Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems

Michael Freitag
Technische Universität München
freitagm@in.tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Pure in-memory database systems offer outstanding performance but degrade heavily if the working set does not fit into DRAM, which is problematic in view of declining main memory growth rates. In contrast, recently proposed memory-optimized disk-based systems such as Umbra leverage large in-memory buffers for query processing but rely on fast solid-state disks for persistent storage. They offer near in-memory performance while the working set is cached, and scale gracefully to arbitrarily large data sets far beyond main memory capacity. Past research has shown that this architecture is indeed feasible for read-heavy analytical workloads.

We continue this line of work in the following paper, and present a novel multi-version concurrency control approach that enables a memory-optimized disk-based system to achieve excellent performance on transactional workloads as well. Our approach exploits that the vast majority of versioning information can be maintained entirely in-memory without ever being persisted to stable storage, which minimizes the overhead of concurrency control. Large write transactions for which this is not possible are extremely rare, and handled transparently by a lightweight fallback mechanism. Our experiments show that the proposed approach achieves transaction throughput up to an order of magnitude higher than competing disk-based systems, confirming its viability in a real-world setting.

PVLDB Reference Format:
Michael Freitag, Alfons Kemper, and Thomas Neumann.
Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. PVLDB, 15(11): 2797 - 2810, 2022.
doi:10.14778/3551793.3551832

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/freitmi/experiments-vldb2022.

1 INTRODUCTION

Over the past decade, we have observed a divergence of relational database system design into two competing species. On the one hand, there are pure in-memory systems that offer unprecedented performance, but do not handle large data sets well. On the other hand, traditional disk-based systems do scale to data sets much larger than main memory transparently and gracefully, but due to a variety of factors they exhibit suboptimal performance even if all data fits into main memory. We argue that this dichotomy has become obsolete, as recent hardware trends make it feasible and in fact necessary to move towards a novel type of memory-optimized disk-based database architecture which unifies the performance of an in-memory system with the scalability of a disk-based system.

As in-memory databases were conceived, it was assumed that main memory sizes would rise in accord with the amount of data in need of processing for the foreseeable future [22, 23]. In reality, however, affordable main memory sizes have increased only marginally since then, effectively reaching a plateau of at most a few TB [46]. In view of this development, serious concerns have been raised about the viability of pure main memory systems and we currently observe a renewed interest in disk-based databases [41, 46].

However, traditional disk-based database systems do not meet the expectations for such a memory-optimized system. This is due to several profound differences between modern hardware platforms, and the hardware platforms for which these traditional systems were originally designed. First of all, main memory has become much more plentiful, with database servers routinely having access to hundreds of GB to several TB of RAM. Furthermore, persistent solid-state storage media have made several quantum leaps in recent years, and now achieve excellent IO throughput up to multiple GB per second, orders of magnitude higher than the previously used rotating disks. Finally, modern multi-core CPUs allow for massive parallelism on a single machine, both in analytical and in transactional workloads.

Past research has shown that in order to fully exploit the capabilities of such modern hardware, it is required to redesign the majority of components of a typical disk-based database system [20, 35, 46]. The result of this process is a novel type of memory-optimized disk-based system which offers excellent performance as long as the working set fits into main memory, while scaling transparently and gracefully to the out-of-memory case. Of course, such a system will usually fall short of a pure in-memory system in terms of maximum attainable raw performance, but this is offset by its far superior robustness since most in-memory systems simply cease operation when they run out of memory [10, 20, 23, 61]. In this spirit, we recently presented the Umbra general-purpose database system as the evolution of HyPer towards this kind of system, and discussed key design characteristics that allow it to achieve close to in-memory performance on analytical workloads [46].

In the following paper, we build upon this foundation and present a novel multi-version concurrency control (MVCC) approach that allows a memory-optimized disk-based system to achieve the same goal on transactional workloads. Our approach is guided by the key insight that due to the large amount of RAM available to a modern database system, the versioning information for the vast majority of transactions can easily be maintained entirely in-memory. For

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551832

instance, any given TPCC transaction updates substantially fewer than 100 tuples. We extend the buffer manager to transparently maintain a minimal mapping layer which associates logical data objects on the database pages with memory-resident versioning information in a decentralized way. Since none of this data needs to be written to persistent storage, the impact of concurrency control on the overall system performance is minimized. Obviously, extremely large write transactions for which this approach is not feasible can still realistically occur, for example during database bulk loading. However, such transactions are rare during regular operation, and we propose a lightweight versioning mechanism that stores some minimal information on each database page in order to isolate bulk operations from concurrent readers.

All techniques presented in this paper have been integrated and evaluated within the general-purpose database system Umbra [46]. We thus provide a detailed architecture blueprint for the transaction processing infrastructure within a general-purpose disk-based database system. It makes full use of modern hardware, allowing it to combine both excellent scalability and excellent performance within a single system. As we will demonstrate in our experimental evaluation, the proposed system architecture achieves transaction throughput numbers up to an order of magnitude higher than traditional disk-based database systems, which further confirms the viability of the memory-optimized disk-based paradigm.

In summary, the key contributions of this paper are:

(1) A novel, low-overhead MVCC approach for disk-based systems which exploits that most versioning information does not need to be persisted to disk. This prevents bloating of database files, and tremendously expedites transaction processing for the common case of small transactions.
(2) A transparent fallback mechanism which allows the system to support arbitrarily large write transactions whose footprint exceeds the available main memory size.
(3) Full integration and thorough evaluation of the proposed approach within the general-purpose Umbra DBMS, validating that the proposed system architecture is viable in a real-world setting.

The remainder of this paper is structured as follows. In Section 2 we present the architecture of a prototypical memory-optimized disk-based system as well as essential background information on multi-version concurrency control. Subsequently, we discuss our MVCC approach in Section 3, and conduct a detailed experimental evaluation of our system in Section 4. Finally, we outline relevant related work in Section 5 and draw conclusions in Section 6.

2 FOUNDATIONS

In the following, we outline key architectural characteristics of a modern memory-optimized disk-based database system. Furthermore, we provide some essential background on multi-version concurrency control and its implementation within a practical system.

2.1 Memory-Optimized Disk-Based Systems

As outlined above, previous work has demonstrated that most components of a traditional disk-based database system exhibit suboptimal performance on modern hardware [5, 19, 20, 35, 36, 46, 62]. Of particular interest to our discussion are the buffer manager and the logging subsystem that are found in virtually all disk-based systems [21], since their characteristics directly influence the design of our proposed approach.

Disk-based system designs that rely on a buffer manager provide an intelligent global replacement strategy across all persistent data structures, and hide the complexities of IO buffering behind a simple interface. These characteristics are desirable within a memory-optimized system as well, but classical buffer managers incur a number of major inefficiencies mostly related to excessive global synchronization [35]. Recently proposed storage managers such as LeanStore overcome these problems through a number of key techniques, which minimizes their overhead in the common case that the entire working set fits into main memory [35]. Most importantly, *pointer swizzling* allows the translation from logical page identifiers to physical buffer frame addresses to be implemented in a decentralized way [15]. Here, any reference to a database page is represented as a fixed-size integer called a *swip*. A swip can either store a logical page identifier in case the referenced page is currently residing on persistent storage, or a physical pointer to a buffer frame in case it is currently cached in the buffer manager [35, 46]. In contrast to traditional designs, pointer swizzling allows cached pages to be accessed with minimal overhead and without any global synchronization since no lookup into any centralized translation data structure is required. Efficient thread synchronization is achieved through a lightweight and highly scalable *hybrid mutex* implementation that supports both pessimistic and optimistic latching modes [5, 36].

Most disk-based database systems employ some form of ARIES-style write-ahead logging in order to guarantee durability of committed transactions and enable recovery from system failure [20, 21]. Similar to buffer management, ARIES is attractive since it offers a wide range of important features, but it suffers from poor scalability [19, 45]. All worker threads must append their log records to a single global log which requires expensive synchronization. More recently, *decentralized logging* approaches have been proposed which retain the core features of ARIES but minimize its overhead [20, 62]. Conceptually, these approaches assign a private thread-local log to each worker threads, while retaining a sufficient global ordering of log records for recovery [20, 62].

2.2 Multi-Version Concurrency Control

Robust and well-defined transaction isolation is one of the major selling points of a general-purpose DBMS. Historically, concurrency control algorithms often relied on locking to ensure transaction isolation, e.g. the well-known *two-phase locking* protocol in which the database maintains read and write locks to coordinate conflicting transactions [64]. However, locking-based approaches typically suffer from major scalability problems as readers can block writers and vice-versa [47]. In contrast, *multi-version concurrency control* allows much higher concurrency between readers and writers [4, 45, 47, 64]. Under MVCC any update of a data object creates a new version of that object while initially retaining the old version, so that concurrent readers can still access it. Consequently, writers can proceed even if there are concurrent readers, and read-only transactions will never have to wait at all. Since this is a highly desirable property, MVCC has emerged as the concurrency control algorithm of choice both in disk-based systems such as

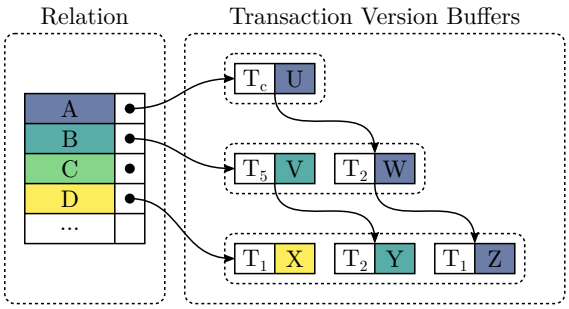


Figure 1: Illustration of decentralized version maintenance in an in-memory system. Relations store the most recent version of a tuple which is linked to a chain of before-images stored in the transaction version buffers.

MySQL [1], SQL Server [43], Oracle [2], or PostgreSQL [16, 51], and in in main memory databases such as HyPer [47], Hekaton [10, 30], SAP HANA [13, 56], or Oracle TimesTen [29].

Our approach is based on the decentralized MVCC implementation first proposed for the HyPer in-memory database system [47, 65]. We choose this approach since it introduces little overhead and requires only minimal synchronization, which matches the general design objectives for a memory-optimized disk-based system. Furthermore, it allows for a garbage collection scheme that is well-optimized and has been proven to be highly effective [5, 47, 65]. The key idea of this approach is to perform updates in-place, and copy the previous values of the updated attributes to the private version buffer of the updating transaction (cf. Figure 1). These *before-images* form a chain for each tuple which possibly spans the version buffers of multiple transactions. The entries in a given chain are ordered in the direction from newest to oldest change, and can be traversed in order to reconstruct a previous version of a tuple. Outdated versions that are no longer relevant to any transaction are garbage collected continually, which is facilitated by having them clustered within the transaction version buffers.

Each new transaction is associated with two timestamps, namely a unique transaction identifier $T_{id} \in \{2^{63}, \ldots, 2^{64} - 1\}$ and a start timestamp $T_{start} \in \{0, \ldots, 2^{63} - 1\}$ which corresponds to the most recent committed transaction. Together, these timestamps determine the range of versions that are visible to the transaction. During commit processing, transactions draw the next available commit timestamp T_{commit} from the same sequence that is used to generate the start timestamps. Each version v stores a single timestamp $T(v)$ that is initially set to the identifier T_{id} of the transaction that created the version, and later updated to the commit timestamp T_{commit}. Thereby, the uncommitted state of a tuple is initially only accessible to the transaction that modified the tuple, and all other transactions reconstruct an old state of the tuple by traversing the chain of before-images associated with the tuple.

Specifically, a transaction that accesses a tuple first reads the most recent state of the tuple. Subsequently it traverses all versions v in the corresponding version chain, applying the respective before-images along the way until the following stopping criterion is met:

$$v = \emptyset \ \vee \ T(v) = T_{id} \ \vee \ T(v) \leq T_{start}.$$

The first term of the disjunction simply terminates traversal if there are no more entries in a chain, the second term allows a transaction to see its own changes, and the third term ensures that transactions reconstruct the state of a tuple that was committed at the time of transaction begin. Consider, for example, the scenario depicted in Figure 1 and assume that there is an active transaction with $T_{id} = T_c$ and $T_{start} = T_2$. A scan of the relation would then yield the values A (since the update $U \rightarrow A$ was performed by T_c itself), V (since the update $V \rightarrow B$ by T_5 is invisible due to $T_5 > T_{start}$, but the update $Y \rightarrow V$ by T_2 is visible to T_c due to $T_2 \leq T_{start}$), C (since the tuple is unversioned), and D (since the update $X \rightarrow D$ by T_1 is visible to T_c due to $T_1 \leq T_{start}$).

3 MEMORY-OPTIMIZED MVCC

Main memory databases offer superior performance over traditional disk-based systems, and consequently most recent work on high-performance MVCC implementations has focused on the in-memory case [53, 58, 65]. This allows several key simplifications that are not immediately applicable to a disk-based system, such as assuming that all relation and version data will reside in-memory at all times. In comparison, little attention has been devoted to exploring novel MVCC approaches in a disk-based setting [44, 54]. Existing systems such as PostgreSQL still rely on MVCC implementations that were devised decades ago [16, 65], and thus fail to optimally exploit the capabilities of modern hardware (cf. Section 1). In particular, these systems often assume that almost no database pages and version data at all can be maintained in-memory. We bridge this gap and present a novel approach that is well-suited for a memory-optimized disk-based system.

Our proposal is based on the fundamental observation that the vast majority of write transactions encountered during regular transaction processing are extremely small. In particular, they generate comparably few versions which consume many orders of magnitude less main memory than the amount typically available on modern hardware. In these cases, a memory-optimized disk-based system can easily maintain all versioning information required by MVCC entirely in-memory. By carefully relying on the logging subsystem that is already in place anyway, it is in fact possible to ensure that this versioning information is truly ephemeral and will never be written to persistent storage. As discussed in further detail in Section 3.1, we can therefore design our MVCC approach for the most part like a pure in-memory implementation and adopt many of the existing innovations and optimizations for this scenario. Not only does this lead to excellent performance in the common case that the working set fits into main memory, but it also dramatically reduces the amount of redundant data that has to be written to disk, since generally only the most recent version of a data object will be present on the actual database pages. Of course, large write transactions with a footprint larger than main memory do realistically occur, e.g. during bulk loading, and an efficient technique for providing transaction isolation in these cases is required. We argue that the main objective here is to allow read-only OLAP transactions to continue unimpeded by a concurrent bulk operation, and consequently present a lightweight versioning scheme that does not consume any additional main memory as a transparent fallback mechanism (cf. Section 3.2).

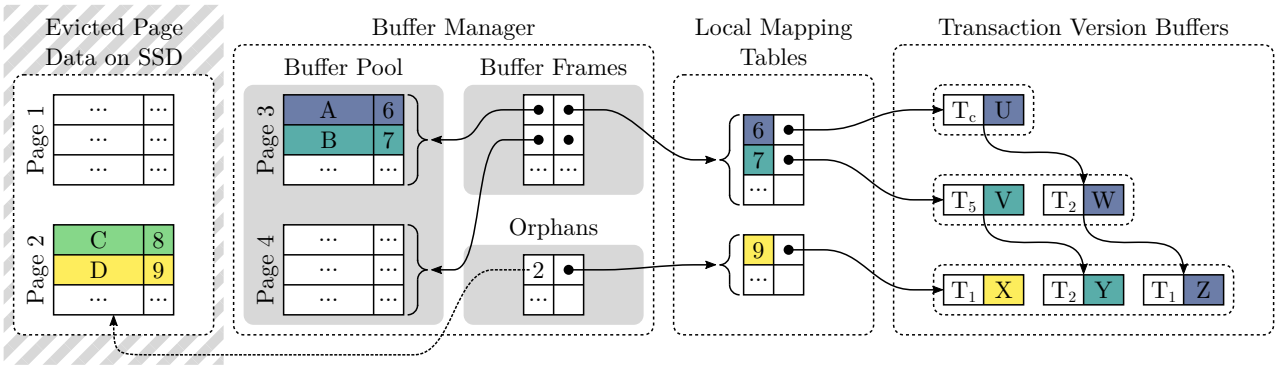


Figure 2: Overview of version maintenance within our proposed approach. Solid arrows represent physical pointers while dotted arrows indicate logical references. Database pages store only the most recent version of a data object, and all additional versioning information resides exclusively in-memory. The buffer manager maintains a local mapping table for each versioned database page which associates stable data object identifiers with version chains.

3.1 In-Memory Version Maintenance

A buffer-managed database system typically employs a steal policy, i.e. database pages containing uncommitted changes may be evicted to persistent storage. This is essential in order to allow the system to scale gracefully beyond main memory, but poses a key challenge when integrating any MVCC approach within such a system. The logical data objects comprising the database contents are stored on database pages which may be evicted from main memory at any time, yet at the same time they have to be associated with their respective version chain in some way.

Existing systems resolve this challenge in a wide variety of different ways, but unfortunately none of these solutions are immediately applicable to a memory-optimized disk-based system. A straightforward option is to physically materialize all versions of a data object within the same storage space, e.g. all versions of a tuple within the corresponding relation [65]. Although this approach is taken by established systems such as the disk-based PostgreSQL [51] and the in-memory Hekaton [10, 31], we argue that it leads to suboptimal resource utilization and performance. Since they are maintained within the same physical storage space, all versions necessarily become part of the persistent database state that is written to disk, whereas only the most recent version of a data object is actually required to be durable if write-ahead logging is used. Thus, a large amount of redundant data is persisted leading to severe write amplification. Append-only storage was extremely useful historically since it allowed MVCC to be implemented with very few in-memory data structures, but this is no longer necessary or desirable on modern hardware.

In order to avoid such write amplification it makes sense to store any additional versions separately from the most recent version of a data object and only include the latter in the persistent database state [65]. Variations of this basic scheme are widespread in existing systems such as SQL Server, Oracle DB, MySQL, SAP HANA, or HyPer [65]. Disk-based systems typically employ a global version storage data structure which maps stable logical identifiers to actual versions. Each version of a data object, including the master version that is persisted to disk, contains such a logical identifier as an additional attribute to form a link to the next version in the corresponding chain. However, a global data structure can easily become a major scalability bottleneck [6]. Moreover, versions can only be accessed through a non-trivial lookup into this data structure, which makes even uncontended version chain traversals rather expensive. In contrast, pure in-memory systems like HyPer store versions in a decentralized way, and use raw pointers to directly link individual versions within a chain instead of relying on logical identifiers [47]. This approach is of course much more efficient, but a minimal logical mapping layer is still required in our case since we cannot store raw pointer on database pages [35].

We thus propose the following high-level architecture for an efficient MVCC implementation within a memory-optimized disk-based system (cf. Figure 2). Database pages that can be evicted to disk store only the most recent version of a data object. If a given database page contains any versioned data objects at all, we maintain a *local mapping table* for this specific database page exclusively in-memory. While the page is pinned in the buffer manager, the respective buffer frame stores a pointer to the associated mapping table, allowing direct access without consulting any global data structures. The table maps suitable stable logical identifiers of the versioned data objects (e.g. tuple identifiers) residing on the page to the corresponding version chains which are maintained in-memory. Any buffer-managed data structures are thus decoupled from the actual version chain implementation and overall MVCC protocol, allowing the latter to be chosen flexibly from a range of existing in-memory MVCC implementations. As discussed previously, we argue that a decentralized version maintenance scheme is best suited for our use-case and thus adopt the MVCC approach outlined in Section 2.2. Garbage collection is based on the highly scalable Steam algorithm devised for in-memory systems, albeit with some extensions to account for the local mapping tables [6].

3.1.1 Version Maintenance. As outlined above, the logical versioning information required by MVCC is physically highly decentralized within our proposed system. Central to our approach are the local mapping tables that establish a link between data objects on a page and their associated version chains, if any. A pointer to the

mapping table is stored in the corresponding buffer frame while a page resides in the buffer pool, and can be accessed through the same latching protocol that is already in place to access the page itself (cf. Section 2.1). That is, no additional synchronization overhead is introduced since the system can request access to both the database page and the associated mapping table with a single latch acquisition [46]. The buffer manager can still evict arbitrary pages as usual, but only the page data is actually written to disk. Any orphaned mapping tables are retained in-memory by the buffer manager within a hash table that maps the identifier of the corresponding page to the mapping table. Once a page is loaded back into memory at a later point in time, the buffer manager probes this hash table to check whether a mapping table exists for the page and reattaches it to the respective buffer frame if necessary. A mapping table entry only stores a pointer to the corresponding version chain that is maintained separately within the transaction-local version buffers (cf. Section 2.2). This is extremely useful, since it allows transactions to efficiently update the timestamps of their versions during commit processing. Specifically, we do not have to update any mapping tables which would require latching database pages.

Since we continuously reclaim expired versions, the vast majority of pages will have no attached mapping table. Semantically, this means that there are no version chains and thus the most recent state of all data objects on that page is globally visible to all transactions. Note that such a page may still contain logically deleted data objects that have not yet been physically reclaimed. A mapping table is initialized lazily once a write transaction actually modifies a data object on a previously unversioned page. Subsequently, writers can insert mappings into this table in order to associate newly created version chains with currently unversioned data objects, or retrieve existing mappings to apply further modifications to an already versioned data object. When reading from a versioned page, a lookup into the mapping table is required to determine whether a version chain exists for a given data object. These lookups are only performed in case that a page actually contains versioned data objects, which we can determine at the granularity of pages by checking whether a mapping table is present. In all other cases we can employ an optimized scan implementation that unconditionally reads all non-deleted data objects from a page, minimizing the overhead of our approach.

Consider, for example, the situation illustrated in Figure 2 which mirrors the in-memory scenario shown previously in Figure 1. Pages 1 and 4 have no associated local mapping table and thus contain no versioned data objects. In contrast, pages 2 and 3 do contain versioned data objects which is indicated by the presence of a local mapping table for these pages. Page 3 is currently loaded into the buffer pool, so the respective buffer frame contains a pointer to this mapping table. Page 2 is currently evicted, for which reason the buffer manager remembers the pointer to the associated mapping table within the separate orphan table. It will be reattached to the corresponding buffer frame once page 2 is loaded back into memory.

3.1.2 Garbage Collection. Like all MVCC implementations, our approach must ensure that outdated versioning information is reclaimed in a timely manner to prevent the system from quickly running out of memory. For this purpose, we adapt the Steam garbage collection approach to our proposed system architecture. This approach has been shown to exhibit superior performance in comparison to a number of alternative garbage collection schemes [6]. Moreover, Steam can be integrated smoothly into our proposed system since it assumes a similar versioning protocol [47].

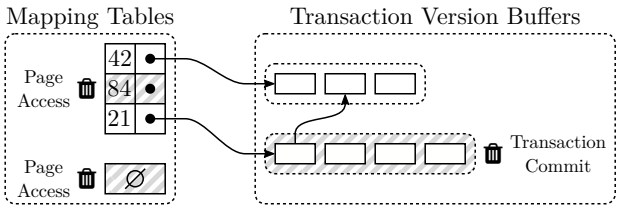


Figure 3: Illustration of garbage collection within our proposed approach. Empty version chain mappings and mapping tables are pruned on page access. Version buffers of globally visible transactions are reclaimed upon transaction commit.

Garbage accumulates in two different forms within our approach (cf. Figure 3). First, the version buffers maintained by the transactions must be reclaimed once they are no longer relevant for any active transaction. Second, each mapping table attached to a database page must be pruned regularly until there are no more versioned data objects on the page and the mapping table itself can be discarded. We deliberately split responsibility for these different manifestations of garbage between several components of our system in order to exploit the decentralized nature of our approach and minimize the communication overhead incurred by garbage collection. Specifically, during commit processing the transaction version buffers are cleaned up but the mapping tables are not modified in any way, since this would require latching the corresponding database pages. Instead, they are pruned whenever a page is accessed during regular query processing and we have to acquire a suitable latch on the page anyway. Of course, on its own this only guarantees timely garbage collection of the mapping tables for hot pages that are frequently accessed, and we additionally rely on the buffer manager to prune the mapping tables of cold pages. Finally, as proposed by Boettcher et al. individual obsolete versions can be pruned eagerly during version chain traversal in order to ensure that the number of versions per data object is limited to the number of active transactions [6]. This serves to minimize the number of versions that have to be retained in the presence of long-running readers, which otherwise could quickly cause obsolete versions to accumulate.

In order to facilitate garbage collection of the transaction-local version buffers, we maintain active and recently committed transactions in two ordered linked lists (cf. Figure 4). A transaction is appended to the active list when it begins, and moved to the recently committed list when it commits so that the versions it created can be retained as long as they are still relevant to other active transactions. Read-only transactions that did not create any versions can be discarded immediately upon committing [6, 47]. As part of the commit processing, we reclaim all recently committed transactions with a commit timestamp that is less than the minimum start timestamp of any active transaction. Note that we may unlink the last version of a chain during this process, resulting in an empty

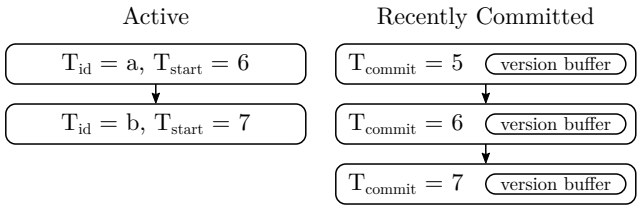


Figure 4: Transaction lists for garbage collection. Once transaction *a* from the active transaction list commits, we can reclaim the oldest two recently committed transactions.

version chain that is still associated with a data object through a local mapping table. A data object which has an empty associated version chain is by definition globally visible, so this obviously does not affect correctness. However, we still want to remove such empty mappings as fast as possible in order to limit the size of the local mapping tables and retain high scan performance.

For this purpose, we extend the regular page-level maintenance processing such as page compaction that is performed by a typical relation and index implementation whenever it acquires a latch on a page. Before any implementation-specific maintenance work is done, we first attempt to prune any local mapping table that may be associated with the page. That is, we iterate over the entries within the mapping table, discard mappings that reference empty version chains, and finally remove the entire mapping table if it has become empty. In order to avoid excessively many traversals of the local mapping tables, we track some minimal statistics about the number of empty version chains within the mapping tables, and only attempt pruning if the fraction of empty version chains within a mapping table exceeds a certain threshold, e.g. 5 %.

Usually, write activity in the database will be focused on a comparably small number of hot pages and the corresponding mapping tables will be continuously pruned. Nevertheless, it is possible that versioned pages become cold and are not accessed anymore, in which case they may even be evicted entirely by the buffer manager. In order to limit the number of orphaned mapping tables within the buffer manager, the worker threads employ the same pruning approach on the orphaned mapping tables whenever they perform disk IO within the buffer manager. Since we handle large write transactions through an entirely separate mechanism that does not generate any physical versioning information at all, we expect eviction of versioned pages to be extremely unlikely during regular operation. Correspondingly the worker threads will rarely, if at all, have to perform garbage collection duties on cold pages.

3.1.3 Recovery. As outlined briefly above, all versioning information maintained by our MVCC implementation is ephemeral, meaning that it will never be written to disk and is thus lost in case of system failure. This does not affect correctness, however, since this information is only necessary in order to provide transaction isolation during forward processing. Recovery exclusively relies on the information captured in the write-ahead log generated during forward processing, and does not require any concurrency control. After recovery, the database is in a globally consistent state without any active transactions, and consequently no version chains at all are present within the system. This allows the state of our MVCC implementation to be initialized in the same way every time the system is (re-)started, e.g. the transaction timestamp counters always start at their initial values listed in Section 2.2 and the transaction lists are initially empty.

This property of our approach allows us to almost completely decouple the logging and concurrency control subsystems, which greatly simplifies the overall system design. One important exception to this strict separation is transaction rollback, which has to be implemented carefully to account for both components. In particular, an ARIES-style write-ahead logging protocol requires that we write exactly one compensation log record whenever we revert an existing log record, so that recovery can skip these log records in the undo pass [45]. Since a single log record may encode changes to multiple data objects, our system must implement rollback by scanning log records. When reverting changes to a data object, the most recent version of that object on the database page is overwritten with the before-image stored in the log record, and the corresponding irrelevant version is unlinked from the respective chain. This differs from a pure in-memory system which does not require undo logging and can thus simply scan the version buffers and revert all changes to the affected data objects individually.

3.1.4 Implementation Details. In order to ensure that garbage collection scales well, our actual implementation avoids centralized data structures wherever possible, which is especially important on multi-socket systems. Specifically, we maintain additional active and recently committed transaction lists locally within each worker thread as proposed by the Steam framework [6]. Small write transactions are pinned to a single worker thread, and subject to thread-local garbage collection according to the approach outlined above. Larger transactions can be executed on multiple worker threads, and are maintained within the global transaction lists for garbage collection. These are protected through a regular latch, but this does not constitute a major scalability bottleneck since it is unlikely that a large number of multi-threaded transactions execute simultaneously (cf. Section 3.2). Internally, a multi-threaded transaction maintains a separate version buffer for each worker thread, so that version allocation requests do not result in contention on a single centralized data structure.

Since garbage collection directly operates on individual versions without accessing them through the local mapping tables, it is possible that another transaction tries to access the same versions concurrently during regular forward processing. In order to ensure proper synchronization in this case, each individual version chain contains a lightweight latch implemented using a single integer which has to be acquired for any modification of the version chain [5]. All modifying operations are implemented carefully in such a way that readers can still traverse version chains without latching them, using only atomic operations.

3.2 Out-of-Memory Version Maintenance

Obviously, the in-memory versioning approach discussed in the previous section fails for transactions which generate more version data than the amount of available working memory. It is optimized for throughput in OLTP workloads, where we expect a high influx of concurrent but comparably small transactions. In contrast, OLAP workloads typically consist of expensive read-only queries, with

occasional ingestion of large amounts of data. Additionally, a user may issue large write transactions at any point during regular operation, for example intentionally for administrative purposes, or unintentionally due to a buggy query. In all of these cases a robust mechanism is required to process such bulk operations and allow the system to scale gracefully beyond main memory.

We argue that unlike the general-purpose MVCC implementations in existing disk-based systems, our fallback mechanism only has to support limited concurrency which allows for a streamlined implementation. In particular, a large write transaction will ideally saturate the available write bandwidth anyway, so there is no benefit in allowing multiple such transactions to execute in parallel. Furthermore, since a bulk operation by definition touches a large fraction of the entire database, any concurrent writer substantially increases the likelihood of write-write conflicts which could force the bulk operation to abort. Due to the large amount of data that is modified, this is extremely undesirable.

We thus give bulk operations exclusive write access to the entire database, and only allow read transactions to execute concurrently. This greatly simplifies concurrency control, and additionally ensures that bulk operations will never abort due to write-write conflicts. Conceptually, our approach allows bulk operations to create *virtual* versions which encode creation or deletion of a data object. This is sufficient to support transaction isolation for arbitrary modifications in bulk operations, provided that bulk updates are performed out-of-place. For the purpose of visibility checks, these virtual versions are treated just like regular versions in our MVCC protocol. That is, a data object can be associated both with virtual versions created by a bulk operation, and regular versions created by the in-memory versioning approach. Crucially, such virtual versions require no physical memory allocation, allowing our approach to process arbitrarily large write transactions.

3.2.1 Versioning Protocol. Our proposed versioning protocol for bulk operations is based on a central monotonically increasing *bulk operation epoch* counter maintained by the database. Similar to the timestamps employed in the in-memory case, each transaction is associated with a *start epoch* $E_{start} \in \{0, \ldots, 2^{64}-1\}$ taken from this sequence. A virtual version v^* is marked with an epoch $E(v^*)$ which is set to the start epoch E_{start} of the bulk operation that created the virtual version. A virtual version v^* is visible to a transaction iff

$$E(v^*) \leq E_{start}.$$

For regular transactions, the start epoch is simply set to the current value of the central counter when they begin. This allows them to see any virtual versions that were created by bulk operations that committed before they started. In bulk transactions, E_{start} is set to the next available value of the central counter. Therefore, any virtual versions created by a bulk operation are initially invisible to concurrent readers. Upon commit, a bulk transaction atomically increments the central bulk operation epoch, which makes all virtual versions it created visible to subsequent transactions. Note that a single epoch value per transaction is sufficient here, since we do not allow multiple concurrent bulk operations.

The central bulk operation epoch is persistent across system restarts, and any changes thereof are properly logged to ensure durability. This allows us to implement virtual versions with extremely low overhead by storing a single *reference bulk load epoch* in the header of each database page containing potentially versioned data objects. Within each data object, two Boolean flags are maintained which indicate whether the object has an associated virtual creation or deletion version, which are implicitly marked with the reference bulk load epoch of the page. The reference epoch is initially set to an impossible value indicating that no virtual versions are present on the page. When a bulk operation later modifies a data object, it first sets the reference bulk load epoch of the page to its start epoch E_{start}. Subsequently, it updates the data object and sets the appropriate virtual version flag. Note that these flags do not actually consume any additional space on the pages in our implementation, since we can pack them into some unused bits of the tuple identifier stored in each data object.

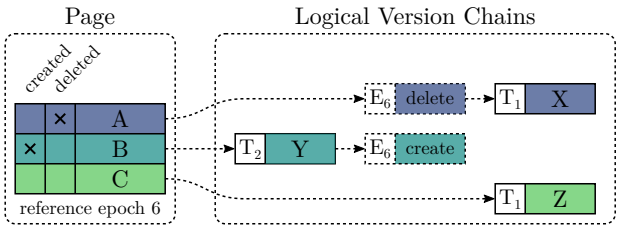


Figure 5: Bulk operations create virtual versions (illustrated as dashed boxes) by setting Boolean flags on the database pages. In this example, two regular transactions T_1 and T_2 updated tuples on a database page, whereas a bulk transaction E_6 created one tuple and deleted another. The local mapping table for the database page and the individual transaction version buffers are omitted for clarity.

Within the version chain associated with a given data object, a virtual version implicitly constitutes the oldest (in case of creation) or newest (in case of deletion) version (cf. Figure 5). These virtual versions are processed together with the in-memory versions during version chain traversal, and the visibility of the data object is computed according to the visibility criterion given above. Therefore, our approach for bulk operations does not require any intrusive modifications of the high-level MVCC protocol implemented within our system, which ensures that it incurs negligible overhead during regular transaction processing.

Consider, for example, Figure 5 where three subsequent transactions modified tuples on a given database page. A regular transaction with commit timestamp T_1 updated the first tuple from X to A and the third tuple from Z to C. Subsequently, a bulk transaction with epoch E_6 deleted the first tuple, and created the second tuple with value Y. Instead of allocating physical versions like a regular transaction, this information is recorded by setting the reference bulk load epoch and the respective Boolean flags on the database page. Readers interpret these flags as virtual versions with epoch E_6 when scanning the page (illustrated as dashed boxes in Figure 5). Finally, another regular transaction with commit timestamp T_2 updated the second tuple from Y to B. Therefore, a scan with $T_{start} = T_1$ and $E_{start} = 5$ would return A and C, whereas a scan with $T_{start} = T_1$ and $E_{start} = 6$ would return Y and C.

In theory, it would be possible to directly use the transaction timestamps for marking virtual versions but this has several major disadvantages. First of all, it requires making changes to these timestamps durable since our persistent database pages can reference them. Thus, every transaction commit would need to write some additional data to disk. Most importantly, the in-memory versioning protocol requires that all versions generated by a transaction are retimestamped during commit. While this is practicable for small transactions, it is prohibitively expensive for bulk operations that potentially modify a large number of database pages.

3.2.2 Synchronization. As outlined above, our approach requires some synchronization between transactions of different kinds. For this purpose we maintain a single global mutex within the database. Read transactions never need to latch this mutex since they are always allowed to proceed. Regular write transactions acquire a shared latch on this mutex, allowing multiple regular write transactions to be executed concurrently. Finally, bulk transactions acquire an exclusive latch on this mutex. Despite requiring a global mutex, our approach introduces negligible contention since latch acquisitions never block unless a bulk transaction is currently executing. Note that our approach would allow for more fine-grained synchronization of writers on the relation or partition level, so that multiple independent bulk operations can execute concurrently. While this increases implementation complexity, it is attractive on multi-socket systems where a centralized latch could introduce noticeable overhead.

A side-effect of our virtual version implementation is that we cannot allow a new bulk operation to begin until after any previous bulk operations have become globally visible to all active transactions. The reference bulk load epoch stored on a database page is used to implicitly determine the visibility of all virtual versions on the page, i.e. we cannot store virtual versions with multiple visibilities on a single page. Thus, when committing a bulk operation we do not immediately release the exclusive write latch on the database, but initially only downgrade it to a shared latch. This allows regular write transactions to begin immediately after a bulk operation has committed, but prevents another bulk operation from starting until the shared latch is released once the previous bulk operation has become globally visible. A desirable consequence of this restriction is that long-running readers delay the next bulk operation instead of forcing the system to maintain an excessive number of obsolete versions.

3.2.3 Detecting Bulk Operations. Detecting bulk operations in the first place poses a challenge in itself. The preferred way is to receive explicit instructions from the user to execute a transaction as a bulk operation, e.g. in an interactive administration session or when ingesting large amounts of data. For this purpose, the database system can provide a `SET TRANSACTION BULK WRITE` statement, for instance. Naturally, this mechanism is inherently unreliable since it relies on correct user input. It is thus not sufficient on its own, and we provide several fallback options to alleviate this. First, we additionally try to infer the write behavior of statements during query optimization, and automatically switch to bulk processing if the first statement within a transaction is likely to modify a large amount of data. As a last resort, e.g. during subsequent statements of a multi-statement transaction or in case the optimizer incorrectly deduced the write behavior of a statement, we also track the amount of memory consumed by the version buffers of the transaction. If the system is in risk of running out of memory, the transaction is aborted and the user can restart it explicitly as a bulk operation.

3.2.4 Garbage Collection. Since our versioning approach for bulk operations does not generate any physical versions, garbage collection can be performed more lazily than in the in-memory approach. Whenever a page containing virtual versions is accessed, we check whether the corresponding reference bulk load epoch is globally visible. If this is the case, we clear the virtual version flags of all data objects on the page, and reset the reference bulk load epoch to an impossible value. These operations can be performed alongside the pruning of local mapping tables during each page access, where a suitable latch on the database page has already been acquired.

3.3 Further Considerations

In the following we briefly discuss further relevant aspects of the proposed approach.

3.3.1 Scalability to Multi-Socket Systems. Although modern CPUs already feature up to 100 logical threads, multi-socket server configurations promise even greater parallelism. However, this comes at the cost of a non-uniform memory access (NUMA) topology which can impede performance in case of excessive cross-socket communication. As outlined in more detail above, the proposed MVCC approach itself takes care to avoid centralized data structures whenever possible to reduce potential scalability bottlenecks. Cross-socket communication could be reduced further by leveraging information about data locality and making the buffer manager and scheduler aware of the NUMA topology [34, 35]. Write operations can then be scheduled in such a way that most local mapping tables and their associated version chains are created within the same NUMA region as the corresponding database pages. If necessary, larger operations can also be split into smaller fragments that are then scheduled individually [34].

3.3.2 Serializability Validation. Our approach as described in this paper guarantees snapshot isolation for all transactions processed by the system [47]. Full serializability could be achieved through the precision locking approach proposed by Neumann et al. [47]. Conceptually, this requires a validation phase during transaction commit in which we validate all reads against potentially conflicting writes by recently committed transactions. For regular transactions, this can be achieved efficiently by scanning the version buffers maintained in the recently committed transaction list [47]. This is not possible for bulk transactions since they create no versions in main memory. Therefore, we have to fall back to actually repeating the reads performed by the transaction under validation in this case. However, this is still reasonably performant since we can use the reference bulk load epoch stored on the database pages to quickly determine whether a page could contain any potentially conflicting writes.

4 EXPERIMENTS

In the following we provide a thorough evaluation of our proposed system architecture as it is implemented within the disk-based relational database management system Umbra [46].

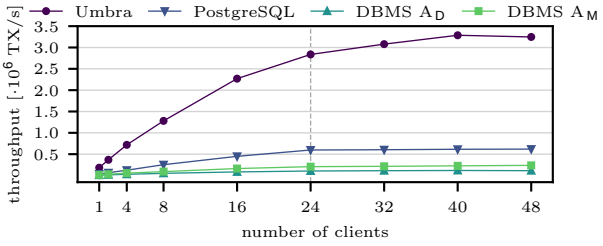
The design of Umbra closely follows the architecture presented in Section 2.1. Its buffer manager is derived from LeanStore, and durability is provided by a low-overhead decentralized logging scheme [35, 46]. Our system implements group commit by default, i.e. the committing thread simply waits for the commit record to become stable. For high-throughput scenarios that allow for relaxed commit semantics, our system additionally supports asynchronous commit. Umbra relies on a compiling execution engine for query processing [46, 47]. Users can either issue individual ad-hoc SQL statements to the system, or implement more complex transaction logic in user-defined functions using the SQL-based programming language UmbraScript. In both cases, we first generate an optimized logical query plan which is subsequently transformed into efficient machine code through a series of lowering steps [24, 28, 46].

4.1 Setup

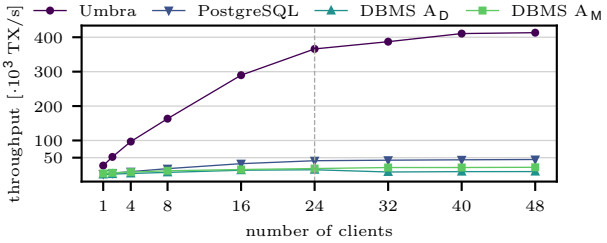
In order to demonstrate the feasibility of our proposed approach within a real-world setting, all experiments are performed through an external benchmark driver that communicates with the database system server over a standard communication protocol. We compare our implementation to PostgreSQL version 14 and another widely used commercial database management system referred to as DBMS A in the following [16]. We consider both the disk-based (DBMS A_D) and the in-memory (DBMS A_M) storage engines provided by DBMS A in our evaluation. The respective workloads are implemented as stored procedures that require minimum communication with the benchmark driver, i.e. we carefully avoid any unnecessary data transfer. In case of Umbra, we make use of the UmbraScript scripting language for this purpose. For PostgreSQL, the workloads are implemented using PL/pgSQL and for DMBS A we rely on its proprietary scripting language.

In case of both Umbra and PostgreSQL, the benchmark driver uses `libpq` version 14 for communication through the PostgreSQL message protocol. We make use of the message pipelining capabilities provided by this protocol in order to minimize the communication overhead in both cases. For DBMS A, communication with the database server occurs through ODBC where we simulate message pipelining by issuing batches of prepared statements. All systems are configured to employ snapshot isolation in conjunction with asynchronous commit semantics which ensures that throughput results are not affected by the latency of the storage device. Finally, we ensure that a separate DBMS worker thread is available to process the requests by a given benchmark driver client thread.

Experiments are run on a server system equipped with 192 GB of RAM and an Intel Xeon Gold 6212U CPU providing 24 physical cores and 48 hyper-threads at a base frequency of 2.4 GHz. The write-ahead log resides on a 768 GB Intel Optane DC Persistent Memory device, while all remaining database files are placed on a PCIe-attached Samsung 970 Pro 1 TB NVMe SSD, both of which are formatted as `ext4`. Note that we only rely on the persistent memory device since it is able to absorb the large volume of log data written during the benchmark runs, i.e. we do not exploit any properties specific to persistent memory. As Haas et al. have demonstrated, comparable write bandwidth could be obtained through directly-attached NVMe arrays which we expect to become widely available in future server configurations [18].



(a) TATP throughput results.



(b) TPCC throughput results.

Figure 6: Transaction throughput on the OLTP workloads (y-axis) in relation to the number of client threads (x-axis).

4.2 System Comparison

We begin our experiments with an end-to-end system comparison between Umbra, PostgreSQL, and DBMS A. For this purpose we select the well-known TATP and TPCC transaction processing benchmarks [48, 59]. For TATP we populate the database with 10 000 000 subscribers and run the default transaction mix consisting of 80 % read transactions and 20 % write transactions with uniformly distributed keys. For TPCC, we use 100 warehouses and run the full transaction mix consisting of about 8 % read transactions and 92 % write transactions. Depending on the system, the initial database population including indexes requires between 7 GB to 8 GB for TATP, and between 11 GB to 12 GB for TPCC. Umbra is configured to employ hash partitioning on the warehouse number for the TPCC database, i.e. it internally creates separate relation and index instances for each warehouse hash value in order to minimize latch contention. Partitioning is disabled for the other systems, as our preliminary experiments showed that it has a negative effect on their overall performance. The systems are configured to use 100 GB of main memory for their buffer pool, which is sufficient to accommodate the entire working set throughout the benchmarks. Therefore, they are executed under ideal conditions for high performance since only minimal disk IO is required, which allows us to investigate to which extent the different systems can exploit the capabilities offered by modern hardware platforms. All benchmarks first run for 30 seconds to warm up any caches and internal data structures, after which throughput numbers are measured over another 30 seconds.

4.2.1 Performance Results. Throughput results on the TATP and TPCC benchmarks in relation to the number of client threads are shown in Figure 6. In both cases Umbra outperforms its competitors by up to an order of magnitude, reaching a maximum speedup of 9.2× over PostgreSQL, 27.6× over DBMS A_D, and 18.8× over

DBMS A_M. Transaction throughput universally scales well with the number of client threads on the TATP benchmark. With a single client thread, the systems respectively process 183 000 TX/s (Umbra), 33 400 TX/s (PostgreSQL), 7 900 TX/s (DBMS A_D), and 15 000 TX/s (DBMS A_M). Umbra, PostgreSQL, and DBMS A_M attain their maximum throughput at 48 client threads with 3 247 000 TX/s, 618 700 TX/s, and 237 900 TX/s, respectively. DBMS A_D achieves its maximum of 117 700 TX/s at 40 client threads after which throughput decreases marginally to 113 000 TX/s. Since TPCC is much more write-heavy than TATP, we observe generally lower throughput, starting at 27 000 TX/s (Umbra), 2 600 TX/s (PostgreSQL), 1 100 TX/s (DBMS A_D), and 4 000 TX/s (DBMS A_M) with a single client thread. Nevertheless, performance scales well for Umbra, PostgreSQL, and DBMS A_M as the number of client threads is increased, and they reach maximum throughput at 48 client threads with 413 300 TX/s, 44 700 TX/s, and 22 000 TX/s respectively. In contrast DBMS A_D struggles to achieve good scalability, and attains its maximum throughput of 14 900 TX/s at 24 client threads beyond which performance decreases again down to around 9 000 TX/s.

4.2.2 Discussion. Our experiments clearly demonstrate that traditionally designed disk-based database systems such as PostgreSQL and DBMS A_D cannot fully exploit the capabilities of modern hardware, which confirms earlier such results from related work [20, 35, 46]. The single-threaded throughput numbers constitute particularly strong evidence for this conclusion, as system performance is far from being bound by IO throughput in this case. In fact, even the mature in-memory system DBMS A_M falls short of Umbra although it can avoid many of the complexities encountered in a disk-based system. Note that despite the low absolute performance of DBMS A_M, its relative speedup over DBMS A_D matches the corresponding performance metrics published by the manufacturer. The large speedup of Umbra over its competitors is almost entirely due to the greatly reduced overhead of its novel memory-optimized system architecture and the proposed MVCC implementation. We can in fact exclude communication overhead as a source of the observed speedup over PostgreSQL, since the benchmark drivers for Umbra and PostgreSQL rely on exactly the same client-server communication protocol (cf. Section 4.1). Although PostgreSQL does scale well on both TATP and TPCC, adding more client threads cannot resolve the inherent performance impediment caused by the excessive implementation overhead.

In contrast, Umbra is much better suited to exploit the large amount of main memory and high IO bandwidth offered by the benchmark platform. Its low-overhead buffer manager and decentralized logging framework ensure that virtually no overhead is introduced while accessing and modifying database pages [20, 35], despite generating slightly over 1 GB/s of log data in order to guarantee durability. Since the proposed memory-optimized MVCC implementation is highly decentralized and closely integrated with the buffer manager, it introduces little additional overhead and negligible contention (cf. Section 4.3.1). As outlined above, the storage engine employed by Umbra is derived from LeanStore which is one of the fastest disk-based storage engines currently published [35]. Therefore, our experiments provide a unique opportunity to quantify the additional implementation overhead that is required to provide general-purpose relational database functionality on top of such a state-of-the-art storage manager. On the same benchmark platform as used for our experiments, a standalone implementation of LeanStore achieves a single-threaded TPCC throughput of 41 000 TX/s which scales to 857 000 TX/s with 48 threads, albeit without any concurrency control [20]. Various factors contribute to the observed performance differential. For example, data is manipulated through SQL in Umbra, and its relation and index implementations have to generically support arbitrary tuple layouts. In contrast, both data layout and manipulation are hard-coded in the LeanStore benchmark driver. A further contributing factor is that LeanStore employs clustered relations, whereas Umbra only supports non-clustered relations which roughly doubles the number of lookup operations that Umbra has to perform [46]. A well-optimized in-memory system can operate with even lower overhead since it can employ highly specialized data structures that are not applicable to a disk-based setting [37]. For instance, we measured the single-threaded TPCC throughput of HyPer in our benchmark environment to be 58 800 TX/s, using the `libpq` driver executable also employed for our experiments with Umbra and PostgreSQL. In summary, our results show that a memory-optimized disk-based system architecture, and in particular the MVCC implementation proposed in this paper, are viable in a real-world setting and achieve excellent performance even when integrated into a general-purpose database system.

Table 1: Breakdown of the impact that various components of the proposed approach have on the overall performance of Umbra. We show TPCC transaction throughput for 1 and 24 client threads, along with the slowdown relative to non-transactional Umbra in parentheses.

	TPCC throughput [$\cdot 10^3$ TX/s]	
	1 client	24 clients
non-transactional Umbra	32.8	452.0
+ transaction lists	31.6 (-1.04 ×)	440.7 (-1.03 ×)
+ shared writer latches	31.3 (-1.05 ×)	429.8 (-1.05 ×)
+ snapshot isolation	27.0 (-1.22 ×)	366.0 (-1.23 ×)
- in-place updates	6.4 (-5.13 ×)	86.2 (-5.24 ×)

4.3 Detailed Evaluation

In the following we present additional experiments within Umbra in order to investigate key characteristics of our proposed approach in more detail.

4.3.1 Impact of MVCC Implementation. In our first experiment we quantify the impact of various components of our MVCC implementation (cf. Table 1). Specifically, we begin my measuring the TPCC throughput achieved by Umbra without any of the transactional features discussed in this paper, i.e. under read uncommitted isolation semantics. Subsequently, we successively enable the transaction lists discussed in Section 3.1.2, and the centralized shared writer latch introduced in Section 3.2, both of which require some coordination between worker threads. Note that we do not yet perform any versioning in these measurements in order to isolate the overhead introduced by the respective components. The

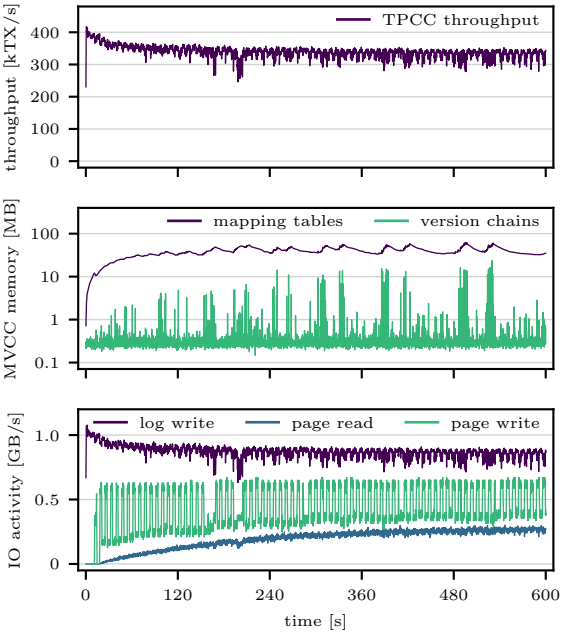


Figure 7: Umbra performance metrics (y-axis) sampled over time (x-axis) in 100 ms intervals. We run TPCC with 24 client threads and a restricted buffer pool size in this experiment, resulting in heavy memory pressure.

efficient latch implementation employed by Umbra ensures that the combined slowdown is barely noticeable at 1.05× for both 1 and 24 client threads [5]. Next, we enable the actual MVCC implementation and thus change the transaction semantics to snapshot isolation. As expected, this affects transaction throughput which decreases by a factor of about 1.2× relative to the non-transactional system configuration. Nevertheless, these results demonstrate that our MVCC implementation allows the system to retain both good scalability and high transaction throughput. Finally, we disable in-place updates in our approach, forcing all versions to be physically materialized within the same storage space. The resulting system configuration thus imitates the append-only version storage scheme employed by established systems such as PostgreSQL and Hekaton [65]. As shown in Table 1, this causes throughput to drop dramatically by more than 5× relative to the maximum attainable value, although the system still outperforms its competitors due to other optimizations such as a compiling query execution engine. Based on previously published results, we expect the benefit of supporting in-place updates to be even more pronounced in scan-heavy workloads since they prevent excessive fragmentation of the relations [47]. In summary, the experiment confirms that the proposed MVCC implementation has a crucial impact on the performance of a memory-optimized disk-based system.

4.3.2 Scalability Beyond Main Memory. As we repeatedly emphasize throughout this paper, one of the major selling points of a memory-optimized disk-based system is its scalability to working set sizes which exceed the available main memory capacity. We demonstrate the feasibility of the proposed MVCC approach within such a system by running the TPCC benchmark with 24 client threads and an artificially constrained buffer pool size of 16 GB in order to simulate such an out-of-memory scenario. Since Umbra requires roughly 12 GB of database pages in order to store the initial TPCC population, this quickly results in heavy memory pressure. Figure 7 shows the development of various performance metrics over 10 minutes of running this workload. The final database state after the experiment contains close to 150 GB of database pages.

During roughly the first half of the experiment, we observe a smooth and graceful transition from pure in-memory transaction processing to steady-state operation beyond main memory. During this transition, Umbra transparently begins to swap database pages to disk as memory pressure increases, while retaining high write throughput for the write-ahead log. Regular phases of increased page write activity are caused by the checkpointer which continuously writes dirty pages to disk in order to ensure bounded recovery time [20]. Subsequently, the system continues to exhibit stable performance during the second half of the experiment. Raw transaction throughput settles at ~340 000 TX/s, which unsurprisingly is slightly lower than the ~370 000 TX/s achieved in the corresponding in-memory experiments presented thus far. Crucially, the experiment confirms that it is entirely feasible to maintain all versioning information in main memory even under otherwise heavily constrained conditions. The decentralized garbage collection approach proposed in Section 3.1.2 is able to keep the memory consumption of the proposed MVCC approach bounded by continuously reclaiming unnecessary versions and local mapping tables. On average, the system requires ~38 MB to store the local mapping tables, and an additional ~400 KB to store the actual versions. Occasionally, memory consumption exhibits some minor spikes, although they never reach beyond 60 MB. Spikes generally occur during brief times in which the available IO bandwidth decreases due to operating system interference and transaction latencies increase correspondingly. Overall, however, the amount of memory required by our MVCC approach remains several orders of magnitude below the amount of available main memory, and is effectively constant over time.

Table 2: Time and version memory required to load the initial TPCH database population at scale factor 10, depending on whether MVCC is enabled and the optimized versioning scheme for bulk operations is used.

MVCC	Bulk Op.	Time [s]	Version Memory [GB]
no	N/A	11.7	0
yes	no	15.0	2.9
yes	yes	13.0	0

4.3.3 Bulk Operations. We conclude our experiments by studying the impact of the optimized versioning scheme for bulk operations introduced in Section 3.2. For this purpose, we first measure the time and amount of version memory required to load the initial TPCH database population at scale factor 10 without any indexes [60], the results of which for different system configurations are displayed in Table 2. Unsurprisingly, bulk loading requires the least time at 11.7 s when versioning is disabled entirely, and no memory at all is allocated for versioning information in this case. If the specific

Table 3: Multi-threaded query throughput on TPCH at scale factor 10 with and without a concurrent update stream.

TPCH throughput [queries/s]		
No Updates	Regular Updates	Bulk Updates
28.6	23.8	25.0

circumstances allow for relaxed transaction isolation, this is a viable option for ingesting large amounts of data. In contrast, the system has to allocate 2.9 GB of memory for storing versioning information and bulk loading time increases to 15.0 s if we enable MVCC but disable the optimized bulk versioning scheme. Since the amount of versioning information is directly proportional to the amount of data ingested within a single transaction, this quickly becomes problematic for large data set sizes. The optimized bulk versioning scheme resolves this problem by creating virtual versions that do not require any physical memory, completing bulk loading in 13.0 s without allocating any memory for versioning information.

Furthermore, we execute a workload inspired by the TPCH power test, i.e. we continuously submit batches of analytical queries from a single client thread, while another client thread simultaneously updates the `orders` and `lineitem` tables by ingesting new data and deleting old data [60]. Note that the analytical queries are sufficiently complex to benefit from parallelization across all available CPU cores, i.e. this setup fully utilizes the underlying hardware platform. Without any concurrent updates, Umbra can process 28.6 analytical queries per second, which drops to 23.8 queries per second when concurrent updates are performed using the regular in-memory versioning scheme. When using bulk operations to perform the updates, throughput is slightly higher at 25.0 queries per second (cf. Table 3). In summary, our results show that the proposed bulk versioning scheme allows the system to transparently process arbitrarily large write transactions without having a negative impact on system performance. In fact, performance is generally improved slightly since creating and interpreting virtual versions introduces less overhead to both readers and writers than the full in-memory versioning scheme.

5 RELATED WORK

Multi-version concurrency control was first proposed towards the end of the 1970s [52]. Due to its immediately obvious advantages over alternative concurrency control algorithms (cf. Section 2.2), the field quickly developed through some initial theoretical considerations [4, 7, 50] into a vast area of both active research and practical relevance. A large number of both disk-based and in-memory database management systems rely on MVCC for transaction isolation [1, 2, 10, 13, 16, 26, 29, 31–33, 39, 40, 43, 47, 51, 56], ranging from fully featured commercial solutions to prototype systems exploring novel approaches. Active research focuses on many aspects of MVCC, among them variations of the underlying multi-versioning protocol [11, 40, 42, 54], physical version maintenance [3, 25, 47, 57], scalability [6, 17, 40], serializability [8, 12, 47, 51, 63], or support for mixed workloads [3, 6, 25, 26, 30, 49].

Recent work on the practical aspects of implementing these MVCC approaches within a larger system is mostly focused on pure main-memory systems, even though the underlying theoretical concepts are often more widely applicable [53, 58, 65, 66]. At the same time, many in-memory systems acknowledge the importance of scaling beyond main memory, and have added some form of fallback support for extremely large data sets. However, Leis et al. argue that adding such functionality as an afterthought leads to a suboptimal system design [35]. For instance, these approaches commonly require index structures to remain memory-resident which constitutes a major limitation [9, 14, 35, 55].

Many disk-based MVCC implementations are found within established commercial database systems with a rigid architecture that cannot easily be adapted to modern hardware [1, 2, 16, 35, 43]. As outlined in Section 3.1, these implementations consequently suffer from several drawbacks such as substantial overhead, severe write amplification, or poor scalability. In view of these issues, several novel disk-based system designs have been proposed recently. LLAMA is a log-structured storage engine on top of which the Deuteronomy component uses MVCC to provide a transactional key-value store [38, 39]. Versions are physically stored in the recovery log, and accessed through a latch-free but centralized hash table. This architecture thus incurs a non-negligible overhead during version chain traversal. Similar to our approach, the BTrim architecture recognizes that modern hardware platforms provide sufficient RAM for disk-based systems to maintain a large amount of data purely in-memory [17]. It adds a transparent in-memory row store on top of the buffer-managed SAP ASE system, although the main objective here is to reduce contention. The design of FOEDUS is based on the same fundamental observation, combining a buffer manager with large in-memory buffers for optimistic concurrency control [27]. It achieves excellent scalability by avoiding most latch acquisitions, but requires specialized hardware such as Phase Change Memory.

6 CONCLUSIONS

In this paper, we developed a novel multi-version concurrency control approach which is designed specifically for memory-optimized disk-based database systems deployed on modern hardware. The proposed approach allows such systems to achieve excellent transaction throughput in the common case that the entire working set fits into main memory, and offers transparent and graceful scalability to working sets exceeding main memory capacity. Specifically, we exploit that most versioning information can be maintained entirely in main memory on modern hardware, which allows for a highly optimized implementation that directly attaches this information to buffer frames. In line with previous results on the subject, our experiments demonstrate that such a memory-optimized disk-based system is indeed viable in a real-world setting, and far outperforms traditionally designed systems. Our paper thus presents strong evidence in favor of a paradigm shift towards a memory-optimized disk-based system architecture for the next generation of general-purpose database systems.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation program (grant agreement number 725286).

REFERENCES

[1] Oracle Corporation and/or its affiliates. 2022. *MySQL.* Retrieved February 7, 2022 from https://www.mysql.com/

[2] Oracle Corporation and/or its affiliates. 2022. *Oracle.* Retrieved February 7, 2022 from https://www.oracle.com/database/

[3] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD Conference*. ACM, 583–598.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[5] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and Robust Latches for Database Systems. In *DaMoN*. ACM, 2:1–2:8.

[6] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (2019), 128–141.

[7] Michael J. Carey and Waleed A. Muhanna. 1986. The Performance of Multiversion Concurrency Control Algorithms. *ACM Trans. Comput. Syst.* 4, 4 (1986), 338–378.

[8] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction Repair for Multi-Version Concurrency Control. In *SIGMOD Conference*. ACM, 235–250.

[9] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD Conference*. ACM, 1243–1254.

[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (2018), 169–182.

[12] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201.

[13] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA Database – Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (2011), 45–51.

[14] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *Proc. VLDB Endow.* 5, 11 (2012), 1424–1435.

[15] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48.

[16] The PostgreSQL Global Development Group. 2022. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* Retrieved February 7, 2022 from https://www.postgresql.org/

[17] Aditya Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, and Zhan-Feng Ma. 2018. BTrim - Hybrid In-Memory Database Architecture for Extreme Transaction Processing in VLDBs. *Proc. VLDB Endow.* 11, 12 (2018), 1889–1901.

[18] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*. www.cidrdb.org.

[19] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD Conference*. ACM, 981–992.

[20] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD Conference*. ACM, 877–892.

[21] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (2007), 141–259. https://doi.org/10.1561/1900000002

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[24] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[25] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *SIGMOD Conference*. ACM, 938–950.

[26] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conference*. ACM, 1675–1687.

[27] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD Conference*. ACM, 691–706.

[28] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. IEEE Computer Society, 197–208.

[29] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

[30] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (2011), 298–309.

[31] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.* 36, 2 (2013), 34–40.

[32] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.

[33] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *SIGMOD Conference*. ACM, 1307–1318.

[34] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD Conference*. ACM, 743–754.

[35] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[36] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE Computer Society, 38–49.

[38] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (2013), 877–888.

[39] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *CIDR*. www.cidrdb.org.

[40] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD Conference*. ACM, 21–35.

[41] David B. Lomet. 2019. Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. In *ICDE Workshops*. IEEE, 140.

[42] David B. Lomet, Alan D. Fekete, Rui Wang, and Peter Ward. 2012. Multi-version Concurrency via Timestamp Range Conflict Management. In *ICDE*. IEEE Computer Society, 714–725.

[43] Microsoft. 2022. *Microsoft Data Platform.* Retrieved February 7, 2022 from https://www.microsoft.com/en-us/sql-server/

[44] Pulkit A. Misra, Jeffrey S. Chase, Johannes Gehrke, and Alvin R. Lebeck. 2019. Multi-version Indexing in Flash-based Key-Value Stores. *CoRR* abs/1912.00580 (2019).

[45] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. 1992. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *SIGMOD Conference*. ACM Press, 124–133.

[46] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[47] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 677–689.

[48] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecom Application Transaction Processing Benchmark. Retrieved January 18, 2022 from http://tatpbenchmark.sourceforge.net/

[49] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD Conference*. ACM, 1771–1775.

[50] Christos H. Papadimitriou and Paris C. Kanellakis. 1984. On Concurrency Control by Multiple Versions. *ACM Trans. Database Syst.* 9, 1 (1984), 89–99.

[51] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[52] David P. Reed. 1978. *Naming and synchronization in a decentralized computer system.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[53] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction Processing on Modern Hardware.* Morgan & Claypool Publishers.

[54] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-Version Concurrency. *Proc. VLDB Endow.* 7, 13 (2014), 1331–1342.

[55] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. 2019. Native Store Extension for SAP HANA. *Proc. VLDB Endow.* 12, 12 (2019), 2047–2058.

[56] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD Conference*. ACM, 731–742.

[57] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes. *Proc. VLDB Endow.* 13, 2 (2019), 211–225.

[58] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An Analysis of Concurrency Control Protocols for In-Memory Database with CCBench. *Proc. VLDB Endow.* 13, 13 (2020), 3531–3544.

[59] Transaction Processing Performance Council (TPC). 2010. TPC benchmark C: Standard specification. Retrieved January 18, 2022 from http://www.tpc.org/

[60] Transaction Processing Performance Council (TPC). 2021. TPC benchmark H: Standard specification. Retrieved January 26, 2022 from http://www.tpc.org/

[61] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[62] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (2014), 865–876.

[63] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.

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

[65] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[66] Ling Zhang, Matthew Butrovich, Tianyu Li, Andrew Pavlo, Yash Nannapaneni, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel J. Eppinger, Jordi E. Gonzalez, Wan Shen Lim, Jianqiao Liu, Lin Ma, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak, Amadou Ngom, Dong Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang, Yao Yu, and William Zhang. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In *CIDR*. www.cidrdb.org.