Guarantees of Atomicity and Isolation

- Serializability
  - conflict, view
- Concurrency Control
  - locking
    - 2-phase locking, strict, rigorous
    - granularity, intention locks
- Other approaches to concurrency control
  - timestamp-based
  - optimistic
  - snapshot isolation
- Quiz 8
- Weak levels of concurrency
- Recovery

- Extensions:
  - Assign 7 by two days
  - Assign 8 by a week

---

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

(not always done)
Granularity Hierarchy

The highest level in the example hierarchy is the entire database. The levels below are of relation and tuple in that order. Can lock at any level in the hierarchy.

Intention Locks

- **New lock mode, called intention 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 might also want exclusive locks on some nodes below

- **Protocol:**
  - Before acquiring a lock on a data item, all the ancestors must be locked as well, at least in the intention mode
  - Lock acquisition order is from the root down to the desired node.
Intention Locks

(1) Want to lock $t_1$ in shared mode, $DB$ and then $R_1$ must be locked in at least IS mode (but IX, SIX, S, X are okay too), then $t_1$ in S mode.

(2) Want to lock $t_4$ in exclusive mode, $DB$ and then $R_2$ must be locked in at least IX mode (SIX, X are okay too), then $t_4$ must be locked in X mode.

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>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>x</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

● holder requestor
Example

- Assume:
  - \( T_1 \) wants *shared* lock on \( t_2 \)
  - \( T_2 \) wants *exclusive* lock on \( t_4 \)

\[ \text{R1} \]

- \( T_1 \) (IS) \( T_1 \) (S) \[ \text{R1} \]
- \( T_1 \) (X) \[ \text{R1} \]

\( t_2 \)

- \( T_2 \) (IX) \[ \text{R1} \]
- \( T_2 \) (X) \[ \text{R1} \]

\( t_4 \)

\( t_1 \)

\( t_3 \)

\( t_2.1 \)

\( t_2.2 \)

\( t_4.2 \)

T2 Needs Locks...But T1 already there...

Can \( T_2 \) access object \( t_{2.2} \) in \( X \) mode?

What locks will \( T_2 \) get?
other
Concurrency Control Schemes

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
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 the database
  - Never aborted

• **Update queries (transactions):**
  - Reads processed as above on a snapshot
  - Writes are done in private storage. However, *the writes are visible to the transaction that made them.*
  - At commit time, for each object that was written, check if some other transaction updated 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)

---

**Snapshot Isolation**

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits

**First-committer-wins rule:**
  - Commits only if no other concurrent transaction has already written data that T1 intends to write.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(Y := 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(X) → 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(Y) → 1</td>
<td></td>
</tr>
</tbody>
</table>

W(X := 2)
<table>
<thead>
<tr>
<th>Commit</th>
</tr>
</thead>
</table>

W(Z := 3)

R(Z) → 0
|          |
| R(Y) → 1 |          |

W(X := 3)
| Commit-Req |
| Abort      |

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back
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

But:

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

\[x = y = 0\]

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

- Also called *write skew*

----

Snapshot Isolation: Multi-Version Implementation

- each write to Q creates a new version of Q
- reads are parameterized by transaction’s *timestamp*
  - satisfied by last write before that timestamp
- Implementing Snapshot Isolation w/ MVCC
  - transaction gets \(StartTS(T_i), CommitTS(T_i)\)
  - write by \(T_i\) saved with \(CommitTS(T_i)\)
  - read by \(T_i\) satisfied by last version w/ time \(< StartTS(T_i)\)
  - as a result:
    - transaction only see writes committed prior to start
    - i.e. a *snapshot*
Snapshot Isolation: Multi-Version Implementation

Two approaches: *first-commiter-wins*, and *first-updater-wins*.

$T_j$ is said to be concurrent with a transaction $T_i$ if either:

\[ \text{StartTS}(T_j) \leq \text{StartTS}(T_i) \leq \text{CommitTS}(T_j), \text{ or } \]

\[ \text{StartTS}(T_i) \leq \text{StartTS}(T_j) \leq \text{CommitTS}(T_i) \]

Under *first-commiter-wins*, $T_i$ checks at commit time to see if any concurrent transaction has written an object that it is trying to write. If so, $T_i$ aborts.

Under *first-updater-wins*, $T_i$ checks at each write. Before writing $Q$, $T_i$:

- Attempts to acquire a write lock on $Q$. If the lock is acquired, $T_i$ aborts if a concurrent transaction $T_j$ has already written $Q$, or commits otherwise.
- If the lock was not successful, $T_i$ waits to see if $T_j$ commits or aborts. If $T_j$ commits, $T_i$ aborts. If $T_j$ aborts:
  - $T_i$ repeats the check for a concurrent writer having updated $Q$. If found,
    - $T_i$ aborts.
  - else
    - $T_i$ commits

Snapshot Isolation

- **Advantages:**
  - Read query don’t block at all, and run very fast
  - If conflicts are rare, update transactions don’t abort either
  - Overall better performance than locking-based protocols

- **Major disadvantage:**
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
Guarantees of Atomicity and Isolation

- Serializability
  - conflict, view
- Concurrency Control
  - locking
    - 2-phase locking, strict, rigorous
    - granularity, intention locks
- Other approaches to concurrency control
  - timestamp-based
  - optimistic
  - snapshot isolation
- Recovery

Time-stamp Based Concurrency Control

- No locks
- Transactions issued time-stamps when started
- Time-stamps determine the serializability order
- If T1 enters before T2, then T1 < T2 in 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, or wrote, that data item, then the write is rejected and T1 is aborted
  - Aborted transactions are restarted with a new timestamp
    - Possibility of starvation
    - Optimistic
## Timestamp-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>write($X$)</td>
</tr>
<tr>
<td></td>
<td>read($X$)</td>
<td>read($X$)</td>
<td>write($Y$)</td>
<td>read($Z$)</td>
<td>write($Y$)</td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td>abort</td>
<td>write($Z$)</td>
<td>abort</td>
<td>write($Z$)</td>
</tr>
</tbody>
</table>

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

## Timestamp-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 $S'$
**Timestamp-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 wrote, 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:**

<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>$T_3$ &lt; $T_4$</td>
<td></td>
</tr>
</tbody>
</table>

**Ignore if $T_3 < T_4$**

---

**Timestamp-based CC**

- **As discussed here, has a few issues**
  - Starvation
  - Non-recoverable
  - Cascading rollbacks required

- **Most can be solved fairly easily**
  - Read up

- **We can always add more restrictions to ensure these things**
  - The goal is to find the minimal set of restrictions to as to not hinder concurrency