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

Thomas Neumann Tobias Mühlbauer Alfons Kemper

Technische Universität München
{neumann, muehlbau, kemper}@in.tum.de

ABSTRACT

Multi-Version Concurrency Control (MVCC) is a widely employed concurrency control mechanism, as it allows for execution modes where readers never block writers. However, most systems implement only snapshot isolation (SI) instead of full serializability. Adding serializability guarantees to existing SI implementations tends to be *prohibitively expensive*.

We present a novel MVCC implementation for main-memory database systems that has very little overhead compared to serial execution with single-version concurrency control, even when maintaining serializability guarantees. Updating data in-place and storing versions as before-image deltas in undo buffers not only allows us to retain the high scan performance of single-version systems but also forms the basis of our cheap and fine-grained serializability validation mechanism. The novel idea is based on an adaptation of precision locking and verifies that the (extensional) writes of recently committed transactions do not intersect with the (intensional) read predicate space of a committing transaction. We experimentally show that our MVCC model allows very fast processing of transactions with point accesses *as well as* read-heavy transactions and that there is little need to prefer SI over full serializability any longer.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Multi-Version Concurrency Control; MVCC; Serializability

1. INTRODUCTION

Transaction isolation is one of the most fundamental features offered by a database management system (DBMS). It provides the user with the illusion of being alone in the database system, even in the presence of multiple concurrent users, which greatly simplifies application development. In the background, the DBMS ensures that the resulting concurrent access patterns are safe, ideally by being serializable.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749436.

Serializability is a great concept, but it is hard to implement efficiently. A classical way to ensure serializability is to rely on a variant of *Two-Phase Locking* (2PL) [42]. Using 2PL, the DBMS maintains read and write locks to ensure that conflicting transactions are executed in a well-defined order, which results in serializable execution schedules. Locking, however, has several major disadvantages: First, readers and writers block each other. Second, most transactions are read-only [33] and therefore harmless from a transaction-ordering perspective. Using a locking-based isolation mechanism, no update transaction is allowed to change a data object that has been read by a potentially long-running read transaction and thus has to wait until the read transaction finishes. This severely limits the degree of concurrency in the system.

Multi-Version Concurrency Control (MVCC) [42, 3, 28] offers an elegant solution to this problem. Instead of updating data objects in-place, each update creates a new version of that data object, such that concurrent readers can still see the old version while the update transaction proceeds concurrently. As a consequence, read-only transactions never have to wait, and in fact do not have to use locking at all. This is an extremely desirable property and the reason why many DBMSs implement MVCC, e.g., Oracle, Microsoft SQL Server [8, 23], SAP HANA [10, 37], and PostgreSQL [34]. However, most systems that use MVCC do not guarantee serializability, but the weaker isolation level *Snapshot Isolation* (SI). Under SI, every transaction sees the database in a certain state (typically the last committed state at the beginning of the transaction) and the DBMS ensures that two concurrent transactions do not update the same data object. Although SI offers fairly good isolation, some non-serializable schedules are still allowed [1, 2]. This is often reluctantly accepted because making SI serializable tends to be *prohibitively expensive* [7]. In particular, the known solutions require keeping track of the entire read set of every transaction, which creates a huge overhead for read-heavy (e.g., analytical) workloads. Still, it is desirable to detect serializability conflicts as they can lead to silent data corruption, which in turn can cause hard-to-detect bugs.

In this paper we introduce a novel way to implement MVCC that is very fast and efficient, both for SI and for full serializability. Our SI implementation is admittedly more carefully engineered than totally new, as MVCC is a well understood approach that recently received renewed interest in the context of main-memory DBMSs [23]. Careful engineering, however, matters as the performance of version maintenance greatly affects transaction *and* query processing. It

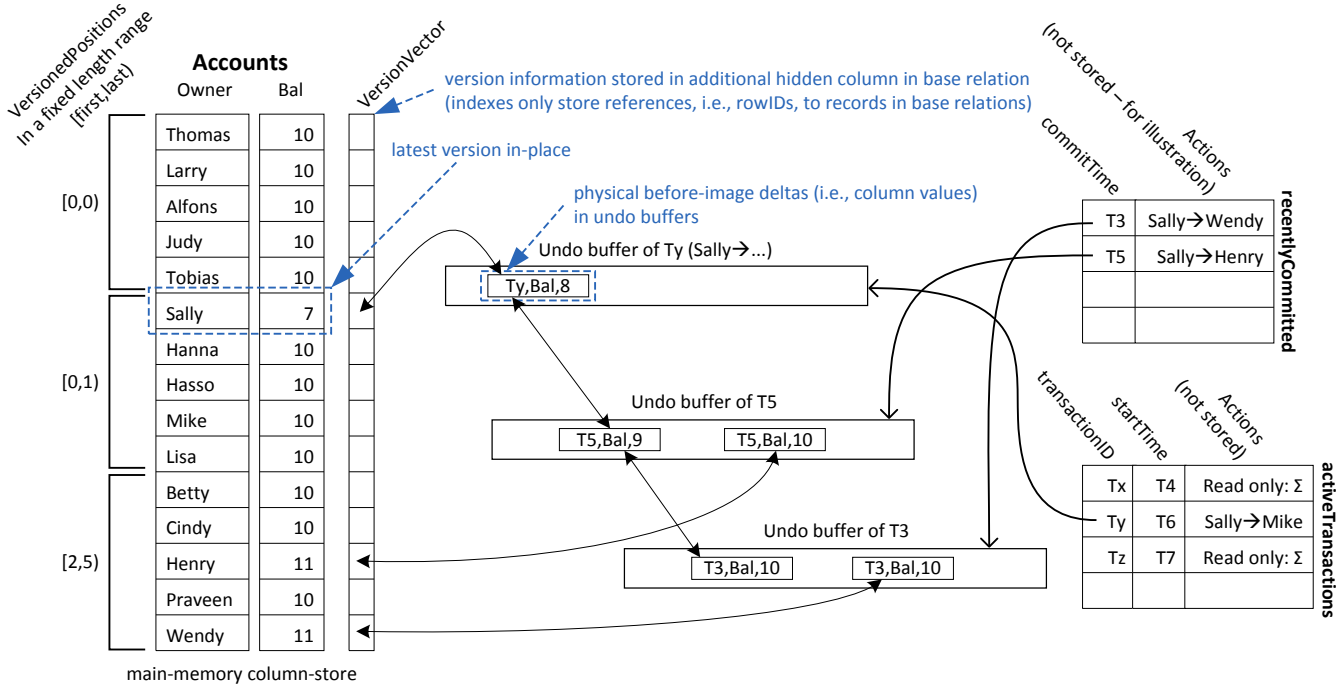


Figure 1: Multi-version concurrency control example: transferring $1 between Accounts (from → to) and summing all Balances (Σ)

is also the basis of our cheap serializability check, which exploits the structure of our versioning information. We further retain the very high scan performance of single-version systems using synopses of positions of versioned records in order to efficiently support analytical transactions.

In particular, the main contributions of this paper are:

1. A novel MVCC implementation that is integrated into our high-performance hybrid OLTP and OLAP main-memory datbase system HyPer [21]. Our MVCC model creates very little overhead for both transactional *and* analytical workloads and thereby enables very fast and efficient logical transaction isolation for hybrid systems that support these workloads simultaneously.

2. Based upon that, a novel approach to guarantee serializability for snapshot isolation (SI) that is both precise and cheap in terms of additional space consumption and validation time. Our approach is based on an adaptation of *Precision Locking* [42] and does not require explicit read locks, but still allows for more concurrency than 2PL.

3. A synopses-based approach (*VersionedPositions*) to retain the high scan performance of single-version systems for read-heavy and analytical transactions, which are common in today's workloads [33].

4. Extensive experiments that demonstrate the high performance and trade-offs of our MVCC implementation.

Our novel MVCC implementation is integrated into our HyPer main-memory DBMS [21], which supports SQL-92 query and ACID-compliant transaction processing (defined in a PL/SQL-like scripting language [20]). For queries and transactions, HyPer generates LLVM code that is then just-in-time compiled to optimized machine code [31]. In the past, HyPer relied on single-version concurrency control and thus did not efficiently support interactive and *sliced* transactions, i.e., transactions that are decomposed into multiple tasks such as stored procedure calls or individual SQL statements. Due to application roundtrip latencies and other factors, it is desirable to interleave the execution of these tasks. Our novel MVCC model enables this logical concurrency with excellent performance, even when maintaining serializability guarantees.

2. MVCC IMPLEMENTATION

We explain our MVCC model and its implementation initially by way of an example. The formalism of our serializability theory and proofs are then given in Section 3. Figure 1 illustrates the version maintenance using a traditional banking example. For simplicity, the database consists of a single *Accounts* table that contains just two attributes, *Owner* and *Bal*ance. In order to retain maximum scan performance we refrain from creating new versions in newly allocated areas as in Hekaton [8, 23]; instead we *update in-place* and maintain the backward delta between the updated (yet uncommitted) and the replaced version in the undo buffer of the updating transaction. Updating data in-place retains the contiguity of the data vectors that is essential for high scan performance. In contrast to positional delta trees (PDTs) [15], which were designed to allow more efficient updates in column stores, we refrain from using complex data structures for the deltas to allow for a high concurrent transactional throughput.

Upon committing a transaction, the newly generated version deltas have to be re-timestamped to determine their validity interval. Clustering all version deltas of a transaction in its undo buffer expedites this commit processing tremendously. Furthermore, using the undo buffers for ver-

sion maintenance, our MVCC model incurs almost no storage overhead as we need to maintain the version deltas (i.e., the before-images of the changes) during transaction processing anyway for transactional rollbacks. The only difference is that the undo buffers are (possibly) maintained for a slightly longer duration, i.e., for as long as an active transaction may still need to access the versions contained in the undo buffer. Thus, the *VersionVector* shown in Figure 1 anchors a chain of version reconstruction deltas (i.e., column values) in "newest-to-oldest" direction, possibly spanning across undo buffers of different transactions. Even for our column store backend, there is a single *VersionVector* entry per record for the version chain, so the version chain in general connects before-images of different columns of one record. Actually, for garbage collection this chain is maintained bidirectionally, as illustrated for Sally's *Bal*-versions.

2.1 Version Maintenance

Only a tiny fraction of the database will be versioned, as we continuously garbage collect versions that are no longer needed. A version (reconstruction delta) becomes obsolete if all active transactions have started after this delta was timestamped. The *VersionVector* contains *null* whenever the corresponding record is unversioned and a pointer to the most recently replaced version in an undo buffer otherwise.

For our illustrative example only two transaction types are considered: *transfer* transactions are marked as "from → to" and transfer $1 *from* one account *to* another by first subtracting 1 from one account's *Bal* and then adding 1 to the other account's *Bal*. For brevity we omit the discussion of object deletions and creations in the example. Initially, all *Bal*ances were set to 10. The read-only transactions denoted Σ sum all *Bal*ances and — in our "closed world" example — should always compute $150, no matter under what *startTime*-stamp they operate.

All new transactions entering the system are associated with two timestamps: *transactionID* and *startTime*-stamps. Upon commit, update transactions receive a third timestamp, the *commitTime*-stamp that determines their serialization order. Initially all transactions are assigned identifiers that are higher than any *startTime*-stamp of any transaction. We generate *startTime*-stamps from 0 upwards and *transactionID*s from 2^{63} upwards to guarantee that *transactionID*s are all higher than the *startTime*s. Update transactions modify data *in-place*. However, they retain the old version of the data in their undo buffer. This old version serves two purposes: (1) it is needed as a before-image in case the transaction is rolled back (undone) and (2) it serves as a committed version that was valid up to now. This most recently replaced version is inserted in front of the (possibly empty) version chain starting at the *VersionVector*. While the updater is still running, the newly created version is marked with its *transactionID*, whereby the uncommitted version is only accessible by the update transaction itself (as checked in the second condition of the version access predicate, cf., Section 2.2). At commit time an update transaction receives a *commitTime*-stamp with which its version deltas (undo logs) are marked as being irrelevant for transactions that start from "now" on. This *commitTime*-stamp is taken from the same sequence counter that generates the *startTime*-stamps. In our example, the first update transaction that committed at timestamp T_3 (Sally → Wendy) created in its undo buffer the version deltas timestamped T_3 for Sally's and Wendy's balances, respectively. The timestamp indicates that these version deltas have to be applied for transactions whose *startTime* is below T_3 and that the successor version is valid from there on for transactions starting after T_3. In our example, at *startTime* T_4 a reader transaction with *transactionID* T_x entered the system and is still active. It will read Sally's *Bal*ance at reconstructed value 9, Henry's at reconstructed value 10, and Wendy's at value 11. Another update transaction (Sally → Henry) committed at timestamp T_5 and correspondingly marked the version deltas it created with the validity timestamp T_5. Again, the versions belonging to Sally's and Wendy's balances that were valid just before T_5's update are maintained as before images in the undo buffer of T_5. Note that a reconstructed version is valid *from* its predecessor's timestamp *until* its own timestamp. Sally's *Bal*ance version reconstructed with T_5's undo buffer is thus valid from timestamp T_3 until timestamp T_5. If a version delta has no predecessor (indicated by a null pointer) such as Henry's balance version in T_5's undo buffer its validity is determined as from virtual timestamp "0" until timestamp T_5. Any read access of a transaction with *startTime* below T_5 applies this version delta and any read access with a *startTime* above or equal to T_5 ignores it and thus reads the in-place version in the *Accounts* table.

As said before, the deltas of not yet committed versions receive a temporary timestamp that exceeds any "real" timestamp of a committed transaction. This is exemplified for the update transaction (Sally → Henry) that is assigned the *transactionID* timestamp T_y of the updater. This temporary, very large timestamp is initially assigned to Sally's *Bal*ance version delta in T_y's undo buffer. Any read access, except for those of transaction T_y, with a *startTime*-stamp above T_5 (and obviously below T_y) apply this version delta to obtain value 8. The uncommitted in-place version of Sally's balance with value 7 is only visible to T_y.

2.2 Version Access

To access its visible version of a record, a transaction T first reads the in-place record (e.g., from the column- or row-store) and then undoes all version changes along the (possibly empty) version chain of undo buffer entries — by overwriting updated attributes in the copied record with the before-images from the undo buffers — up to the first version v, for which the following condition holds (*pred* points to the predecessor; *TS* denotes the associated timestamp):

$$v.pred = null \vee v.pred.TS = T \vee v.pred.TS < T.startTime$$

The first condition holds if there is no older version available because it never existed or it was (safely) garbage collected in the meantime. The second condition allows a transaction to access its own updates (remember that the initial *transactionID* timestamps assigned to an active transaction are very high numbers exceeding any start time of a transaction). The third condition allows reading a version that was valid at the start time of the transaction. Once the termination condition is satisfied, the visible version has been re-materialized by "having undone" all changes that have occurred in the meantime. Note that, as shown in Section 4, version "reconstruction" is actually cheap, as we store the physical before-image deltas and thus do not have to inversely apply functions on the in-place after-image.

Traversing the version chain guarantees that all reads are performed in the state that existed at the start of the trans-

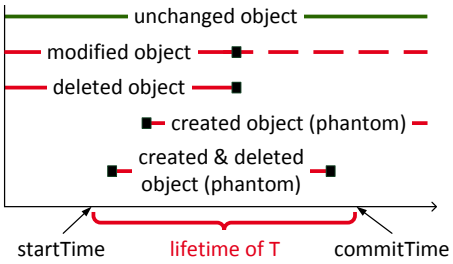


Figure 2: Modifications/deletions/creations of data objects relative to the lifetime of transaction T

action. This is sufficient for serializability of read-only transactions. However, for update transactions we need a validation phase that (conceptually) verifies that its entire read set did not change during the execution of the transaction. In previous approaches, this task is inherently complex as the read set can be very large, especially for main-memory database systems that tend to rely on full-table scans much more frequently than traditional disk-based applications [33]. Fortunately, we found a way to limit this validation to the objects that were actually changed and are still available in the undo buffers.

2.3 Serializability Validation

We deliberately avoid write-write conflicts in our MVCC model, as they may lead to cascading rollbacks. If another transaction tries to update an uncommitted data object (as indicated by the large *transactionID* timestamp in its predecessor version), it is aborted and restarted. Therefore, the first *VersionVector* pointer always leads to an undo buffer that contains a committed version — except for unversioned records where the pointer is null. If the same transaction modifies the same data object multiple times, there is an internal chain of pointers within the same undo buffer that eventually leads to the committed version.

In order to retain a scalable lock-free system we rely on optimistic execution [22] in our MVCC model. To guarantee serializability, we thus need a validation phase at the end of a transaction. We have to ensure that all reads during transaction processing could have been (logically) at the very end of the transaction without any observable change (as shown for the object on the top of Figure 2). In terms of this figure, we will detect the four (lower) transitions: modification, deletion, creation, and creation & deletion of an object that is "really" relevant for the transaction T. For this purpose, transactions draw a *commitTime*-stamp from the counter that is also "giving out" the *startTime*-stamps. The newly drawn number determines the serialization order of the transaction. Only updates that were committed during T's lifetime, i.e., in between the *startTime* and the *commitTime*, are potentially relevant for the validation. In terms of Figure 2, all events except for the top-most may lead to an abort, but *only* if these modified/deleted/created objects *really* intersect with T's read *predicate space*.

In previous approaches for serializability validation, such as in Microsoft's Hekaton [8, 23] and PostgreSQL [34], the entire *read set* of a transaction needs to be tracked (e.g., by using SIREAD locks as in PostgreSQL) and needs to be re-checked at the end of the transaction — by redoing all the read accesses. This is prohibitively expensive for large read sets that are very typical for scan-heavy main-memory

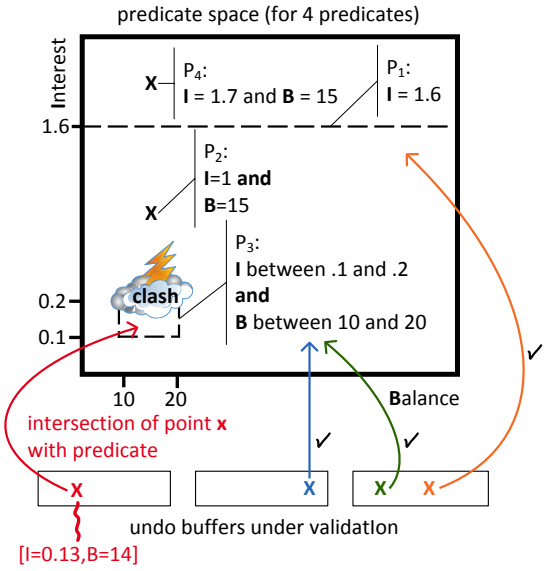


Figure 3: Checking data points in the undo buffers against the predicate space of a transaction

database applications [33], including analytical transactions. Here, our novel idea of using the undo-buffers for validation comes into play. Thereby, we limit the validation to the number of recently changed and committed data objects, no matter how large the read set of the transaction was. For this purpose, we adapt an old (and largely "forgotten") technique called *Precision Locking* [17] that eliminates the inherent satisfiability test problem of predicate locking. Our variation of precision locking tests discrete writes (updates, deletions, and insertions of records) of recently committed transactions against predicate-oriented reads of the transaction that is being validated. Thus, a validation fails if such an extensional write intersects with the intensional reads of the transaction under validation [42]. The validation is illustrated in Figure 3, where we assume that transaction T has read objects under the four different predicates P_1, P_2, P_3, and P_4, which form T's predicate space. We need to validate the three undo buffers at the bottom and validate that their objects (i.e., data points) do not intersect with T's predicates. This is done by evaluating the predicates for those objects. If the predicates do not match, then there is no intersection and the validation passes, otherwise, there is a conflict. This object-by-predicate based validation eliminates the undecidability problem inherent in other approaches that require predicate-by-predicate validation.

In order to find the extensional writes of other transactions that committed during the lifetime of a transaction T, we maintain a list of *recentlyCommitted* transactions, which contains pointers to the corresponding undo buffers (cf., Figure 1). We start our validation with the undo buffers of the oldest transaction that committed after T's *startTime* and traverse to the youngest one (at the bottom of the list). Each of the undo buffers is examined as follows: For each newly created version, we check whether it satisfies any of T's selection predicates. If this is the case, T's read set is inconsistent because of the detected phantom and it has to be aborted. For a deletion, we check whether or not the deleted object belonged to T's read set. If so, we have to abort T. For a modification (update) we have to check both,

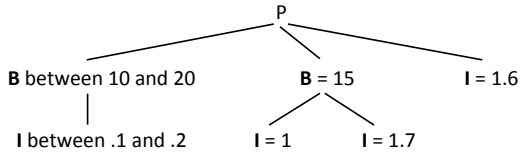


Figure 4: Predicate Tree (PT) for the predicate space of Figure 3

the before image as well as the after image. If either intersects with T's predicate space we abort T. This situation is shown in Figure 3, where the data point x of the left-most undo buffer satisfies predicate P_3, meaning that it intersects with T's predicate space.

After successful validation, a transaction T is committed by first writing its commit into the redo-log (which is required for durability). Thereafter, all of T's *transactionID* timestamps are changed to its newly assigned *commitTime*-stamp. Due to our version maintenance in the undo buffers, all these changes are local and therefore very cheap. In case of an abort due to a failed validation, the usual undo-rollback takes place, which also removes the version delta from the version chain. Note that the serializability validation in our MVCC model can be performed in parallel by several transactions whose serialization order has been determined by drawing the *commitTime*-stamps.

2.3.1 Predicate Logging

Instead of the read set, we log the predicates during the execution of a transaction for our serializability validation. Note that, in contrast to Hekaton [23], HyPer not only allows to access records through an index, but also through a base table scan. We log predicates of both access patterns in our implementation. Predicates of a base table access are expressed as restrictions on one or more attributes of the table. We log these restrictions in our predicate log on a per-relation basis. Index accesses are treated similarly by logging the point and range lookups on the index.

Index nested loop joins are treated differently. In this case, we log all values that we read from the index as predicates. As we potentially read many values from the index, we subsequently coarsen these values to ranges and store these ranges as predicates in the predicate log instead. Other join types are not treated this way. These joins are preceded by (potentially restricted) base table accesses.

2.3.2 Implementation Details

From an implementation perspective, a transaction logs its data accesses as read predicates on a per-relation basis in a designated predicate log. We always use 64 bit integer comparison summaries per attribute to allow for efficient predicate checks based on cheap integer operations and to keep the size of the predicate log small. Variable-length data objects such as strings are hashed to 64 bit summaries.

Traditional serializable MVCC models detect conflicts at the granularity of records (e.g., by "locking" the record). In our implementation we log the comparison summaries for restricted attributes (predicates), which is sufficient to detect serializability conflicts at the record-level (SR-RL). However, sometimes a record is too coarse. If the sets of read and written attributes of transactions do not overlap, a false positive conflict could be detected. To eliminate these false positives, which would lead to false aborts, we also implemented a way to check for serializability conflicts at the granularity of attributes (SR-AL): in addition to the restricted attributes we further log which attributes are accessed, i.e., read, without a restriction. During validation we then know which attributes were accessed and can thus skip the validation of versions that modified attributes that were not accessed. The evaluation in Section 4.4 shows that serializability checks at the attribute-level (SR-AL) reduce the number of false positives compared to serializability checks at the record-level (SR-RL) while barely increasing the overhead of predicate logging and validation.

Serializability validation works as follows: At the beginning of the validation of a committing transaction, a *Predicate Tree* (PT) is built on a per-relation basis from the predicate log. PTs are directed trees with a root node P. The PT for the predicate space in Figure 3 is exemplified in Figure 4. The nodes of a PT are single-attribute predicates, e.g., $B = 15$. Edges connect nodes with a logical AND, e.g., $B = 15 \wedge I = 1$. The logical OR of all paths in the tree then defines the predicate space. Nodes for the same predicate that share the same root are merged together, e.g., for $B = 15$ in Figure 4. During validation, data objects are checked whether they satisfy the PT, i.e., whether there is a path in the PT that the data object satisfies.

2.4 Garbage Collection

Garbage collection of undo buffers is continuously performed whenever a transaction commits. After each commit, our MVCC implementation determines the now oldest visible *transactionID*, i.e., the oldest timestamp of a transaction that has updates that are visible by at least one active transaction. Then, all committed transactions whose *transactionID* is older than that timestamp are removed from the list of recently committed transactions, the references to their undo buffers are atomically removed from the version lists, and the undo buffers themselves are marked with a tombstone. Note that it is not possible to immediately reuse the memory of a marked undo buffer, as other transactions can still have references to this buffer; although the buffer is definitely not relevant for these transactions, it may still be needed to terminate version chain traversals. It is safe to reuse a marked undo buffer as soon as the oldest active transaction has started after the undo buffer had been marked. As in our system, this can be implemented with very little overhead, e.g., by maintaining high water marks.

2.5 Handling of Index Structures

Unlike other MVCC implementations in Hekaton [8, 23] and PostgreSQL [34], our MVCC implementation does not use (predicate) locks and timestamps to mark read and modified keys in indexes. To guarantee SI and serializability, our implementation proceeds as follows: If an update updates only non-indexed attributes, updates are performed as usual. If an update updates an indexed attribute, the record is deleted and re-inserted into the relation and both, the deleted and the re-inserted record, are stored in the index. Thus, indexes retain references to all records that are visible by any active transaction. Just like undo buffers, indexes are cleaned up during garbage collection.

We ensure the uniqueness of primary keys by aborting a transaction that inserts a primary key that exists either (i) in the snapshot that is visible to the transaction, (ii) in the last committed version of the key's record, or (iii) uncommitted

as an insert in an undo buffer. Note that these are the only cases that need to be checked, as updates of indexed attributes are performed as a deletion and insertion.

For foreign key constraints we need to detect the case when an active transaction deletes a primary key and a concurrent transaction inserts a foreign key reference to that key. In this case, we abort the inserting transaction as it detects the (possibly uncommitted) delete. The inserting transaction is aborted pro-actively, even if the delete is uncommited, because transactions usually commit and only rarely abort.

2.6 Efficient Scanning

Main-memory database systems for real-time business intelligence, i.e., systems that efficiently handle transactional and analytical workloads in the same database, rely heavily on "clock-rate" scan performance [43, 26]. Therefore, testing each data object individually (using a branch statement) whether or not it is versioned would severely jeopardize performance. Our MVCC implementation in HyPer uses LLVM code generation and just-in-time compilation [31] to generate efficient scan code at runtime. To mitigate the negative performance implications of repeated version branches, the generated code uses synopses of versioned record positions to determine ranges that can be scanned at maximum speed.

The generated scan code proceeds under consideration of these synopses, called *VersionedPositions*, shown on the left-hand side of Figure 1. These synopses maintain the position of the first and the last versioned record for a fixed range of records (e.g., 1024) in a 32 bit integer, where the position of the first versioned record is stored in the high 16 bit and the position of the last versioned record is stored in the low 16 bit, respectively. Maintenance of *VersionedPositions* is very cheap as insertions and deletions of positions require only a few logical operations (cf., evaluation in Section 4). Further, deletions are handled fuzzily and *VersionedPositions* are corrected during the next scan where the necessary operations can be hidden behind memory accesses.

Note that the versions are continuously garbage collected; therefore, most ranges do not contain any versions at all, which is denoted by an empty interval $[x, x)$ (i.e., the lower and upper bound of the half-open interval are identical). E.g., this is the case for the synopsis for the first 5 records in Figure 1. Using the *VersionedPositions* synopses, adjacent unversioned records are accumulated to one range where version checking is not necessary. In this range, the scan code proceeds at maximum speed without any branches for version checks. For modified records, the *VersionVector* is consulted and the version of the record that is visible to the transaction is reconstructed (cf., Section 2.2). Again, a range for modified records is determined in advance by scanning the *VersionVector* for set version pointers to avoid repeated testing whether a record is versioned.

Looking at Figure 1, we observe that for strides $0 \ldots 4$ and $6 \ldots 10$ the loop on the unversioned records scans the *Bal*ance vector at maximum speed without having to check if the records are versioned. Given the fact that the strides in between two versioned objects are in the order of millions in a practical setting, the scan performance penalty incurred by our MVCC is marginal (as evaluated in Section 4.1). Determining the ranges of versioned objects further ensures that the *VersionedPositions* synopses are not consulted in hotspot areas where all records are modified.

2.7 Synchronization of Data Structures

In this work, we focus on providing an efficient and elegant mechanism to allow for logical concurrency of transactions, which is required to support interactive and *sliced transactions*, i.e., transactions that are decomposed into multiple tasks such as stored procedure calls or individual SQL statements. Due to application roundtrips and other factors, it is desirable to interleave the execution of these decomposed tasks, and our serializable MVCC model enables this logical concurrency. Thread-level concurrency is a largely orthogonal topic. We thus only briefly describe how our MVCC data structures can be synchronized and how transactional workloads can be processed in multiple threads.

To guarantee thread-safe synchronization in our implementation, we obtain short-term latches on the MVCC data structures for the duration of one task (a transaction typically consists of multiple such calls). The commit processing of writing transactions is done in a short exclusive critical section by first drawing the *commitTime*-stamp, validating the transaction, and inserting commit records into the redo log. Updating the validity timestamps in the undo buffers can be carried out unsynchronized thereafter by using atomic operations. Our lock-free garbage collection that continuously reclaims undo log buffers has been detailed in Section 2.4. Currently we use conventional latching-based synchronization of index structures, but could adapt to lock-free structures like the Bw-Tree [25] in the future.

In future work, we want to optimize the thread-parallelization of our implementation further. We currently still rely on classical short-term latches to avoid race conditions between concurrent threads. These latches can largely be avoided by using hardware transactional memory (HTM) [24] during version retrieval, as it can protect a reader from the unlikely event of a concurrent (i.e., racy) updater. Note that such a conflict is very unlikely as it has to happen in a time frame of a few CPU cycles. A combination of our MVCC model and HTM is very promising and in initial experiments indeed outperforms our current implementation.

3. THEORY

In this section, we analyze our serializable MVCC model more formally in the context of serializability theory.

3.1 Discussion of our MVCC Model

In order to formalize our MVCC scheme we need to introduce some notation that is illustrated in Figure 5. On the top of the figure a schedule consisting of four transactions is shown. These transactions start at times S_1, S_2, S_3, and S_4, respectively. As they access different versions of the data objects, we need a version ordering/numbering scheme in order to differentiate their reads and their version creations. This is shown for the same four-transaction-schedule at the bottom of the figure.

Transactions are allowed to proceed concurrently. They are, however, committed serially. An update transaction draws its *commitTime*-stamp from the same counter that generates the *startTime*-stamps. The *commitTime*-stamps determine the commit order and, as we will see, they also determine the serialization order of the transactions. Read-only transactions do not need to draw a commit order timestamp; they reuse their *startTime*-stamp. Therefore, in our example the transaction that started at S_1 obtained the *commitTime*-stamp T_6, because the transaction that started at

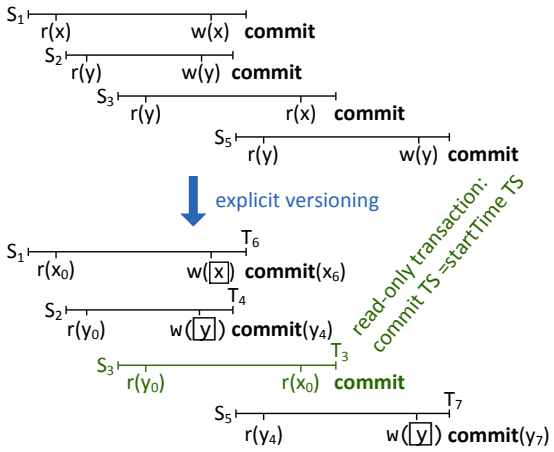


Figure 5: Example of the explicit versioning notation in our MVCC model

S_2 committed earlier at timestamp T_4. The read-only transaction that started at timestamp S_3 logically also commits at timestamp T_3.

Transactions read all the data in the version that was committed (i.e., created) most recently before their *startTime*-stamp. Versions are only committed at the end of a transaction and therefore receive the identifiers corresponding to the *commitTime*-stamps of the transaction that creates the version. The transaction schedule of Figure 5 creates the version chains $y_0 \rightarrow y_4 \rightarrow y_7$ and $x_0 \rightarrow x_6$. Note that versions are not themselves (densely) numbered because of our scheme of identifying versions with the *commitTime*-stamp of the creating transaction. As we will prove in Section 3.2, our MVCC model guarantees equivalence to a serial mono-version schedule in *commitTime*-stamp order. Therefore, the resulting schedule of Figure 5 is equivalent to the serial mono-version execution: $r_3(y)$, $r_3(x)$, c_3, $r_4(y)$, $w_4(y)$, c_4, $r_6(x)$, $w_6(x)$, c_6, $r_7(y)$, $w_7(y)$, c_7. Here all the operations are subscripted with the transaction's *commitTime*-stamp.

Local writing is denoted as $w(\boxed{x})$. Such a "dirty" data object is only visible to the transaction that wrote it. In our implementation (cf., Section 2.1), we use the very large transaction identifiers to make the dirty objects invisible to other transactions. In our formal model we do not need these identifiers. As we perform updates in-place, other transactions trying to (over-)write $\boxed{x}$ are aborted and restarted. Note that reading x is always possible, because a transaction's reads are directed to the version of x that was committed most recently before the transaction's *startTime*-stamp — with one exception: if a transaction updates an object x, i.e., $w(\boxed{x})$, it will subsequently read its own update, i.e., $r(\boxed{x})$. This is exemplified for transaction (S_1, T_2) on the upper left hand side of Figure 6(a). In our implementation this read-your-own-writes scheme is again realized by assigning very large transaction identifiers to dirty data versions.

Figure 6(a) further exemplifies a cycle of rw-dependencies, often also referred to as rw-antidependencies [11]. rw-antidependencies play a crucial role in non-serializable schedules that are compliant under SI. The first rw-antidependency involving $r(y_0)$ and $w(\boxed{y})$ in the figure could not have been detected immediately as the write of y in (S_4, T_7) happens after the read of y; the second rw-antidependency involving $r(x_2)$ and $w(\boxed{x})$ on the other hand could have been detected immediately, but in our MVCC model we opted to validate all reads at commit time. After all, the rw-antidependencies could have been resolved by an abort of (S_4, T_7) or by a commit of the reading transaction before T_7.

The benefits of MVCC are illustrated in Figure 6(b), where transaction (S_5, T_6) managed to "slip in front" of transaction (S_4, T_7) even though it read x after (S_4, T_7) wrote x. Obviously, with a single-version scheduler this degree of logical concurrency would not have been possible. The figure also illustrates the benefits of our MVCC scheme that keeps an arbitrary number of versions instead of only two as in [36]. The "long" read transaction (S_1, T_1) needs to access x_0 even though in the meantime the two newer versions x_3 and x_7 were created. Versions are only garbage collected after they are definitely no longer needed by other active transactions.

Our novel use of precision locking consisting of collecting the read predicates and validating recently committed versions against these predicates is illustrated in Figure 6(c). Here, transaction (S_2, T_5) reads x_0 with predicate P, denoted $r_P(x_0)$. When the transaction that started at S_2 tries to commit, it validates the before- and after-images of versions that were committed in the meantime. In particular, $P(x_0)$ is true and therefore leads to an abort and restart of the transaction. Likewise, phantoms and deletions are detected as exemplified for the insert $i(\boxed{o})$ and the delete $d(\boxed{u})$ of transaction (S_1, T_4). Neither the inserted object nor the deleted object are allowed to intersect with the predicates of concurrent transactions that commit after T_4.

3.2 Proof of Serializability Guarantee

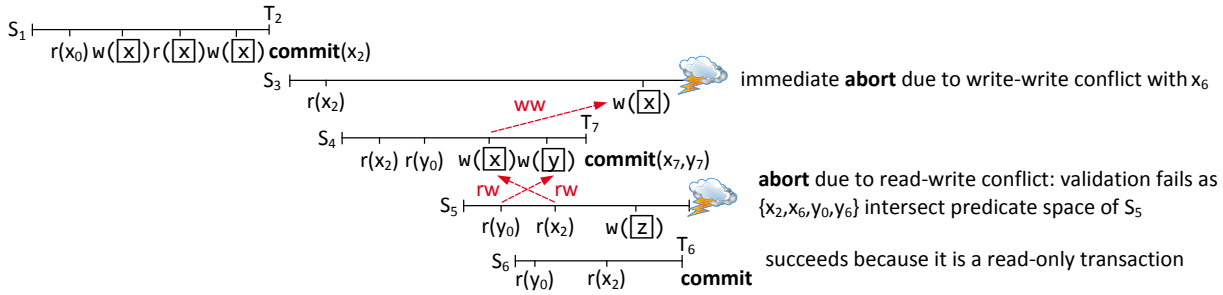
We will now prove that our MVCC scheme with predicate space validation guarantees that any execution is serializable in commit order.

Theorem. *The committed projection of any multi-version schedule H that adheres to our protocol is conflict equivalent to a serial mono-version schedule H' where the committed transactions are ordered according to their commitTime-stamps and the uncommitted transactions are removed.*

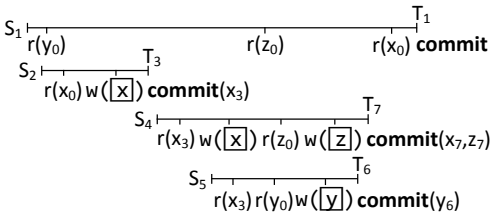
Proof. Due to the nature of the MVCC protocol, the effects of any uncommitted transaction can never be seen by any other successful transaction (reads will ignore the uncommitted writes, writes will either not see the uncommitted writes or lead to aborts). Therefore, it is sufficient to consider only committed transactions in this proof.

Basically, we will now show that all dependencies are in the direction of the order of their *commitTime*-stamps and thus any execution is serializable in commit order. Read-only transactions see a stable snapshot of the database at time S_b, and get assigned the same *commitTime*-stamp $T_b = S_b$, or, in other words, they behave as if they were executed at the point in time of their *commitTime*-stamp, which is the same as their *startTime*-stamp.

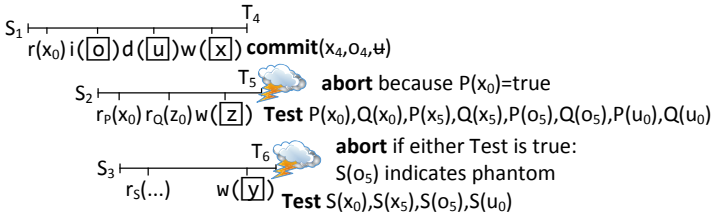
Update transactions are started at S_b, and get assigned a *commitTime*-stamp T_c with $T_c > S_b$. We will now prove by contradiction, that the transactions behave as if they were executed at the time point T_c. Assume T is an update-transaction from the committed projection of H (i.e., T has committed successfully), but T could not have been delayed to the point T_c. That is, T performed an operation o_1 that conflicts with another operation o_2 by a second transaction T' with $o_1 < o_2$ and T' committed during T's lifetime, i.e., within the time period $S_b \leq T'_c < T_c$. If T' committed after



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



(b) benefits of our MVCC model



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

Figure 6: Example schedules in our MVCC model

T, i.e., $T'_c > T_c$, we could delay T' (and thus o_2) until after T_c, thus we only have to consider the case $T'_c < T_c$.

There are four possible combinations for the operations o_1 and o_2. If both are reads, we can swap the order of both, which is a contradiction to our assumption that o_1 and o_2 are conflicting. If both are writes, T' would have aborted due to to our protocol of immediately aborting upon detecting *ww*-conflicts. Thus, there is a contradiction to the assumption that both, T and T', are committed transactions. If o_1 is a read and o_2 is a write, the update o_2 is already in the undo buffers when T commits, as $T'_c < T_c$ and the predicate P of the read of o_1 has been logged. The predicate validation at T_c then checks if o_1 is affected by o_2 by testing whether P is satisfied for either the before- or the after-image of o_2 (i.e., if the read should have seen the write), as illustrated in Figure 6(c). If not, that is a contradiction to the assumption that o_1 and o_2 are conflicting. If yes, that is a contradiction to the assumption that T has committed successfully as T would have been aborted when P was satisfied. If o_1 is a write and o_2 is a read, the read has ignored the effect of o_1 in the MVCC mechanism, as $T'_c > S_b$, which is a contradiction to the assumption that o_1 and o_2 are conflicting. The theorem follows. □

4. EVALUATION

In this section we evaluate our MVCC implementation in our HyPer main-memory database system [21] that supports SQL-92 queries and transactions (defined in a PL/SQL-like scripting language [20]) and provides ACID guarantees.

HyPer supports both, column- and a row-based storage of relations. Unless otherwise noted, we used the column-store backend, enabled continuous garbage collection, and stored the redo log in local memory. Redo log entries are generated in memory and log entries are submitted in small groups (group commit), which mitigates system call overheads and barely increases transaction latency. We evaluated HyPer with single-version concurrency control, our novel MVCC model, and a MVCC model similar to [23], which we mimiced by updating whole records and not using *VersionedPositions* synopses in our MVCC model. We further experimented with DBMS-X, a commercial main-memory DBMS with a MVCC implementation similar to [23]. DBMS-X was run in Windows 7 on our evaluation machine. Due to licensing agreements we can not disclose the name of DBMS-X.

The experiments were executed on a 2-socket Intel Xeon E5-2660v2 2.20 GHz (3 GHz maximum turbo) NUMA system with 256 GB DDR3 1866 MHz memory (128 GB per CPU) running Linux 3.13. Each CPU has 10 cores and a 25 MB shared L3 cache. Each core has a 32 KB L1-I and L1-D cache as well as a 256 KB L2 cache.

4.1 Scan Performance

Initially we demonstrate the high scan performance of our MVCC implementation. We implemented a benchmark similar to the SIBENCH [7] benchmark and our bank accounts example (cf., Figure 1). The benchmark operates on a single relation that contains integer (key, value) pairs. The workload consists of update transactions which modify a (key, value) pair by incrementing the value and a scan transaction that scans the relation to sum up the values.

Figure 7 shows the per-core performance of scan transactions on a relation with 100M (key, value) records. To demonstrate the effect of scanning versioned records, we disable the continuous garbage collection and perform updates before scanning the relations. We vary both, the number of dirty records and the number of versions per dirty record. Additionally, we distinguish two cases: (i) the scan transaction is started before the updates (*scan oldest*) and thus needs to undo the effects of the update transactions and (ii) the scan transaction is started after the updates (*scan newest*) and thus only needs to verify that the dirty records are visible to the scan transaction. For all cases, the results show that our MVCC implementation sustains the high scan throughput of our single-version concurrency control implementation for realistic numbers of dirty records; and even under high contention with multiple versions per record.

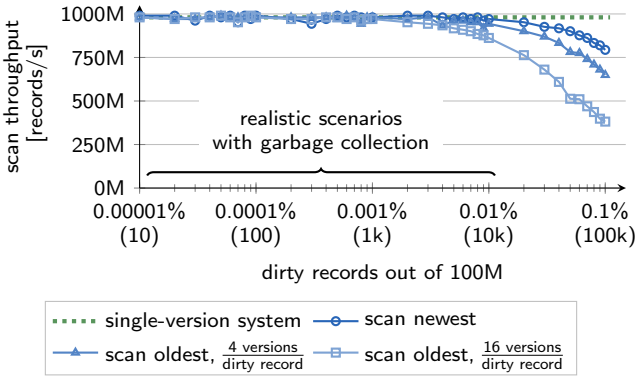


Figure 7: Scan performance with disabled garbage collection: the *scan newest* transaction only needs to verify the visibility of records while the *scan oldest* transaction needs to undo updates.

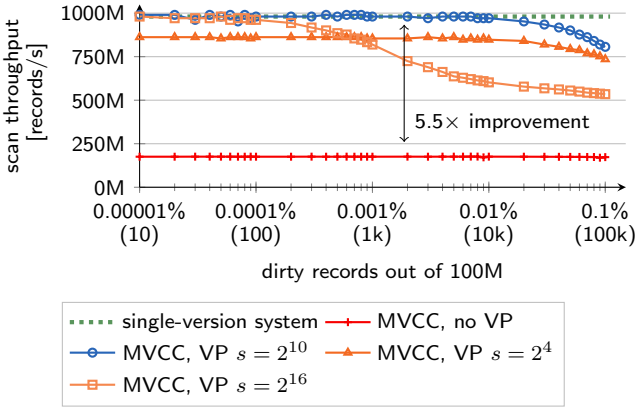


Figure 8: Effect of *VersionedPositions* (VP) synopses per s records on scan performance

To validate our assumptions for the number of dirty records and versions we consider Amazon.com as an example. 6.9 million copies of *Harry Potter and the Half-Blood Prince*, one of the best-selling books of all time, were sold within the first 24 hours in the United States. Even if we make the highly pessimistic assumptions that all books are sold through Amazon within 20 hours of that day and that Amazon operates only six warehouses, 16 copies of that book are sold per warehouse per second. Our experiment suggests that in order to measure a significant drop in scan performance there need to be hundreds of thousands of such best-selling items and a transaction that is open for a long period of time. Remember that in this case the long-running transaction can be aborted and restarted on a snapshot [29].

Figure 8 shows the performance effect of having *VersionedPositions* synopses (see Section 2.6) on scan performance. Our implementation maintains *VersionedPositions* per 1024 records. The experiment suggests that increasing or decreasing the number of records per *VersionedPositions* degrades scan performance. Compared to not using *VersionedPositions* at all, scan performance is improved by more than 5.5×. 1024 records seems to be a sweetspot where the size of the *VersionedPositions* vector is still reasonable and the synopses already encode meaningful ranges, i.e., ranges that include mostly modified records. A breakdown of CPU cycles in Figure 9 shows that our MVCC functions are very cheap for realistic numbers of versioned records. We measured 2.8 instructions per cycle (IPC) during the scans.

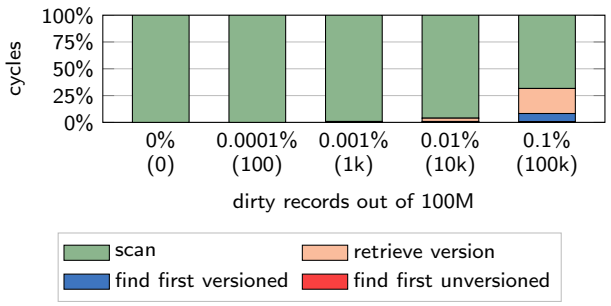


Figure 9: Cycle breakdowns of *scan-oldest* transactions that need to undo 4 updates per dirty record

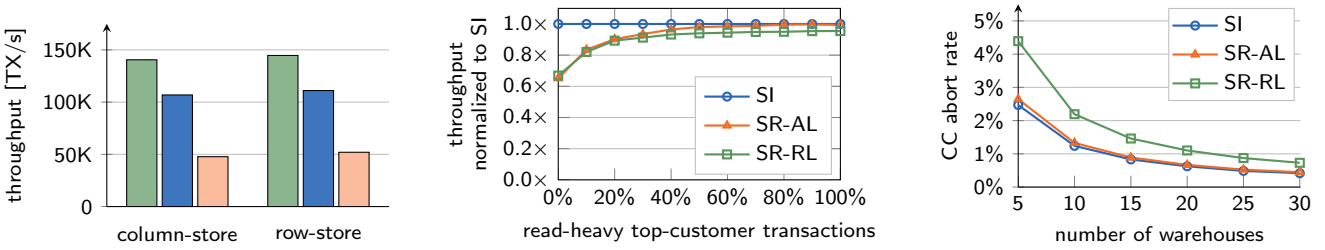
We further compared the scan performance of our MVCC implementation to DBMS-X. DBMS-X achieves a scan speed of 7.4M records/s with no dirty records and 2.5M records/s with 10k dirty records (> 100× slower than our MVCC implementation). Of course, we "misused" DBMS-X with its point-query-only-optimized model for large full-table scans, which would be necessary for analytical transactions. The Hekaton model is only optimized for point queries and performs all accesses through an index, which severely degrades performance for scans-based analytics.

4.2 Insert/Update/Delete Benchmarks

We also evaluated the per-core performance of insert, update, and "delete and insert" (delin) operations on a relation with 10 integer attributes and 100M records. As expected, compared to our single-version concurrency control implementation (5.9M inserts/s, 3.4M updates/s, 1.1M delins/s), performance with our MVCC implementation is slightly degraded due to visibility checks and the maintenance of the *VersionVector* and the *VersionedPositions* (4M inserts/s, 2M updates/s, 1M delins/s). The number of logically concurrent active transactions, however, has no performance impact. As the newest version is stored in-place and the version record of the previous version is inserted at the beginning of the version chain, performance of updates is also independent of the total number of versions.

4.3 TPC-C and TATP Results

TPC-C is a write-heavy benchmark and simulates the principal activities of an order-entry environment. Its workload mix consists of 8% read-only transactions and 92% write transactions. Some of the transactions in the TPC-C perform aggregations and reads with range predicates. Figure 10(a) shows the per-core performance of our MVCC implementation for the TPC-C benchmark with 5 warehouses and no think times. Compared to our single-version concurrency control implementation, our MVCC implementation costs around 20% of performance. Still, more than 100k transactions/s are processed. This is true for our column- and a row-based storage backends. We also compared these numbers to a 2PL implementation in HyPer and a MVCC model similar to [23]. 2PL is prohibitively expensive and achieves a ∼5× smaller throughput. The MVCC model of [23] achieves a throughput of around 50k transactions/s.

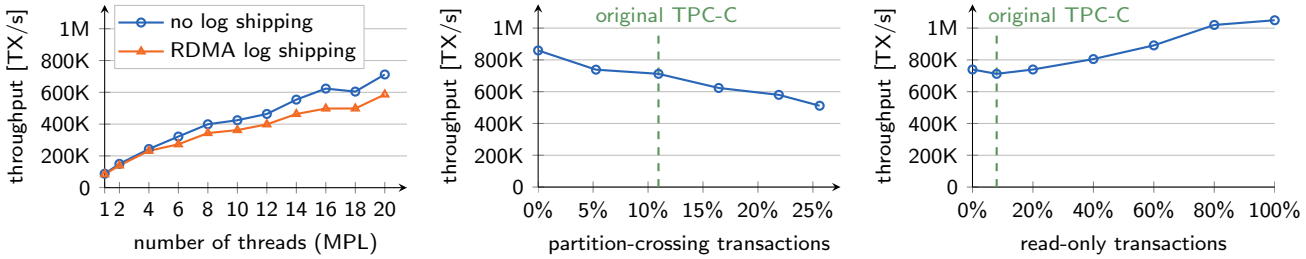


(a) HyPer with single-version concurrency control ( ), our MVCC model ( ), and a MVCC model that mimics the behavior of [23] by updating whole records and not using *VersionedPositions* ( )

(b) throughput of interleaved transactions under serializability with attribute- (SR-AL) and record-level predicate logs (SR-RL) relative to SI throughput for a varying percentage of read-heavy *top-customer* transactions in the workload mix

(c) concurrency control abort rate running interleaved transactions with a varying number of warehouses under snapshot isolation (SI) and serializability with attribute- (SR-AL) and record-level predicate logs (SR-RL)

Figure 10: Single-threaded TPC-C experiments with a default of 5 warehouses



(a) scalability

(b) 20 threads, varying contention

(c) 20 threads, impact of read-only TX

Figure 11: Multi-threaded TPC-C experiments with 20 warehouses in HyPer with our MVCC model

We further executed the TPC-C with multiple threads. Like in H-Store [19]/VoltDB, we partition the database according to warehouses. Partitions are assigned to threads and threads are pinned to cores similar to the DORA system [32]. These threads process transactions that primarily access data belonging to their partition. Unlike DORA, partition-crossing accesses, which, e.g., occur in 11% of the TPC-C transactions, are carried out by the primary thread to which the transaction was assigned. The scalability experiment (see Figure 11(a)) shows that our system scales near linearly up to 20 cores. Going beyond 20 cores might require the reduction of global synchronization like in the SILO system [41]. We further varied the contention on the partitions by varying the percentage of partition-crossing transactions as shown in Figure 11(b). Finally, as shown in Figure 11(c), we also measured the impact of read-only transactions and proportionally varied the percentage of the two read-only transactions in the workload mix.

Figure 11(a) also shows the scalability of HyPer when shipping the redo log using remote direct memory accesses (RDMA) over Infiniband. RDMA-based log shipping generates an overhead of 17% with 20 threads. Our evaluation system has a Mellanox ConnectX-3 Infiniband network adapter, which operates at 4×QDR. The maximum write bandwidth of our setup is 3.5 GB/s with a latency of 1.3 μs. This bandwidth is sufficient to ship the redo log entries: for 100k TPC-C transactions we generate 85 MB of redo log entries. In our setup, the receiving node can act as a high-availability failover but could also write the log to disk.

The Telecommunication Application Transaction Processing (TATP) benchmark simulates a typical telecommunications application. The workload mix consists of 80% read-only transactions and 20% write transactions. Thereby, the read transactions all perform point accesses and records are mostly updated as a whole. Thus, the TATP benchmark is a best-case scenario for the MVCC model of [23]. We ran the benchmark with a scale-factor of 1M subscribers. Compared to running the benchmark with single-version concurrency control (421,940 transactions/s), our MVCC implementation creates just a tiny overhead (407,564 transactions/s). As expected, the mimicked MVCC model of [23] also performs quite well in this benchmark, but still trails performance of our MVCC implementation by around 20% (340,715 transactions/s).

4.4 Serializability

To determine the cost of our serializability validation approach (cf., Section 2.3), we first measured the cost of predicate logging in isolation from predicate validation by running the TPC-C and TATP benchmarks each in a single serial stream. Without predicate logging, i.e., under SI, we achieve a TPC-C throughput of 112,610 transactions/s, with record-level predicate logging (SR-RL) 107,365 transactions/s, and with attribute-level predicate logging (SR-AL) 105,030 transactions/s. This means that there is a mere 5% overhead for SR-RL and a mere 7% overhead for SR-AL. For the TATP benchmark, we measured an overhead of only 1% for SR-RL and 2% for SR-AL. We also measured the predicate logging overhead for the mostly read-only TPC-H decision support benchmark, which resulted in an even smaller overhead. In terms of size, predicate logs generated by our implementation are quite small. Unlike other serializable MVCC implementations, we do not need to track the entire read set of a transaction. To illustrate the difference in size

imagine a *top-customer* transaction on the TPC-C schema that, for a specific warehouse and district, retrieves the customer with the highest account balance and a good credit rating (GC) and performs a small update:

```
select
  c_w_id, c_d_id, max(c_balance)
from
  customer
where
  c_credit = 'GC'
  and c_w_id = :w_id
  and c_d_id = :d_id
group by
  c_w_id, c_d_id
update ...
```

For such a query, serializable MVCC models that need to track the read set then have to either copy all read records or set a flag (e.g., the SIREAD lock in PostgreSQL [34]). If we assume that at least one byte per read record is needed for book-keeping, then these approaches need to track at least 3 KB of data (a district of a warehouse serves 3k customers in TPC-C). Our SR-AL on the other hand just stores the read attributes and the aggregate that has been read which is less than 100 bytes. This is 30× less than what traditional read set book-keeping consumes; and for true OLAP-style queries that read a lot of data, predicate logging saves even more. E.g., the read set of an analytical TPC-H query usually comprises millions of records and tracking the read set can easily consume multiple MBs of space.

To determine the cost of predicate validation, we again ran the TPC-C benchmark but this time interleaved transactions (by decomposing the transactions into smaller tasks) such that transactions are running logically concurrent, which makes predicate validation necessary. We further added the aforementioned *top-customer* transaction to the workload mix and varied its share in the mix from 0% to 100%. The results are shown in Figure 10(b). As TPC-C transactions are very short, it would be an option to skip the step of building the predicate tree and instead just apply all the predicates as-is and thus make predicate validation much faster. However, the predicate tree has much better asymptotic behavior, and is therefore much faster and more robust when transaction complexity grows. We therefore use it all the time instead of optimizing for very cheap transactions. And the figure also shows that with more read-only transactions in the workload mix (as it is the case in real-world workloads [33]), the overhead of serializability validation almost disappears. Building the predicate trees takes between 2 μs and 15 μs for the TPC-C transactions on our system; and between 4 μs and 24 μs for the analytical TPC-H queries (9.5 μs geometric mean). In comparison to traditional validation approaches that repeat all reads, our system has, as mentioned before, a much lower book-keeping overhead. A comparison of validation times by themselves is more complicated. Validation time in traditional approaches depends on the size of the read set of the committing transaction $|R|$ and how fast reads can be repeated (usually scan speed and index lookup speed); in our approach, it mostly depends on the size of the write set $|W|$ that has been committed during the runtime of the committing transaction. In our system, checking the predicates of a TPC-C transaction or a TPC-H query against a versioned record that has been reconstructed from undo buffers is a bit faster than an index lookup. In general, our approach thus favors workloads where $|R| \geq |W|$. In our opinion this is mostly the case, as modern workloads tend to be read-heavy [33] and the time that a transaction is active tends to be short (long-running transactions would be deferred to a "safe snapshot").

Finally, we evaluated the concurrency control abort rates, i.e., the aborts caused by concurrency control conflicts, of our MVCC implementation in HyPer. We again ran TPC-C with logically interleaved transactions and varied the number of TPC-C warehouses. As the TPC-C is largely partitionable by warehouse, the intuition is that concurrency control conflicts are reduced with an increasing number of warehouses. The results are shown in Figure 10(c). We acknowledge that TPC-C does not show anomalies under SI [11], but of course the database system does not know this, and this benchmark therefore tests for false positive aborts. The aborts under SI are "real" conflicts, i.e., two transaction try to modify the same data item concurrently. Serializability validation with SR-AL creates almost no false positive aborts. The only false positive aborts stem from the minimum (*min*) aggregation in *delivery*, as it sometimes conflicts with concurrent inserts. Predicate logging of minimum and maximum aggregates is currently not implemented in our system but can easily be added in the future. SR-RL creates more false positives than SR-AL, because reads are not only checked against updated attributes but rather any change to a record is considered a conflict, even though the updated attribute might not even have been read by the original transaction.

5. RELATED WORK

Transaction isolation and concurrency control are among the most fundamental features of a database management system. Hence, several excellent books and survey papers have been written on this topic in the past [42, 3, 2, 38]. In the following we further highlight three categories of work that are particularly related to this paper, most notably multi-version concurrency control and serializability.

5.1 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) [42, 3, 28] is a popular concurrency control mechanism that has been implemented in various database systems because of the desirable property that readers never block writers. Among these DBMSs are commercial systems such as Microsoft SQL Server's Hekaton [8, 23] and SAP HANA [10, 37] as well as open-source systems such as PostgreSQL [34].

Hekaton [23] is similar to our implementation in that it is based on a timestamp-based optimistic concurrency control [22] variant of MVCC and uses code generation [12] to compile efficient code for transactions at runtime. In the context of Hekaton, Larson et al. [23] compared a pessimistic locking-based with an optimistic validation-based MVCC scheme and proposed a novel MVCC model for main-memory DBMSs. Similar to what we have seen in our experiments, the optimistic scheme performs better in their evaluation. In comparison to Hekaton, our serializable MVCC model does not update records as a whole but *in-place* and at the *attribute-level*. Further, we do not restrict data accesses to index lookups and optimized our model for high scan speeds that are required for OLAP-style transactions. Finally, we use a novel serializability validation mechanism based on an adaptation of precision locking [17]. Lomet et al. [27] propose another MVCC scheme for main-memory

database systems where the main idea is to use ranges of timestamps for a transaction. In contrast to classical MVCC models, we previously proposed using virtual memory snapshots for long-running transactions [29], where updates are merged back into the database at commit-time. Snapshotting and merging, however, can be very expensive depending on the size of the database.

Hyder [5, 4] is a data-sharing system that stores indexed records as a multi-version log-structured database on shared flash storage. Transaction conflicts are detected by a *meld* algorithm that merges committed updates from the log into the in-memory DBMS cache. This architecture promises to scale out without partitioning. While our MVCC model uses the undo log only for validation of serializability violations, in Hyder, the durable log *is* the database. In contrast, our implementation stores data in a main-memory row- or column-store and writes a redo log for durability. OctopusDB [9] is another DBMS that uses the log as the database and proposes a unification of OLTP, OLAP, and streaming databases in one system.

5.2 Serializability

In contrast to PostgreSQL and our MVCC implementation, most other MVCC-based DBMSs only offer the weaker isolation level *Snapshot Isolation* (SI) instead of serializability. Berenson et al. [2], however, have shown that there exist schedules that are valid under SI but are non-serializable. In this context, Cahill and Fekete et al. [7, 11] developed a theory of SI anomalies. They further developed the *Serializable Snapshot Isolation* (SSI) approach [7], which has been implemented in PostgreSQL [34]. To guarantee serializability, SSI tracks commit dependencies and tests for "dangerous structures" consisting of rw-antidependencies between concurrent transactions. Unfortunately this requires keeping track of every single read, similar to read-locks, which can be quite expensive for large read transactions. In contrast to SSI, our MVCC model proposes a novel serializability validation mechanism based on an adaptation of precision locking [17]. Our approach does not track dependencies but read predicates and validates the predicates against the undo log entries, which are retained for as long as they are visible.

Jorwekar et al. [18] tackled the problem of automatically detecting SI anomalies. [14] proposes a scalable SSI implementation for multi-core CPUs. Checking updates against a predicate space is related to SharedDB [13], which optimizes the processing of multiple queries in parallel.

5.3 Scalability of OLTP Systems

Orthogonal to logical transaction isolation, there is also a plethora of research on how to scale transaction processing out to multiple cores on modern CPUs. H-Store [19], which has been commercialized as VoltDB, relies on static partitioning of the database. Transactions that access only a single partition are then processed serially and without any locking. Jones et al. [16] describe optimizations for partition-crossing transactions. Our HyPer [21] main-memory DBMS optimizes for OLTP and OLAP workloads and follows the partitioned transaction execution model of H-Store. Prior to our MVCC integration, HyPer, just like H-Store, could only process holistic pre-canned transactions. With the serializable MVCC model introduced in this work, we provide a logical transaction isolation mechanism that allows for interactive and sliced transactions.

Silo [41] proposes a scalable commit protocol that guarantees serializability. To achieve good scalability on modern multi-core CPUs, Silo's design is centered around avoiding most points of global synchronization. The proposed techniques can be integrated into our MVCC implementation in order to reduce global synchronization, which could allow for better scalability. Pandis et al. [32] show that the centralized lock manager of traditional DBMSs is often a scalability bottleneck. To solve this bottleneck, they propose the DORA system, which partitions a database among physical CPU cores and decomposes transactions into smaller actions. These are then assigned to threads that own the data needed for the action, such that the interaction with the lock manager is minimized during transaction processing. Very lightweight locking [35] reduces the lock-manager overhead by co-locating lock information with the records.

The availability of hardware transactional memory (HTM) in recent mainstream CPUs enables a new promising transaction processing model that reduces the substantial overhead from locking and latching [24]. HTM further allows multi-core scalability without statically partitioning the database [24]. In future work we thus intend to employ HTM to efficiently scale out our MVCC implementation, even in the presence of partition-crossing transactions.

Deterministic database systems [39, 40] propose the execution of transactions according to a pre-defined serial order. In contrast to our MVCC model transactions need to be known beforehand, e.g., by relying on holistic pre-canned transactions, and do not easily allow for interactive and sliced transactions. In the context of distributed DBMSs, [6] proposes a middleware for replicated DBMSs that adds global one-copy serializability for replicas that run under SI.

6. CONCLUSION

The multi-version concurrency control (MVCC) implementation presented in this work is carefully engineered to accommodate high-performance processing of both, transactions with point accesses *as well as* read-heavy transactions and even OLAP scenarios. For the latter, the high scan performance of single-version main-memory database systems was retained by an *update-in-place* version mechanism and by using synopses of versioned record positions, called *VersionedPositions*. Furthermore, our novel serializabiliy validation technique that checks the before-image deltas in undo buffers against a committing transaction's predicate space incurs only a marginal space and time overhead — no matter how large the read set is. This results in a very attractive and efficient transaction isolation mechanism for main-memory database systems. In particular, our serializable MVCC model targets database systems that support OLTP and OLAP processing simultaneously and in the same database, such as SAP HANA [10, 37] and our HyPer [21] system, but could also be implemented in high-performance transactional systems that currently only support holistic pre-canned transactions such as H-Store [19]/VoltDB. From a performance perspective, we have shown that the integration of our MVCC model in our HyPer system achieves excellent performance, even when maintaining serializability guarantees. Therefore, at least from a performance perspective, there is little need to prefer snapshot isolation over full serializability any longer. Future work focusses on better single-node scalability using hardware transactional memory [24] and the scale-out of our MVCC model [30].

7. REFERENCES

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, 2000.

[2] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, et al. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, 1995.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1986.

[4] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[5] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic Concurrency Control by Melding Trees. *PVLDB*, 4(11), 2011.

[6] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-Copy Serializability with Snapshot Isolation under the Hood. In *ICDE*, 2011.

[7] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. *TODS*, 34(4), 2009.

[8] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, et al. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, 2013.

[9] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, 2011.

[10] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record*, 40(4), 2012.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *TODS*, 30(2), 2005.

[12] C. Freedman, E. Ismert, and P.-Å. Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *DEBU*, 37(1), 2014.

[13] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *PVLDB*, 5(6), 2012.

[14] H. Han, S. Park, H. Jung, A. Fekete, U. Röhm, et al. Scalable Serializable Snapshot Isolation for Multicore Systems. In *ICDE*, 2014.

[15] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional Update Handling in Column Stores. In *SIGMOD*, 2010.

[16] E. P. Jones, D. J. Abadi, and S. Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD*, 2010.

[17] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. In *SIGMOD*, 1981.

[18] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB*, 2007.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, et al. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2), 2008.

[20] A. Kemper et al. Transaction Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *DEBU*, 36(2), 2013.

[21] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[22] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2), 1981.

[23] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, et al. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4), 2011.

[24] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[25] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware. In *ICDE*, 2013.

[26] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[27] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-Version Concurrency via Timestamp Range Conflict Management. In *ICDE*, 2012.

[28] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-only Transactions. *SIGMOD Record*, 21(2), 1992.

[29] H. Mühe, A. Kemper, and T. Neumann. Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems. In *CIDR*, 2013.

[30] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: Elastic OLAP Throughput on Transactional Data. In *DanaC*, 2013.

[31] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9), 2011.

[32] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented Transaction Execution. *PVLDB*, 3, 2010.

[33] H. Plattner. The Impact of Columnar In-Memory Databases on Enterprise Systems: Implications of Eliminating Transaction-Maintained Aggregates. *PVLDB*, 7(13), 2014.

[34] D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. *PVLDB*, 5(12), 2012.

[35] K. Ren, A. Thomson, and D. J. Abadi. Lightweight Locking for Main Memory Database Systems. *PVLDB*, 6(2), 2012.

[36] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB*, 7(13), 2014.

[37] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, et al. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*, 2012.

[38] A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. *CSUR*, 30(1), 1998.

[39] A. Thomson and D. J. Abadi. The Case for Determinism in Database Systems. *PVLDB*, 3, 2010.

[40] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012.

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.

[42] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[43] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1), 2009.