Concurrency Control

continued...

Strict 2PL
- 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
  - Read up in the book

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 transaction 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>lock-S(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>lock-S(B)</td>
</tr>
</tbody>
</table>
```

Rolling back transactions can be costly...

---

**Deadlocks**

- 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></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B ← B-50</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>lock-S(B)</td>
</tr>
</tbody>
</table>
```

Rolling back transactions can be costly...
Preventing deadlocks

- **Graph-based protocols**
  - Acquire locks only in a well-known order

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

  - Might not know locks in advance

Detecting existing deadlocks

- **Timeouts (local information)**
- **cycles in *waits-for graph* (global information):**
  - edge $T_i \rightarrow T_j$ when $T_i$ waiting for $T_j$

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

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

- Deadlock detected, now what?
  - Will need to abort some transaction

- Victim selection
  - 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.

- Issues
  - Prefer to prefer transactions with the most work done
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing

Locking granularity
Locking granularity (not always done)

- 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)
Compatibility Matrix with Intention Lock Modes

- Locks from different transactions:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</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>SIX</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

T₁(IS), T₂(IX)

T₁(S)

T₂(X)
Examples

Can T2 access object f2.2 in X mode? What locks will T2 get?

Other CC Schemes

- **Time-stamp based**
  - Transactions are issued time-stamps when they start
  - Time-stamps determine the *serializability order*
  - If T1 enters before T2, then T1 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*
Time-stamp based CC

- Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($X$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($X$) abort</td>
<td>read($X$)</td>
<td>write($Y$)</td>
<td>write($Z$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td>abort</td>
<td>write($Z$) abort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$TS(T_1) < TS(T_2) < TS(T_3) < TS(T_4) < TS(T_5)$

Time-stamp based CC

- The following set of instructions is not conflict-serializable:

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Q$)</td>
<td>write($Q$)</td>
</tr>
<tr>
<td></td>
<td>write($Q$)</td>
<td></td>
</tr>
</tbody>
</table>

- As discussed before, not even view-serializable:
  - if $T_i$ reads initial value of $Q$ in $S$, must also in $S'$
  - if $T_i$ reads value written from $T_j$ in $S$, must also in $S'$
  - if $T_i$ performs final write to $Q$ in $S$, must also in $S'$
Time-stamp based CC

- Thomas’ Write Rule
  - Ignore obsolete writes

- 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 or written that data item, then the write is rejected and T1 is aborted
    - If a transaction with larger time-stamp already written that data item, then the write is ignored

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td></td>
<td>write(Q)</td>
</tr>
<tr>
<td>write(Q)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other CC Schemes

- Time-stamp based
  - As discussed here, has 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

Isolation Levels: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol

- A type of “optimistic concurrency control”

- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated
Other CC Schemes: Snapshot Isolation

- **Read queries:**
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of committed database

- **Update queries (transactions):**
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction committed the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)
    - *first committer vs first updater*

Other CC Schemes: Snapshot Isolation

- **Advantages:**
  - Read queries do not block, and run very fast
  - As long as conflicts are rare, update transactions don’t abort
  - Overall better performance than locking-based protocols

- **Major disadvantage:**
  - Not serializable!

\[ x = y = 0 \]

\[
\begin{array}{c|c|c}
T_1 & T_2 \\
\hline
w(x)1 & w(y)1 \\
r(y)0 & r(x)0 \\
\end{array}
\]