Concurrency Control
Part 2

From Chapters 16, 17
Outline

- Deadlock prevention and detection
- Advanced locking techniques
- Lower degrees of isolation
- Concurrency control for index structures
Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds.

- Wait-Die:

- Wound-wait:

- If a transaction re-starts, make sure it has ___________ timestamp
Deadlock Detection

- Create a waits-for graph:
  - Nodes:
  - Edges:

- Periodically check for cycles in the waits-for graph
Deadlock Detection

Example:

T1:  S(A), R(A), S(B)
T2:  X(B), W(B) X(C)
T3:  S(C), R(C) X(A)
T4:  X(B)
Outline

- Deadlock prevention and detection
- Advanced locking techniques
- Lower degrees of isolation
- Concurrency control for index structures
Multiple-Granularity Locks

- Hard to decide what granularity to lock
- Shouldn’t have to decide!
- Data “containers” are nested:
  - Database
    - Tables
      - Pages
        - Tuples
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new “intention” locks:

  - Before locking an item:

  - Unlock:

  - SIX mode:

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</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>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples

- T1 scans R, and updates a few tuples:

- T2 uses an index to read only part of R:

- T3 reads all of R:

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</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>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with \( \text{rating} = 1 \), and finds \textit{oldest} sailor
  - Next, T2 inserts a new sailor; \( \text{rating} = 1, \text{age} = 96 \).
  - T2 also deletes oldest sailor with \( \text{rating} = 2 \) (\texttt{age} = 80), and commits.
  - T1 now locks all pages containing sailor records with \( \text{rating} = 2 \), and finds \textit{oldest}

- No consistent DB state where T1 is “correct”!
The Problem - Phantom

- T1 implicitly assumes that it has locked the set of all sailor records with \textit{rating} = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
Index Locking

- If there is a dense index on the *rating* field using Alternative (2), T1 should lock the index page containing the data entries with *rating* = 1.
  - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with *rating* = 1 are added.
Outline

- Deadlock prevention and detection
- Advanced locking techniques
- Lower degrees of isolation
- Concurrency control for index structures
Transaction Support in SQL-92

Each transaction has an access mode, a diagnostics size, and an isolation level.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Database Management Systems, R. Ramakrishnan and Johannes Gehrke
Outline

- Deadlock prevention and detection
- Advanced locking techniques
- Lower degrees of isolation
- Concurrency control for index structures
Locking in B+ Trees

- How can we enable “safe” concurrent access to index structures?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- Problem?
Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be X-locked, only if __________________________________________
- (Similar point holds w.r.t. deletes.)
A Simple Tree Locking Algorithm

- **Search**: Start at _____ and go _____; repeatedly, ___ lock ___ then unlock ______

- **Insert/Delete**: Start at _____ and go _____, obtaining ___ locks as needed. Once child is locked, check if it is **safe**:
  - If child is safe,______________________________.

- **Safe node**:
  - Inserts:
  - Deletes:
Example

Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*

Database Management Systems, R. Ramakrishnan and Johannes Gehrke
A Better Tree Locking Algorithm

- **Search:** As before.

- **Insert/Delete:**
  - Set locks as if for search, get to leaf, and set X lock on leaf.
  - If leaf is not safe, release all locks, and restart Xact using previous Insert/Delete protocol.
Example

Do:
1) Delete 38*
2) Insert 25*
4) Insert 45*
5) Insert 45*, then 46*

Database Management Systems, R. Ramakrishnan and Johannes Gehrke