High-Performance Concurrency Control Mechanisms for Main-Memory Databases

Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling

Presented by Jamie Shin
Introduction

- Memory prices continue to decline
- The majority of OLTP databases will fit in memory
- DBMS optimized for in-memory storage running on a multicore processor will support high transaction rates
- Efficiently ensuring isolation between concurrent transactions is challenging
- In traditional systems, lock managers become a bottleneck
- Long read-only transactions are also problematic
This paper proposes...

- Optimistic: MVCC method relying on validation
- Pessimistic: redesign of two locking-based concurrency control methods
  - single-version
  - multiversion
- Evaluation of the three concurrent control methods for different workloads
**MV Storage Engine**

**Snapshot Isolation (SI):**
- Widely used MVCC method
- Does not guarantee serializability
- Reads happen at the beginning, and writes at the end of transaction
- A transaction is *serializable* if we can guarantee it would see the same data if all its reads repeated at the end of the transaction.

For serializability, we need to guarantee:

1. **Read stability**: if a transaction reads some version, that version doesn’t change til the end
2. **Phantom avoidance**: transaction’s scans would not return new versions
Outline

1. Preliminaries of multiversioning
2. Optimistic scheme
3. Pessimistic scheme
4. Evaluation
Storage and Indexing

- Hash indexes with lock-free hash tables
- A table can have many indexes
- Records are always accessed via index lookup
- No direct access to a record without going through an index

Figure 1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Reads

- Every read specifies a logical read time
- Only versions whose valid time overlaps the read time are visible to the read

Figure 1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Updates

- Transaction 75 transfers $20 from Larry to John

Figure 1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Updates

- Update creates new versions for Larry and John

Figure 1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Updates

- Tx ID in End field serves as lock
- Tx ID in Begin informs the version may not yet be committed

Figure 1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Updates

- Tx 75 commits with End timestamp of 100 for old versions and Begin of 100 for new versions

![Diagram showing transactions between Larry and John with a $20 transfer from Larry to John.](http://www.flaticon.com/authors/freepik from www.flaticon.com)

**Figure 1:** Example account table with one hash index. Transaction 75 has transferred $20 from Larry’s account to John’s account but has not yet committed.
Transaction Phases

A transaction can be in one of four states:
1. Active
2. Preparing
3. Committed
4. Aborted

A transaction goes through three different phases:
1. Normal processing phase
2. Preparation phase
3. Postprocessing phase
Version Visibility

- **RT**: logical read time used by a transaction \( T \)
- To determine if a version \( V \) is visible to \( T \), check \( V \)’s Begin end End fields

Three cases:

1. Begin and End fields contain timestamps
2. Begin field contains a transaction ID
3. End field contains a transaction ID
Version Visibility

1. **Begin** and **End** fields contain timestamps
   - Visible if a transaction’s read time falls between two timestamps
2. **Begin** field contains a transaction ID of a transaction **TB**

<table>
<thead>
<tr>
<th>TB’s state</th>
<th>TB’s end timestamp</th>
<th>Action to take when transaction T checks visibility of version V.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Not set</td>
<td>V is visible only if TB=T and V’s end timestamp equals infinity.</td>
</tr>
<tr>
<td>Preparing</td>
<td>TS</td>
<td>V’s begin timestamp will be TS but V is not yet committed. Use TS as V’s begin time when testing visibility. If the test is true, allow T to <em>speculatively</em> read V.</td>
</tr>
<tr>
<td>Committed</td>
<td>TS</td>
<td>V’s begin timestamp will be TS and V is committed. Use TS as V’s begin time to test visibility.</td>
</tr>
<tr>
<td>Aborted</td>
<td>Irrelevant</td>
<td>Ignore V; it’s a garbage version.</td>
</tr>
<tr>
<td>Terminated or not found</td>
<td>Irrelevant</td>
<td>Reread V’s Begin field. TB has terminated so it must have finalized the timestamp.</td>
</tr>
</tbody>
</table>
Version Visibility

3. **End** field contains a transaction ID of a transaction **TE**
   - If **TE** is Preparing, its end timestamp **TS** that will become the end timestamp of **V** if **TE** does commit. If it is greater than **RT**, **V** will be visible.
Version Visibility

3. **End** field contains a transaction ID of a transaction **TE**
   - When **TS** is less than **RT**, **V** will not be visible if **TE** commits, but will be visible if it aborts
   - In such case, allow **T** to **speculatively ignore V**

<table>
<thead>
<tr>
<th>TE’s state</th>
<th>TE’s end timestamp</th>
<th>Action to take when transaction T checks visibility of a version V as of read time RT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Not set</td>
<td>V is visible only if V was created by transaction T and V’s end timestamp equals infinity.</td>
</tr>
<tr>
<td>Preparing</td>
<td>TS</td>
<td>V’s end timestamp will be TS provided that TE commits. If TS &gt; RT, V is visible to T. If TS &lt; RT, T speculatively ignores V.</td>
</tr>
<tr>
<td>Committed</td>
<td>TS</td>
<td>V’s end timestamp will be TS and V is committed. Use TS as V’s end timestamp when testing visibility</td>
</tr>
<tr>
<td>Aborted</td>
<td>Irrelevant</td>
<td>V is visible.</td>
</tr>
<tr>
<td>Terminated or not found</td>
<td>Irrelevant</td>
<td>Reread V’s End field. TE has terminated so it must have finalized the timestamp.</td>
</tr>
</tbody>
</table>
Version Visibility

3. End field contains a transaction ID of a transaction TE
   - If TE has aborted, some other tx TO may have sneaked in after T read V's End field, updated V
   - Doesn’t matter as TO’s end timestamp must be later than RT

<table>
<thead>
<tr>
<th>TE’s state</th>
<th>TE’s end timestamp</th>
<th>Action to take when transaction T checks visibility of a version V as of read time RT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Not set</td>
<td>V is visible only if V was created by transaction T and V’s end timestamp equals infinity.</td>
</tr>
<tr>
<td>Preparing</td>
<td>TS</td>
<td>V’s end timestamp will be TS provided that TE commits. If TS &gt; RT, V is visible to T. If TS &lt; RT, T speculatively ignores V.</td>
</tr>
<tr>
<td>Committed</td>
<td>TS</td>
<td>V’s end timestamp will be TS and V is committed. Use TS as V’s end timestamp when testing visibility</td>
</tr>
<tr>
<td>Aborted</td>
<td>Irrelevant</td>
<td>V is visible.</td>
</tr>
<tr>
<td>Terminated or not found</td>
<td>Irrelevant</td>
<td>Reread V’s End field. TE has terminated so it must have finalized the timestamp.</td>
</tr>
</tbody>
</table>
Updating a Version

- A version is updatable only if it is the latest version
- If a version’s End field contains ID of a transaction TE (Active/Preparing), V is the latest committed version but there is a later uncommitted version.
- In such a case, we force transaction T to abort
Commit Dependencies

- Cascading aborts are possible
- Commit dependencies are implemented by a register-and-report approach
- T1 registers its dependency with T2 and T2 informs T1 when it has committed or aborted
- Each transaction has a counter, CommitDepCounter that counts its unresolved commit dependencies
- A transaction cannot commit until the counter is zero
- AbortNow
- CommitDepSet
- Transactions may or may not have to wait before commit
- Deadlock is impossible
Outline

1. Preliminaries of multiversioning
2. Optimistic scheme
3. Pessimistic scheme
4. Evaluation
Optimistic Transactions

● Backward validation method is used (check version visibility at the end of tx)
● A transaction object contains three sets:
  ○ ReadSet: pointers to versions read
  ○ ScanSet: information needed to repeat scans
  ○ WriteSet: pointers to versions updated, deleted, and inserted
Normal Processing Phase

Outline of a scan when a transaction T runs at serializable isolation level:

- Start scan
- Check predicate
- Check visibility
- Read version
- Check updatability
- Update version
- Delete version
Normal Processing Phase

Outline of a scan when a transaction $T$ runs at serializable isolation level:

- **Start scan**
  - Scan is registered in $T$’s ScanSet so $T$ can check for phantoms during validation

- **Check predicate**
  - If a version doesn’t satisfy the given predicate, it is ignored

- **Check visibility**
  - We check whether version $V$ is visible to transaction $T$ as of time $RT$
  - A commit dependency may be registered if necessary

- **Read version**
  - $T$ records the read by adding a pointer to $V$ to its ReadSet

- **Check updatability**
  - If $T$ intends to update or delete $V$, check the updatability of the version
  - Updatable if the End field equals infinity or contains a tx ID and the tx is aborted
Normal Processing Phase

Outline of a scan when a transaction $T$ runs at serializable isolation level:

- **Update version**
  - $T$ first creates a new version $VN$ and atomically sets $V$’s End field to $T$’s transaction ID
  - If there is a write-write conflict, $T$ aborts
  - If $T$ succeeds in setting $V$’s End field, it serves as an exclusive write lock
  - $T$ records the update by adding two pointers to its $\text{WriteSet}$: a pointer to $V$ and a pointer to $VN$
  - The pointers are used for:
    - Logging new versions during commit
    - Postprocessing after commit or abort
    - Locating old versions when they are no longer needed
Normal Processing Phase

Outline of a scan when a transaction $T$ runs at serializable isolation level:

- **Delete version**
  - Does not create new versions
  - If successfully set the end timestamp of $V$, a pointer to $V$ (old) is added to the WriteSet and the delete is complete
- **When $T$ reaches the end, it precommits** and begins the preparation phase
- **Precommit** consists of
  - acquiring the transaction’s end timestamp
  - setting the transaction state to Preparing
Preparation Phase

Consists of three steps:
1. Read validation
2. Waiting for commit dependencies
3. Logging

- If T fails validation, it is not serializable and must abort
- Otherwise it must wait for outstanding commit dependencies to be resolved

Figure 3: Possible validation outcomes.
Preparation Phase

Consists of three steps:
1. Read validation
2. Waiting for commit dependencies
3. Logging
   - To complete the commit, T scans its WriteSet and writes new versions to a persistent log.
   - After the log writes, T sets its state to Committed

Figure 3: Possible validation outcomes.

<table>
<thead>
<tr>
<th>Read Validation</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>V2 Fail</td>
<td>Pass</td>
</tr>
<tr>
<td>V3 NA</td>
<td>Pass</td>
</tr>
<tr>
<td>V4 NA</td>
<td>Fail</td>
</tr>
</tbody>
</table>
Postprocessing

- A committed transaction TC propagates its end timestamp to Begin and End fields of new and old versions, listed in its WriteSet.
- An aborted transaction TA sets the Begin field of its new versions to infinity, and attempts to reset the End fields of its old versions to infinity.
- If another transaction reset the End field of the old version, TA leaves it unchanged.
- The transaction then processes commit dependencies:
  - If aborted, force all dependent transactions to abort.
  - If committed, decrease their dependents’ CommitDepCounter and wake up those whose count become zero.
Outline

1. Preliminaries of multiversioning
2. Optimistic scheme
3. Pessimistic scheme
4. Evaluation
Pessimistic Transactions

- Prevents its reads from being invalidated with read locks
- A serializable pessimistic transaction must keep track of:
  - Which versions it read
  - Which hash buckets it scanned
  - Its new and old versions
- Transaction maintains three sets:
  - **ReadSet**: pointers to versions read locked by transaction
  - **BucketLockSet**: pointers to hash buckets visited and locked
  - **WriteSet**: references to versions updated (old\&new), versions deleted (old), and versions inserted(new).
Lock Types

- **Record lock**
  - Ensures read stability
  - Required only for the latest version of a record
  - Embedded in the End field of versions so no extra space required
  - NoMoreReadLocks, ReadLockCount, WriteLock

- **Bucket lock (range lock)**
  - Placed on (hash) buckets to prevent phantoms
  - When a transaction TS begins a scan of a hash bucket, it locks the bucket
  - Multiple (serializable) transactions can have a bucket locked
  - LockCount, LockList
Eager Updates

- To avoid blocking, we allow a transaction **TU** to eagerly update/delete a read locked version **V** but it cannot **precommit** until all read locks on **V** are released.
- Similarly, a transaction **TR** can acquire a read lock on a version that is already write locked by **TU**. If so, **TU** cannot precommit until **TR** has released its lock.
- The same applies to locked buckets.
Wait-For Dependencies

- Used to enforce correct serialization order
- Forces an update transaction $TU$ to wait before it can acquire an end timestamp and begin commit processing

**Incoming dependency:** T waits on some other transaction
- WaitForCounter
- NoMoreWaitFors

**Outgoing dependency:** T some other transaction waits on T
- WaitingTxnList
Wait-For Dependencies

- Read lock dependencies
  - A transaction $TU$ that updated or deleted a version $V$ has a wait-for dependency on $V$ as long as $V$ is read locked
  - $TU$ is not allowed to acquire an end timestamp and begin commit unless $V$’s ReadLockCount is zero

- Bucket lock dependencies
  - A serializable transaction $TS$ acquires a lock on a bucket $B$ by incrementing $B$’s LockCounter and adding its transaction ID to $B$’s LockList
  - Another tx $TU$ can add a version to $B$ but then $TU$ cannot precommit until $TS$ releases its lock on $B$
Normal Processing Phase

• For snapshot isolation, the read time is always the tx begin time
• For all others, it is the current time which has the effect that the read sees the latest version of a record

• Start scan
• Check predicate
• Check visibility
• Read version
• Check updatability
• Update version
• Delete version
Normal Processing Phase

- **Start scan**
  - If T is serializable, it puts a bucket lock on B to prevent phantoms and records in BucketLockSet
- **Check predicate**
- **Check visibility**
- **Read version**
  - If T runs under a lower isolation level than serializable, no read lock is required
- **Check updatability**
- **Update version**
  - T creates a new version N, sets V’s WriteLock and, if V was read locked, takes out a wait-for dependency on V by incrementing its own WaitForCounter
- **Delete version**: Essentially an update of V that doesn’t create a new version
Processing Phases

- **Preparation Phase**
  - Pessimistic transactions require no validation
  - T waits if there is any outstanding commit dependencies

- **Postprocessing Phase**
  - Same as for optimistic transactions
Deadlock Detection

- Wait-for graph by analyzing wait-for dependencies of all transactions that are currently blocked
- Any algorithm for finding cycles in graphs can be used; Tarjan’s algorithm used
Outline

1. Preliminaries of multiversioning
2. Optimistic scheme
3. Pessimistic scheme
4. Evaluation
Experimental Results

- **1V**: Single version locking
- **MV/O**: Multi-version engine when transactions run **optimistically**
- **MV/L**: Multi-version engine when transactions run **pessimistically**

- Single-version locking works well when tx’s are short and contention is low
- Multiversion schemes have higher overhead but are less sensitive to hotspots and the presence of long-running transactions

1. **Homogeneous workload**: A single transaction type that performs R reads and W writes against a table of N records with a unique key
2. **Heterogeneous workload**: Added read-only transactions in the workload mix
Homogeneous workload: Scalability under low contention
Homogeneous workload: Scalability under high contention
Heterogeneous workload: impact of read-only tx’s
Heterogeneous workload: impact of long read tx’s
Conclusions

- Single-version locking can be implemented efficiently and without lock acquisition becoming a bottleneck.
- Single-version locking is fragile; it performs well when transactions are short and contention is low but suffers under more demanding conditions.
- The MVCC schemes have higher overhead but are more resilient, retaining good throughput even in the presence of hotspots and long read-only transactions.
- The optimistic MVCC scheme consistently achieves higher throughput than the pessimistic scheme.