Multiple-Granularity Locks: Motivation

- What is appropriate object granularity to lock (records vs. pages vs. tables?)
- **Examples**
  - Xact wants to lock entire table... lock each record? ➔ a lot of locks to grab!
  - Xact wants to read only a few records... lock whole table? ➔ limits concurrency!

Multiple-Granularity Locks

Observation: hierarchy of data “containers”

- Lock a node *explicitly* (S or X) ➔ locks all descendants *implicitly*

Suppose T1 gets X-lock on Relation A.
Then T2 wants lock on Record 2... what happens?
Now T3 wants lock on the Database... how to tell that it shouldn’t get it?
Solution: New Lock Modes, Protocol

• Allow Xacts to lock at each level, but with a special protocol using new *intention* locks.

• Still need S and X locks, but before locking an item, xact must have *proper intention locks on all its ancestors* in the granularity hierarchy.

  • IS – Intent to get S lock(s) at finer granularity.
  • IX – Intent to get X lock(s) at finer granularity.
  • SIX mode: Have S & IX at the same time on a node.

Multiple Granularity Lock Protocol

• Each xact starts locking from the root of the hierarchy
  – To get S or IS lock on a node, must hold IS or IX on parent node
  – To get X or IX or SIX on a node, must hold IX or SIX on parent node

• Must release locks in bottom-up order

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exercise 2

<table>
<thead>
<tr>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Isolation: Protection from Concurrency

- **Isolation**: each xact executes *as if* it was running by itself

- Two basic categories:
  - **Pessimistic** — don’t let problems arise in the first place... *use locks*
  - **Optimistic** — assume conflicts are rare, deal with them *after* they happen.

Optimistic CC: Motivation

Locking is a conservative approach in which conflicts are prevented. Disadvantages:

- Lock management overhead
- Deadlock detection/resolution
- Lock contention for heavily used objects

- Locking is “pessimistic” because it assumes that conflicts will happen.

- *If conflicts are rare, we might get better performance by not locking*, and instead checking for conflicts at commit

OCC: Kung-Robinson Model

- Xacts have three phases:
  - **READ**: Xacts read from the database, but make changes to private copies of objects.
  - **VALIDATE**: Check for conflicts with other Xacts
  - **WRITE**: Make local copies of changes public

  *Key idea*: check in validate phase if the Xact has behaved in a serializable manner

OCC: Validate Phase

- Each Xact is assigned a numeric id
  - Just use a timestamp (call it $T_i$)
  - Timestamps are assigned at end of *READ* phase, just before validation begins

  *Main question*: is the timestamp-ordering of xacts equivalent to a serial ordering

- The DBMS also keeps track of timestamp when each Xact starts and finishes

- Check for conflicts regarding:
  - $\text{ReadSet}(T_i)$: Set of objects read by Xact $T_i$
  - $\text{WriteSet}(T_i)$: Set of objects modified by $T_i$
What Can Go Wrong: Example 1

- Serial order for T1 and T2 determined by order they start validate phase
  - Can think of each *xact* executing instantaneously when its validation starts
- It is possible that T2 read A before T1 wrote A... but we don’t know!
  - We have to *abort* T2 just in case it didn’t see T1’s change
    (violating the presumed serial order)

What Can Go Wrong: Example 2

- Serial order: T1, T2
  - Final value of A should be T2’s version
- If we let T2 validate, *T2 could write A before T1 does*
  - (violating the presumed serial order)
  - Must *abort* T2

Validate Phase: Checking for Conflicts

- In both examples, serial order should be: T1, T2

- Example 1 issue:
  - T1 was still in write phase while T2 was reading, *and*
  - WriteSet(T1) overlaps ReadSet(T2)

- Example 2 issue:
  - T1 was still in write phase while T2 tried validating, *and*
  - WriteSet(T1) overlaps WriteSet(T2)

Test 1 – Non-overlapping

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

  Ti

  R V W

  Tj

  R V W

Tj sees changes made by Ti, ok since Ti happened serially before it

Need a set of tests to check for issues!
Test 2 – No Write Phase Overlap

- For all `i` and `j` such that `T_i < T_j`, check that:
  Ti completes before Tj begins its Write phase
  AND WriteSet(Ti) ∩ ReadSet(Tj) is empty.

Test 3 – Overlapping Write Phases

- For all `i` and `j` such that `T_i < T_j`, check that:
  Ti completes Read phase before Tj does
  AND WriteSet(Ti) ∩ ReadSet(Tj) is empty
  AND WriteSet(Ti) ∩ WriteSet(Tj) is empty.

Exercise 3

- How do the three tests prevent anomalies from RW, WR, WW conflicts?

  - Rule 2 takes care of RW and WR, since Tj can’t see Ti’s writes
  - Rule 3 takes care of WW: no lost updates if the WriteSets do not overlap

Exercise 4 (a-c)

- Which transaction(s) will commit successfully in each schedule?

  (a) Both will commit
  (b) Both will commit
  (c) T2 will abort, since its ReadSet overlaps T1’s WriteSet
Implementing OCC

- **Backwards serial validation** for a xact $T_v$:
  - Make sure serializability not violated with respect to all xacts $T_i$ that committed *after* $T_v$ started.

- In practice, implement a xact’s validate and write phases together in a critical section.
  - Only one xact can be executing its critical section at a time.

Serial Validation: Applying Tests 1 & 2 (backwards validation)

- To validate Xact $T_v$:
  ```
  valid = true;
  // S = set of Xacts that committed after Start($T_v$)
  // (above definition implements Test 1)
  // The following is done in critical section
  < foreach $T_i$ in S do {
    if ReadSet($T_v$) intersects WriteSet($T_i$)
      then valid = false;
  } 
  if valid then { install updates; // Write phase
    Commit $T$ } >
  else Abort $T$
  ```

Reflections on Serial Validation

- Applies Test 2, with $T_v$ playing the role of $T_j$ and each Xact in $T_s$ (in turn) being $T_i$.
  - So $T_v$ is looking “backwards” in time.

- Assignment of Xact id (timestamp), Validation, and the Write phase are inside a critical section!
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- cannot happen.

- Exercise 5
  - If Write phase is long, major drawback.
  - Optimization for read-only Xacts:
    - Don’t need critical section (because there is no Write phase).

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.

- Must check for conflicts during validation, and must make validated writes “global”.
  - Critical section can reduce concurrency.

- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted.
  - But this might not happen that much!
Finding Order for Xacts

- **Lock-based CC**
  - Conflicting actions ordered by the order that xacts acquired locks
  - This effectively orders the xacts

- **OCC (Kung-Robinson model)**
  - Xacts ordered by time they enter validate phase
  - Validation checks that any conflicting actions occur in that same order

- Another approach: **Timestamp CC**
  - Idea: If action $a_i$ of Xact $T_i$ conflicts with action $a_j$ of Xact $T_j$, and $\text{Start}(T_j) < \text{Start}(T_i)$, then only okay if $a_i$ occurs before $a_j$

**OK!**

- Objects get a read-timestamp (RTS) and a write-timestamp (WTS)
- Each Xact gets timestamp (TS) when it starts

---

MVCC and Snapshot Isolation (SI)

- **Idea**: avoid waiting to read a database object by having a xact read the most recent version of the object

- **Multi-version concurrency control (MVCC)**
  - Maintain several versions of objects with timestamps
  - Xact Ti reads most recent version of database before Ti started

- **Snapshot Isolation** is an implementation of MVCC
  - Xact Ti operates on its snapshot in isolation
  - *First Committer Wins* policy for conflicting xacts Ti and Tj

---

Reflection: Snapshot Isolation

- **The good**
  - Reading is *never* blocked, and reads don’t block writes
  - **Avoids common anomalies**
  - Matches common understanding of isolation: concurrent transactions are not aware of one another’s changes

- **The bad/ugly**
  - It turns out that it does *not* actually guarantee *serializability*
  - e.g., *write-skew anomaly*: two xacts read same data (e.g. A and B), but modify disjoint sets of data (A or B)

Some systems used to implement “Serializable” isolation level with SI