Reducing Database Locking Contention Through Multi-version Concurrency

Mohammad Sadoghi[1], Mustafa Canim[1], Bishwaranjan Bhattacharjee[1], Fabian Nagel[3], Kenneth A. Ross[1,2]

[1]IBM T.J. Watson Research Center
[2]Columbia University
[3] University of Edinburgh

ABSTRACT

In multi-version databases, updates and deletions of records by transactions require appending a new record to tables rather than performing in-place updates. This mechanism incurs non-negligible performance overhead in the presence of multiple indexes on a table, where changes need to be propagated to all indexes. Additionally, an uncommitted record update will block other active transactions from using the index to fetch the most recently committed values for the updated record. In general, in order to support snapshot isolation and/or multi-version concurrency, either each active transaction is forced to search a database temporary area (e.g., rollback segments) to fetch old values of desired records, or each transaction is forced to scan the entire table to find the older versions of the record in a multi-version database (in the absence of specialized temporal indexes).

In this work, we describe a novel kV-Indirection structure to enable efficient (parallelizable) optimistic and pessimistic multi-version concurrency control by utilizing the old versions of records (at most two versions of each record) to provide direct access to the recent changes of records without the need of temporal indexes. As a result, our technique results in higher degree of concurrency by reducing the clashes between readers and writers of data and avoiding extended lock delays. We have a working prototype of our concurrency model and kV-Indirection structure in a commercial database and conducted an extensive evaluation to demonstrate the benefits of our multi-version concurrency control, and we obtained orders of magnitude speed up over the single-version concurrency control.

1. INTRODUCTION

In a multi-version database system, new records do not physically replace old ones. Instead, a new version of the record is created, which becomes visible to other transactions at commit time. Conceptually, there may be many rows for a record, each corresponding to the state of the database at some point in the past. Older versions may be garbage-collected as the need for old data diminishes, in order to reclaim space for new data.

Proceedings of the VLDB Endowment, Vol. 7, No. 13

When indexing data, one typically indexes only the most recent version of the data, since that version is most commonly accessed and to reduce the overhead of maintaining older versions of the records which affects the index fanout. In such a setting, record insertions, deletions and updates trigger I/O to keep the indexes up to date. With a traditional index structure, the deletion of a record requires the traversal of each index and the removal of the row-identifier (RID) from the leaf node. The update of a record (changing one attribute value to another) creates a new version, again triggering a traversal of all indexes to change the RIDs to the new version's RID. In the case of a modified attribute, the position of the record in the index may also change. For a newly inserted record, the new RID must be inserted into each index. Therefore, managing a multi-version database further increases the cost of index maintenance.

The concurrent access of different transactions during index maintenance poses another obstacle when relying on traditional locking because reader and writer transactions are incompatible and block each other. Therefore, as the concurrency increases and resource contention between the readers and writers increases (this is an emerging hardware trend: exponential increase of processor's core count and increase in the size of main-memory [7, 24, 16]) the overall utilization of a system deteriorates. This effect is further magnified when in addition to typical short update transactions, there are long running read-only transactions that hold read locks for an extended period of time, which could essentially bring the database to a complete stall.

The conflict between readers and writers, especially those of long readers, limits the prospect of single-version concurrency. A naive (and rather common) approach is to deal with this limitation by relaxing the consistency model and settling for transaction-inconsistent answers to queries, or by relying on an existing multi-version concurrency model (MVCC) such as [4, 1, 12, 17, 18]. In this work, we address these limitations, namely, the clashes of both short and long running transactions, and our contributions are as follows.

- We propose a latch-free optimistic and pessimistic 2-version concurrency control (2VCC) model to eliminate transaction blocking and avoid prolonged lock wait times.

- We develop a parallel and optimistic 2VCC variant to reduce the length of time that locks are held. This method avoids locking resources prior to transferring data from a slow to a fast medium, which would prevent other active transactions from accessing the held resources.

- We design a novel kV-Indirection structure, which is an efficient method to transparently extend indexes in order to incorporate the 2VCC model by keeping at most k transient references to the last k versions of the record.
- We propose a cost-effective and simple mapping to express the 2VCC model using the existing single-version two-phase locking (2PL) infrastructure found in most commercial database systems.
- We develop a working prototype of the 2VCC model and kV-Indirection structure in a commercial database and conducted an extensive evaluation to demonstrate their benefits.

2. MULTI-VERSION DATABASES OVERVIEW

By keeping old data versions, a system can enable queries about the state of the database at points in the past. The ability to query the past has a number of important applications [28], for example, (1) a financial firm is required to retain any changes made to client information for up to five years in accordance with auditing regulations; (2) a retailer ensures that they offer only one discount for each product at any given time; (3) a bank needs to retroactively correct an error for miscalculating the promised introductory interest rate. In addition to these business-specific scenarios, there is an inherent algorithmic benefit from retaining the old versions of the record and avoiding in-place update, that is, to utilize efficient optimistic locking and latch-free data structures.

A simple implementation of a multi-version database would store the row-identifier (RID) of the old version within the row of the new version, defining a linked list of versions. Such an implementation allows for the easy identification of old versions of each row, but puts the burden of reconstructing consistent states at particular times on the application, which would need to keep timing information within each row.

To relieve applications of such burdens, a multi-version database system can maintain explicit timing information for each row. In a valid time temporal model [10] each row is associated with an interval [begin-time,end-time) for which it was/is current. Several implementation choices exist for such a model. One could store the begin-time with each new row, and infer the end-time as the begin-time of the next version. Compared with storing both the begin-time and end-time explicitly for each row, this choice saves space and also saves some write I/O to update the old version. On the other hand, queries over historical versions are more complex because they need to consult more rows to reconstruct validity intervals.

In this work we do not commit to any one of these implementation options, each of which might be a valid choice for some workloads. For any of these choices, our proposed methods will not only reduce the I/O burden of index updates using the indirection technique [26], but leverage these versions to efficiently implement multi-version concurrency control using existing locking infrastructure in commercial databases.

2.1 Physical Organization

There are several options for the physical organization of a multi-version database. For example, one organization option appends old versions of records to a "history" table and only keeps the most recent version in the main table, updating it in-place. Commercial systems have implemented this technique: In IBM DB2 it is called "System-period data versioning" [13], and it is used whenever a table employs transaction time as the temporal attribute. The Oracle Flashback Archive [23] also uses a history table. Such an organization clusters the history table by end-time, and does not impose a clustering order on the main table. Updates need to read and write the main table, and also write to the end of the history table. Because updates to the main table are in-place an index needs to be updated only when the corresponding attribute value changes. For insertions and deletions, all indexes need to be updated. In short, using the history table approach, the temporal ordering of the data is lost and additional random I/Os are required to perform in-place updates of records.

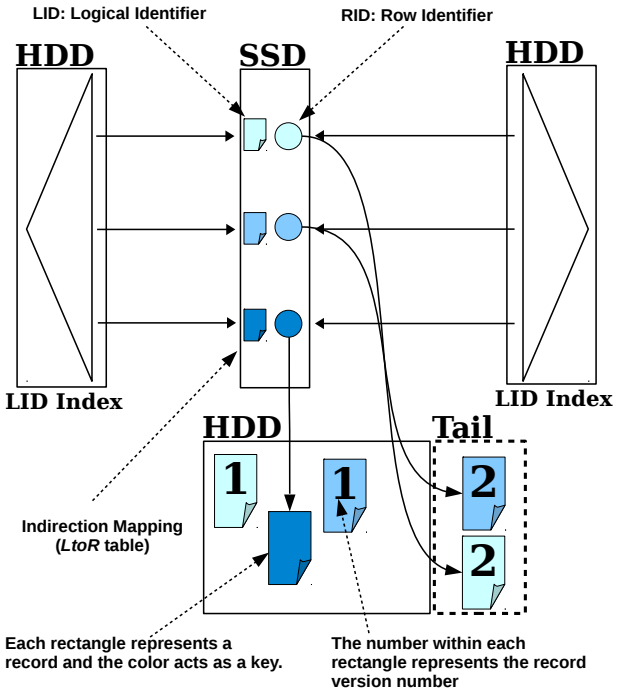


Figure 1: LID index using the indirection technique

In this paper, we assume an organization in which there is a single table containing both current and historical data; however, we are not limited to this design choice. Commercial systems that implement this technique include Oracle 11g where the concept is called "version-enabled tables" [22]. IBM's DB2 also uses this approach for tables whose only temporal attribute is the application time. New rows are appended to the table, so that the entire table is clustered by begin-time. Updates need to read the table once and write a new version of the record to the end of the table.

2.2 Index Maintenance Overview

Traditional index structures directly reference a record via a pointer known as a physical row-identifier (RID). The RID usually encodes a combination of the database partition identifier, the page number within the partition, and the row number within the page.

The choice of a physical identifier hinders the update performance of a multi-version database in which updates result in a new physical location for the updated record. Changes to the record induce random access for every index, even indexes on "unaffected" attributes, i.e., attributes that have not changed. Random accesses are required to modify leaf pages.

To avoid random accesses for indexes on unaffected attributes, we rely on an existing technique for decoupling the physical and logical representations of records spanning many versions [26]. Thus, we distinguish between a physical row-identifier (RID) and a logical record identifier (LID). For any given record, there may be many RIDs for that record corresponding to the physical placement of all of the versions of that record. In contrast, the LID is a refer-

ence to the RID representing the most recent version of the record. For now, one can think of a table *LtoR(LID,RID)* that has LID as the primary key. Indexes now contain LIDs rather than RIDs in their leaves.

Using this indirection technique [26], an index traversal must convert a LID to a RID using the *LtoR* table. A missing LID, or a LID with a NULL RID in the *LtoR* table are treated as deleted rows, and are ignored during search. The new index design is demonstrated in Figure 1.

When an existing record is modified, a new version of that record is created. The *LtoR* table is updated to associate the new row's RID to the existing LID. That way, indexes on unchanged attributes remain valid. Only for the changed attribute value will random access to the index be required.

When a record is deleted, the (LID,RID) pair for this record in the *LtoR* table is deleted. Index traversals ignore missing LIDs. Indexes can lazily update their leaves during traversal, when a read I/O is performed anyway. At that time, any missing LIDs encountered lead to the removal of those LIDs from the index leaf page. After a long period of activity, indexes should be validated offline against the *LtoR* table to remove deleted LIDs that have subsequently never been searched for.

When a new record is added, the new record is appended to the tail of the relation and its RID is fetched and associated to a new LID. The (LID, RID) pair for the new record is added to the *LtoR* table. All indexes are also updated with the new record LID accordingly.

3. OVERVIEW OF KV-INDIRECTION

We extend the indirection to further differentiate between up to $k-1$ committed and one uncommitted versions of each record through our indirection mapping. By decoupling committed and uncommitted versions, we avoid clashes between readers of currently committed data and writers of newly updated/inserted records, without changing the semantics or the structure of the index. An overview of this structure is presented in Figure 2, in which for clarity we only show one committed and one uncommitted version of each record.

In 2V-Indirection, the currently committed version of every record is given by *cRID* and the outstanding uncommitted version is given by *uRID*. The triplet *(LID, cRID, uRID)* presents a conceptual and logical connection; but, it does not dictate that the indirection mapping table must be physically extended such that it pre-allocates enough space for the *uRID*. The *uRID* could be maintained for only the active set of transactions to reduce space overhead.

There is an important subtlety that arises when combining indexes with the 2V-Indirection mapping. Suppose we update a record on column col_i, where an index is also defined on col_i. Now whenever a record value for col_i is changed, then both the old value and the new value of the column are associated to the record's LID. This allows readers to detect that both values are referring to the same record due to the common LID and also gives an option of reading either committed or uncommitted values from the index. This is shown in Figure 3. By examining the index leaf page, it can easily be determined whether value and LID pair is committed or uncommitted using a single bit to indicate whether an entry is committed or not.

Through 2V-Indirection, concurrent readers are able to access the currently committed version of every record without interfering with writers. Similarly, writers are able install an updated uncommitted version of a record without blocking current readers. By placing a reference to the uncommitted version of every record, it also enables the readers to speculatively read uncommitted data

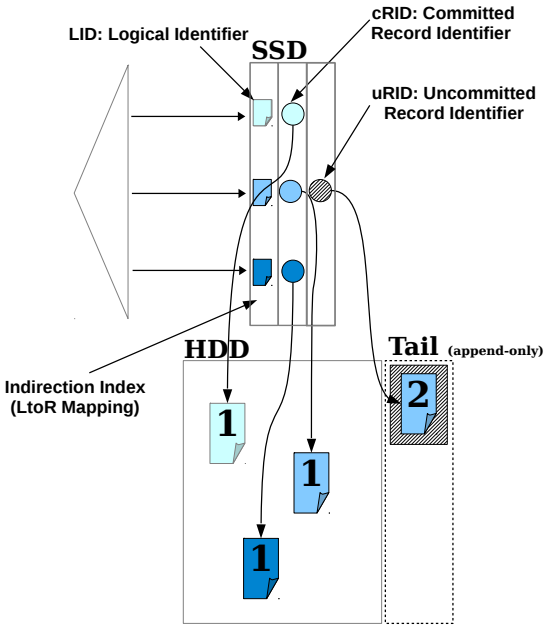


Figure 2: 2V-Indirection structure

and ignore the committed version or vice versa. Therefore, kV-Indirection seamlessly allows access to multiple versions of a record without changing the underlying structure. kV-Indirection is transient in nature because it maintains only references to at most k recent versions of the record.

With 8-byte LIDs, 8-byte *cRID*s, and 8-byte *uRID*s we need 24 bytes per database row (or the row in the active set). This could further be compressed to less than half the size, since LIDs tend to have many leading zeroes, and there are well-known techniques for compressing RIDs. In addition, if a table is partitioned into set of smaller tablets, a 4-byte *cRID* and *uRID* could be sufficient. Alternatively, if we could guarantee that the number of LIDs are chosen consecutively starting from 1 (for example use the original RID of the record), one could omit the LID entirely, and just use position in the table as a surrogate for the LID). Even without any of the above schemes to reduce the size of indirection table, a table with 240-byte rows incurs of an overhead of only 10% of the base table footprint (regardless of how many indexes are defined on the table).

4. ACCESS OVERVIEW OF 2V-INDIRECTION

The 2V-Indirection mapping keeps only the *cRID* of the most recently committed version of a record and the *uRID* of the uncommitted version of the record (at most one uncommitted version is allowed). The indirection mapping table is keyed by LID (e.g., a hash table) to enable fast lookups to both *cRID* and *uRID*. Indexes would no longer point to RIDs, but to LIDs instead such that each index may now indirectly point to different versions of the record, a committed and uncommitted RID. The 2V-Indirection enables selection of the right version based on the transaction concurrency need as to whether a committed or the uncommitted version is required.

Updates of uncommitted transactions will physically create a new version of a record referenced by an uncommitted *uRID* address in the 2V-Indirection. But all other concurrent transactions, when searching through indexes will read the committed version of the record (and not the uncommitted version); thus, always following *cRID* pointers (unless the uncommitted/dirty read is tolerated,

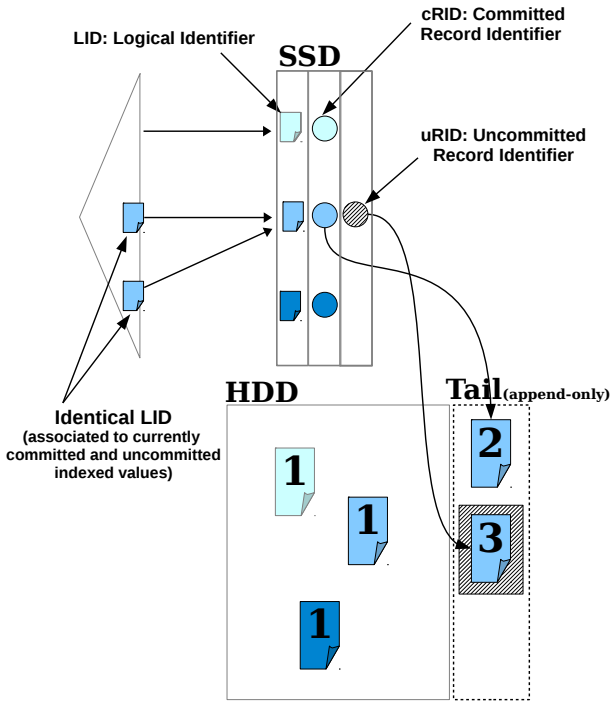


Figure 3: 2V-Indirection structure with one-to-many LID-to-RID association

which is efficiently supported by 2V-Indirection). Only at the time of committing the updates, the *uRID* address will replace the existing *cRID* and the current *uRID* is set to null. Old key values associated to the older version of the record will be removed from the affected indexes and new key values are inserted. Indexes on unaffected columns are not modified. If a transaction aborts, then the *uRID* is set to null and any new key added to the affected indexes is removed (undo procedure); however, since the old key is retained in the index until commit time, chasing the *cRID* is still valid in the 2V-Indirection; the old value does not need to be re-inserted.

5. MULTI-VERSION CONCURRENCY

We present our optimistic and pessimistic 2-version concurrency control (2VCC) model with both transaction blocking and non-blocking behaviour, inspired by 2V2PL [4]. The original 2V2PL protocol is strictly a pessimistic lock-based (i.e., blocking) multi-version concurrency control method that requires retaining at most two versions of every record in order to guarantee transaction serializability [4]. We characterize optimistic and pessimistic control behaviour based on how a transaction validates its reading set as follows:

- *pessimistic*, validating the reads during the transaction, or
- *optimistic*, validating the reads only before committing the transaction.

Another aspect of our 2VCC model (for both pessimistic and optimistic) is whether a transaction waits for the validation, i.e., whether the concurrency protocol relies on locking (blocking) or not (non-blocking).

Our concurrency model is capable of supporting full range of isolation levels including:

- *read uncommitted* ensures that there are no lost updates but reads could be uncommitted (dirty reads are possible)
- *read committed* ensures that there are no lost updates and reads are committed (reads are not repeatable)
- *repeatable read* ensures that there are no lost updates and reads are repeatable (phantoms are possible)
- *serializable* ensures that there are no lost updates, reads are repeatable, and there are no phantoms
- *snapshot* ensures reads are repeatable, i.e., transaction consistent, with respect to a point in time such as transaction begin time and there are no phantoms

What is common among these isolation variations is that by retaining old versions of a record (for non-snapshot isolation levels only the last committed and the uncommitted versions are required), we can avoid conflicts between readers and writers of the data as the level concurrency and contention increases. Most importantly, we describe an efficient protocol to realize our proposed concurrency protocol using kV-Indirection, which we have implemented in an existing commercial database system.

5.1 Pessimistic Concurrency Control

The original lock-based pessimistic 2V2PL avoids clashes of readers and writers by ensuring that writers certify their writes prior to committing. The protocol is formalized as follows (as presented in [4]):

- *reading* $r(x)$, a read lock $rl(x)$ is set prior to reading the currently committed version of x; the current version is read from the *cRID* column of 2V-Indirection. For phantom detection, the range-predicate of the query is registered or the next-key locking technique is employed.
- *writing* $w(x)$, a new uncommitted version of a record is installed by locking $wl(x)$, a write lock is set prior to modifying x.
- *certifying writes*, a certify lock $cl(x)$ is set prior to finalizing the transaction, in parallel, on every data item x modified by the transaction (lock promotions) in order to ensure that no active transaction with repeatable read isolation or higher is currently reading the current value of records. The certification is also extended to satisfy the registered range-predicates.
- *commit*, newly committed versions are installed, and all read and write locks are released.

THEOREM 1. *The parallel implementation of 2V2PL is conflict-free serializable.*[1]

Supporting read committed semantics using pessimistic 2V2PL is straightforward (following standard 2PL protocol for cursor stability). The read locks are released as soon as the records are read in the reading phase. For uncommitted reads, no read locks are acquired prior to reading the record.

Next we describe our latch-free (non-blocking) pessimistic 2VCC mechanism. The latch-free operations are implemented using an atomic compare-and-swap (CAS) operator.

[1]All proofs are presented in Appendix.

- *reading* $r(x)$, if the read counter of item x is greater or equal to 0, then the read counter is incremented [implemented latch-free using CAS operator] prior to reading the current version of item x; the current *cRID* value of x is read from the 2V-Indirection structure. If the read counter is smaller than 0, then reading fails and the transaction is aborted and rolled back.

- *writing* $w(x)$, a new version of the record is installed by detecting *ww-conflict(x)*: a write-write conflict is detected prior to modifying x. If no conflict is detected a new uncommitted version of x is written and *uRID* is updated accordingly. The value of *uRID* itself is an indicator of write-write conflict: a null value means that no other transaction is currently changing the corresponding record and a non-null value means that the record is already being changed [implemented latch-free using CAS operator].

- *certifying writes*, every item in the writeset is certified prior to finalizing the transaction in order to ensure that no active transaction with repeatable read isolation or higher is currently reading the current value of records. Certify is satisfied if the read counter is 0 for each item in the writeset. If the read counter is 0, then on certify, set the counter to -1 [implemented latch-free using CAS operator], indicating that the item is in the process of certification and its read counter cannot be incremented. If the read counter is not 0, then the write cannot be certified and the transaction is aborted and rolled back.

- *commit*, once the transaction is committed; the increased read counters for items in the readset are decremented, the read counters of the writeset is set back to 0 from -1, the *cRID* is replaced with the *uRID*, and the *uRID* of items in the writeset are set to null.

THEOREM 2. *The proposed latch-free pessimistic* 2VCC *is conflict-free serializable.*

Similar to blocking pessimistic 2VCC, supporting read committed semantics using latch-free pessimistic 2VCC is also straightforward. For read committed, the read counter is decremented as soon as the records are read in the reading phase (to support only read-committed isolation, no read counter is required). For the uncommitted read, no read counter is incremented prior to reading the record, essentially the counter is acting as a light-weight locking mechanism.[2]

5.2 Optimistic Concurrency Control

Optimistic concurrency is relevant when validation is needed, namely, for repeatable read and serializability. The read committed isolation level always reads the committed version and does not require validation, and snapshot isolation reads the view of the database from an instantaneous point in time, which again does not require validation. We focus first on repeatable read and not phantom detection required for full serializability.

To efficiently implement our optimistic 2VCC model, we again rely on our proposed 2V-Indirection that provides a fast mechanism for the validation phase. We require each transaction to keep track of the (LID, *cRID*) that was read during the read phase of the transaction. We also drop the need for holding read locks for the readset of a transaction. In the final phase of a transaction, during the certify/validation phase, a transaction re-fetches the current (LID, *cRID*) pairs from the 2V-Indirection structure (and at this stage it also acquires read locks on these records). If the *cRID* has not changed for the entire readset, then the transaction satisfies the validation phase and continues with the rest of the certify phase of 2VCC (read locks acquired during the validation are released once the certify phase is also completed). If either the *cRID* has changed or the read lock cannot be acquired due to a certify lock already held on the record by another active transaction, then the current transaction is aborted.

In order to deal with phantoms, we propose two alternative approaches. (1) A standard technique repeats the entire range scan when validating instead of just validating each element in the readset. (2) Rely on range-predicate phantom detection (or key-range locking), in which when range scans are submitted, a predicate that covers the range is also issued; thus, each writer transaction must validate all of its writes against the (relevant) range predicates of all active transactions during the certify stage to avoid invalidating repeatability requirement of other active transactions. Extending the notion of certification to include range-predicate validation prevents phantoms.

The validation phase can be carried out efficiently and in parallel because the 2V-Indirection structure has the necessary information to complete the validation, and it is expected to be maintained on a fast medium that supports random access (e.g., main memory or storage-class memory). Once the transaction is ready to commit, at which point the entire readset is also known, the readset can be verified using 2V-Indirection in parallel. Validation time is reduced including the time that read locks are maintained, which further reduces the contention likelihood between concurrent transactions. Since the readset is known at the end of transaction, effective batching techniques to acquire read locks in bulk can also be employed. Lastly, since the most restrictive lock types are held only during the final transaction phase, in which data most likely resides in-memory (or even processor's cache), exclusive lock duration is reduced because a lock is rarely held while transferring data from slow to fast memory.

The proposed optimistic 2VCC is formalized as follows:

- *reading* $r(x)$, the currently committed version of x is read; the current version is read from the *cRID* column of the 2V-Indirection structure. For phantom detection, the range-predicate of the query is also registered.

- *writing* $w(x)$, a new uncommitted version of record is installed by locking $wl(x)$, a write lock is set prior to modifying x.

- *validating reads*, a read lock $rl(x)$ is set prior to reading the current version of x in the readset; the current *cRID* value of x is read from the 2V-Indirection structure, for each x if its *cRID* value has not changed from when it was first read, then the validation is satisfied.

- *certifying writes*, a certify lock $cl(x)$ is set prior to finalizing the transaction on every data item x modified by the transaction (lock promotions) in order to ensure that no active transaction with repeatable read isolation or higher is currently reading the current value of records. The certification is also extended to satisfy the registered range-predicates.

- *commit*, newly committed versions are installed, and all read and write locks are released.

THEOREM 3. *The proposed optimistic* 2VCC *is conflict-free serializable.*

[2]Note that the read counter could be embedded within the 8 byte pointer of the 2V-Indirection *uRID* (if not persisted) to avoid incurred cache misses when updating counters.

Supporting read committed/uncommitted semantics for optimistic 2VCC is achieved by not requiring validation of the reads prior to committing the transaction. Next we outline our latch-free (non-blocking) optimistic 2VCC mechanism. The latch-free operations are implemented using an atomic compare-and-swap (CAS) operator.

- *reading* $r(x)$, the currently committed version of x is read; the current version is read from the *cRID* column of the 2V-Indirection structure. For phantom detection, the range-predicate of the query is also registered.
- *writing* $w(x)$, a new uncommitted version of record is installed by detecting *ww-conflict(x)*, a write-write conflict is detected prior to modifying x. If no conflict is detected a new uncommitted version of x is written and *uRID* is updated accordingly. The value of *uRID* itself is an indicator of write-write conflict: the null value means that no other transaction is currently changing the corresponding record and a non-null value means that the record is already being changed [implemented latch-free using the CAS operator].
- *validating reads*, for each item x in the readset, if its read counter is greater or equal to 0, then the read counter is incremented [implemented latch-free using the CAS operator] prior to reading the current version of item x; the current *cRID* value of x is read from the 2V-Indirection structure. For each item x if its *cRID* value has not changed from when it was first read, then the validation is satisfied. If the read counter is smaller than 0, the validation fails and the transaction is rolled back.
- *certifying writes*, every item in the writeset is certified prior to finalizing the transaction in order to ensure that no active transaction with repeatable read isolation or higher is currently reading the current value of records. Certify is satisfied if the read counter is 0 for each item in the writeset. If the read counter is 0, then certify sets the counter to -1 [implemented latch-free using CAS operator], which indicates that item is in the process of certification and its read counter cannot be incremented. If the read counter is not 0, then the write cannot be certified and the transaction is aborted and rolled back. The certification is also extended to satisfy the registered range-predicates.
- *commit*, when the transaction is committed; the reference to the uncommitted versions replace the reference to committed versions, the increased read counters for items in the readset are decremented, the read counters of the writeset is set back to 0 from -1, and *uRID* of items in the writeset are set to null.

THEOREM 4. *The proposed latch-free optimistic* 2VCC *is conflict-free serializable.*

Supporting read committed/uncommitted semantics for latch-free optimistic 2VCC is achieved by not requiring validation of the reads prior to committing the transactions.

5.3 Pessimistic and Optimistic Co-existence

The blocking versions of pessimistic and optimistic 2VCC can naturally co-exist because both rely on the lock manager, and the only difference is the time when the locks are acquired. Appropriate locks, including certify locks, must be acquired. The situation is slightly more complex when we also include non-blocking concurrency mechanisms.

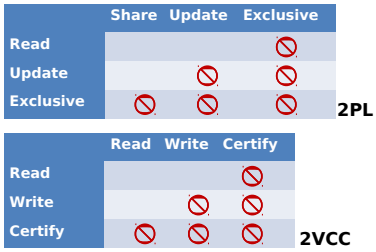


Figure 4: Lock compatibility comparison for 2PL and 2VCC

Consider the case of mixing latch-free pessimistic and optimistic transactions first. Again, both models respect how read counters are incremented, ensure that there is at most one outstanding write for each record, and certify writes always. The only difference between the two models is that read counters are incremented at the beginning of a transaction for the pessimistic approach while the increment is deferred to the end of transaction for optimistic transactions. Therefore, latch-free pessimistic and latch-free optimistic concurrency can peacefully co-exist.

However, the co-existence of blocking and non-blocking (i.e., latch-free) concurrency is non-trivial irrespective of whether they are pessimistic or optimistic. Blocking transactions are relying on the lock manager and its queuing capability while the non-blocking transactions altogether bypass the lock manager. In other words, there is no obvious mechanism to enable coordination between blocking and non-blocking transactions.[3]

We propose a light-weight coordination mechanism based on 2V-Indirection and the lock manager to enable the co-existence of both blocking and non-blocking transactions. First, we require that blocking transactions increment the read counters in addition to acquiring the read locks (depending on pessimistic or optimistic behavior, the read counter could be incremented either at the beginning or the end of the transaction, respectively). If the read counter cannot be incremented due to certification of an outstanding write by a latch-free transaction, then the blocked transaction also acquires a wait-for dependency on the writer transaction. The writes for a blocking concurrency must also detect the possible write-write conflicts (using the *uRID* column of the 2V-Indirection structure) and acquire a wait-for dependency again using the lock manager. Therefore, using our proposed coordination, we can support the mix of blocking and non-blocking of both pessimistic and optimistic concurrency.

5.4 Mapping 2VCC-to-2PL Infrastructure

We incorporated the 2VCC protocol efficiently using 2V-Indirection within an existing commercial database system based on 2PL. The existing 2PL lock manager consists of two main locks: *shared* and *exclusive*. Shared locks are for reading and can be shared across transactions and exclusive locks are for writing and prevent other transactions from reading or writing records. In addition, most commercial systems have *update* lock which is an intention to write, but does not actually allow the changing of records; the lock must be promoted to an exclusive lock first. This 2PL characterization nicely fits into our 2VCC lock scheme (as shown in Figure 4 by mapping *read* lock to *shared* lock, *write* lock to *update* lock, and *certify* lock to *exclusive* lock. In addition, we can also relax update

[3]Certain (existing) applications may require the blocking feature and expect the database to manage conflicting transactions as opposed to simply aborting such transactions and returning them to the user.

lock conditions to allow the transaction that holds an update lock to physically change the record. Using this lock mapping, we efficiently transform the locking infrastructure designed for a database with single-version concurrency to multi-version concurrency with minimal changes.

5.5 Impacts on recovery

The persistence of the 2V-Indirection mapping table (together with the append-only approach for retaining all versions of the record) plays a central role in rapid recovery, so that the indexes are still valid. If we store LIDs with rows in the log, we can recover the mapping table after a failure. The recovery procedure can drop the *uRID* from the affected indexes while leaving the *cRID* in 2V-Indirection. Also since the old values of committed versions are retained, the undo process of recovery (or roll back) is much simpler, and it does not affect transactions accessing the committed version (also accessible through the index).

The append-only insertion of records further improves recovery time, because the redo will strictly be limited to the tail of the relation (also exhibiting a fast sequential I/O pattern) and only undo for rare and very long running transactions may require some random accesses to mark an earlier inserted record as invalid/deleted. All dirty pages are naturally flushed and committed even under a no-force policy (enabling fast fuzzy checkpoints), which results in examining a limited number of dirty pages and redo pages during recovery. In addition, simple bufferpool eviction policies (such as stealing the oldest dirty page) or frequently committing the tail of a relation ensure a provably bounded recovery time without any checkpointing.

5.6 Proposed concurrency model benefits

The key advantages of our 2VCC model using kV-Indirection are summarized as follows. The transaction writeset can be protected adaptively based on the workload characteristics. Update locks can be taken when there is contention among writer transactions. Our proposed latch-free write-write conflict detection using 2V-Indirectionś *uRID*s can be taken when there is contention between reader and writer transactions. For the latter case, it is notable that even our pessimistic 2VCC protocol is optimistic by design and can seamlessly benefit from an optimistic philosophy. If there is always contention between readers and writers, but transactions are short lived and touch only a few records, then the reader of an older version of a record could be already completed before a writer commits a newer version of the record. Therefore, certification can be completed quickly.

During the validate stage, the readset is verified by checking in parallel using kV-Indirection the current (LID, *cRID*); if *cRID* has changed, then abort the transaction, roll back, and release all locks:

- Unlike an in-place update scheme, the checking of the RID using kV-Indirection is an indicator of change (for monotonically increasing RIDs); it could simulate a global timestamp.
- Using the *RID* for validation, the need for a global clock is avoided, the new disk location indicates a change and guarantees correctness.
- Writers do not wait for non-repeatable readers. Only readers must validate their readset for repeatable read and read stability isolation levels.
- A transaction does not take locks in a serial order because at the end of transactions, if a lock is needed, the list of required locks is known and can be acquired in bulk. Thus, parallel requests of locks for the entire readset is possible.
- No exclusive locks are held while data are being transferred from slow to fast memory during reads.

Our latch-free 2VCC model uses a light-weight locking mechanism relying on counters. Our proposed optimistic and pessimistic concurrency control supports both blocking and non-blocking behaviours and can be implemented using the existing 2PL infrastructures, allowing both multi-version and single-version concurrency to co-exist peacefully.

6. EXPERIMENTAL EVALUATION

In this section, we present a comprehensive evaluation of the proposed MVCC techniques in a commercial database prototype. Experiments were conducted on a machine running Red Hat Enterprise Linux Server release 6.4 using two 8-core Intel Xeon E7-4820 CPUs clocked at 2.0GHz (providing total of 32 hardware threads) and having 32GB of RAM. Our experiment is based on modifying an existing commercial database engine to implement our proposed kV-Indirection and 2VCC techniques.

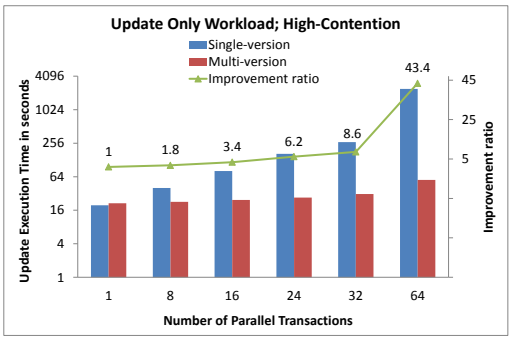
We focus our study on an existing micro benchmark proposed in [17]. The goal of this benchmark was to narrow down the impact of concurrency with respect to the database active set, which determines the degree of the readers' and writers' lock contention. Three types of workload were studied: (1) low contention, where the active set is 10M records; medium contention, where the active set is 10K records; and high contention, where the active set is 1k records. It is important to note that the database size is not limited to the active set, and can be much larger (millions or billions of records). Similar to [17], we consider two classes of transactions: read-only transactions that scan up to 10% of the data (to model TPC-H style queries) and short update transactions (to model TPC-C and TPC-E transactions) that consist of 10 reads and 2 writes. In addition, we studied varying the ratio of read/writes in these update transactions to model different customer scenarios with different read/write degrees. Like [17], we focus on the read-committed isolation level for update transactions.[4] For read-only transactions, we focus on getting transaction-consistent view, which is efficiently achieved using snapshot isolation for a multi-version database by retaining the history of all records while acquiring a consistent view is only possible using repeatable read isolation semantics for the single-version database. In this study, we took a vanilla commercial database that relies on single-version concurrency and enhanced it with the 2VCC model that avoids in-place updates and maintains the complete history of all records.

Unlike the work in [17] that dealt with an in-memory database, our database prototype is disk-based. We acknowledge the potential trend toward memory-optimized databases, in which the entire transactional workload will fit in memory. In a memory-optimized database, where the disk I/O is virtually eliminated, there is an even a higher degree of concurrency (due to a lower latency) and a higher degree of lock contention. To capture this higher degree of contention, we adjusted the database bufferpool such that all reads (including indirection mapping accesses) are served from memory,[5] and writes did not result in page cleaning. Additionally, we placed the log on an enterprise FusionIO SSD card and enabled OS file-system caching. We essentially eliminated any log-related bottlenecks as well.

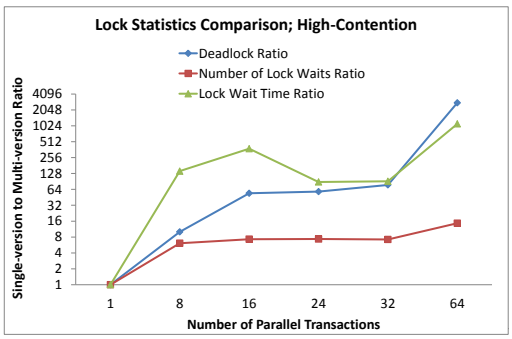
Effects of varying contention: We study the effect of varying contention among short update transactions to model OLTP type

[4]Therefore, we focus our experiments on the latch-free optimistic/pessimistic transaction model.

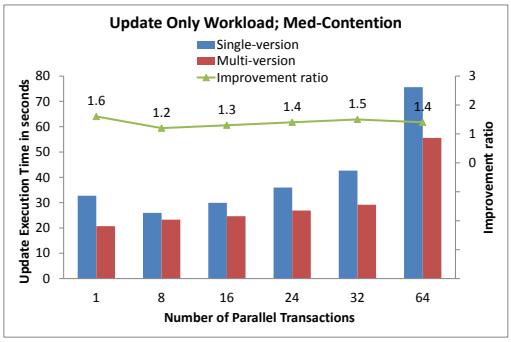
[5]We also ensured that the bufferpool is warmed up prior to running measured experiments.



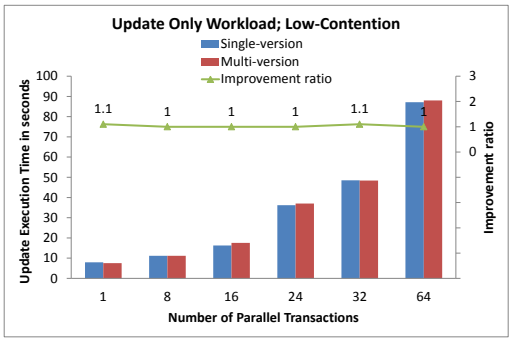
(a) High Contention (log scale)



(b) Lock-related statistics for High Contention



(c) Medium Contention



(d) Low Contention

Figure 5: Effects of varying the number of parallel short update transactions (update execution time)

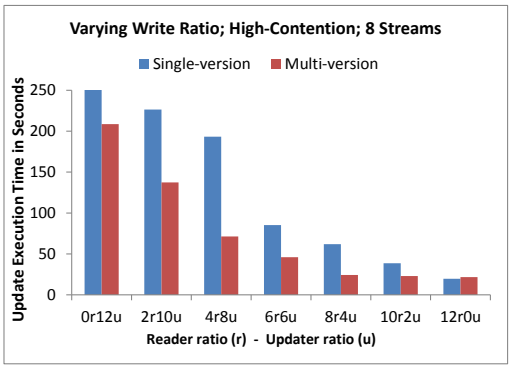
workloads. We control the degree of contention by varying the database active set size and varying the number of parallel update transactions. We construct up to 64 parallel streams of update transactions, where in each stream we issue up to 10,000 transactions in sequence, and we report the total execution of all streams. As we increase the number of concurrent short update transactions from 1 to 64, we observed that the our 2VCC model using 2V-Indirection outperforms the database with a single-version concurrency model consistently for both high (up to 43x speed up) and medium contention (up to 1.6x speed up) and achieves a comparable performance when dealing with low contention (up to 1.1x speed up), as shown in Figure 5. The reasons for improvement are two-fold: (1) fewer read locks are acquired and/or latch-free reads are utilized without acquiring any read locks, and (2) readers and writers do not conflict because when an updater writes a new version of a record, a read could continue reading the currently committed version of the record without any blockage.

These improvements are also supported by statistics from the database such as the number of lock waits issued, the total lock wait time, and the number of deadlocks. As shown in Figure 5(b), for the high-contention workload, as the number of parallel threads increases, the lock wait time and the deadlock ratio of single-version over mutli-version is substantially increased due to readers and writers conflicts.[6]

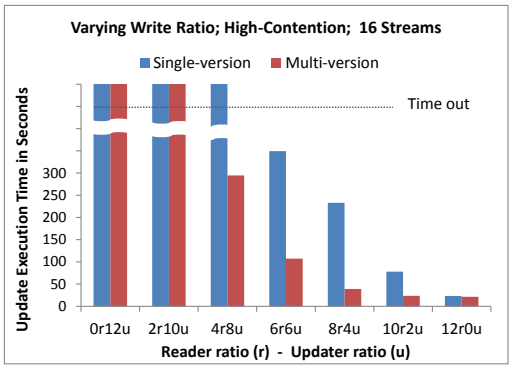
Effects of varying workload read/write ratio: A key property of update transactions is the ratio between reads and writes. In the previous experiments, we considered only a read/write ratio of 10:2. We now explore the spectrum from a read-intensive workload having read/write ratio of 12:0 to a write-intensive workload having read/write ratio of 0:12 for both high and medium contention. Interestingly, in Figure 6 we observe that for the read-intensive workload, the single- and multi-version database performs the same (true also for low contention, which is not shown here) despite the fact that multi-version database with the latch-free feature avoids taking any read locks. This shows the maturity of the commercial database lock infrastructure, where lock avoidance logic is effective and, in general, acquiring and releasing locks without any lock wait are not expensive. However, this picture quickly changes as soon as we introduce writes into the mix, where the lock avoidance logic fails, and acquiring read locks conflicts with the exclusive locks (resulting in an extended lock wait time) that are held for the entire duration of the transaction in the single-version database. The clashes between readers and writers are eliminated by acquiring only update locks during transaction life-time, and holding exclusive locks for a much shorter period only during the commit time.

Effects of combining OLTP- and OLAP-type queries: Another major benefit of our approach manifests itself when long read-only transactions are also present, modelling an operational data store where both OLTP and OLAP queries are run. The long-running queries on average touch 10% of the base table. A desired isolation semantics for read-only query is to have transaction consistent view; this is efficiently achieved through snapshot (through table scan and by retaining the history of all records) in a multi-version database, but in a single-version database, a repeatable read isolation level is needed to achieve a consistent view. However, in a

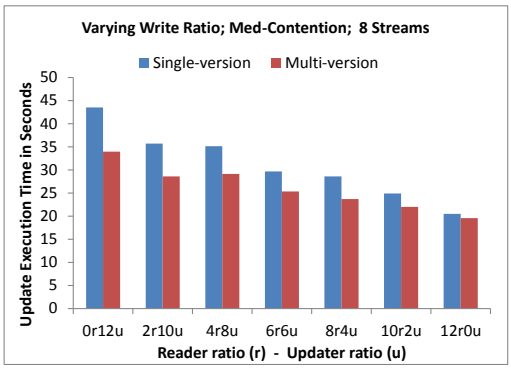
[6]For the medium and low contention workloads similar trends were also observed, but they are omitted in the interest of space.



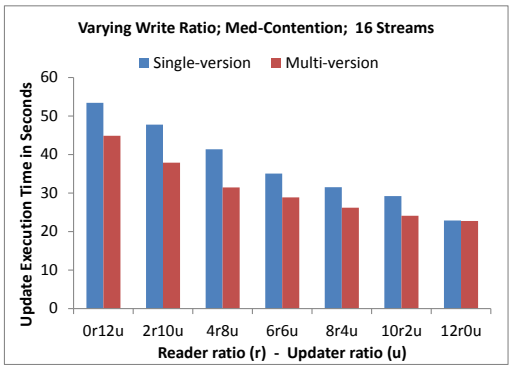
(a) High Contention



(b) High Contention



(c) Medium Contention



(d) Medium Contention

Figure 6: Effects of varying the read/write ratio of short update transactions (update execution time)

single-version database, the execution of long running queries are increased due to frequent lock wait times on records that are being modified. To address this challenge, we rely on what refer to as last committed (LC) isolation level, which is a highly efficient mechanism to support long-running queries by eliminating the read and write contention at the of cost weaker isolation semantics. Basically, under the LC semantics, if a query attempts to read a record that is being modified by an update transaction, it simply reads the last committed version of that record from the log space;[7] thus, LC semantics may not provide a transaction consistent view. We benchmark our proposal against both ends of the spectrum: RR and LC isolation levels for read-only queries in a single-version database.

We further consider two scenarios: first, keeping the number of parallel update transactions constant at 16 while increasing the number of concurrent read-only transactions from 1 to 16 and, second, keeping the number of parallel read-only transactions constant at 16 while increasing the number of concurrent short update transactions from 1 to 16. We present results for high and medium contention and report both read and update throughput as shown in Figure 7.

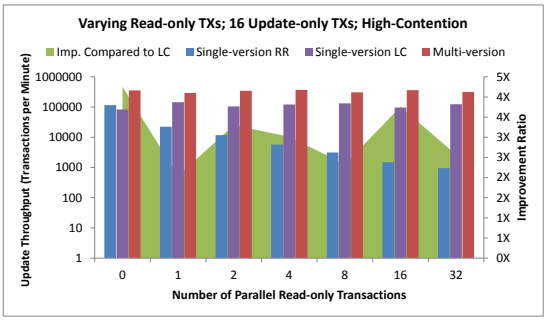
We observe, as the number of long running queries is increased (Figures 7(a) and 7(b)), the contention is increased significantly between the read locks of long running read-only transactions and the write locks (exclusive locks) of update transactions in a single-version database. This increased contention is less significant for single-version with LC isolation (2VCC benefit over LC reaches up to 16x). In fact, once the number of read-only transactions exceeded two, both the lock wait time and the number of deadlocks are increased substantially when a consistent transaction view is required in a single-version database. The clashes between heavy reader transactions and update transactions were avoided by avoiding in-place updates and retaining all the old versions in our 2VCC model. Similar performance advantages were observed when the number of read-only queries is fixed and the number of update transactions is varied. These results are demonstrated in Figures 7(c) and 7(d). We also repeat these experiments for medium contention scenarios as shown in Figures 7(e) and 7(f), although the gap is reduced, yet 2VCC model outperformed the single-version with RR and LC semantics by orders of magnitudes and a factor up to 2x, respectively.

[7]Note that reads from the log does not necessarily mean disk I/O because the last committed versions may be in the log buffer and in most cases it would require only memory access to fetch the old versions.

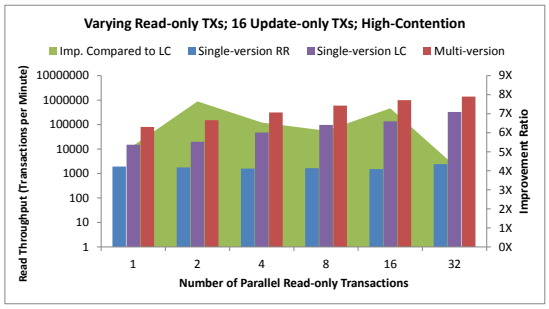
7. RELATED WORK

Database concurrency theory [4], an old age problem, has recently been revived by industry [7, 17, 18] and academia [25, 33, 6, 32, 29, 14, 16, 34] due to hardware trends (e.g., multi-cores and large main memory) and application requirements (e.g., millions of transactions per second in algorithmic trading).

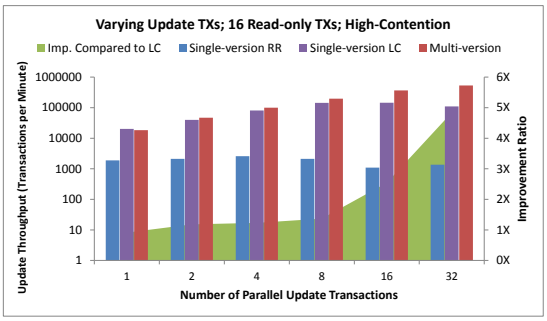
Microsoft Hekaton focuses primarily on optimistic concurrency by assuming that roll backs are inexpensive and conflicts are rare (despite the high contention) [7, 17]. Hekaton avoids the use of a lock manager and relies on read validation to ensure repeatable



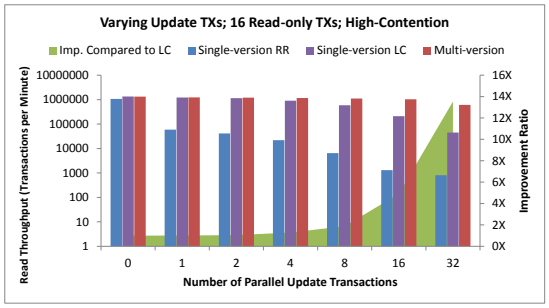
(a) Varying the number of read-only transaction



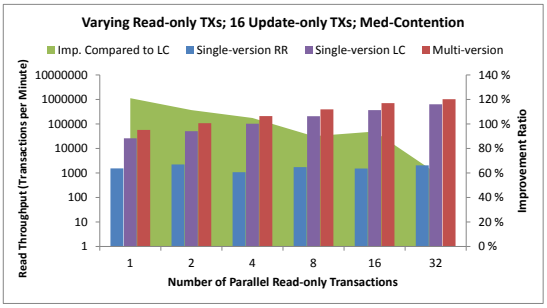
(b) Varying the number of read-only transaction



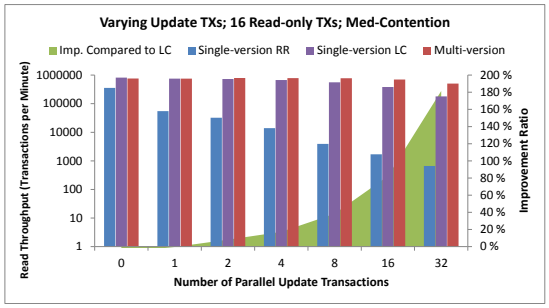
(c) Varying the number of update transaction



(d) Varying the number of update transaction



(e) Varying the number of read-only transaction



(f) Varying the number of update transaction

Figure 7: Effects of varying the number of read-only vs. short update transactions (High to Medium Contention)

reads, performs re-execution of all range queries to achieve serializability, and detects write-write conflicts by using a CAS operator, aborting the second writer to avoid any blocking. This optimistic outlook is implicitly rejected by recent academic work [25, 33, 6, 32, 14, 16] that argues validation is expensive and, thus, advocates a deterministic execution that avoids both the need for a lock manager and validation. However, this deterministic paradigm is practical only in an ideal world where workloads are completely partitionable.

For example, H-Store proposes a disjoint partitioning of the workload across cores, and by assuming that a transaction only access a data within the assigned partition, all transactions for each core are run serially [16] without any concurrency control mechanism. This simple and potentially effective model breaks as soon as a transaction attempts to access data on more than one partition. To address the problem of multi-partition transactions (that are common in many customer settings and unavoidable for any general purpose commercial database systems [7]), speculative execution and lightweight locking mechanisms are proposed [25, 33, 6, 32, 14] for both central and distributed execution. To overcome the challenges of statically assigning transactions/partitions to a core, a data-oriented transaction model is introduced in DORA [24], which continues to rely on a traditional lock manager, but reduces the size of the transaction critical path. However, DORA threading is substantially different from the conventional assignment of a transaction-to-thread (or to an agent) and relies a on producer-consumer model, where the work for each transaction is spread across many threads based on the data access pattern (the assignment is typically determined based on the available indexes). Although DORA could in principle address some of the shortcomings of multi-partition transactions, its adoption is limited due to the high cost of redesigning the transactional model of existing systems. The producer-consumer rendezvous engine must incur the necessary coordination overhead especially when dealing with multi-

socket and non-uniform memory accesses to communicate between threads. None of these data partitioning methods consider a multi-version concurrency model.

Our kV-Indirection model using 2VCC differ from Hekaton by providing both efficient (latch-free) pessimistic and optimistic models. Similar to Hekaton, it does not focus on a specific type of workload, like partitionable workloads [25, 33, 6, 32, 14, 16], and does not require redesigning the transactional threading model of existing systems [24].

In [8], Dou et al. propose specialized index structures and algorithms that support querying of historical data in flash-equipped sensor devices. Since the sensor devices have limited memory capacity (SRAM) and the underlying flash devices have certain limitations, there are challenges in maintaining and querying indexes.

Many specialized indexes for (transient) versioned and temporal data have been proposed. A comprehensive survey of temporal indexing methods is provided in [27]. Tree based indexes on temporal data include the multi-version B-tree [3], Interval B-tree [2], Interval B+-tree [5], TP-Index [30], Append-only Tree [11] Monotonic B+tree [9], and distributed multi-version B-Tree [31]. Efficiently indexing data with branched evolution is discussed by Jouni et al. [15], who build efficient structures to run queries on both current and historical data. A transient versioned database introduced by Mohan et al. aim to isolate reader and writer transactions, similar to our technique, at the cost of slightly outdated but a consistent snapshot for readers[21]. However, our scope is not limited to supporting consistent snapshot read and we focus on the general problem of (latch-free) multi-version concurrency control. Our approach avoids challenges associated with advancing outdated snapshot reads presented in [21] by explicitly retaining the history of all records and constructing the database snapshot based on the time records were inserted or deleted. For example, the maintained transient snapshot can only advance when there are no outstanding read queries in the system (the age of snapshot is proportional to the duration of longest running queries in the system) or the system is forced to explicitly maintain as many concurrent snapshots as there are active readers in the system [21].

Specialized transaction time database systems such as Immortal DB [19, 20] provide high performance for temporal applications. Lomet et al. [20] describe how a temporal indexing technique, the TSB-tree, is integrated into SQL Server. The paper also describes an efficient page layout for multi-version databases.

8. CONCLUSIONS

We presented our commercial database working prototype enhanced with our multi-version concurrency control. We described a kV-Indirection mapping that translates a version-independent logical ID into the version-dependent RID. Our proposed 2V-Indirection mapping maintains two RIDs: the *cRID* that points to the most recently committed version of the record and the *uRID* that points to the most recent uncommitted version of the record. Indexes no longer point to RIDs, but to LIDs instead. And each index may point to at least two versions of a record through a single LID via a (*cRID*, *uRID*) pair.

Exploiting the 2V-Indirection mapping not only has the advantages of limiting updates or deletes to affected indexes but also provides a mechanism for implementing multi-version concurrency control (such as 2VCC) because at any point in time the 2VCC requires efficient and fast access to both last committed and last uncommitted values, and the indirection mapping allows the direct access to this information. In addition, our proposed latch-free pessimistic/optimistic 2VCC can efficiently be implemented using 2V-Indirection. In general, kVCC can be generalized to kV-Indirection in order to access the last k versions of the index efficiently.

9. REFERENCES

[1] Postgresql: Open source object-relational database system. `http://www.postgresql.org/`.

[2] C.-H. Ang and K.-P. Tan. The interval B-tree. *Inf. Process. Lett.*, 53(2):85–89, Jan. 1995.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-Tree. *VLDB J.*, 5(4):264–275, 1996.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] T. Bozkaya and M. Özsoyoğlu. Indexing valid time intervals. *Lecture Notes in Computer Science*, 1460:541–550, 1998.

[6] K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 2012.

[7] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[8] A. J. Dou, S. Lin, and V. Kalogeraki. Real-time querying of historical data in flash-equipped sensor devices. In *IEEE Real-Time Systems Symposium*, pages 335–344, 2008.

[9] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B+-tree. In *Temporal Databases*, pages 433–456. 1993.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[11] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Trans. on Knowledge and Data Eng.*, 5(3):496, June 1993.

[12] F. D. Hinshaw, C. S. Harris, and S. K. Sarin. Controlling visibility in multi-version database systems, 2007. US 7305386 Patent, Netezza Corporation.

[13] DB2 10 for z/OS. `ftp://public.dhe.ibm.com/software/systemz/whitepapers/DB210_for_zOS_Upgrade_ebook.pdf`.

[14] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 603–614. ACM, 2010.

[15] K. Jouini and G. Jomier. Indexing multiversion databases. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 915–918, New York, NY, USA, 2007. ACM.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[17] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[18] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering*, ICDE '13, Washington, DC, USA, 2013. IEEE Computer Society.

[19] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD*

international conference on Management of data, SIGMOD '05, pages 939–941, New York, NY, USA, 2005. ACM.

[20] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. *Proc. VLDB Endow.*, 1(1):870–881, Aug. 2008.

[21] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 124–133, New York, NY, USA, 1992. ACM.

[22] Oracle database 11g workspace manager overview. `http://www.oracle.com/technetwork/database/twp-appdev-workspace-manager-11g-128289.pdf`.

[23] Oracle total recall/flashback data archive. `http://www.oracle.com/technetwork/issue-archive/2008/08-jul/flashback-data-archive-whitepaper-129145.pdf`.

[24] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[25] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, 2012.

[26] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *Proc. VLDB Endow.*, 6(11):997–1008, Aug. 2013.

[27] Salzberg and Tsotras. Comparison of access methods for time-evolving data. *CSURV: Computing Surveys*, 31, 1999.

[28] C. M. Saracco, M. Nicola, and L. Gandhi. A matter of time: Temporal data management in DB2 for z/OS, 2010.

[29] R. Shaull, L. Shrira, and H. Xu. Skippy: A new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 637–648, New York, NY, USA, 2008. ACM.

[30] H. Shen, B. Chin, and O. H. Lu. The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. In *In Proceedings of the Tenth International Conference on Data Engineering*, pages 274–281. IEEE, 1994.

[31] B. Sowell, W. Golab, and M. A. Shah. Minuet: A scalable distributed multiversion B-tree. *Proc. VLDB Endow.*, 5(9):884–895, May 2012.

[32] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[33] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In Candan et al. [6], pages 1–12.

[34] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

APPENDIX

This section contains the proofs of pessimistic and optimistic concurrency theorems.

THEOREM 1. *The parallel implementation of 2V2PL is conflict-free serializable.*

PROOF. This is directly follows by the fact that every history of transactions produced by 2V2PL is 1-serializable (1SR) [4], and our implementation of the 2VCC protocol using 2V-Indirection does not change its semantics nor the order in which locks are acquired. □

THEOREM 2. *The proposed latch-free pessimistic* 2VCC *is conflict-free serializable.*

PROOF. Our latch-free pessimistic 2VCC prevents any write operation on items read by active transactions; therefore, the certification is always trivially satisfied.

The proposed 2VCC with write-write conflict is also 1SR because it aborts and avoids queuing transactions that discover their writeset is being changed by other active transactions. Since the only modification is to abort certain transactions, there are no lost updates.

Finally, the pessimistic 2VCC certification guarantees that the version read by transactions throughout the transaction remains unchanged throughout the life of the transaction. The only subtle point is to ensure that certification itself is done in a critical section, meaning once an item in the writeset is certified no new readers are admitted. This is achieved by setting the read counter of each item in the writeset to -1 if the counter is 0. If the read counter is greater than 0, then the certification fails and the transaction is aborted in order to ensure read stability for active transaction without blocking the certifying transaction. Hence, all histories produced by our proposed 2VCC are 1SR. □

THEOREM 3. *The proposed optimistic* 2VCC *is conflict-free serializable.*

PROOF. The optimistic 2VCC with validation guarantees that the version read by transaction remains unchanged throughout the life of transaction. The only subtle point is to ensure that validation itself is done in a critical section, meaning once an item in the readset is validated no other transaction can change the item until validation is completed. This is achieved by holding a read lock for every item in the readset before validating it and releasing the read lock after transaction is committed. Hence, all histories produced by our proposed optimistic 2VCC are 1SR. □

THEOREM 4. *The proposed latch-free optimistic* 2VCC *is conflict-free serializable.*

PROOF. Our latch-free optimistic 2VCC is based on the read-counter mechanism preventing any write operation on items read by active transactions; therefore, the certification is always trivially satisfied.

The proposed 2VCC with write-write conflict is also 1SR because it aborts and avoids queuing transactions that discovered that their writeset is being changed by other active transactions. Since the only modification is to abort certain transactions, there are no lost updates.

The optimistic 2VCC ensures the correctness of the validation by guaranteeing that the version read by transaction remains unchanged throughout the life of transaction. The only subtle point is to ensure that validation itself is done in a critical section, meaning once an item in the readset is validated (its reader counter incremented) no other transaction can change the item until validation is completed. This is achieved by incrementing the counter for items in the readset before validating, and only decrement them once the transaction is committed.

Finally, the optimistic 2VCC certification guarantees that the older versions read by other active transaction remains unchanged throughout the life of transaction. The only subtle point is to ensure that certification is also done in a critical section, meaning once an item in the writeset is certified no new readers are admitted. This is achieved by setting the read counter of each item in the writeset to -1 if the counter is 0. If the read counter is greater than 0, then the certification fails and the transaction is aborted in order to ensure read stability for active transactions without blocking the certifying transaction. Hence, all histories produced by our proposed optimistic 2VCC are 1SR. □