process, or a separate module

- Uses a *lock table* to keep track of currently assigned locks and the requests for locks
- Black rectangles indicate granted locks, white ones indicate waiting requests.
- Lock table also records the type of lock granted or requested.
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
- If transaction aborts, all waiting or granted requests of the transaction are deleted.

★ lock manager may keep a list of locks held by each transaction, to implement this efficiently.
Recap so far...

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc

- Lock-based protocols
  - Use locks to prevent multiple transactions accessing the same data items

- 2 Phase Locking
  - Locks acquired during growing phase, released during shrinking phase

- Strict 2PL, Rigorous 2PL
More Locking Issues: Deadlocks

- No xction proceeds:

  Deadlock
  - T1 waits for T2 to unlock A
  - T2 waits for T1 to unlock B

  Rollback transactions
  Can be costly...

- 2PL does not prevent deadlock
  - Strict doesn’t either

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ⊲ B-50</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>lock-X(A)</td>
<td></td>
</tr>
</tbody>
</table>
Preventing deadlocks

- **Solution 1:** A transaction must acquire all locks before it begins
  - Not acceptable in most cases

- **Solution 2:** A transaction must acquire locks in a particular order over the data items
  - Also called *graph-based protocols*

- **Solution 3:** Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.

- **Solution 4:** Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn’t get it by then
Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen.

- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from $T_i$ to $T_j$
  - $T_i$ waiting for $T_j$

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S(V)$</td>
<td>$X(V)$</td>
<td>$X(Z)$</td>
<td>$X(W)$</td>
</tr>
<tr>
<td></td>
<td>$S(W)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$S(V)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose $T_4$ requests lock-$S(Z)$....
Dealing with Deadlocks

- Deadlock detected, now what?
  - Will need to abort some transaction
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing
What are we taking locks on? Tables, tuples, attributes?

Coarse granularity
- e.g. take locks on tables
- less overhead (the number of tables is not that high)
- very low concurrency

Fine granularity
- e.g. take locks on tuples
- much higher overhead
- much higher concurrency
- What if I want to lock 90% of the tuples of a table?
  - Prefer to lock the whole table in that case
The highest level in the example hierarchy is the entire database. The levels below are of type *area*, *file or relation* and *record* in that order.

Can lock at any level in the hierarchy
Granularity Hierarchy

- New lock mode, called *intentional* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - SIX: *shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below

- Protocol:
  - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
  - So you always start at the top *root* node
Granularity Hierarchy

(1) Want to lock $F_a$ in shared mode, $DB$ and $A1$ must be locked in at least IS mode (but IX, SIX, S, X are okay too)

(2) Want to lock $rc1$ in exclusive mode, $DB$, $A2,Fc$ must be locked in at least IX mode (SIX, X are okay too)
## Granularity Hierarchy

<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IS, S</td>
</tr>
<tr>
<td>IX</td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td>S</td>
<td>[S, IS] not necessary</td>
</tr>
<tr>
<td>SIX</td>
<td>X, IX, [SIX]</td>
</tr>
<tr>
<td>X</td>
<td>none</td>
</tr>
</tbody>
</table>
The compatibility matrix (which locks can be present simultaneously on the same data item) for all lock modes is:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>S IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S IX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Example

R1

- $t_1$
- $t_2$
- $t_3$
- $t_4$

$T_1(IS), T_2(IX)$

$T_1(S)$

$T_2(X)$
Can T2 access object f2.2 in X mode?
What locks will T2 get?
Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IS</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>
Recap, Next…

- **Deadlocks**
  - Detection, prevention, recovery

- **Locking granularity**
  - Arranged in a hierarchy
  - Intentional locks

- **Next video…**
  - Brief discussion of some other concurrency schemes