Concurrency Control
More!
R & G - Chapter 19

Smile, it is the key that fits the lock of everybody’s heart.
Anthony J. D’Angelo,
The College Blue Book

Review: Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Can result in Cascading Aborts!
  - STRICT (!) 2PL “Avoids Cascading Aborts” (ACA)

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
  - requires latches ("semaphores"), which ensure that the process is not interrupted while managing lock table entries
  - see CS162 for implementations of semaphores
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure it gets its original timestamp
  - Why?
Deadlock Detection (Continued)

Example:

T1: S(A), S(D), S(B)
T2: X(B), X(C)
T3: S(D), S(C), X(A)
T4: X(B), X(A)

Deadlock Detection (cont.)

• In practice, most systems do detection
  – Experiments show that most waits-for cycles are length 2 or 3
  – Hence few transactions need to be aborted
  – Implementations can vary
    • Can construct the graph and periodically look for cycles
    – When is the graph created?
      • Either continuously or at cycle checking time
      – Which process checks for cycles?
        • Separate deadlock detector
    • Can do a "time-out" scheme: if you’ve been waiting on a
      lock for a long time, assume you’re deadlock and abort

Summary

• Correctness criterion for isolation is "serializability".
  – In practice, we use "conflict serializability", which is
    somewhat more restrictive but easy to enforce.
• There are several lock-based concurrency control
  schemes (Strict 2PL, 2PL). Locks directly implement
  the notions of conflict.
  – The lock manager keeps track of the locks issued.
  Deadlocks can either be prevented or detected.

Things We’re Glossing Over

• What should we lock?
  – We assume tuples here, but that can be expensive!
  – If we do table locks, that’s too conservative
    • Multi-granularity locking
• Mechanisms
  – Locks and Latches
• Repeatability
  – In a Xact, what if a query is run again?
  – Are more records (phantoms) tolerable?

Multiple-Granularity Locks

• Hard to decide what granularity to lock (tuples vs. pages vs. tables).
• Shouldn’t have to make same decision for all transactions!
• Data “containers” are nested:

<table>
<thead>
<tr>
<th>Database</th>
<th>Tables</th>
<th>Pages</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 intension 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: Like S & IX at the same time. Why useful?
Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds SIX on parent? S on parent?
- 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.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Examples – 2 level hierarchy

<table>
<thead>
<tr>
<th>Tuples</th>
<th>IX</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Locks and Latches

- What’s common?
  - Both used to synchronize concurrent tasks
- What’s different?
  - Locks are used for logical consistency
  - Latches are used for physical consistency
- Why treat ‘em differently?
  - Database people like to reason about our data
- Where are latches used?
  - In a lock manager!
  - In a shared memory buffer manager
  - In a B+ Tree index
  - In a log/transaction/recovery manager

Locks vs Latches

<table>
<thead>
<tr>
<th></th>
<th>Latches</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ownership</td>
<td>Processes</td>
<td>Transactions</td>
</tr>
<tr>
<td>Duration</td>
<td>Very short</td>
<td>Long (Xact duration)</td>
</tr>
<tr>
<td>Deadlocks</td>
<td>No detection - code carefully!</td>
<td>Checked for deadlocks</td>
</tr>
<tr>
<td>Overhead</td>
<td>Cheap - 10s of instructions (latch is directly addressable)</td>
<td>Costly - 100s of instructions (got to search for lock)</td>
</tr>
<tr>
<td>Modes</td>
<td>S, X</td>
<td>S, X, IS, IX, SIX</td>
</tr>
<tr>
<td>Granularity</td>
<td>Flat - no hierarchy</td>
<td>Hierarchical</td>
</tr>
</tbody>
</table>

Dynamic Databases – The “Phantom” Problem

- If we relax the assumption that the DB is a fixed collection of objects, even strict 2PL (on individual items) will not assure serializability!
- Consider T1: “Find oldest sailor for each rating”
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).
  - Next, T2 inserts a new sailor; rating = 1, age = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits.
  - T1 now locks all pages containing sailor records with rating = 2, and finds oldest (say, age = 63).
- No serial execution where T1’s result could happen!
  - Let’s try it and see!

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with 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.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
  - e.g. table locks
### Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. `age > 2*salary`.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead.
  - too expensive!

### Instead of predicate locking

- Table scans lock entire tables
- Index lookups do "next-key" locking
  - physical stand-in for a logical range!