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)
The highest level in the example hierarchy is the entire database. The levels below are of type and tuple 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 \( t1 \) in shared mode, \( DB \) and then \( R1 \) must be locked in at least IS mode (but IX, SIX, S, X are okay too), then \( t1 \) in S mode.

(2) Want to lock \( t4 \) in exclusive mode, \( DB \) and then \( R2 \) must be locked in at least IX mode (SIX, X are okay too), then \( t4 \) 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>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Example

Can T2 access object t2.2 in X mode?
What locks will T2 get?

Examples
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, 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

---

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>read(Y)</td>
<td>read(Y)</td>
<td>write(Y)</td>
<td>write(Z)</td>
<td>read(Z)</td>
<td></td>
</tr>
<tr>
<td>read(X)</td>
<td>read(X)</td>
<td>write(Z)</td>
<td>write(Y)</td>
<td>write(Z)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>abort</td>
<td>abort</td>
<td>abort</td>
<td>abort</td>
<td></td>
</tr>
</tbody>
</table>
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>read(Q)</td>
<td></td>
<td>write(Q)</td>
</tr>
<tr>
<td>write(Q)</td>
<td></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'

---

### 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
**Time-stamp based CC**

- **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”
  - Execution:
    - read phase: at time $\text{Start}(T_i)$ reads any needed values into local variables
    - validation phase: at time $\text{Validation}(T_i)$, for all transactions $T_k$ s.t. $\text{TS}(T_k) < \text{TS}(T_i)$, either:
      - $\text{Finish}(T_k) < \text{Start}(T_i)$, or
      - data written by $T_k$ does not intersect w/ those read by $T_i$, and $\text{Finish}(T_k) < \text{Validation}(T_i)$
    - write phase: at $\text{Finish}(T_i)$ all writes complete
Other CC Schemes

- Optimistic concurrency control
  - Also called “validation-based”
  - Execution:
    - read phase: at time \( \text{Start}(T_i) \) reads any needed values into local variables
    - validation phase: at time \( \text{Validation}(T_i) \), for all transactions \( T_k \) s.t. \( TS(T_k) < TS(T_i) \), either:
      - \( \text{Finish}(T_k) < \text{Start}(T_i) \), or
      - data written by \( T_k \) does not intersect w/ those read by \( T_i \), and \( \text{Finish}(T_k) < \text{Validation}(T_i) \)
    - write phase: at \( \text{Finish}(T_i) \) all writes complete

- 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: *Hope not too many problems/aborts*

Other CC Schemes: Optimistic CC

- serialization order:
  - \( TS(T_2) < TS(T_1) < TS(T_3) < TS(T_4) \)
- good validations:
  - \( T_2 \)
  - \( \text{not } T_1 \)
  - \( T_3 \)
  - \( T_4 \)
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

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(x)₁</td>
<td>w(y)₁</td>
</tr>
<tr>
<td>r(y)₀</td>
<td>r(x)₀</td>
</tr>
<tr>
<td>commit?</td>
<td>commit?</td>
</tr>
</tbody>
</table>

- Major disadvantage:
  - Not serializable!