Admin

• Signup sheet for project presentations
Recap...1

- ACID properties:
  - Atomicity (recovery)
  - Consistency (transaction design, concurrency control, recovery)
  - Isolation (concurrency control)
  - Durability (recovery)
Recap

• Concurrency Control Scheme
  – A way to guarantee serializability, recoverability etc

• Lock-based protocols
  – Use \textit{locks} to prevent multiple transactions accessing the same data items

• 2 Phase Locking
  – Locks acquired during \textit{growing phase}, released during \textit{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

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

Rollback transactions
Can be costly...
2PL and Deadlocks

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

- > 2 transactions involved?
  - Rollbacks expensive

<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 Ti to Tj
    - Ti waiting for Tj

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

Suppose T4 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
Locking granularity

• Locking granularity
  – 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
Granularity Hierarchy

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>
Compatibility Matrix with Intention Lock Modes

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>holder</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>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.
Recap, Next....

• Deadlocks
  – Detection, prevention, recovery

• Locking granularity
  – Arranged in a hierarchy
  – Intentional locks

• Next...
  – Brief discussion of some other concurrency schemes
Other CC Schemes

• Time-stamp based
  – Transactions are issued time-stamps when they enter the system
  – The time-stamps determine the serializability order
  – So if T1 entered before T2, then T1 should be before T2 in the serializability order
  – Say \( \text{timestamp}(T1) < \text{timestamp}(T2) \)
  – If T1 wants to read data item A
    • If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is aborted
  – If T1 wants to write data item A
    • If a transaction with larger time-stamp already read that data item or written it, then the write is rejected and T1 is aborted
  – Aborted transaction are restarted with a new timestamp
    • Possibility of starvation
Other CC Schemes

• Time-stamp based
  – As discussed here, has too many problems
    • Starvation
    • Non-recoverable
    • Cascading rollbacks required

– Most can be solved fairly easily
  • Read up

– Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
  • The goal is to find the minimal set of restrictions to as to not hinder concurrency
Other CC Schemes

• Optimistic concurrency control
  – Also called validation-based

  – Intuition
    • Let the transactions execute as they wish
    • At the very end when they are about to commit, check if there might be any problems/conflicts etc
      – If no, let it commit
      – If yes, abort and restart

  – Optimistic: The hope is that there won’t be too many problems/aborts

• Rarely used any more
The “Phantom” problem

• An interesting problem that comes up for dynamic databases

• Schema: \texttt{accounts}(branchname, acct\_no, balance, \ldots)

• Transaction 1: Find the maximum balance in each branch

• Transaction 2: Insert \texttt{<"branch1", acctX, $10000000>, \text{and \ delete \texttt{<"branch2", acctY, $100000000>.}}}
  – Both maximum entries in the corresponding branches

• Execution sequence:
  – T1 locks all tuples corresponding to “branch1”, finds the maximum balance and releases the locks
  – T2 does its two insert/deletes
  – T1 locks all tuples corresponding to “branch2”, finds the maximum balance and releases the locks

• Not serializable