Scalable Garbage Collection for In-Memory MVCC Systems

Jan Böttcher Viktor Leis* Thomas Neumann Alfons Kemper
Technische Universität München Friedrich-Schiller-Universität Jena*
{boettcher,neumann,kemper}@in.tum.de viktor.leis@uni-jena.de

ABSTRACT

To support Hybrid Transaction and Analytical Processing (HTAP), database systems generally rely on Multi-Version Concurrency Control (MVCC). While MVCC elegantly enables lightweight isolation of readers and writers, it also generates outdated tuple versions, which, eventually, have to be reclaimed. Surprisingly, we have found that in HTAP workloads, this reclamation of old versions, i.e., garbage collection, often becomes the performance bottleneck.

It turns out that in the presence of long-running queries, state-of-the-art garbage collectors are too coarse-grained. As a consequence, the number of versions grows quickly slowing down the entire system. Moreover, the standard background cleaning approach makes the system vulnerable to sudden spikes in workloads.

In this work, we propose a novel garbage collection (GC) approach that prunes obsolete versions eagerly. Its seamless integration into the transaction processing keeps the GC overhead minimal and ensures good scalability. We show that our approach handles mixed workloads well and also speeds up pure OLTP workloads like TPC-C compared to existing state-of-the-art approaches.

PVLDB Reference Format:
Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. *PVLDB*, 13(2): 128-141, 2019.
DOI: https://doi.org/10.14778/3364324.3364328

1. INTRODUCTION

Multi-Version Concurrency Control (MVCC) is the most common concurrency control mechanism in database systems. Depending on the implementation, it guarantees snapshot isolation or full serializability if complemented with precision locking [28]. MVCC has become the default for many commercial systems such as MemSQL [25], MySQL [27], Microsoft SQL Server [40], Hekaton [18], NuoDB [29], PostgreSQL [35], SAP HANA [9], and Oracle [30] and state-of-the-art research systems like HyPer [14] and Peloton [34].

The core idea of MVCC is simple yet powerful: whenever a tuple is updated, its previous version is kept alive by the system. Thereby, transactions can work on a consistent snapshot of the data without blocking others. In contrast to other concurrency control protocols, readers can access older snapshots of the tuple, while writers are creating new versions. Although multi-versioning itself is non-blocking and scalable, it has inherent problems in mixed workloads. If there are many updates in the presence of long-running transactions, the number of active versions grows quickly. No version can be discarded as long as it might be needed by an active transaction.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364328

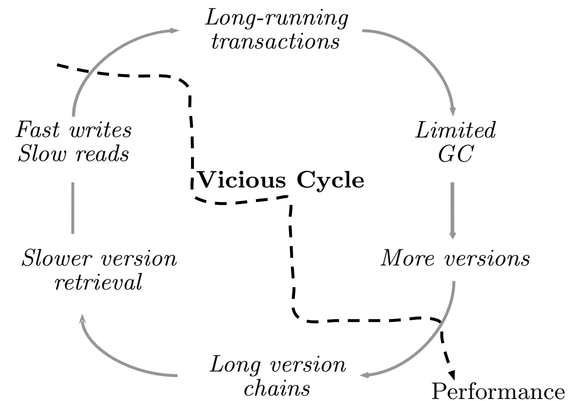


Figure 1: **MVCC's vicious cycle of garbage** – *Old versions cannot be garbage collected as long as there are long-running transactions that have to retrieve them*

For this reason, long-running transactions can lead to a "vicious cycle" as depicted in Figure 1. During the lifetime of a transaction, newly-added versions cannot be garbage collected. The number of active versions accumulates and leads to long version chains. With increasing chain lengths, it becomes more expensive to retrieve the required versions. Version retrievals slow down long-running transactions further, which amplifies the effects even more. Write transactions are initially hardly affected by longer version chains as they do not have to traverse the entire chain. They only add new versions to the beginning of the chain. Thereby, the gap between fast write transactions and slow read transactions increases, quickly producing more and more versions. At some point, the write performance is also affected by the increasing contention on the version chains as the insertion of new versions is blocked while the chain is latched for GC. The system also loses processing time for transactions when the threads clean the versions in the foreground.

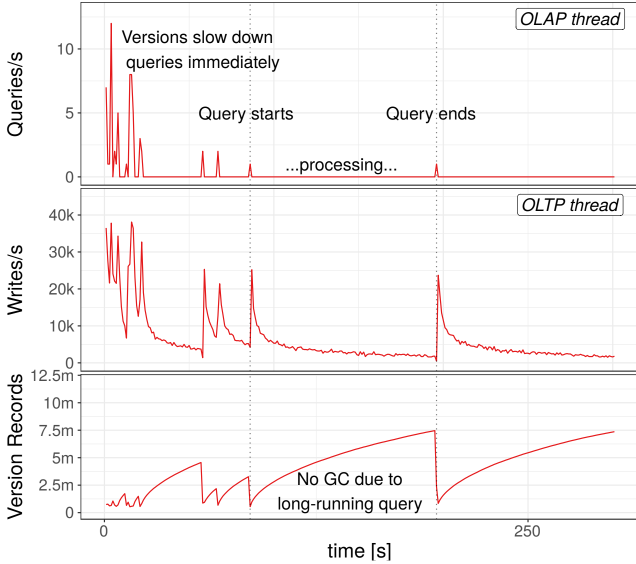


Figure 2: **Practical Impacts** – *The system's performance drops within minutes in a mixed workload using a standard garbage collection strategy*

In Figure 2 we visualize the practical implications of the described "vicious cycle" by monitoring an MVCC system in the mixed CH benchmark[1]. The OLTP thread continuously runs short-lived TPC-C style transactions, while the OLAP thread issues analytical queries. We see that the read performance collapses within seconds, while the writes are slowed down by long periods of GC. With higher write volumes or more concurrent readers, the negative effects would be even more pronounced. However, even low-volume workloads can run into this problem as soon as GC is blocked by a very long-running transaction (e.g., by an interactive user transaction).

The fact that GC is a major practical problem, causing increased memory usage, contention, and CPU spikes, has been observed by others [33, 22]. Nevertheless, in comparison with the number of papers on MVCC protocols and implementations, there is little research on GC. Except for of SAP HANA [20] and Hekaton [18], most research papers discuss GC only cursorily.

In this paper, we show that the garbage collector is a crucial component of an MVCC system. Its implementation can have a huge impact on the system's overall performance as it affects the management of transactions. Thus, it is important for all classes of workloads—not only mixed, "garbage-heavy" workloads [17, 16]. Our experimental results emphasize the importance of GC in modern many-core database systems.

As a solution, we propose *Steam*—a lean and lock-free GC design that outperforms previous implementations. Steam prunes every version chain eagerly whenever it traverses one. It removes all versions that are not required by any active transaction but would be missed by the standard high watermark approach used by most systems.

The remainder of this paper is organized as follows. Section 2 introduces basic version management and garbage

[1]Section 2.2 describes this experiment in more detail.

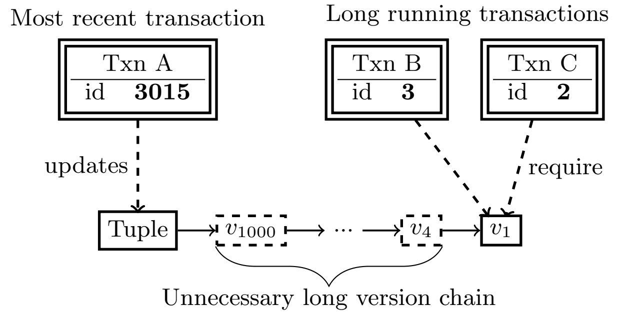


Figure 3: **Long version chain** – *Containing many unnecessary versions that are not GC'ed by traditional approaches*

collection in MVCC systems and challenges regarding mixed workloads and scalability. We then provide an in-depth survey of existing GCs and design decisions in Section 3. In Section 4, we propose our scalable and robust garbage collector *Steam* that decreases the vulnerability to long-running transactions. We present our experimental evaluation of Steam in comparison to different state-of-the-art GC implementations in Section 5. Lastly, we conclude with related work on HTAP workloads and garbage collection in Section 6.

2. VERSIONING IN MVCC

MVCC is a concurrency control protocol that "backs up" old versions of tuples, whenever tuples are modified. For every tuple, a transaction can retrieve the version that was valid when the transaction started. Thereby, all transactions can observe a consistent snapshot of the table.

The versions of a tuple are managed in an ordered chain of version records. Every version record contains the old version of the tuple and a timestamp indicating its visibility. Under snapshot isolation, a version is visible to a transaction if it was committed before its start. Hence, the timestamp equals the transaction's commit timestamp or a high temporary number, if it is still in-flight [28].

MVCC can maintain multiple versions (snapshots) of a tuple, whereas every update adds a new version record to the chain. The chain is ordered by the timestamp to facilitate the retrieval of visible versions.

Figure 3 shows a version chain for a tuple that was updated multiple times. Since Transaction B and C started before v_4 was committed, they have to traverse the chain (to the very end in this case) to retrieve the visible version v_1.

2.1 Identifying Obsolete Versions

Before discussing efficient garbage collection, we revisit when it is safe to remove a version. In general, a version must be preserved as long as an active transaction requires it to observe a consistent snapshot of the database. Essentially this means, that all versions that are visible to an active transaction must be kept. It does not matter whether the versions will be actually retrieved since the database system generally cannot predict the accessed tuples of a transaction—especially in the case of interactive user queries. Therefore, it always has to keep the visible versions as long as they could be accessed in future.

The set of visible versions is determined by the currently active transactions. When a version is no longer needed by any active transaction, it can be removed safely. Future transactions will not need them because they will already work on newer snapshots of the database. Hence, the required lifetime of every version only depends on the currently active transactions.

In the best case, a garbage collector can identify and remove all unnecessary versions. Looking at Figure 3: version record v_1 must not be garbage collected because it is required by Transactions B and C. All the preceding version records could be garbage collected safely and the length of the chain could be reduced significantly from 1000 to only 1 version. However, traditional garbage collectors only keep track of the start timestamp of the oldest active transaction. Thereby, they only get a crude estimation of the reclaimable version records. Essentially, only the versions that were committed before the start of the oldest active transaction are identified as obsolete. This leads to several "missed" versions in the case of multiple updates and long-running transactions. To overcome this problem, we propose a more fine-grained approach in Section 4.3 that prunes the unnecessary in-between versions.

2.2 Practical Impacts of GC

Figure 2 demonstrates the practical weaknesses of a standard GC. For this experiment, we ran the mixed CH benchmark which combines the transactional TPC-C and analytical TPC-H workload [2]. One OLAP and OLTP thread are enough to overstrain the capabilities of a traditional high watermark GC. Having only one warehouse, the isolated query execution times are reasonably fast (5-500 ms). However, compared to the duration of a write (0.02 ms), some of the queries are already long-running enough to run into the "vicious cycle". By adding more threads and/or warehouses the effects would be even worse.

The query throughput drops significantly after some seconds and queries start to last seconds (instead of milliseconds as before). These long-running queries show up in the topmost plot as the increasing periods of 0 queries/s. As long as the query is running, the number of version records stack up. This leads to the "shark fin" appearance in the number of version records. Only when the reader is completed, the writer starts to clean up the version records. For these periods of GC, it cannot achieve any additional write progress. Over time, the effects get worse and the amplitude of the number of version records increases while the read and write performance drops to almost 0. The query latencies increase significantly by the additional version retrieval work while the write processing suffers from the additional contention caused by the GC. In this setup—with only one write thread—the back pressure on the GC thread is already too high and the number of versions grows constantly. Especially the effects on the read performance are tremendous if the GC thread cannot catch up with the write thread(s). At some point, the entire system would run out of memory.

In summary, traditional garbage collectors have several fundamental limitations: (1) scalability due to global synchronization, (2) vulnerability to long-living transactions caused by its (3) inaccuracy in garbage identification. The general high watermark approach cannot clean in-between versions long version chains.

3. GARBAGE COLLECTION SURVEY

Our survey compares the GC implementations of modern in-memory MVCC systems with our novel approach Steam, which we describe in detail in Section 4.

Steam is a highly scalable garbage collector that builds on HyPer's transaction and version management [28]. Long version chains are avoided by pruning them precisely based on the currently active transactions. This is done using an interval-based algorithm similar to that in HANA, except that the version pruning does not happen in the background but is actively done in the foreground by piggy-backing it onto transaction processing [20]. A chain is pruned eagerly whenever it would grow due to an update or insert. This makes the costs of pruning negligibly small as the chain is already latched and accessed anyway by the corresponding update operation.

Hekaton also cleans versions during regular transaction processing [18]. In contrast to Steam, it cleans only those obsolete versions that are traversed during scans, whereas Steam already removes obsolete versions before a reader might have to traverse them. Essentially, Steam prunes a version chain whenever it would grow due to the insertion of a new version—limiting the length of a chain to the number of active transactions. Additionally, Hekaton only reclaims versions based on a more coarse-grained high watermark criterion, while Steam cleans all obsolete versions of a chain.

On a high-level, Steam can be seen as a practical combination and extension of various existing techniques found in HANA, Hekaton, and HyPer. As will show experimentally, seemingly-minor differences have a dramatic impact on performance, scalability, and reliability. In the remainder of the section, we discuss different design decisions in more details and summarize them in Table 1.

Tracking Level Database systems use different granularities to track versions for garbage collection. The most fine-grained approach is GC on a *tuple-level*. The GC identifies obsolete versions by scanning over individual tuples. Commonly this is implemented using a background vacuum process that is called periodically. However, it is also possible to find and clean the versions in the foreground during regular transaction processing. For instance, Hekaton's worker threads clean up all obsolete versions they see during query processing. Since this approach only cleans the traversed versions, Hekaton still needs an additional background thread to find the remaining versions [4].

Alternatively, the system can collect versions based on transactions. All versions created by the same transaction share the same commit timestamp. Thus, multiple obsolete versions can be identified and cleaned at once. While this makes memory management and version management easier, it might delay the reclamation of individual versions compared to the more fine-grained tuple-level approach.

Epoch-based systems go a step further by grouping multiple transactions into one epoch. An epoch is advanced based on a threshold criterion like the amount of allocated memory or the number of versions. BOHM also uses epochs, but since it executes transactions in batches, it also tracks GC on a batch level.

The coarsest granularity is to reclaim versions per table. This makes sense when it is certain that a given set of transactions will never access a table. Only then the system can

Table 1: **Garbage Collection Overview** – *Categorizing different GC implementations of main-memory database systems*

	Tracking Level	Frequency (Precision)	Version Storage	Identification	Removal
BOHM [7]	Txn Batch	Batch (`watermark`)	Write Set (`Full-N2O`)	Epoch Guard (`FG`)	Interspersed
Deuteronomy [21]	Epoch	Threshold (`watermark`)	Hash Table (`Full-N2O`)[1]	Epoch Guard (`FG`)	Interspersed
ERMIA [15]	Epoch	Threshold (`watermark`)	Logs (`Full-N2O`)	Epoch Guard (`FG`)	Interspersed
HANA [20]	Tuple/Txn/Table	1/10s (`watermark/exact`)	Hash Table (`Full-N2O`)[2]	Snapshot Tracker (`BG`)	Background
Hekaton [3, 4, 18]	Transaction	1 min (`watermark`)[3]	Relation (`Full-O2N`)	Txn Map (`BG`)	On-the-fly+Inter.[4]
HyPer [28]	Transaction	Commit (`watermark`)	Undo Log (`Delta-N2O`)	Global Txn List (`FG`)	Interspersed
Peloton [34]	Epoch	Threshold (`watermark`)	Hash Table (`Full-N2O`)	Global Txn List (`FG`)	Background
Steam	Tuple/Txn	Version Access (`exact`)	Undo Log (`Delta-N2O`)	Local Txn Lists (`FG`)	On-creation+Inter.

[1] The version records in the hash table only contain a logical version offset while the actual data is stored in a separate version manager.
[2] HANA keeps the oldest version in-place.
[3] Default value: Hekaton changes the GC frequency according to the workload.
[4] GC work is assigned ("distributed") by the background thread.

remove all of the table's versions without having to wait for the completion of these transactions. Since this only works for special workloads with a fixed set of given operations, e.g., stored procedures or prepared statements, this approach is rarely used. HANA is the only system we are aware of that applies this approach as an extension to its tuple and transaction-level GC [20]. In general, the database system cannot predict with certainty which tables will be accessed during the lifetime of a transaction.

Frequency and Precision Frequency and precision indicate how quickly and thoroughly a GC identifies and cleans obsolete versions. If a GC is not triggered regularly or does not work precisely, it keeps versions longer than necessary. The epoch-based systems control GC by advancing their global epoch based on a certain *threshold* count or memory limit. Thus, the frequency highly depends on the threshold setting.

Systems building on a background thread for GC, trigger the background thread *periodically*. Thus, the frequency of GC depends on how often the background thread is called. Since HANA and Hekaton use the background thread to refresh their high watermark, garbage collection decisions are made based on outdated information if the GC is called too infrequently. In the worst case, GC is stalled until the next invocation of the background thread. Systems like Hekaton, change the interval adaptively based on the current load [18].

BOHM's organizes and executes its transactions in *batches*. GC is done at the end of a batch to ensure that all of its transactions have finished executing. Only versions of previously executed batches, except for the latest state of a tuple, can be GC'ed safely.

Besides the frequency of GC, its thoroughness is mostly determined by the way a GC identifies versions as removable. Timestamp-based identification is not as thorough as an interval-based approach. The timestamp approach is more approximate because it only removes versions whose strictly chronological timestamps have fallen behind the *high watermark* which is set by the minimum start timestamp of the currently active transactions. Since the high watermark is bound to the oldest active transaction, long-running transactions can block the entire GC progress as long as they are active. In these cases, an interval-based GC can still make progress by excising obsolete versions from the middle of chains. In general, an interval-based GC only keeps required versions and thereby cleans the database *exactly*.

Version Storage Most systems store the version records in global data structures like hash tables. This allows the system to reclaim every single version independently. The downside is that the standard case, where all versions of an entire transaction fall behind the watermark, becomes more complex, as the versions have to be identified in the global storage. Depending on the implementation, this can require a periodical background vacuum process.

For this reason, HyPer and Steam store their versions directly within the transaction, namely the *Undo Log*. When a transaction falls behind the high watermark, all of its versions can be reclaimed together as their memory is owned by the transaction object. Nevertheless, single versions can still be pruned (unlinked) from version chains. Only the reclamation of their memory is delayed until the owning transaction object is released. In general, using the transaction's undo log as version storage is also appealing since the undo log is needed for rollbacks anyway. Using an undo log entry as a version record is straightforward as the stored-before images contain all information to restore the previous version of a tuple. For space reasons, we only store the *delta*, i.e., the changed attributes, in the version records. If a system stores the *entire tuple*, updating wide tables or tables with var-size attributes like strings or BLOBs can lead to several unnecessary copy operations [46].

Hekaton's version management is special in the sense that it does not use a contiguous table space with in-place tuples. The versions of a tuple are only accessible from indexes. For this reason, Hekaton does not distinguish between a version record and a tuple. Additionally, it is the only of the considered system that orders the records from oldest-to-newest (`O2N`). This order forces transactions to traverse the entire chain to find the latest version which makes the system's performance highly dependent on its ability to prune old versions quickly [46]. O2N-ordering also makes the detection of write-write conflicts more expensive as the transactions have to traverse the entire chain to detect the existence of a conflicting version. The same holds for rollbacks which also need to traverse entire chains to revert and remove previously installed versions.

Identification If commit timestamps are assigned monotonically, they can be used to identify obsolete versions. All versions committed before the start of the oldest active transaction can be reclaimed safely. The start timestamp of the oldest active transaction can be determined in constant time when the active transactions are managed in an ordered data structure like a *global txn list*, or a *txn map*.

Since pure timestamp-based approaches miss in-between versions as discussed in Section 2.1, systems like HANA and Steam complement it with a more fine-grained interval-based approach. While this approach keeps the lengths of version chains minimal, it is also more complex to implement it. The systems have to keep track of all active transactions and perform interval-based intersections for every version chain. HANA does this by tracking all transactions that started at the same time using a reference-counted list ("*Global STS Tracker*" [20]). In Section 4.3, we propose a more scalable alternative implementation using *local txn lists*.

For a more coarse-grained garbage collection, it is also possible to control the lifetimes of versions in epochs. This essentially approximates the more exact timestamp-based watermark used by the other systems. Nevertheless, epoch-based memory management is an appealing technique in database systems as it can be used to control the reclamation of all kinds of objects—not only versions. When a transaction starts, it registers itself in the current epoch by entering the epoch. This causes the *epoch guard* to postpone all memory deallocations/version removals made by the transaction until all other threads have left this epoch and thus will not access them anymore. While managing the versions in epochs limits the precision of the GC, it allows a system to execute transactions without having monotonically increasing transaction timestamps. For instance, in timestamp ordering-based MVCC systems like Deuteronomy or BOHM versions might be created or accessed in a different order than their logical timestamps suggest [7, 21].

Independent of the chosen data structure, the identification which versions are obsolete can either be done periodically by a background (*BG*) thread or actively in the foreground (*FG*).

Removal In HANA, the entire GC work is done by a dedicated background thread which is triggered periodically. Hekaton cleans all versions *on-the-fly* during transaction processing. Whenever a thread traverses an obsolete version, it removes it from the chain. Note, that this only works for *O2N*, when the obsolete (old) versions are stored in the beginning and thus are always traversed by the transactions. To clean infrequently-visited tuples as well, Hekaton runs a background thread that scans the entire database for versions that were missed so far. The background thread then assigns the removal of those versions to the worker threads which intersperse the GC work with their regular transaction processing.

A common pattern in epoch-based systems is to add committed versions along with the current epoch information to a free list. When a transaction requires a new version, it checks whether it can reclaim an old version from the free list based on the current epoch. Thereby, version removal essentially happens interspersed with normal transaction processing. However, the epoch guard should periodically release more than the newly required versions. Otherwise, the overall number of versions can only go up over time as all reused versions eventually end up in the free-list again. Deuteronomy addresses this by limiting the maximum number of versions. When the hard limit is reached, no more version creations are permitted and the threads are co-opted into performing GC until the number of versions is under control again [21].

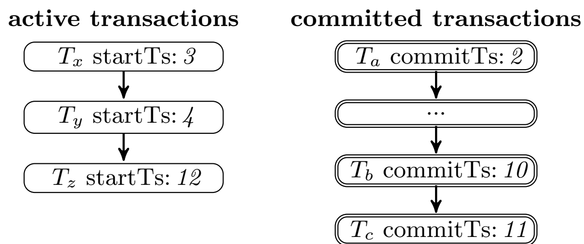


Figure 4: **Transaction lists** – *Ordered for fast GC*

HyPer and Steam also perform the entire GC work in the foreground by *interspersing* the GC tasks between the execution of transactions. If there are obsolete versions, the worker threads reclaim them directly after every commit. Thereby, GC becomes a natural part of the transaction processing without the need for an additional background thread. This makes the system self-regulating and robust to peaks at the cost of a slightly increased commit latency. Steam, additionally, prunes obsolete versions *on-creation* whenever it inserts a new version into a chain. Thereby, Steam ensures that the "polluters" are responsible for the removal of garbage, which relieves the (potentially already slow) readers.

4. STEAM GARBAGE COLLECTION

Garbage collection of versions is inherently important in an MVCC system as it keeps the memory footprint low and reduces the number of expensive version retrievals. In this section, we propose an efficient and robust solution for garbage collection in MVCC systems. We target three main areas: scalability (→ 4.2), long-running transactions (→ 4.3), and memory-efficient design (→ 4.4).

4.1 Basic Design

Steam builds on HyPer's MVCC implementation and extends it to become more robust and scalable [28]. To keep track of the active and committed transactions, HyPer uses two linked lists as sketched in Figure 4.

While HANA and Hekaton use different data structures (a reference-counted list and a map), the high-level properties are the same. All implementations implicitly keep the transactions ordered and adding or removing of a transaction can be done in constant time. To start a new transaction, the system appends it to the *active transactions* list. When an active transaction commits, the system moves it to the *committed transactions* list to preserve the versions it created. Completed read-only transactions, that did not create any tuple versions, are discarded directly.

By appending new or committed transactions to the lists, the transaction lists are implicitly ordered by their timestamps. This ordering allows one to retrieve the minimum *startTs* efficiently by looking at the first element of the *active transactions* list. The versions of a *committed transaction* with $commitId \leq min(startTs)$ can be reclaimed safely. Since the *committed transaction* list is also ordered, the system can reclaim all transactions until it hits a transaction that was committed after the oldest active transaction.

4.2 Scalable Synchronization

While the previously described basic design offers constant access times for GC operations, its scalability is limited

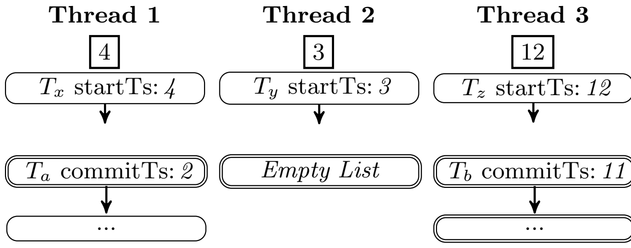


Figure 5: **Thread-local design** – *Each thread manages a subset of the transactions*

by the global transaction lists: Both lists need to be protected by a global mutex. For scalability reasons, we aim to avoid data structures that introduce global contention. Hekaton avoids a global mutex by using a latch-free transaction map for this problem. Steam, in contrast, follows the paradigm that it is best to use algorithms that do not require synchronization at all [8]. For GC, we exploit the domain-specific fact that the correctness is not affected by keeping versions slightly longer than necessary—the versions can still be reclaimed in the "next round" [33]. Steam's implementation does not require any synchronized communication at all. Instead of using global lists, every thread in Steam manages a disjoint subset of transactions. A thread only shares the information about its thread-local minimum globally by exposing it using an atomic 64-bit integer. This thread-local *startTs* can be read by other threads to determine the global minimum.

The local minimum always corresponds to the first active transaction. If there is no active transaction, it is set to the highest possible value ($2^{64} - 1$). In Figure 5 the local minimums are 4, 3, and, 12. To determine the global minimum for GC, every thread scans the local minimums of the other threads. Although this design does not require any latching, the global minimum can still be determined in $O(\#threads)$. Updating the thread-local minimum does not introduce any write contention either since every thread updates only its own *minStartTs*.

Managing all transactions in thread-local data structures reduces contention. On the downside, this can lead to problems when a thread becomes inactive due to a lack of work. Since every thread cleans its obsolete versions during transaction processing, GC can be delayed if the thread becomes idle. To avoid this problem, the scheduler periodically checks if threads have become inactive and triggers GC if necessary.

4.3 Eager Pruning of Obsolete Versions

During initial testing, we noticed significant performance degradations in mixed workloads. Slow OLAP queries block the collection of garbage because the global minimum is not advanced as long as a long-running query is active. Depending on the complexity of the analytical query, this can pause GC for a long time. With concurrent update transactions, the number of versions goes up quickly over the lifetime of a query. This can easily lead to the vicious cycle as described in Section 1. In practice, this effect can be amplified further by skewed updates which leads to even longer version chains.

Figure 3 shows how the versions of a tuple can form a long chain in which the majority of versions is useless for the active transactions. The useless versions slow down the long-running transactions when they have to traverse the entire chain to retrieve the required versions in the end. For this reason, we designed *Eager Pruning of Obsolete Versions* (EPO) that removes all versions that are not required by any active transaction. To identify obsolete versions, every thread periodically retrieves the start timestamps of the currently active transactions and stores them in a sorted list. The active timestamps are fetched efficiently without additional synchronization as described later in Section 4.3.1. Throughout the transaction processing, the thread identifies and removes all versions that are not required by any of the currently active transactions. Whenever a thread touches a version chain, it applies the following algorithm to prune all obsolete versions:

```
input:  active timestamps A (sorted)
output: pruned version chain
v_current ← getFirstVersion(chain)
for a_i in A
  v_visible ← retrieveVisibleVersion(a_i, chain)
  // prune obsolete in-between versions
  for v in (v_current, v_visible)
    // ensure that the final version covers all attributes
    if attrs(v) ⊄ attrs(v_visible)
      merge(v, v_visible)
    chain.remove(v)
  // update current version iterator
  v_current ← v_visible
```

We only store the changed attributes in the version record to save memory. For this reason, we have to check whether all of v's attributes are covered by $v_{visible}$. If there are additional attributes, we merge them into the final version. Systems that store the entire tuple would not need this check and could discard the in-between versions directly.

Figure 6 shows the pruning of a version chain for one active transaction started at timestamp 20. It shows the relatively-simple case when all attributes are covered by $v_{visible}$ and the more complex case, when the in-between versions contain additional attributes. In this case, we add the missing versions to the final version. When an attribute is updated multiple times, we overwrite it when we find an older version of it while approaching the visible version v_{visble}. In our example, A_{50} is overwritten by A_{25}. After the pruning, $v_{current}$ is set to the current value of $v_{visible}$ and $v_{visible}$ is advanced to the version that is visible to the next older (smaller) active id. As we only have one active transaction in our example, we can stop at this point.

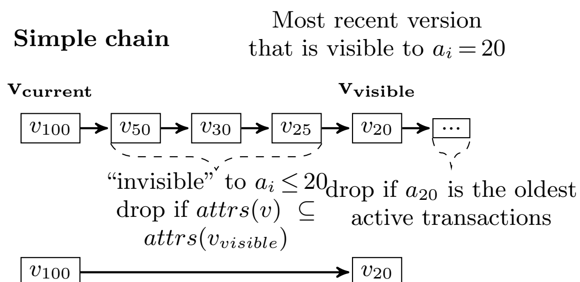
Since the version chain and the active timestamps are sorted and duplicate-free, every version is only touched once by the algorithm.

4.3.1 Short-Lived Transactions

EPO is designed for mixed workloads in which some transactions (mostly OLAP queries) are significantly slower than others. If all transactions are equally fast, it does not help as the commit timestamps hardly diverge from the id of the oldest active transaction.

A standard GC using a global minimum already works perfectly fine here. Thus, creating a set of active transactions will hardly pay off, as the number of reducible version chains is small. Ideally, we can avoid the overhead of retrieving the current set of transaction timestamps.

However, in general, the characteristics of a workload cannot be known by the database system and change over time.



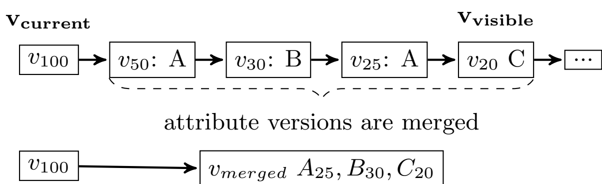


Figure 6: **Prunable version chain** – *Example for an active transaction with id 20*

So instead of turning EPO off, we reduce its overhead without compromising its effectiveness in mixed workloads.

The only measurable overhead of the approach is the creation of the sorted list of currently active transactions. The creation of the list only adds several cycles to the processing of every transaction (for a system using 10 worker threads that are 10 load instructions[2] and sorting them) but it is still noticeable in high volume micro-benchmarks.

To reduce this overhead, every thread reuses its lists of active transactions if it is still reasonably up-to-date. Thereby, the costs are amortized over multiple short-lived transactions and the overhead becomes negligible. For transactions running for more than 1 ms the costs of fetching the active transaction timestamps become insignificantly small. The quality of EPO is not affected as the set of long-running transactions changes significantly less frequently than the active transactions lists are updated.

During micro-benchmarks with cheap key-value update transactions, we noticed that the update period can be set to as low as 5 ms without causing any measurable overhead. This update period is still significantly smaller than the lifetime of even "short long-running" transactions.

4.3.2 HANA's Interval-Based GC

HANA's interval GC builds on a similar technique to shorten unnecessary long version chains, yet it differs in important aspects, which are summarized in Table 2. The biggest difference is how the version chains are accessed for pruning. In Steam, the pruning happens during every update of a tuple, i.e., whenever the version chain is extended by a new version. Thereby, a chain will never grow to more versions than the current number of active transactions and will never contain obsolete versions.

In HANA, in contrast, the pruning is done by a dedicated background thread which is triggered only every 10

[2]We only schedule as many concurrent transactions as we have threads.

Table 2: **Comparison with HANA's Interval GC**

HANA	Steam
Dedicated GC thread scans	Every thread scans
all committed versions	the accessed version chains
lazily every 10 s	eagerly
causing additional version and latching	"piggybacking" the costs while the chain is locked anyway

Table 3: **Data Layout of Version Records**

	Update	Delete	Insert	*Bytes*
	Common Header			
Type	✓	✓	✓	*1*
Version	✓	✓	✓	*4*
RelationId	✓	✓	✓	*2*
	Additional Fields			
Next Pointer	✓	✓	–	*4*
TupleId	✓	✓	–	*4*
NumTuples	–	–	✓	*4*
AttributeMask	✓	–	–	*4*
	Payload			
BeforeImages	✓	–	–	*var*
Tuple Ids	–	–	✓	$8 \times t$
Total Bytes	19+*var*	15	$11{+}8{\times}t$	

seconds. When HANA's GC thread is triggered, it scans the set of versions that were committed after the start of the oldest active transaction. For each of these versions, it checks if it is obsolete within its corresponding version chain using a merge-based algorithm similar to ours. This causes additional chain accesses, whereas Steam can "piggyback" this work on normal processing. Since HANA calls the interval-based GC only periodically, the version chains are not pruned and grow until the GC is invoked again.

4.4 Layout of Version Records

The design a version record should be space and computationally efficient. All operations that involve versions (insert, update, delete, lookup, and rollback) should work as efficiently as possible. Additionally, the layout should be in favor of GC itself, especially our algorithm for pruning intermediate versions.

Table 3 shows the basic layout of a version record. It has a *Type* (Insert/Update/Delete) and visibility information encoded in the *Version*. At commit time, the *Version* is set to the commit timestamp, which makes the version visible to all future transactions. To guarantee atomic commits, the *Version* includes a lock bit, which is used when a transaction commits multiple versions at the same time.

When a transaction is rolled back, it uses the *RelationId* and *TupleId* to identify and restore the tuples in the relation. The fields are also used during GC to identify the tuple that owns the version chain. The version chain itself is implemented as a linked list using the *Next Pointer* field. The *Next Pointer* either points to the next version record in the chain or *NULL* if there is none.

For all types of version records except for deletes, we need some additional fields or variations. For deletes, it is enough

to store the timestamp when a tuple has become invisible due to its deletion.

For inserts, we adapt the data layout by reinterpreting the attributes *TupleId* and *Next Pointer* to maintain a list of inserted tuple ids. This allows us to handle bulk-inserts more efficiently because we can use a single version record for all inserted tuples of the same relation. Sharing insert version records decreases the memory footprint (previously every inserted tuple required an own version record) and improves the commit latency. We can now commit multiple versions atomically by updating only a single *Version*. This optimization is possible since new tuples can only be inserted into previously empty slots. Thus, we can reuse the *Next Pointer* field to maintain a list of inserted *Tuple Ids*. For MVCC, we only need the information when the inserted tuple becomes visible. The tuple id list can be further compressed for bulk-inserts by storing ranges of subsequent tuples.

Update version records require the most fields as they contain the tuple's previous version (*Before Images*). To save space, we only store the versions of the changed attributes instead of a full copy of the tuple. Therefore, the version record needs to explicitly indicate which attributes it contains. For all relations with less than 64 attributes, we therefore use a 64-bit *Attribute Mask*, where every changed attribute is marked by a bit. When the relation has more columns, we indicate the changed attributes using a list of the ids of all changed attributes.

While the *Attribute Mask* saves space compared to the list, it also allows us to perform the check if a version record is covered by another (cf. Algorithm line 9) using a single bit-wise or-operation. If the *bit-wise or* of the attribute masks of v_x and v_y equals the attribute mask of v_x, all attributes of v_y are covered by v_x.

5. EVALUATION

In this section, we experimentally evaluate the different GC designs discussed in Section 3. To compare their performance, we implemented and integrated these GC approaches into HyPer [28]. For a fair apples-to-apples comparison, we only change the GC while the other components such as the storage layer or the query engine stay the same.

To distinguish our implementations from the original systems we put their names into quotes, e.g., 'Hekaton'. In our evaluation, we do not include BOHM of our survey in Section 3 as its GC is specifically designed for executing transactions in batches, in which concurrency control and the actual transaction execution are strictly separated into two phases [7]. Epoch-based GC—as used by BOHM—is represented by 'Deuteronomy' and 'Ermia'.

We monitor the systems' performance and capabilities by running the CH benchmark for several minutes. The CH benchmark is a challenging stress test for GCs because its short-lived OLTP transactions face long-living queries [2, 10, 36]. To better understand the general characteristics of the different systems we run some additional experiments. We analyze the scalability and overhead of each approach using the TPC-C benchmark. TPC-C is a pure OLTP benchmark without long-running transactions that could lead to the "vicious cycle of garbage". To evaluate different workload characteristics, we run the updates along with varying percentages of concurrent reads. We also explore the effects of skewed updates as they can be particularly challenging for garbage collectors by leading to potentially long version chains. Finally, we evaluate the effectiveness of *EPO* in keeping version chains short in isolation.

Table 4: **Configuration and Setup**

	Watermark	Exact	Frequ.	Find/Clean
'Deuter.'	Epoch (∞)	–	100 txs	FG
'Ermia'	Epoch (3)	–	1 tx	FG
'Hana'	Txn	Lazy	1 ms	BG
'Hekaton'	Txn	–	1 ms	BG⇒FG
Steam	Txn	Eager	cont.	FG

Table 4 summarizes the key features of our different GC implementations. All systems order the chains from *N2O*. The high watermark is either defined as the start timestamp of the oldest active transaction or epoch. All versions that were committed before that point in time are obsolete, as all active transactions already work on more recent snapshots of the data. Additionally, 'Hana' and Steam use a more *exact* form of GC that prunes intermediate versions in chains (cf. Section 4.3 for details). While 'Deuteronomy' increases its epoch-ids monotonically, 'Ermia' uses a three-phase epoch-guard[3].

Another important implementation detail is the *frequency* of garbage collection. For the epoch-based systems, this is the minimal number of committed transactions before the global epoch is advanced and for 'Hana' and 'Hekaton' this is the time when the background GC thread is invoked. It turns out that the default settings of the systems are not always suitable, so we hand-tuned them to the optimal values. In Section 5.4 we show how big the effect of a poorly chosen GC frequency is. Since Steam runs GC continuously whenever a version chain is accessed, there is no need to find and set an optimal interval.

In 'Hana', the GC work is done solely by the background thread (BG). 'Hekaton' uses the background thread only to refresh the global minimum and to identify obsolete versions. When it finds obsolete versions, it assigns the task of removing them to the worker threads. The other systems intersperse the entire GC work (identification and removal) with their normal transaction processing. Steam additionally prunes version chains eagerly whenever it accesses a version chain.

We evaluate the different approaches on an Ubuntu 18.10 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 "Hyper-Threads") each, resulting in a total of 20 physical cores (40 "Hyper-Threads"). The sockets communicate using a high-speed QPI interconnect (16 GB/s).

5.1 Garbage Collection Over Time

In this experiment, we put critical stress on the GC by running the mixed CH benchmark. This tests the vulnerability of every approach to long-running transactions and the "vicious cycle" of garbage.

The CH benchmark combines TPC-C write transactions with queries inspired by the TPC-H benchmark. This creates a demanding mix of short-lived write transactions and long-running queries. The gap between short-lived writes and long OLAP queries increases over time as the data set

[3]used code from `https://github.com/ermia-db/ermia`

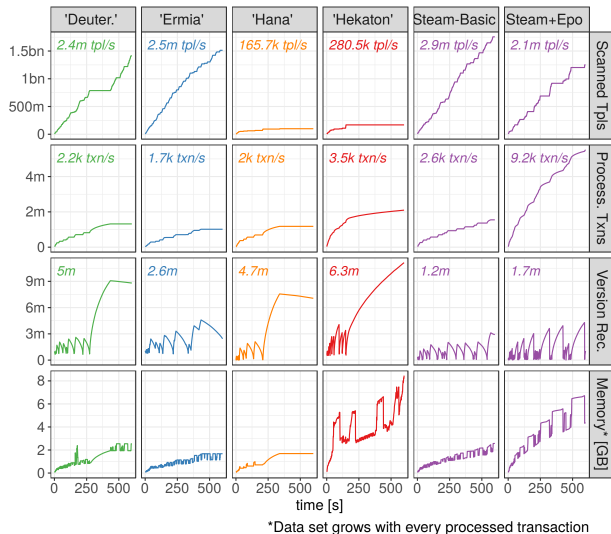


Figure 7: **Performance over time** – *CH benchmark with 1 OLAP and 1 OLTP thread. (Mean values shown in italics)*

grows with the number of processed transactions[4]. This makes our workload particularly challenging for fast systems like Steam that maintain a high write rate throughout the entire experiment. For comparison, it would take 'Ermia' 8356 seconds and thereby about 13× as long as Steam to process the same number of transactions reaching the same level of GC complexity.

To account for the data growth, we normalize the query performance by plotting the number of scanned tuples instead of the raw query throughput, following Funke et al.'s suggestion [10] to normalize the query performance using the increasing cardinalities of the relations. The increasing data size is also the reason why the used memory increases over time independently of the number of used/GC'ed versions.

Figure 7 shows the read, write, version record, and memory statistics over 10 minutes. Pruning all versions eagerly that are not required by any active transaction using EPO proofs to be an effective addition to Steam. Rather surprisingly the main improvements can be seen in the write throughput (roughly 3× compared to the second-best solution) while the read performance stays about the same. This is due to the fact that the main consumer of long version chains are not long-running queries but GC.

During GC we always have to traverse the entire chain to remove the oldest (obsolete) versions, whereas queries just have to retrieve the version that was valid when they started. For this reason, GC benefits most from short chains leaving more time for actual transaction processing. The increased speed of GC becomes visual when looking at the shapes of the version record curves: while the number of version records goes down gradually in all systems at the end of a long-running query, it drops almost immediately and very sharply when using EPO. This happens because hardly any GC has to be done anymore: most version records are already pruned eagerly from the chains and the remaining version records can be identified very quickly as the owning chains have a maximum length of 2, i.e., the number

[4] Every delivery transaction "delivers" 10 orders. Having 45% new-orders and only 4% delivery transactions, approximately 11% of the new orders remain undelivered.

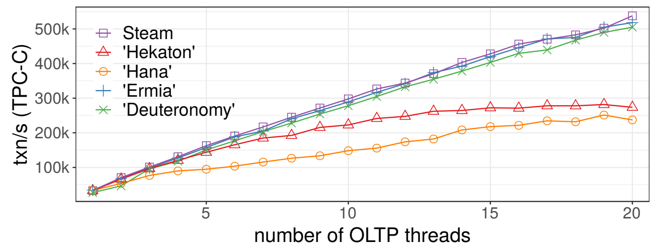


Figure 8: **TPC-C** – *Performance for increasing number of OLTP threads (100 warehouses)*

of active transactions. We analyze and compare those GC performance stats in details in the later Section 5.7.

As a side-effect, due to the highly improved write performance, the overall used memory increases faster than without using EPO. This can be accounted to the nature of the CH benchmark as described above: the data set grows with every processed transaction. What this means, in turn, is that reads also get more expensive as they have to scan more data (cf. memory plot). The increased query response times lead to bigger gaps between the short-lived writes and the long-lived queries, which is why the number of version records is a little bit higher with EPO. However, the average number of active version records only goes up by 42%, whereas the number of writes (which can be directly translated to the number of produced version records) increases significantly by 354%.

The epoch-based systems 'Deuteronomy' and 'Ermia' conceptually follow the same approach as the basic version of Steam using a watermark only. For this reason, the performance looks quite similar. There is only a slight set-back compared to the basic version of Steam, which is probably caused by the epochs being a little bit too coarse-grained for a mixed workload and that maintaining the global epoch introduces a small overhead.

'Hana' runs into more problems because it does the GC work exclusively in its background thread. With increasing gaps between the quick writers and the slow readers, the number of versions becomes too big and the single background thread becomes overwhelmed by the work.

'Hekaton' cleans the versions in the foreground, but it offloads the GC control, i.e., maintaining the high watermark and assignment of GC work, to the background thread. This detached workflow increases the GC latency to a point, where it gets out of control and the number of versions grows quickly.

5.2 TPC-C

While the previous experiment analyzed a mixed workload, we now want to show that the design and choice of a GC is also critical in pure OLTP workloads without any long-running transactions. Since we only interchange the GC, we can directly compare the overhead and scalability of the different approaches.

The TPC-C numbers in Figure 8 show that the foreground-based systems 'Ermia', 'Deuteronomy', and Steam scale best. 'Hana' falls slightly behind because it uses a centralized "*Global Snapshot Tracker*" that requires a global mutex.

While 'Hekaton' is superior to 'Hana', it is still limited by the use of its background thread which coordinates the GC. The background thread periodically retrieves the global

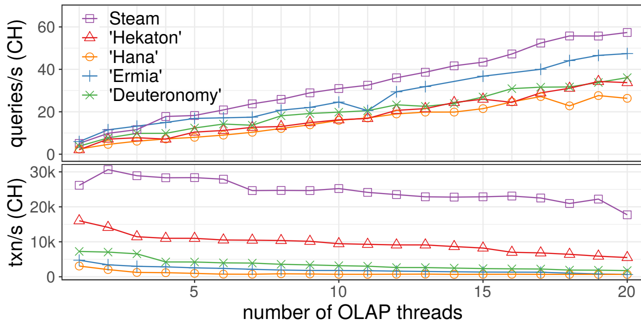


Figure 9: **CH benchmark** – *Performance for increasing number of OLAP threads using 1 OLTP thread*

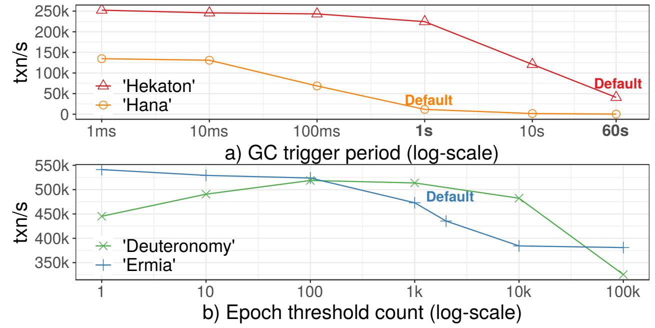


Figure 10: **GC Frequency** – *Varying a) the period when the GC thread is triggered or b) the count of committed transactions before an epoch might be advanced (TPC-C, 20 OLTP threads)*

minimum from the global transaction map and populates it to the threads. Additionally, it collects obsolete versions and assigns them to the work queues of the threads. While this allows the workers to remove the garbage cooperatively, there is still the single-threaded phase of identifying the garbage and "distributing" it. Furthermore, there is a small but constant synchronization overhead caused by the global transaction map. Although it is implemented latch-free, it still falls behind the thread-local implementations of Steam and the epoch-based solutions. This aligns well with recent findings that synchronous communication should be avoided and using latch-free data structures can even have worse performance than traditional locking [8, 45].

These results indicate that GC has a big impact on the system's performance in every kind of high-volume workload and not only in mixed workloads. For efficient GC global data structures and synchronous communication have to be avoided. In Section 5.5 we will see even bigger impacts on the system's scalability when running "cheap" key-value update transactions instead of TPC-C. When the transaction rate becomes very high, the maintenance of a global epoch starts to become a notable bottleneck.

5.3 Scalability in Mixed Workloads

In this section, we take another look at the CH benchmark. This time, we focus on the scalability by varying the number of read threads. In contrast to the previous time-bound experiment, now, every system processes a fixed number of 1 million TPC-C transactions. This makes the throughput numbers more comparable, as the query response times increase with every processed transaction due to growing data [10].

Figure 9 shows that the throughput of the single OLTP thread is highly affected by concurrent OLAP threads. This can be accounted to effects caused by the vicious cycle of garbage. As seen in Section 5.1, the versions accumulate quickly over time slowing down the readers. When the read transactions get slower, the version records have to be retained longer which amplifies this effect further. Additionally, the GC work and the slow readers create increased contention on the tuple latches as they require more time to retrieve a version. Hence, it is crucial to keep the number of version records as low as possible.

Steam's EPO reduces the number of versions effectively by pruning the version chains eagerly. This makes its GC and write performance superior to the other systems which struggle because their GC is too coarse-grained (epochs/high watermark). Even 'Hana' which also uses precise cleaning cannot keep up with Steam since its background pruning is not as effective as Steam's eager pruning (cf .Section 4.3.2 for a detailed comparison). At higher numbers of active read transactions, Steam's write performance degrades slightly because of the increasing likelihood that more versions have to be kept in the chains. Ideally, all transactions started at the same time and Steam only needs to keep one version per chain. This can be achieved by batching the start of readers in groups (similar to a group commit). Having fewer start timestamps improves the performance and effectiveness of EPO. Therefore, the performance could be improved slightly by artificially delaying some queries so that all queries share the same start timestamp. An evaluation of this idea showed gains of a few percents—at the cost of increased query latencies.

5.4 Garbage Collection Frequency

In Steam, GC happens continuously: Version chains are pruned whenever they are updated. Thus, the frequency is implicitly given and self-regulated by the workload. For the other systems, the frequency has to be explicitly set by a parameter which is either a time period in which the background GC thread is triggered (a) or a threshold that has to be reached before the global epoch is advanced (b).

The optimal period depends on the workload and the performance of the system. A faster system with high update rates generates more versions and has to be cleaned more frequently. To determine the optimal setting for the use with HyPer, we run TPC-C with different GC frequencies. Figure 10 shows the throughput when varying the trigger frequency from 1 ms to 60 s and epoch thresholds from 1 to 100k processed transactions.

For all systems, we see the best results when we trigger the GC as frequently as possible. For the background-thread approaches, we achieved the best results by setting the period to 1 ms. The period time cannot be decreased further, as the processing time of the GC thread would exceed its invocation intervals.

For the epoch-based systems, it is also best to set the epoch threshold as low as possible. This means that the system tries to advance the global epoch after every single committed transaction. However, refreshing the global epoch is not for free as this requires entering a critical section and/or scanning of other thread-local epochs. While the three-phase epoch-guard of 'Ermia' handles this case very efficiently, refreshing the global epoch in 'Deuteronomy' which

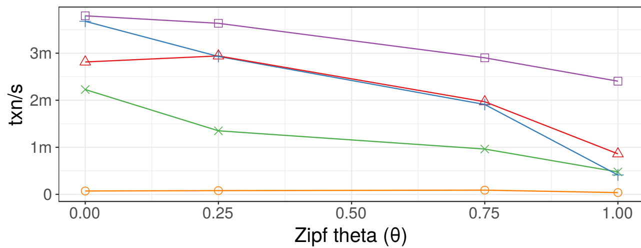


Figure 11: **Cheap key-value updates** – *Increasing the skew in key-value updates (using 20 OLTP threads)*

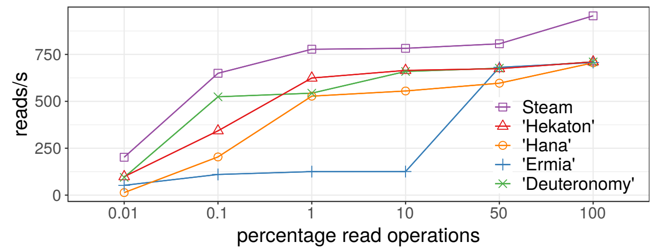


Figure 12: **Varying read-write ratios** – *Mixing table scans and key-value update transactions (20 threads)*

uses infinite epochs is more expensive. For this reason, the best threshold setting for 'Deuteronomy' is slightly higher at 100. This gives the best tradeoff between fast (immediate) GC and the overhead for refreshing the global epoch.

This experiment shows that the choice GC frequency can have a tremendous effect on the system's performance. There is a difference of more than 500× only by changing the frequency parameter. In practice, this could create critical instability if the system does not adjust this setting timely. This indicates that the frequency should be chosen based on the workload, i.e., the number of produced garbage (transactions) and not a fixed time interval. Otherwise, the back pressure on the GC can easily become too high. Even in the worst measured configuration, the epoch-based systems that control GC based on the number of processed transactions outperform the best time-interval-based GC. In Steam, we take this concept even a step further by pruning the chains eagerly whenever a new version is added.

5.5 Skew

When all updates are distributed evenly, every version chain tends to be equally short. However, in the real world, we often have skewed workloads. When certain tuples are updated more often their version chains get longer making GC more expensive. To measure the effectiveness of the GCs in skewed scenarios, we run key-value updates on a table using different Zipfian distributions. Figure 11 shows the throughput for theta values from 0.0 (no skew) to 1.0 (significant skew).

Steam is robust to skew because it deeply integrates GC into the transaction processing. Version chains that would become long can be pruned while, or rather before they grow (during an update). Other systems delay GC for longer: in particular, the time-based systems 'Hana' and 'Hekaton' which trigger GC only periodically, can be affected most. In the worst case, when only one tuple is updated all the time, the length of its version chain grows to the current number of updates per GC interval. At a throughput of 10,000 txn/s that would generate a chain of 10,000 versions assuming a GC interval of 1 s (default for HANA). In our experimental results, this effect is mostly diminished because we decreased the GC to 1 ms, but we can still see the systems falling behind Steam.

Unfortunately, the results for 'Hana' and to some degree 'Deuteronomy' are not very meaningful for increased skew as their performance is mostly dominated by their limited scalability. The results for a theta value of 0.0 indicate an overhead in high-volume workloads. This can be accounted to the use of a global mutex for the snapshot tracker ('Hana') and a relatively expensive refreshing of the global epoch counter in 'Deuteronomy'. By contrast, the three-phase epoch manager of 'Ermia' scales significantly better.

5.6 Varying Read-Write Ratios

In this experiment, we analyze how effective each approach is for different read/write setups. We run two kinds of transactions: write transactions updating tuples and read-only transactions doing full table scans, whereas all transactions operate on the same table. We vary the ratio of reads and writes by increasing the percentage of read operations every thread performs. Figure 12 shows the number of read operations for a decreasing number of writes.

The read performance increases as expected when the workload mix shifts towards being read-only, whereas Steam performs best in all setups. Especially in the read-only case, Steam's minimal overhead is clearly visible: A read-only thread never retrieves the set of active transaction ids (including the global minimum). This is only done when it has recently committed versions (i.e., its committed transaction list is not empty), or lazily during its first update operation. In the read-only case, every thread only has to signal its currently active transaction by adding it to its thread-local list. By contrast, all other systems require at least a basic form of synchronization, i.e., entering an epoch, or registering the transaction in a globally shared transaction map/tracker.

In the more write-heavy cases, EPO helps Steam to control the number of versions speeding up the readers. For high numbers of writes (<10% reads), 'Ermia' falls behind the other systems. While its three-phase epoch guard showed very good scalability in the other experiments, it seems to be too coarse-grained now. The more fine-grained infinite epochs of 'Deuteronomy' perform significantly better in these cases.

5.7 Eager Pruning of Obsolete Versions

To avoid long version chains in mixed workloads, we implemented *EPO* (cf. Section 4.3) to prune the chains eagerly whenever a new version is inserted. EPO removes all versions as soon as they are not required by any active transaction anymore.

Table 5 shows that this reduces the number of traversed versions significantly in the CH benchmark. Steam processes the given set of transactions 5× faster using EPO. Without the optimization, the GC cannot keep the number of versions down effectively since the high watermark approach is too coarse-grained. The version chains grow quickly hitting a maximum length of 30287. When the optimization is enabled, the maximum length goes down to two versions. The "optimized" chain only keeps the most recent version of the writer and an older version that is visible to the reader.

Table 5: **Effect of using EPO** – *CH benchmark, 1 read thread, 1 write thread, 300k transactions in total*

	Standard Watermark		**EPO** Exact	
Version Removal (GC)				
Traversed Versions	1,197m		4.2m	
Avg. Chain Length (max)	287.43 (30287)		1.07 (2)	
Table Scans (Queries)				
Traversed Versions	120m		37m	
Avg. Chain Length (max)	1.00 (141)		1.00 (2)	
Breakdown	Time	[%]	Time	[%]
Fetch Active Txn-Ids	<1 ms	0.01	<1 ms	0.01
Prune Chains (EPO)	–	–	8.4 ms	0.07
Finalize Entire Txns	1.5 s	4.47	81 ms	0.68
Version Retrieval (Scan)	4.2 s	12.26	1.1 s	8.79
Queries/s	4.8		5.1	
Transactions/s	6554		**30,580**	

Rather surprisingly, or even paradoxically, the more thorough and fine-granular we clean our system, the less time we spend cleaning. Using EPO, the system spends less than 100 ms in total on GC, while it requires 1.5 s using the standard watermark approach. This performance difference becomes clear when we look at the number of traversed versions: it is reduced from 1.2 billion to only 4.2 million. Since EPO keeps version chains short at all time, it is always cheap to identify and reclaim obsolete versions. In particular, when an entire transaction falls behind the "watermark" and we finalize its versions, most of its versions are already removed from the chains by EPO (thus, GC of them is a no-op) or belong to very short chains which makes unlinking them from the chain fairly cheap.

We also see that maintaining the set of active transaction ids does not add any overhead. For the watermark-approach, all thread-local minimums have to be fetched anyway. The additional sorting step required by EPO is negligible cheap since at most #-threads integers are sorted.

Faster GC is very beneficial for transaction processing in general, as slow, interspersed GC work can stall the processing of writes. Thus, faster GC gives the worker threads more time to process transactions.

The average length of version chains is significantly higher during version removal than it is during table scans. This happens because some tuples (counter and warehouse statistics) are updated frequently, but are never read by any query [2]. The readers mainly access parts of the tables that are updated evenly. Thus, the positive effect of EPO is not as big for queries as it is for the writes. The maximum chain length during a table scan is "only" 141 without the optimization. With the optimization, the scans have to retrieve 3.24× fewer versions. This is reflected by a slightly improved query performance. The benefit would be significantly higher if the readers would need to access the frequently updated tuples with lengths of more than 30,000.

6. RELATED WORK

In recent years, the performance of systems in mixed workloads (HTAP) was studied extensively [48, 19, 42, 46, 31, 1]. Several systems were developed focusing on scalability in high volume OLTP workloads [32, 37, 12, 23, 15]. A reoccurring topic is to optimize the concurrency control protocol, e.g., by tuning the validation phase or reordering transaction [5, 11, 38, 44]. Although most of the papers mention the use and importance of an efficient garbage collector, the implementation is either described only briefly or not mentioned at all. Recent work on GC is mostly related to large data systems in which the challenges and tasks are very different and not comparable to version reclamation in MVCC systems [47, 24]. In summary, most components of MVCC systems are well-understood, studied, and optimized but there is little research on efficient GC — despite its big impacts on performance.

Handling of long-living transactions is an inherent problem of MVCC systems studied by others. Lee et al. [20] describe practical solutions to this problem such as: (1) flushing old versions to disk if main memory is exceeded, (2) aborting long-running transactions (user gets an error), and (3) closing transactions as soon as possible (e.g., after query results are materialized). However, these solutions are not applicable to high volume workloads. One proposal for such workloads is to create virtual memory snapshots (forks) for read-only queries [26, 39]. However, this strongly affects the overall scalability of the system as it requires a shared mutex per column.

Modern and fast OLTP systems like TicToc or Silo often use a single-version approach instead of MVCC [49, 43, 6, 41]. A single-version system only maintains the latest version of a tuple and thus there is no need for garbage collection. This makes them particularly fast in OLTP workloads. However, by default, they are not designed to handle OLAP or mixed workloads as they would have to maintain a large read set. Since this is can easily lead to aborts, Silo also allows creating snapshots of the data by storing old tuple versions. Due to the costs of snapshots creation, snapshots are only taken periodically, i.e., every second, which results in slightly stale data [43].

Systems that apply Serialization Graph Testing (SGT) instead of timestamps have to keep a transaction and its items until its existence does not influence any other or future transactions [6, 13].

7. CONCLUSION

In this paper, we show the importance of garbage collection for in-memory MVCC systems on modern many-core systems. We find that GC should be based on thread-local data structures and asynchronous communication for optimal performance. Further, it is crucial for HTAP workloads of short-lived writes and long-running reads to keep the number of active versions as low as possible. With traditional high watermark-based approaches, a single long-running transaction blocks GC progress during its lifetime.

Our novel, scalable GC *Steam* speeds up transaction processing and garbage removal by pruning all obsolete versions eagerly whenever a new version is added. Thereby, Steam effectively limits the length of chains to the number of active transactions. Besides HTAP workloads, our experimental results indicate that Steam benefits all kind of workloads from write-only to read-only. Its seamless integration into transaction processing enables superior performance compared to other state-to-the-art GC approaches which detach GC from transaction processing.

8. REFERENCES

[1] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The case for heterogeneous HTAP. In *CIDR*, 2017.

[2] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, New York, NY, USA, 2011. ACM.

[3] K. Delaney. SQL Server in-memory OLTP internals overview. *White Paper of SQL Server*, 2014.

[4] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[5] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2), 2018.

[6] D. Durner and T. Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *ICDE*, 2019.

[7] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[8] J. M. Faleiro and D. J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR*, 2017.

[9] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Record*, 40(4), 2012.

[10] F. Funke, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, A. Nica, M. Poess, and M. Seibold. Metrics for measuring the performance of the mixed workload ch-benchmark. In *TPCTC*, 2011.

[11] J. Guo, P. Cai, J. Wang, W. Qian, and A. Zhou. Adaptive optimistic concurrency control for heterogeneous workloads. *Proceedings of the VLDB Endowment*, 12(5), 2019.

[12] A. Gurajada, D. Gala, F. Zhou, A. Pathak, and Z.-F. Ma. Btrim: hybrid in-memory database architecture for extreme transaction processing in vldbs. *PVLDB*, 11(12), 2018.

[13] T. Hadzilacos and N. Yannakakis. Deleting completed transactions. *JCSS*, 38(2), 1989.

[14] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[15] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*. ACM, 2016.

[16] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper. Analytics on fast data: Main-memory database systems versus modern streaming systems. In *EDBT*, 2017.

[17] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper. Scalable analytics on fast data. *ACM*, 44(1), Jan. 2019.

[18] P. Larson, M. Zwilling, and K. Farlee. The Hekaton memory-optimized OLTP engine. *IEEE Data Eng. Bull.*, 36(2), 2013.

[19] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[20] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *SIGMOD*. ACM, 2016.

[21] J. J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *CIDR*, 2015.

[22] L. Li, G. Wu, G. Wang, and Y. Yuan. Accelerating hybrid transactional/analytical processing using consistent dual-snapshot. In *DASFAA*, 2019.

[23] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017.

[24] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *PVLDB*, 9(12), 2016.

[25] MemSQL. `https://www.memsql.com/`.

[26] H. Mühe, A. Kemper, and T. Neumann. Executing long-running transactions in synchronization-free main memory database systems. In *CIDR*, 2013.

[27] MySQL. `https://www.mysql.com/`.

[28] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, 2015.

[29] NuoDB. `http://www.nuodb.com/`.

[30] Oracle. `https://www.oracle.com/database/`.

[31] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. In *SIGMOD*. ACM, 2017.

[32] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6), 2018.

[33] A. Pavlo. Multi-Version Concurrency Control (Garbage Collection). `https://15721.courses.cs.cmu.edu/spring2019/slides/05-mvcc3.pdf`, January 2019.

[34] Peloton. `https://pelotondb.io/`.

[35] PostgreSQL. `https://www.postgresql.org/`.

[36] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling up mixed workloads: A battle of data freshness, flexibility, and scheduling. In *TPCTC*. Springer, 2014.

[37] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, W. Lehner, and A. Rizk. Oltpshare: the case for sharing in OLTP workloads. *PVLDB*, 11(12), 2018.

[38] C. Reid, P. A. Bernstein, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11), 2011.

[39] A. Sharma, F. M. Schuhknecht, and J. Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018.

[40] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/.

[41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[42] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. *PVLDB*, 11(5), 2018.

[43] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[44] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2), 2016.

[45] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *SIGMOD*, 2018.

[46] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7), 2017.

[47] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei. An experimental evaluation of garbage collectors on big data applications. *PVLDB*, 12(1), Sept. 2018.

[48] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3), 2014.

[49] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *SIGMOD*. ACM, 2016.