Rethinking Serializable Multi-version Concurrency Control

Jose Faleiro and Daniel Abadi
Yale University
Theory: Single- vs Multi-version Systems

Single-version system

- $T_r$ – Read $X$
- $T_w$ – Write $X$
- $X_0$

Multi-version system

- $T_r$ – Read $X$
- $T_w$ – Write $X$
- $X_0$
Theory: Single- vs Multi-version Systems

Single-version system
- $T_r$ – Read $X$
- $T_w$ – Write $X$
- $X_0$

Multi-version system
- $T_r$ – Read $X$
- $T_w$ – Write $X$
- $X_0$

$T_r$ and $T_w$ cannot simultaneously execute
**Theory:**

**Single- vs Multi-version Systems**

<table>
<thead>
<tr>
<th>Single-version system</th>
<th>Tr – Read X</th>
<th>Tw – Write X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="X0" alt="X0" /></td>
<td></td>
</tr>
</tbody>
</table>

**Multi-version system**

<table>
<thead>
<tr>
<th>Tr – Read X</th>
<th>Tw – Write X</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="X0" alt="X0" /></td>
<td><img src="X1" alt="X1" /></td>
</tr>
</tbody>
</table>

- Tr and Tw cannot simultaneously execute
- Tr and Tw can both simultaneously make progress
Theory: Single- vs Multi-version Systems

Multi-versioning obviously buys more concurrency, right?
Practice: Multi-version Systems

\[ T_0: \]
if savings + checking \(\geq\) 100
savings \(\equiv\) 100

\[ T_1: \]
if savings + checking \(\geq\) 75
checking \(\equiv\) 75

Savings: 100

Checking: 50

Constraint:
savings + checking \(\geq\) 0
Practice: Multi-version Systems

$T_0$ commits

$T_0$: if savings + checking $\geq 100$
  savings $\gets 100$

$T_1$: if savings + checking $\geq 75$
  checking $\gets 75$

Constraint: savings + checking $\geq 0$

Savings: 100
Checking: 50

$\rightarrow$

Savings: 0
Practice: Multi-version Systems

**T₀ commits**

*T₀:*
if savings + checking >= 100
savings -= 100

**T₁ commits**

*T₁:*
if savings + checking >= 75
checking -= 75

<table>
<thead>
<tr>
<th>Savings: 100</th>
<th>Savings: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking: 50</td>
<td>Checking: -25</td>
</tr>
</tbody>
</table>

**Constraint:** savings + checking >= 0
## Practice: Multi-version Systems

**T₀ commits**

<table>
<thead>
<tr>
<th>T₀:</th>
<th>if savings + checking ≥ 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>savings = 100</td>
</tr>
</tbody>
</table>

**Savings: 100**

**Checking: 50**

**T₁ commits**

<table>
<thead>
<tr>
<th>T₁:</th>
<th>if savings + checking ≥ 75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>checking = 75</td>
</tr>
</tbody>
</table>

**Savings: 0**

**Checking: -25**

**Constraint:**

```
savings + checking ≥ 0
```
Practice: Multi-version Systems

“Solution”: Use conservative isolation techniques similar to single-version systems
**Practice: Multi-version Systems**

\[ T_0: \]

\[ \text{savings } += 100 \]

**Monotonic timestamp generator**

**Savings: 25**

**begin: 0**  **end: 10**

**Savings: 75**

**begin: 10**  **end: inf**
Practice: Multi-version Systems

\[ T_0: \]
\[ \text{savings} += 100 \]

Begin timestamp?

Monotonic timestamp generator

Savings: 25
begin: 0 end: 10

Savings: 75
begin: 10 end: \( \text{inf} \)
Practice: Multi-version Systems

$T_0$: $\text{savings += 100}$

- **Savings:** 25
  - *begin:* 0, *end:* 10

- **Savings:** 75
  - *begin:* 10, *end:* $\infty$

Monotonic timestamp generator

Determines snapshot visible to txn
Practise: Multi-version Systems

\( T_0: \) savings += 100

<table>
<thead>
<tr>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Savings: 25

<table>
<thead>
<tr>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Savings: 75

<table>
<thead>
<tr>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>inf</td>
</tr>
</tbody>
</table>

Monotonic timestamp generator

End timestamp?
Practice: Multi-version Systems

$T_0$: $savings += 100$

begin: 12 end: 18

Determines visibility of txn’s writes

Monotonic timestamp generator

Savings: 25 begin: 0 end: 10

Savings: 75 begin: 10 end: 18

Savings: 175 begin: 18 end: inf
Practice: Multi-version Systems

\[ T_0: \]
savings += 100

Monotonic timestamp generator

Global counter

Scalability bottleneck!

Savings: 175

begin: 0  end: 10
begin: 10 end: 18
begin: 18 end: inf
Practice: Multi-version Systems

Version management overhead

- Savings: 25
  - begin: 0
  - end: 10

- Savings: 75
  - begin: 10
  - end: 18

- Savings: 175
  - begin: 18
  - end: inf
Practice: Multi-version Systems

• Offer the same amount of concurrency as single-version systems
  • Do not effectively exploit multi-versioning

• Significant sources of overhead
  • Contended counters
  • Version management
Practice: Multi-version Systems

• Offer the same amount of concurrency as single-version systems
  • Do not effectively exploit multi-versioning

• Significant sources of overhead
  • Contended counters
  • Version management

Fundamental issue with concurrency control protocols

Severe performance degradation on multi-core main-memory systems
Root Cause

- Multi-version databases enforce serial order **dynamically**
  - Concurrency control occurs during transaction execution

- Requires conservative concurrency control
  - Same read-write concurrency as single-versioning

- Concurrency control meta-data prone to contention
Our Approach: Bohm

• Separate concurrency control from transaction execution

• Concurrency control determines transaction order and version visibility

• Execution performs logic given concurrency control ordering
Bohm Overview

Concurrency Control

Determine legal schedule

Execution

Perform logic
Extra Requirements

• Transactions’ entire logic must be submitted in one piece

• Transactions’ write-sets must be deducible prior to their execution
Concurrency Control Layer

- Take a totally ordered batch of txns as input
- Create a new version for each write
  - Versions created in sequence order
Concurrent Control Layer

- Partition data across multiple threads
Concurrency Control Layer

- Partition data across multiple threads
- For every write, create a new version
Concurrency Control Layer

- Partition data across multiple threads
- For every write, create a new version

\[ T_{200} \]

\[ \begin{array}{cccc}
  a & b & h & f \\
\end{array} \]

\[ \begin{array}{cccc}
  a & e & g \\
\end{array} \]

\[ \text{value}_0 \]

\[ \begin{array}{cc}
  \text{begin: 0} & \text{end: inf} \\
\end{array} \]
Concurrency Control Layer

• Partition data across multiple threads
• For every write, create a new version

\[ \text{T}_{200} \]

\[ \text{begin: 200} \quad \text{end: inf} \]

\[ \text{prev} \]

\[ \text{value}_0 \]

\[ \text{begin: 0} \quad \text{end: 200} \]
Concurrency Control Layer

• Partition data across multiple threads
• For every write, create a new version

Version contains txn reference
No value yet
Execution Layer

- Begins executing a batch after concurrency control completes
- Perform txn logic, write out data
Execution Layer

- Begins executing a batch after concurrency control completes
- Perform txn logic, write out data

Replace txn reference with actual data

- $T_{200}$
  - begin: 200
  - end: inf
- prev
- $value_0$
  - begin: 0
  - end: 200
• Begins executing a batch after concurrency control completes

• Perform txn logic, write out data

Replace txn reference with actual data
Implications of Design

Concurrency control fixes txn snapshots prior to their execution

Execution only produces values

No logical locking/validation

Concurrent transaction T_{200}

\text{value}_0

begin: 200 \quad end: \text{inf}

begin: 0 \quad end: 200
What have we gained?

- **Strong concurrency guarantees**
  - Reads *never* block writes
  - Write-write conflicts *never* lead to aborts

- **Significant reduction in contention**
  - No contended counters
  - Contention-free concurrency control
Baselines

• **Hekaton**
  • State-of-the-art MVCC protocol
  • No garbage collection, simple array indices

• **Snapshot Isolation**
  • Not serializable
  • Based on Hekaton

• **OCC**
  • Silo – Decentralized timestamps, latch-free validation

• **Two-Phase Locking**
  • No global latches, deadlock free
Evaluation

• Effect of read-write concurrency
  • YCSB-like: 2RMWs + 8 Reads
  • High contention
Conclusions

• State-of-the-art databases are unable to reduce read-write conflicts despite using multi-versioning

• Bohm: Separate concurrency control from execution

• Bohm makes strong concurrency guarantees
  • Reads never block writes
  • Write-write conflicts never lead to aborts

• Bohm eliminates contention due to concurrency control
Thanks!

JMFALEIRO.COM

JOSE.FALEIRO@YALE.EDU
Limitations of Contended Counters

![Graph showing throughput vs. number of threads for different systems (Bohm, 2PL, OCC, SI, Hekaton). The x-axis represents the number of threads ranging from 0 to 44, and the y-axis represents throughput in txns/sec ranging from 0 to 4 M, with a 3x zoom on the y-axis.]
Varying Contention

Throughput (txns/sec)

Increasing contention

Theta

OCC
Bohm
2PL
SI
Hekaton