Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems

By Thomas Neumann, Tobias Mühlbauer, Alfons Kemper

Presented by Samodya Abeysiriwardane

Transaction Isolation

- ACID (Atomicity, Consistency, Isolation, Durability)
- Provides the user with the illusion that it will be executed alone
- Database ensures that concurrent transactions can be safely ordered. Ideally a serializable ordering.

Serializability

- 2PL ensures serializability but limits the degree of concurrency.
  - Readers and writers block each other
  - Although most transactions are readonly
- Serializability is hard to implement efficiently.

MVCC

- Mutli-Version Concurrency Control
- Each Update creates a new versoin of the data object
- Therefore concurrent readers can read the old version => Read-only transactions never have to wait
- But most implementations only provide Snapshot Isolation (SI)

Serializable SI

- SI offers fairly good isolation but some schedules are not serializable
- Known solution: keep track of the entire read set of the transaction and verify that its observed values are consistent with serial order
- Very expensive for read heavy workloads

This paper

- MVCC implementation that is efficient, both for SI and full serializability
- Retain high scan performance of single version systems

Implementation

- Integrated into HyPer main memory database
- ACID compliant transaction processing
- Queries and transactions generate LLVM code

Implementaion

- In place updates
  - High scan speed
- Undo buffer
  - No additional overhead
- VersionVector anchors chain of reconstruction deltas
  - Newest to oldest
- Versioned Positions
- Version access
  - v.pred = null or v.pred.TS = T or v.pred.TS < T.startt

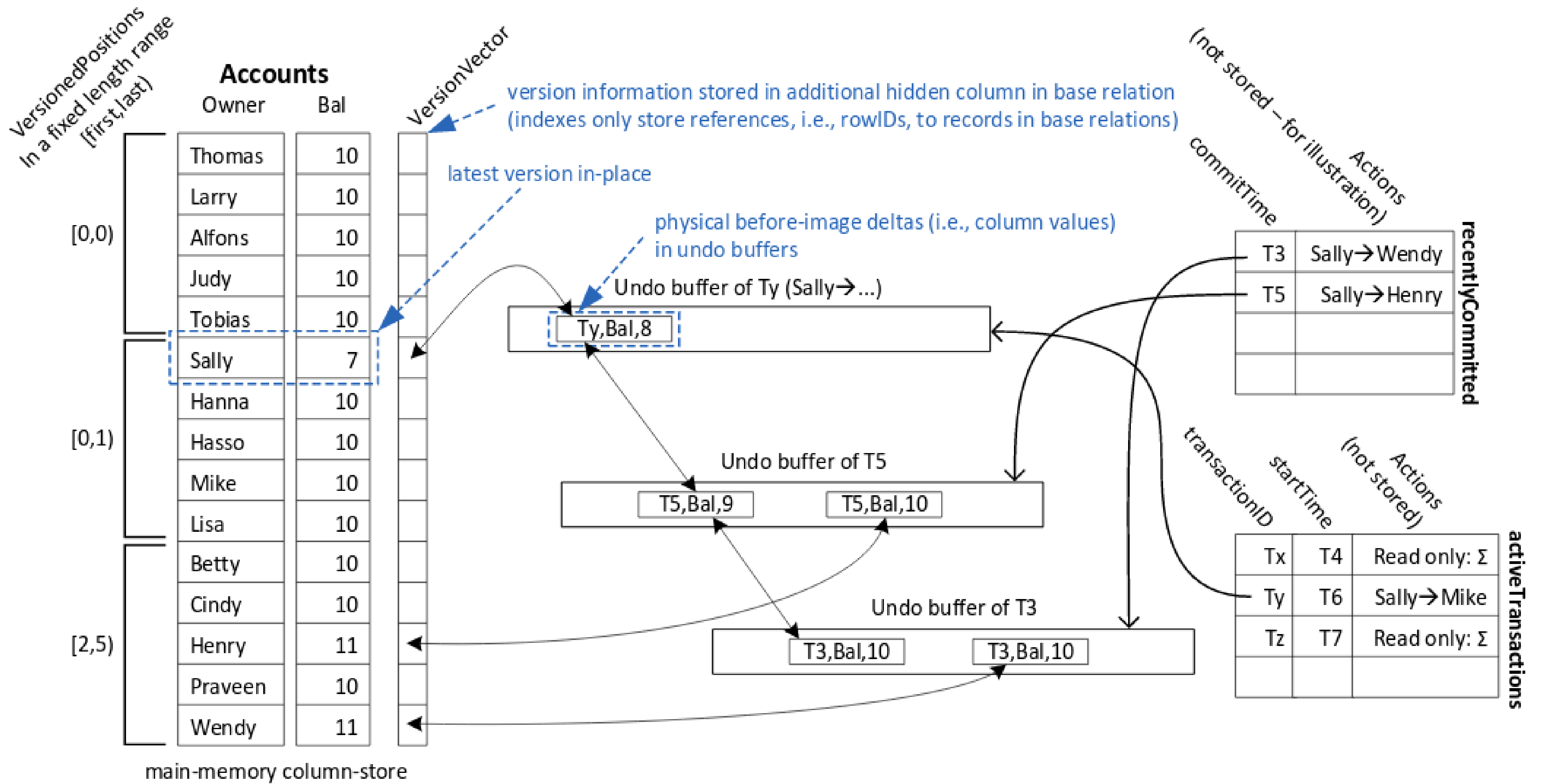


Figure 1: Multi-version concurrency control example: transferring $1 between Accounts (from → to) and summing all Balances (Σ)

Serializability Validation

- Complex in other approaches when readset is large
- Limit the validation to the objects that actually changed

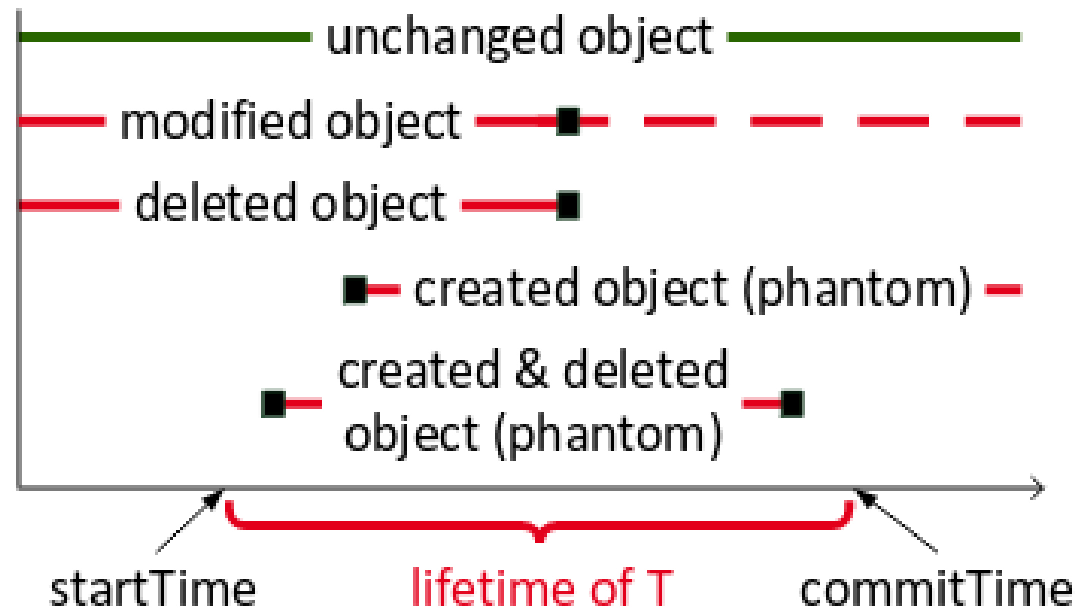


Figure 2: Modifications/deletions/creations of data objects relative to the lifetime of transaction T

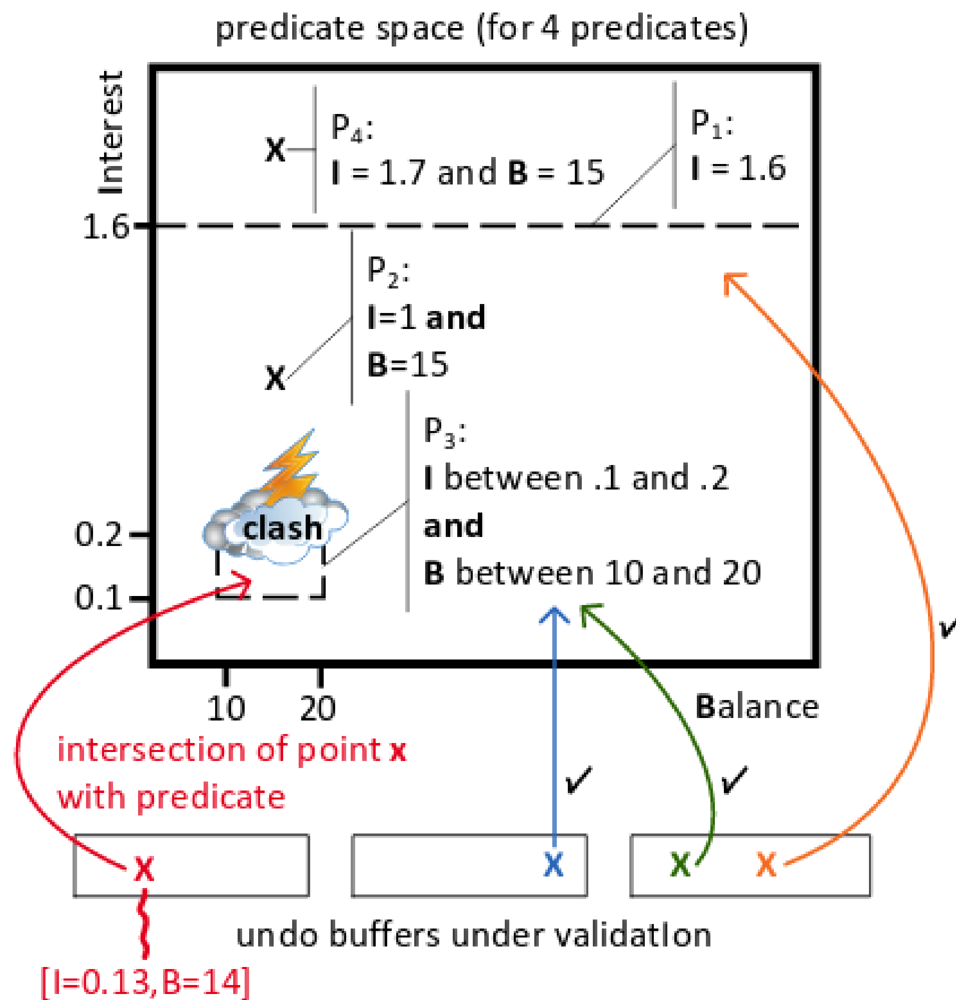
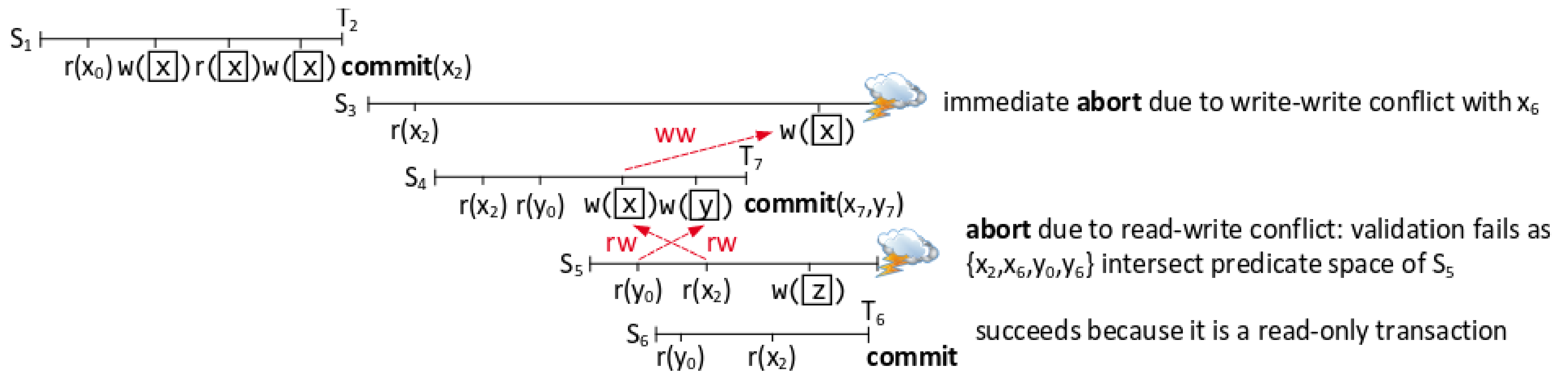


Figure 3: Checking data points in the undo buffers against the predicate space of a transaction

Garbage collection

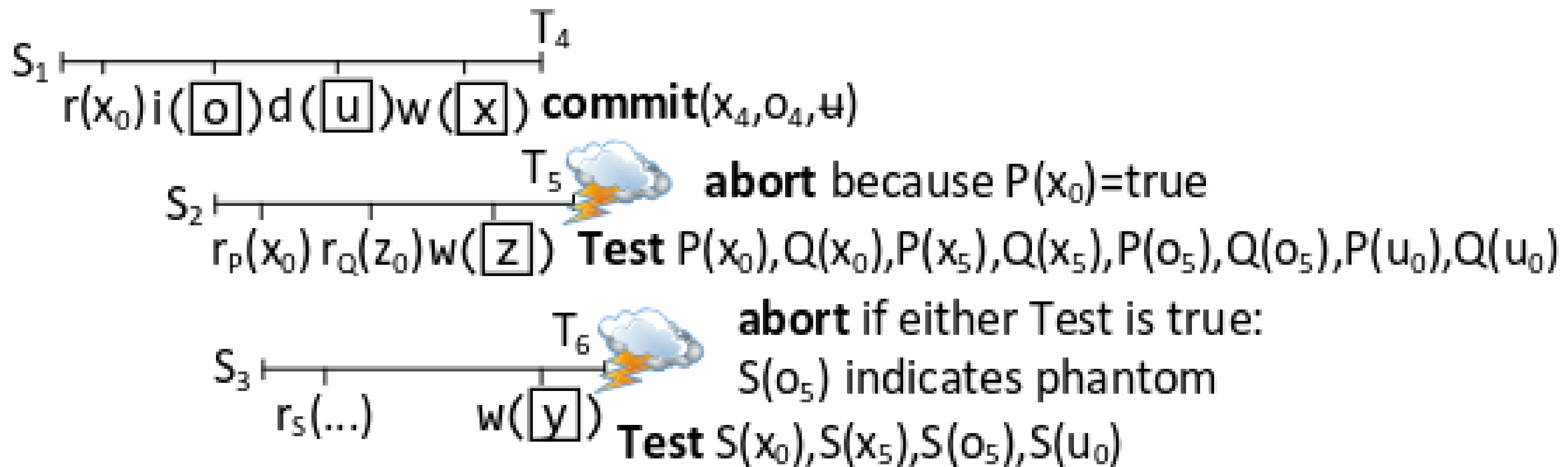
- Buildup of old versions can be a problem
- Whenever transaction commits do Garbage collection
- Find oldest committed transaction that has visible update to an active transaction, then remove all transaction older than that

Examples



(a) a write-write conflict and a read-write conflict

Examples



(c) a read-write conflict, a read-delete conflict, and a phantom conflict

Evaluation

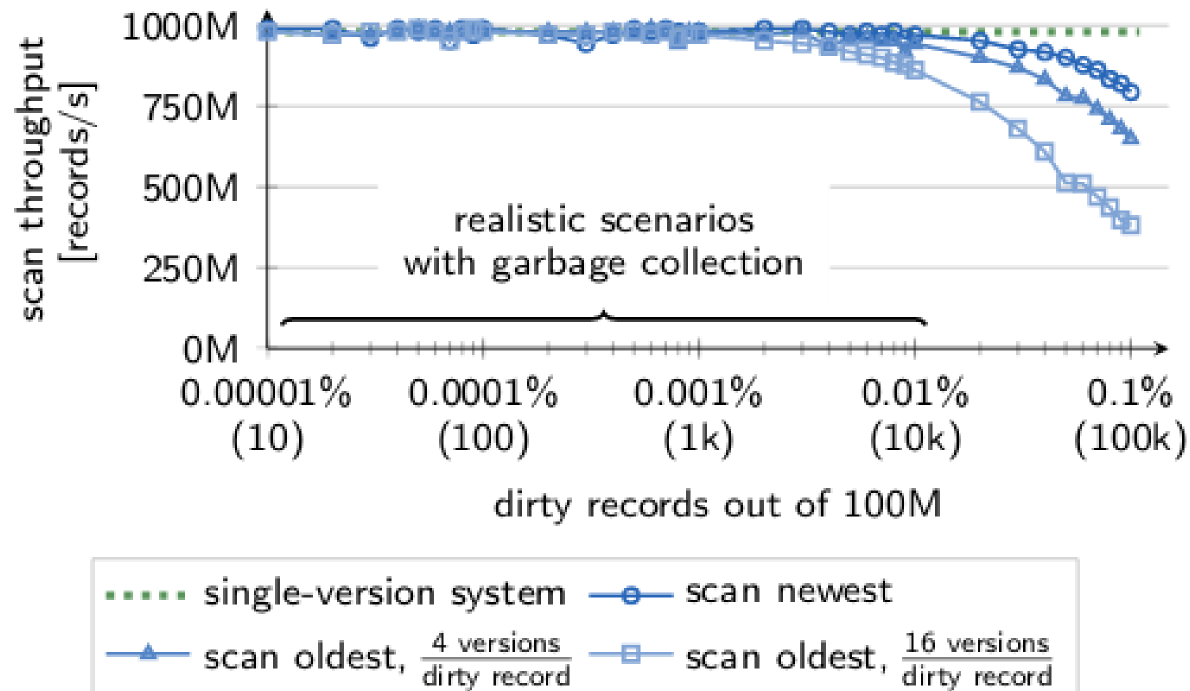


Figure 7: Scan performance with disabled garbage collection: the *scan newest* transaction only needs to verify the visibility of records while the *scan oldest* transaction needs to undo updates.

Evaluation

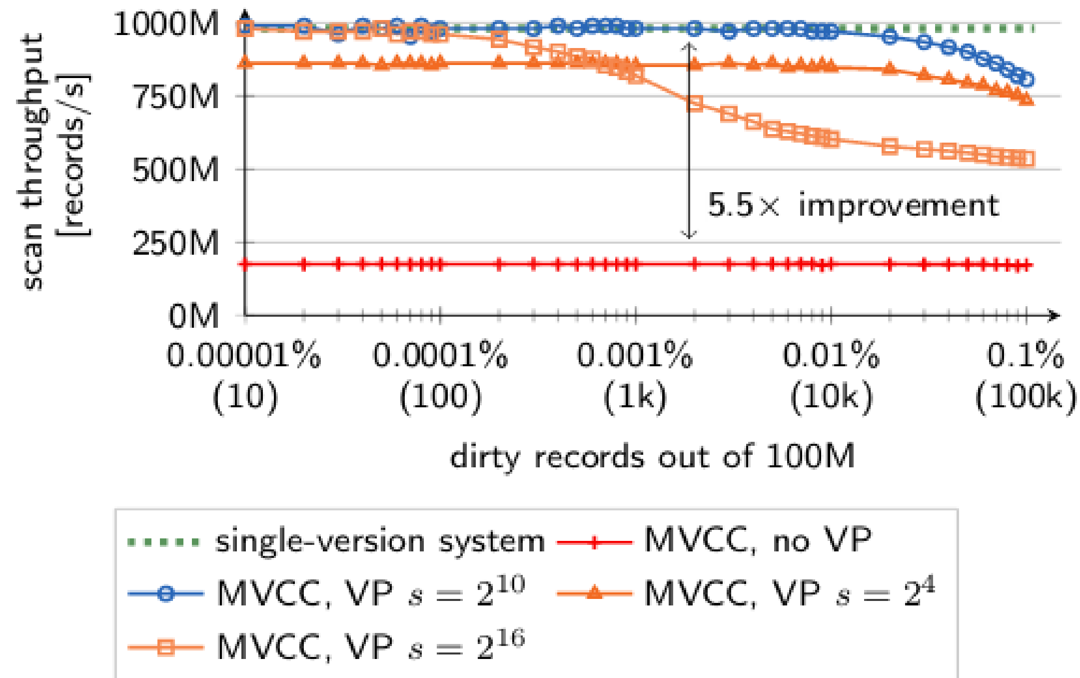


Figure 8: Effect of *VersionedPositions* (VP) synopses per s records on scan performance

Evaluation

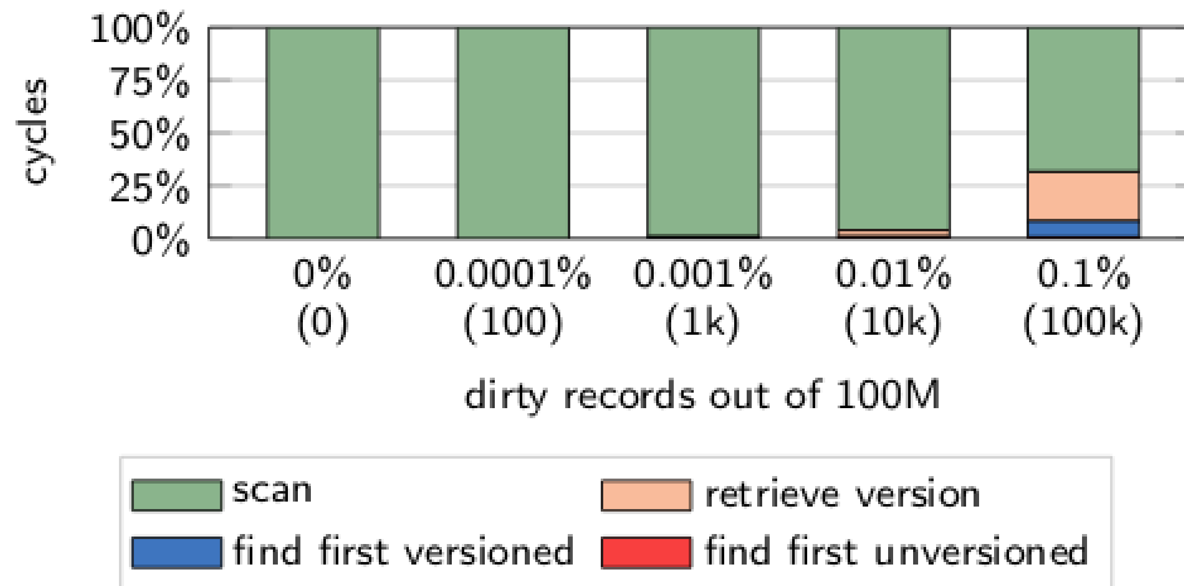


Figure 9: Cycle breakdowns of *scan-oldest* transactions that need to undo 4 updates per dirty record

Conclusion

- An MVCC main-memory database system implementation that provides Snapshot Isolation and full serializability with performance comparable to a single version main memory database system.
- Serializability validation technique that is based on precision locking which does not require to depend on the whole read set of a transaction.

Possible future work

- Can we handle Write-Write conflicts in another way (than just aborting) since we can support for multi versions?
- Extending to Disk instead of focussing on main memory since the main data structures are not necessarily bound to main memory. And also this will let the user better exploit the feature of long running readonly transaction without being blocked by other concurrent transactions.
- Improve the protocol so that it does not depend on an unbounded timestamp counter