Concurrency Control
Concurrency Via Locks

- Idea: Data items modified by one transaction at a time

- Locks
  - Control access to a resource
  - Can block a transaction until lock granted
  - Two modes: Shared (read only) ▪ eXclusive (read & write)

- Requesting locks: Must request before accessing a data item
Example of Locks

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

- No transaction proceeds:
  - T1 waits for T2 to unlock A
  - T2 waits for T1 to unlock B

Rollback transactions can be costly...
The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.

- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks

- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
The Two-Phase Locking Protocol

2PL does not prevent deadlock

> 2 transactions involved: Rollbacks expensive
Strict two phase locking

- Exclusive locks must be held until transaction commits
- Ensures data written by transaction can’t be read by others
- Prevents cascading rollbacks

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-X(A)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(A)</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)</td>
<td>unlock(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;xaction fails&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- Deadlock prevention protocols ensure that the system will never enter into a deadlock state.
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
Deadlock Handling

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- wait-die scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.  
  - a transaction may die several times before acquiring needed data item

- wound-wait scheme — preemptive
  - older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
Deadlock Handling
Timestamp-Based Protocols

**Idea:** If action $p_i$ of Xact $T_i$ conflicts with action $q_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $p_i$ must occur before $q_j$. Otherwise, restart violating Xact.
Timestamp-Based Protocols

When Xact T wants to read Object O

- If $TS(T) < W-TS(O)$, this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)

- If $TS(T) > W-TS(O)$:
  - Allow T to read O.
  - Reset R-TS(O) to $\max(R-TS(O), TS(T))$
Timestamp-Based Protocols

When Xact T wants to Write Object O

- If $TS(T) < R-TS(O)$, then the value of $O$ that $T$ is producing was needed previously, and the system assumed that that value would never be produced. **write rejected, $T$ is rolled back.**

- If $TS(T) < W-TS(O)$, then $T$ is attempting to write an obsolete value of $O$. Hence, this **write operation is rejected, and $T$ is rolled back.**

- Otherwise, the **write operation is executed**, and $W-TS(O)$ is set to $TS(T)$. 
# Log-Based Recovery

**Deferred Database Modification**

**Immediate Database Modification**

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>&lt;T1, start&gt;</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>&lt;T1, A, 1000, 950&gt;</td>
</tr>
<tr>
<td>Write(A)</td>
<td>&lt;T1, B, 2000, 2050&gt;</td>
</tr>
<tr>
<td>Read(B)</td>
<td>&lt;T1, commit&gt;</td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>
Deferred Database Modification

DB updates by reading and executing the log:

<Ti start> …… < Ti commit >

Redo: if both <Ti start > and < Ti commit> are there in the log.
Immediate Database Modification

Database updates of an uncommitted transaction are allowed

• Undo: < Ti, start > is in the log but <Ti, commit> is not.

➢ restore the value of all data items updated by Ti to their old values, going backwards from the last log record for Ti

• Redo: <Ti, start > and <Ti, commit> are both in the log.

➢ sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti
Checkpoint

Objective: avoid redundant redoes

• Put marks in the log indicating that at that point DB and log are consistent.

T1 can be ignored (updates already output to disk due to checkpoint)
T2 and T3 redone.
T4 undone.
Recovery With Concurrent Transactions

Checkpoints for concurrent transactions:

<checkpoint L>

L: the list of transactions active at the time of the checkpoint

We assume no updates are in progress while the checkpoint is carried out

Recovery for concurrent transactions, 3 phases:

Initialize undo-list and redo-list to empty

Scan the log backwards from the end, stopping when the first <checkpoint L> record is found.

For each record found during the backward scan:

  if the record is <Ti commit>, add Ti to redo-list

  if the record is <Ti start>, then if Ti is not in redo-list, add Ti to undo-list

For every Ti in L, if Ti is not in redo-list, add Ti to undo-list

ANALYSIS
Recovery With Concurrent Transactions

Scan log backwards
Perform undo(T) for every transaction in undo-list
Stop when you have seen \(<T, \text{start}>\) for every T in undo-list.

Locate the most recent \(<\text{checkpoint } L>\) record.
Scan log forwards from the \(<\text{checkpoint } L>\) record till the end of the log.
perform redo for each log record that belongs to a transaction on redo-list

UNDO

REDO