High-Performance Concurrency Control Mechanisms for Main-Memory Databases

Per-Åke Larson\textsuperscript{1}, Spyros Blanas\textsuperscript{2}, Cristian Diaconu\textsuperscript{1}, Craig Freedman\textsuperscript{1}, Jignesh M. Patel\textsuperscript{2}, Mike Zwilling\textsuperscript{1}

\textsuperscript{1}Microsoft Corporation \quad \textsuperscript{2}Univ. of Wisconsin-Madison
The problem

Most DBMSs designed for:
• Disk-resident data
• Few CPUs

$50K server in 2012:
• 1TB of RAM
• 40 CPUs

What concurrency control scheme should be used for a high-performance main-memory OLTP system?
Contributions

1. Multi-version optimistic concurrency control
   – Multi-version: readers don’t block writers
   – Optimistic: no waiting on database locks
   – Supports all SQL isolation levels
2. Efficient mechanisms for implementing multi-version and single-version locking
3. Experimental evaluation: High performance (millions of TX/sec) and full serializability without workload-specific knowledge
Recent related work

Our approach:

Redesign DBMS storage engine, make no assumption about workload

• Make existing DBMS storage engine scale:
  – Locking, page latching, B-tree index, logging, ...

• Exploit specific workload property:
  – Partitionable workload
  – Deterministic stored procedures
Designing a main memory storage engine

<table>
<thead>
<tr>
<th>Traditional disk-oriented engine</th>
<th>Our main memory prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk-friendly data structures</td>
<td>Latch-free hash table</td>
</tr>
<tr>
<td>- Pages, B-tree index</td>
<td>stores individual records</td>
</tr>
<tr>
<td>Absorbs high disk latency by</td>
<td></td>
</tr>
<tr>
<td>frequent context switching</td>
<td>Minimizes context switching</td>
</tr>
<tr>
<td>Thread spins for latches</td>
<td>- Usually 1, at most 2 per TX</td>
</tr>
<tr>
<td>TX may yield for locks</td>
<td>Eliminates latches</td>
</tr>
<tr>
<td>Critical sections are</td>
<td>TX never waits for locks</td>
</tr>
<tr>
<td>thousands of instructions</td>
<td>No critical sections</td>
</tr>
<tr>
<td>long, and limit scalability</td>
<td>- Many TXs finish in thousands of instructions</td>
</tr>
</tbody>
</table>
Multi-version optimistic scheme
Snapshot Isolation (SI)

- TXs have two unique timestamps: BEGIN, END
- **Read** as of BEGIN timestamp
- **Write** as of END timestamp

**Logical time**

- BEGIN: 2
- END: 4
- Snapshot Isolation (SI): Sufficient for Read Committed
- But not for Serializable
Making SI serializable

- **Read** as of BEGIN timestamp
- Repeat **Read** as of END timestamp, verify no change
- **Write** as of END timestamp

[Bornea et al, ICDE’11]
What needs to be repeated?

- Depends on the isolation level
- Read Committed or SI: No validation needed
  - Versions were committed at BEGIN, will still be committed at END
- Repeatable Read: Read versions again
  - Ensure no versions have disappeared from the view
- Serializable: Repeat scans with same predicate
  - Ensure no phantoms have appeared in the view
Transaction states

- **Active**: Get Begin Timestamp
- **Validating**: Get End Timestamp
- **Committed**: Log updates, wait for I/O
- **Aborted**: Serializability violation
- **Terminated**: Postprocessing

States:
- **Committed**
- **Validating**
- **Active**
- **Aborted**
- **Terminated**

Additional notes:
- Read only transaction
- User abort or WW conflict
Transaction map

- Stores transaction state, timestamps
- Globally visible

<table>
<thead>
<tr>
<th>TXID</th>
<th>STATE</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>ACTIV</td>
<td>2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Determining version visibility

Visibility as of time T is determined by:
version timestamps and TX state

TRANSACTION MAP

<table>
<thead>
<tr>
<th>TXID</th>
<th>STATE</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>ACTIV</td>
<td>2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Example: Update to $150

<table>
<thead>
<tr>
<th>TXID</th>
<th>STATE</th>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>John</td>
<td>$150</td>
</tr>
</tbody>
</table>

Get Begin Timestamp → Active → Get End Timestamp → Validating → Committed → Postprocessing

Log updates, wait for I/O → Terminated
WW conflicts

First writer wins

TX2 chooses to abort

TX2 updates $100 to $75

TX5

CAS

1  TX5  John  $100

TX5

CAS

8 bytes

TX5 writes John $150
WR conflicts

Q: When is version visible?
A: Depends on TX state

<table>
<thead>
<tr>
<th>TX5 State</th>
<th>Visible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE</td>
<td>No, version is uncommitted</td>
</tr>
<tr>
<td>VALIDATING</td>
<td>Speculate YES now, confirm at end</td>
</tr>
<tr>
<td>COMMITTED</td>
<td>Maybe, check TX5 END timestamp</td>
</tr>
<tr>
<td>ABORTED</td>
<td>No, version is garbage</td>
</tr>
</tbody>
</table>
Commit dependencies

• Impose constraint on serialization order: *Commit B only if A has committed.*

• Implementation: register-and-signal
  – Transform multiple waits on every record access to a single wait at end of TX
  – Dependency wait time “added” to log latency
    • Most common: no wait needed, dependency has cleared

• But: Cascading aborts now possible
Commit dependencies

- **Active**
  - Read only transaction
  - Get Begin Timestamp

- **Committed**
  - Wait for dependencies to clear, then
  - Get End Timestamp
  - Log updates, wait for I/O

- **Validating**
  - Serializability violation

- **Aborted**
  - User abort or WW conflict

- **Terminated**
  - Release dependents Postprocessing

- **Postprocessing**
  - Release dependents

Multi-version optimistic summary

- TXs never wait during the ACTIVE phase
- No deadlock detection is needed
- Lower isolation level = less work
  - Read Committed and SI: No validation at all
Multi-version locking

• Provides lock-like semantics: 
  *Once a version is read by T, it will remain visible to T until commit.*

• No centralized lock table
  – Record lock embedded in version’s END timestamp

• Same context switching overhead: At most 2 per TX

But:

• Deadlock detection necessary

• More write traffic, even readers write to memory
Implementation details

• Independent transaction kernel in C++
• Base data structure: latch-free hash table
  – Perfect sizing, perfect hashing
  – Load factor when idle: 1
Single-version two-phase locking

• Traditional 2PL, optimized for main memory
• No central lock manager
• Lock is pre-allocated in hash table bucket
  – Protects hash bucket, prevents phantoms
  – Multiple-reader, single-writer lock
  – For our experiments, also serves as a record lock
Experimental setup

• 2-socket × 6-core Xeon X5650 with 48GB RAM
• TXs don’t wait for the log (lazy commit)
  – Log records are populated and written to disk
• All transactions run under Serializable

<table>
<thead>
<tr>
<th>MV/O</th>
<th>Multi-version optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV/L</td>
<td>Multi-version locking</td>
</tr>
<tr>
<td>1V</td>
<td>Single-version two-phase locking</td>
</tr>
</tbody>
</table>
Scalability
No contention (10M row table)

Throughput (tx/sec) Millions

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5

0 6 12 18 24

Threads

80% R=10
20% R=10, W=2
Scalability

No contention (10M row table)

Throughput (tx/sec) vs. Threads

<table>
<thead>
<tr>
<th></th>
<th>1V</th>
<th>MV/L</th>
<th>MV/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>80% R=10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20% R=10, W=2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All methods scale

Similar results for TATP
Scalability
Extreme contention (1000 row table)

Throughput (tx/sec) [Millions]

80% R=10
20% R=10, W=2

1V throughput limited by lock thrashing

1V throughput limited by lock thrashing

5x

Threads
Effect of long readers
(10M row table)

Update throughput (tx/sec) Millions

R=1,000,000
R=10, W=2

6 TXs long readers
18 TXs short updaters

All active TXs short updaters
All active TXs long readers
Effect of long readers
(10M row table)

If all TXs do updates, 1V 1.9× faster
Even if 1 long reader, MV/O 2.3× faster

All active TXs short updaters
All active TXs long readers
Conclusions

• Single-version 2PL is fragile
  – Great for update-heavy workloads, little contention
  – But: problematic for hotspots, long read TXs
• Multi-version optimistic scheme is robust
  – Readers don’t block writers, no waiting on locks
• Locking semantics can be offered efficiently
• High performance and full serializability without workload-specific knowledge