One-shot Garbage Collection for In-memory OLTP through Temporality-aware Version Storage

AUNN RAZA, EPFL, Switzerland
PERIKLIS CHRYSOGELOS*, Oracle, Switzerland
ANGELOS CHRISTOS ANADIOTIS†, Oracle, Switzerland
ANASTASIA AILAMAKI, EPFL, Switzerland

Most modern in-memory online transaction processing (OLTP) engines rely on multi-version concurrency control (MVCC) to provide data consistency guarantees in the presence of conflicting data accesses. MVCC improves concurrency by generating a new version of a record on every write, thus increasing the storage requirements. Existing approaches rely on garbage collection and chain consolidation to reduce the length of version chains and reclaim space by freeing unreachable versions. However, finding unreachable versions requires the traversal of long version chains, which incurs random accesses right into the critical path of transaction execution, hence limiting scalability.

This paper introduces OneShotGC, a new multi-version storage design that eliminates version traversal during garbage collection, with minimal discovery and memory management overheads. OneShotGC leverages the temporal correlations across versions to opportunistically cluster them into contiguous memory blocks that can be released in *one shot*. We implement OneShotGC in Proteus, and use YCSB and TPC-C to experimentally evaluate its performance with respect to the state-of-the-art, where we observe an improvement of up to 2x in transactional throughput.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Main memory engines**; *Database transaction processing*; Record and buffer management.

Additional Key Words and Phrases: DBMS, OLTP, MVCC, GC, garbage collection

ACM Reference Format:
Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2023. One-shot Garbage Collection for In-memory OLTP through Temporality-aware Version Storage. *Proc. ACM Manag. Data* 1, 1, Article 19 (May 2023), 25 pages. https://doi.org/10.1145/3588699

1 INTRODUCTION

Modern in-memory online transaction processing (OLTP) engines execute several transactions in parallel to exploit the increasing number of CPU cores and main memory in scale-up servers. As concurrently running transactions may lead to conflicting data accesses, the OLTP engine needs to provide guarantees on data consistency. Most engines today, like Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SAP HANA provide snapshot isolation (SI) guarantees: SI allows increased

*Work done while the author was at EPFL.
†Work done while the author was at EPFL and Ecole Polytechnique.

Authors' addresses: Aunn Raza, aunn.raza@epfl.ch, EPFL, Lausanne, Switzerland; Periklis Chrysogelos, periklis.chrysogelos@oracle.com, Oracle, Zurich, Switzerland; Angelos Christos Anadiotis, angelos.anadiotis@oracle.com, Oracle, Zurich, Switzerland; Anastasia Ailamaki, anastasia.ailamaki@epfl.ch, EPFL, Lausanne, Switzerland.

2836-6573/2023/5-ART19 $15.00
https://doi.org/10.1145/3588699

concurrency while protecting from most anomalies and providing good scalability, enabling OLTP engines to take advantage of the CPU parallelism.

In practice, most systems that provide SI guarantees employ a multi-version concurrency control (MVCC) scheme. MVCC creates a new version on each write operation, to avoid blocking concurrent readers: Read operations can then proceed in parallel with a write operation, as long as there exists a version that does not violate the consistency of the database. However, there is no free lunch: storing several versions of the database increases the memory footprint and the associated maintenance cost [11, 16, 22].

Existing approaches amend multi-versioning overheads by frequently collecting *unreachable versions* through active or passive garbage collection (GC). Versions become unreachable when there can be no transaction that will access them at any point in time. Specifically, all versions created before the most recent version of a record that can be accessed by the least recent transaction currently active in the system are unreachable and, thus, can be dismissed. *Active approaches* [2, 5, 18] place a garbage collection phase at the end or during the execution of each transaction. During active GC, the OLTP engine traverses the version chains to find and release the unreachable versions. However, version chain traversal incurs random accesses over version lists and, thus, it does not scale with the chain length. In active GC, the chain length depends on the write ratio of the workload, since every write appends a new version to the chain. *Passive approaches* [1, 13] register unreachable versions to a background process that periodically vacuums them. In passive GC, the chain length depends both on the workload write ratio and the frequency of the vacuum process. Unfortunately, it is hard to decide on the vacuuming frequency: if it is too high, then it will interfere with transaction execution, whereas if it is too low, it will lead to long version chains. In either case, performance will drop and the tradeoff heavily depends on real-time workload parameters.

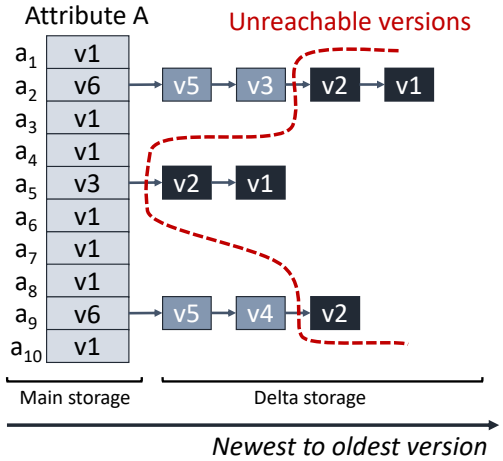


Fig. 1. Multi-versioned storage

The problem in the state-of-the-art GC approaches, with a direct impact on transactional performance and scalability, is the version traversal, which stems from the extremely fine-grained memory allocation performed for every version and results in several random memory accesses for every record. Figure 1 exemplifies the problem by showing the version storage for an attribute A with 10 records a_1 to a_{10}. Suppose that, when GC is triggered, the unreachable versions are the ones separated out by the red dotted line in the figure. In order to trace them, the system will have

to go through every record a_i, traverse all chains (with all the random accesses involved), and then, release the memory allocated for the unreachable versions. To understand the impact of GC on

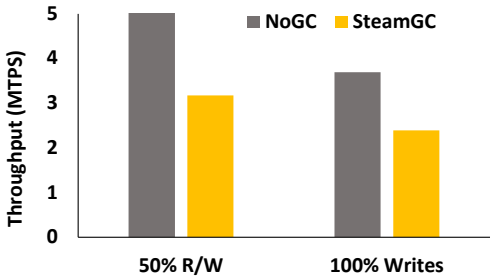


Fig. 2. Impact of GC on transactional performance

transactional performance, we experimentally analyze GC performance by executing the YCSB workload while turning GC on and off, on a database with a single table of 10 columns sized 8 bytes each. We execute the experiment on a server equipped with two NUMA nodes and 28 hardware threads, with each hardware thread associated with a worker database thread. Every transaction has 10 operations accessing data following the Zipfian distribution with theta equal to 0.5, whereas we set the write ratio to be either 100% or 50%. Each worker thread executes the transactions in a closed loop. We implement NoGC by generating versions but never collecting them, thus creating an illusion of infinite memory size. For the GC case, we consider SteamGC [2], a state-of-the-art scalable garbage collector for in-memory OLTP. Figure 2 shows the transactional throughput in each case. We observe that garbage collection causes 36% throughput degradation because the system has to track, identify, unlink, and finally reclaim the memory of transactionally obsolete versions.

In this paper, we leverage the inherent temporal properties of the version storage, where we observe that versions' validity in MVCC is temporally correlated. Effectively, transactions with similar profiles, which are started and executed in parallel in a multi-socket multi-core server, are likely to finish at a similar time. Based on this observation, all versions falling behind a given timestamp threshold can be collected altogether, thereby eliminating the costly version traversals during GC. TPC-C is an example of such a workload, given that `NewOrder` and `Payment` take 88% of the overall transactions in the benchmark. Accordingly, the versions that they created will also become unreachable at about the same time, and hence, we can collect them all together in *one shot*.

Our approach, OneShotGC, temporally partitions the version storage and assigns each partition to a group of transactions that started at (about) the same time. GC takes place at partition-level granularity, provided that all versions included in a given partition have become unreachable. This way, we fully eliminate version traversal at GC time, whereas we also provide mechanisms for transparent cross-partition version traversal, when it is required by a transaction (e.g., to find the proper version to read). Furthermore, a long-running transaction can delay the GC process by blocking the global minimum which leads to longer version chains and increased storage requirements for version storage. Similar to version chain pruning [2, 13], we enable partition-level consolidation to GC partitions that contain part of version chains that are unreadable by any active transactions. In summary, we make the following **contributions**:

- We devise a version storage design that physically partitions the versions according to their temporal properties and allows transparent, cross-partition version traversal.
- We propose a garbage collection algorithm, OneShotGC, that eliminates version traversals during GC, allows version consolidation without the requirement of per-version link updates and enables partition-level chain consolidation.
- We implement our approach in an in-memory OLTP engine and experimentally show how OneShotGC improves transactional throughput by up to 2x compared to the state-of-the-art. In addition, we show that diverse transactional workloads amplify the gains achieved by OneShotGC.

2 BACKGROUND

The core idea of MVCC is that every write operation (insert or update) results in a new version of a database record. A database snapshot consists of a single version of every record, and a consistent snapshot guarantees that there are no two records in the snapshot which have been created by conflicting transactions. A conflict on a record happens when at least one out of a set of two or more concurrent transactions is attempting to write the record. To avoid conflicts, while maximizing concurrency, MVCC allows transactions to read any version of the record as long as it was created before they started. This way, multiple transactions can read a record, while another transaction is updating it, by choosing the snapshot which includes the right version. Instead, there can only be one transaction updating a single record. As read operations can be executed concurrently even on the same version, all transactions are accessing the most recent version of the record that is available to them. Therefore, all versions created before the most recent version of a record that can be accessed by the least recent transaction currently active in the system can be dismissed.

In what follows, we describe the design options of versioned storage, the garbage collection process, and the impact of GC.

2.1 Version Storage

MVCC-based engines create versions on every write, resulting in a new database snapshot. We consider, at a logical level, that there exist two storage layers: (i) the main storage, and (ii) the versioned storage, which holds the version list. Then, the version list can be ordered in two ways: either from the newest to the oldest version (N2O), where the head of the version list is the newest version, or, from the oldest to the newest version (O2N), where the versions are stored in the order of creation. In what follows, we describe the three most common storage architectures [22] employed by MVCC-based OLTP engines.

Append-only storage. In append-only storage, the main and the versioned storage are combined. Upon creating a new version, a new record is inserted in the storage, and pointers from or to the previous versions are created, depending on whether the version ordering is N2O or O2N. For example, Hekaton [5, 12] uses append-only storage with O2N ordering. GC for append-only storage can be seen as a storage compaction process, that is, reusing the memory locations from the obsolete versions for new records or copying active versions from the tail to reduce storage fragmentation.

Time-travel storage. In time-travel storage, versioned storage is separated from the main storage. However, the versioned storage follows the same principle as append-only storage, where versions are stored in an append-only fashion, and they are connected through pointers. For example, a system that uses such an architecture is SAP HANA [13].

Delta storage. In delta storage, the main and the versioned storage are separated. The tuple metadata are stored in the main storage and they include a pointer to the version list if it exists. The version list resides physically in the versioned storage space. Versions are created by requesting the desired amount of space from the version storage, where they store every update. This approach is space-efficient as versions are created in a fine granularity and they may only store the updated

attributes instead of the whole tuple. Systems which use the delta storage design include Hyper [18] and Steam [2].

In this work, we design and optimize delta-storage-based versioning with N2O ordering, keeping the main or persistent storage as the latest tuple version and delta or transient storage as a list of tuple snapshots created in time. N2O ordering reduces the chain traversal in accessing the desired version. The delta-storage scheme maintains the separation of concerns and storage-specific optimization between the persistent and the transient (delta) storage.

2.2 Garbage Collection

Storage maintenance in MVCC is performed during the *garbage collection* (GC) phase, where the OLTP engine dismisses the versions of the records which will not be needed by future transactions. One way to see GC is as a filter on all available versions in the database. Even though GC is not required to ensure consistency, it makes sure that memory is used reasonably. Without GC, the number of snapshots constantly increases, exhausting the system memory and resulting in no usable space for further operations. On the other hand, GC brings significant overheads, and therefore, its optimization is critical for the overall performance [2, 22].

Version *Liveness* Tracking. During GC, the system identifies the versions which are globally unreadable by any active transaction, and then, reclaims them by (i) logically un-linking the version from the version chain, and (ii) physically by reclaiming the allocated storage space. Version identification and tracking [2] requires maintaining metadata about the version creation time, locating the version in the version storage, and determining the *liveness* of a version: whether the version is visible to any active transaction. Tracking and identifying obsolete versions may be done at epoch, batch, or per-transaction level.

Hyper [18] and Steam [2] employs global and thread-local transaction lists for version identification and liveness tracking, per transaction. BOHM [6] tracks at transaction batch and identifies obsolete versions by grouping transactions in epochs. Deuteronomy[14] and ERMIA [10] track and identify versions created through epoch grouping of transactions. Regardless of the approach for version liveness tracking, GC in existing approaches has to consider all records (or tuples, tables, etc., depending on the granularity of the version storage) and then still go through every version and explicitly dismiss it.

Complexity. The complexity of GC depends on the selected version tracking and identification method. In exact GC, where all obsolete versions are reclaimed timely, the complexity of GC is $O(k+n)$, where k is the number of tuples, which have at least one version in the versioned storage, hence a valid version chain, and n is the total number of obsolete versions in the system. This is due to the fact that during GC, the system traverses all the versioned tuples, and reclaims the obsolete versions, summing to $k+n$ operations of random access memory operations.

2.3 Space/Time effects of GC

MVCC is a snapshot-on-write approach, that is, a new version is created per update. As a result, MVCC has significant storage requirements and can generate long version chains, causing performance penalties when accessed.

Storage consumption. While versioning scales readers by having at least one readable snapshot by any active transaction, it comes at the cost of storage space. GC directly impacts the system performance by reclaiming the memory of obsolete versions. In the optimal case, the maximum number of versions in the system should be the number of concurrent updates across all active transactions, that is, every tuple has a readable version for every active transaction. However, depending on the GC frequency and granularity, the version reclamation may be delayed, causing unclaimed memory which is unreadable to any transaction. Furthermore, if the system doesn't

support version consolidation, the GC may be delayed by a long-running transaction, holding off by the tail of the version chain, hence, prohibiting reclamation of in-middle unreadable versions.

Traversal Length. Version chains are generally implemented as linked lists, causing random access on traversals. A transaction, upon reading from versioned storage, incurs random access until it finds the appropriate snapshot in time. Traversal length depends on the version ordering in the chain; In the case of N2O ordering, the reader will have to traverse a chain with as many versions as the expected number of concurrent updates on the same record, performed by transactions running concurrently to the reader. This case is reversed in the case of O2N ordering, where the number of traversals depends on the frequency of GC. If GC occurs too infrequently, the reader has to traverse all obsolete versions before finding the appropriate readable version. In addition to linked lists, vWeaver [9] optimizes for scans in MVCC by using frugal skip-lists per record for the version chains. Thus, vWeaver provides probabilistically logarithmic, with respect to the actual list size, accesses for readers.

Furthermore, in the case of mixed workloads, where a long-running transaction holds the GC process, causing a long-tail version chain, the old-running reader pays the traversal length, even for globally obsolete in-middle versions as concurrent updates cause the chain to grow. With chain consolidation, implemented as EPO [2] in Steam and interval-based GC [13] in SAP HANA, GC also cleans in-middle versions, hence, causing chain-length to be at-max the number of active transaction in the optimal case. OneShotGC supports Steam EPO-based consolidation for reducing chain length and vWeaver-based version chain layouts for efficient version accesses. Furthermore, section 5 introduces a mechanism for partition-level consolidation in OneShotGC for both, reducing chain length as well as reclaiming memory from obsolete versions.

3 SCALABILITY BOTTLENECKS OF GC

Garbage collection includes three steps: (i) Track and identify the obsolete versions, that are, the versions which are no longer readable by any active transaction and can be safely removed; (ii) Unlink obsolete versions from active version chains of records; (iii) Clean and reclaim the memory allocated to obsolete versions. In the following, we explain the scalability bottlenecks in each step of the GC process and provide an overview of how OneShotGC overcomes these bottlenecks.

Tracking/Identifying obsolete versions. The DBMS implements version tracking and identification by introducing intermediate data structures to track the versions created by each write operation. During GC the system identifies which versions are eligible for removal. Tracking is performed at either transaction-, epoch-, or tuple-level and version metadata may be stored in global or local transaction lists or maps, as well as in snapshot tracking data structures. Regardless of the mechanism used to track and identify versions in the system, the DBMS needs to traverse the list of obsolete versions, and process them for GC, thereby incurring random accesses whose number is proportional to the number of versions in the system. As a result, the GC overheads increase with the write throughput of the system, essentially penalizing the system for achieving high performance, since GC will need to clear a big number of versions that are created by the write operations.

OneShotGC builds on the observation that the lifetime of versions in MVCC follows a sliding temporal window fashion. Accordingly, all versions falling behind a global minimum threshold can be collected as a whole, thus eliminating the need for individual version maintenance. Temporality-aware version storage enables OneShotGC to scale by grouping the versions based on their expected deletion time, instead of the traditional, flat version storage used by state-of-the-art GC approaches.

Unlinking/Updating individual version chains. GC requires updating physical or logical pointers to and from other versions. Whenever a version is GC'ed, either the previous version or the head of the version chain needs to be updated to remove invalid pointers, and only then,

the version memory can be safely reclaimed. Unlinking and updating individual pointers require accessing individual versions, limiting the scalability of the GC process.

OneShotGC uses *partition tags* and *tagged pointers* to implicitly invalidate all pointers in a partition whenever a partition is GC'ed.

Memory Reclamation. GC is essential, not only to reduce version chain length and traversals but also to reclaim memory timely. Write operations create new versions, essentially translated to a new database snapshot for every write, in order to allow concurrent readers to access a given snapshot without waiting for the writer to finish. Old data versions become obsolete when no active transaction can read them. The GC removes the obsolete versions in order to reclaim and recycle the memory for the transient storage of the system. Individually reclaiming the memory of each version requires random accesses, limiting the scalability of GC. Further, memory reclamation causes internal memory management overheads due to fragmentation maintenance.

OneShotGC simplifies memory management by abstracting each partition as a linear memory allocator. Then, during GC, the entire partition is reclaimed as a whole, through a single pointer swap, effectively resetting the allocation cursor to zero which is essentially the starting address of the partition.

Version chain consolidation. A long-running transaction delays the advancing of the global minimum for the set of active transactions. Traditional GC approaches only dismiss versions based on global minimum, that is, dismissing versions from the tail of the version list only, allowing very long version chains in the presence of long-running transactions. Recent proposals [2, 8, 13] for chain consolidation extends the traditional GC by dismissing globally unreadable versions within the middle of version chains, essentially removing all versions that are not required by any active transaction in the system. Consolidation helps reduce traversal lengths for long-running transactions and reclaims memory timely. However, chain consolidation requires traversing the active version chains and testing each version for readability by any active transaction, and if not readable, then dismissing the specific versions.

OneShotGC abstracts partitions as a single big version, having min and max readability boundaries, and then performs consolidation at the partition level. OneShotGC enables transparent cross-partition traversals even if the in-middle partition in the version chain was dismissed in the context of consolidation.

4 TEMPORALITY-AWARE VERSION STORAGE

In delta-storage-based MVCC, the OLTP engine maintains two types of storage: The first type is the *persistent storage*, which is the engine's main, transactionally consistent storage backend. The second type is the *transient storage* that the transaction manager uses to temporarily allocate memory, such as the private storage for every transaction and the version storage. This section focuses on the design of the version storage and its functionality for reading and writing operations. It assumes that the version storage has a fixed pre-allocated memory size that it manages independently. Even though changes in the allocation are possible through runtime interaction with the transient storage, they are orthogonal to the main contributions.

The version storage follows the delta storage design principles to provide snapshot isolation guarantees [22]. Every version has a timestamp reflecting its creation time, and it must be retained until no transaction will need to access it. Garbage collection is responsible for removing all obsolete versions and returning all memory allocations to the version storage.

OneShotGC exploits the temporal nature of version creation and reclamation in MVCC and, therefore, partitions the versions into temporal groups that are expected to be collected together. Temporally grouped partitions enable garbage collection to reclaim all the memory allocations belonging to a given partition at once, eliminating the need to *track*, *identify*, *unlink* and *reclaim*

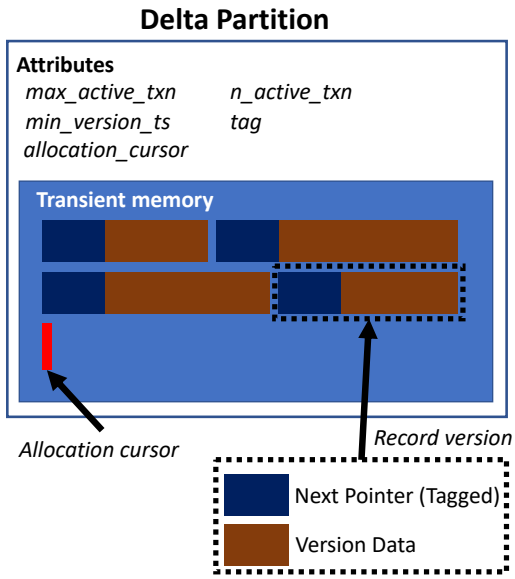


Fig. 3. Delta partition layout

obsolete versions individually. Note that OneShotGC differs from coarse-grained tracking/identification approaches. Specifically, in the latter, even though all obsolete versions can be tracked and identified at the end of epoch or batch boundary, each version must be reclaimed individually, which limits GC scalability. Instead, in OneShotGC, the main design principle is to avoid costly random access during GC by exploiting the temporal nature of versioning in MVCC.

The remainder of this section describes the data structures and protocols which enable OneShotGC to eliminate random-access traversals during the GC process, including invalidating all obsolete version pointers and reclaiming the allocated transient memory.

Delta partitions. The version storage is arranged as a circular buffer of N partitions, where each partition serves as a transient memory provider. On system startup, each partition is initialized and granted ownership of a memory chunk. The allocated memory is used to serve memory requests for record versions and is garbage collected when the partition's visibility window goes out of the scope with respect to active transactions in the system.

Figure 3 shows the layout of a delta partition. Each partition keeps the record of the latest active writer (*max_acrtive_txn*) in the partition, the oldest version (*min_version_ts*), the number of currently active transactions assigned to this partition (*n_active_txn*), the partition's tag, and the allocation cursor. Each record version is stored as a pair of the next pointer and the version data itself, where the next pointer points to the older version in the list if it exists or a null pointer to mark list termination. Essentially, on a high level, each partition can be seen as a single version, having a visibility window from the timestamp of the oldest version in the partition to the latest transaction, which created a version in the given partition. Therefore, whenever the partition's visibility window is not readable by any active transaction, the partition can be safely garbage collected as a whole in *one shot*. Figure 4 depicts the partitioned version storage by specializing the case originally presented in Figure 1. The latest versions are stored in-place in the persistent or the main storage, while older versions are pushed into the version storage. The version chain shows the transaction id that created each version and associated it with a given partition. When the versions

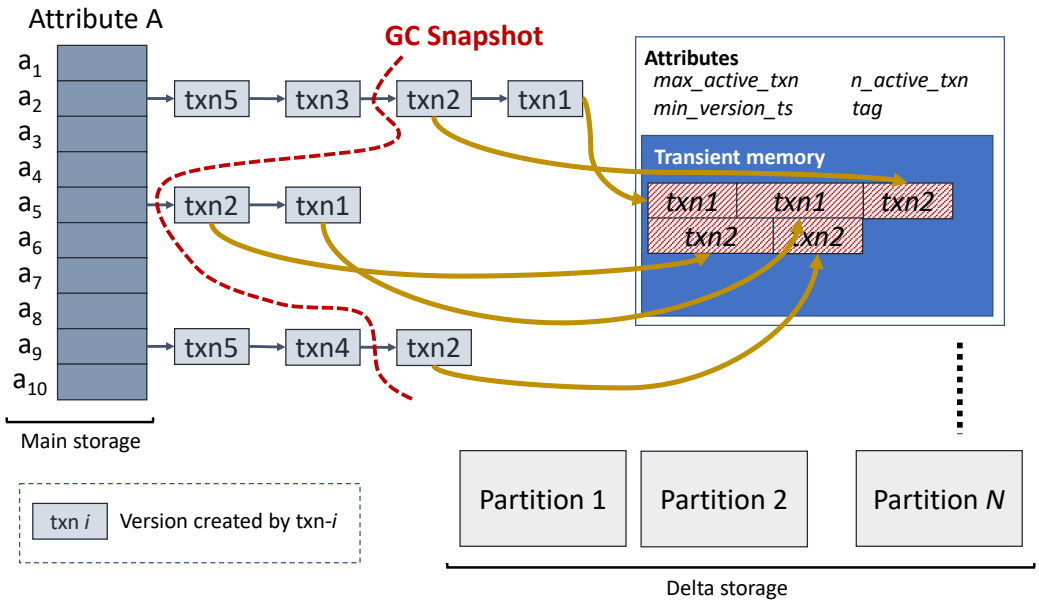


Fig. 4. Partitioned version storage

that belong to the GC snapshot become unreachable, the version storage will reclaim the whole partition P_N. On the other hand, if there was another transaction, say txn_k with a record version in P_N which was not unreadable, then GC would reclaim no memory area of P_N. When a transaction creates a new version for any given record, the version storage places it inside a partition following a given policy. The default policy in our implementation is the incremental assignment policy. In the incremental assignment policy, a number of transactions, say M, are assigned to the first partition, and then the next M transactions will be assigned to the next partition in the sequence. Recall that partitions are arranged in a circular buffer fashion, hence after utilizing all partitions, transactions would be again assigned to the first partition. The insight here is that for a transactional workload with a mostly uniform transactional profile, the first N transactions started are likely to finish at the same time, hence triggering the GC process exactly when the Mth transaction finishes. Whereas, in the case of stragglers, the GC can be delayed until the last transaction finishes in the first batch but does not block the further batches to consume the next partitions in the sequence. Ideally, the M should be greater than the maximum concurrent transactions allowed in the system, allowing a batch of transactions to be assigned to the same partition which is likely to be finished together. However, the policy, the number of partitions, and the partition size are orthogonal to the storage design, hence discussed separately in section 6.

Partition tagging. Every partition maintains a monotonically increasing tag, which denotes the validity of the memory allocated by the corresponding partition in the version storage. Upon garbage collection, the partition tag increases by one, rendering all the previous tags invalid. The partition tag is included in every version, as every memory chunk storing a version is referenced by *tagged pointers* instead of physical memory pointers. Tagged pointers encapsulate the physical memory pointer, the partition identifier, and the partition tag at version creation time. Specifically, in our implementation, tagged pointers are 64-bit unsigned integers, having an 8-bit partition id, 20-bit partition tag, and a 36-bit memory offset which results in 64GB addressable memory per

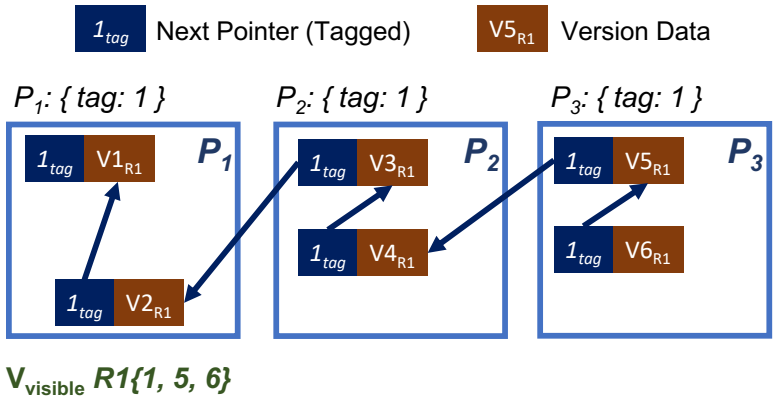


Fig. 5. Version Chains across delta partitions
($V_{visible}$ are the only readable versions by active transactions)

partition while allowing a maximum of 265 partitions. Dereferencing a tagged pointer first checks for tag validity with the corresponding delta partition; then, the physical pointer is formed by adding the memory offset to the partition's base address.

Figure 5 depicts three partitions that include six different versions of the same record. The left-hand side of every memory chunk includes the pointer to the next version in the chain, while the right-hand side includes the record itself. In the case where partition $P1$ was GC'ed, the partition's tag would be incremented to 2 from 1, invalidating all versions within the partition. If any reader tried to dereference the next pointer of $V3_{R1}$, pointer tag validity would fail given the tag mismatch between the tagged pointer and the corresponding partition's tag.

Partition tagging combined with tagged pointers enables efficient *linking/unlinking versions* from the respective versions chains upon a successful GC through incrementing the partition's tag, invalidating all versions contained within the corresponding partition implicitly. Therefore, GC does not need to traverse individual versions to unlink versions from version chains and can manage the validity of versions through the version and partition tags.

Transaction registration. Every transaction that creates a new version registers to the version storage, which assigns the transaction to a partition. Figure 6 shows the algorithm for transaction registration and de-registration. Each partition maintains the number of active transactions assigned to it, and as garbage collection is performed at the partition level, each partition also maintains a partition-wide maximum active transaction timestamp. Upon transaction registration (function *register_txn* in Figure 6), the delta partition waits if the current partition is locked for GC and then registers the transaction by incrementing the active transaction count. Further, it updates the maximum transaction timestamp, if required. Similarly, when a transaction finishes, it deregisters (function *deregister_txn* in Figure 6) itself from the assigned partition by decrementing the number of active transactions. The version storage can safely reclaim the partition when there are no active transactions and when the maximum transaction timestamp is unreachable.

Version chain traversal. Whenever a transaction cannot read a tuple from in-place persistent storage as it is newer than the transaction's start time, it reads a snapshot of the requested tuple from the versioned storage. In our design, version pointers are stored in-place with the tuple meta-data along with tuple visibility timestamps. Recall that in our design, all version pointers are tagged pointers. In case a transaction needs to access the version storage, it dereferences the tagged pointer, and checks if the version is readable; if not, it traverses the version chain until it finds a

```
1  void DeltaPartition::register_txn(timestamp_t xid){
2      while(this->n_active_txn < 0) // wait if GC is in-progress
3          ;
4      this->n_active_txn++;
5      if(this->max_active_txn < xid)
6          this->max_active_txn = xid;
7  }
8  void DeltaPartition::deregister_txn(){
9      if (this->n_active_txn-- == 0)
10         oneShotGC();
11 }
12 void DeltaPartition::oneShotGC(){
13     int64_t expected= 0;
14     // Lock the partition for active readers by setting to negative large number to denote active
           ↪ GC
15     if(n_active_txn.compare_exchange_strong(expected, INT64_MIN)){
16         if(global_minActiveTxn() > this->max_active_txn){
17             tag++;
18             this->reset();
19         }
20         this->n_active_txn.store(0);
21     }
22 }
```

Fig. 6. Algorithm for OneShotGC

readable version. Although in MVCC, if a transaction cannot read the latest version, it is guaranteed that there exists a version in the version chain that should be readable by the transaction, and thus, checking pointer validity is redundant, for sanity reasons, our implementation asserts on pointer validity through the associated partitions tags.

Memory allocation & record update. Every partition acts as a linear memory allocator. It maintains a single base memory pointer and an atomic allocation cursor. Writers request memory from the assigned partitions by providing the memory size and the version's timestamp. The delta partition stores the timestamp if it is older than the oldest version in the partition for facilitating GC in determining partition-wide visibility of any active transaction in the system. Then, it allocates the requested memory through the cursor's $fetch\&add$ operation. Finally, it returns a tagged pointer to the writer, which encapsulates the physical memory pointer and its validity given the partition's current tag. The writer then initializes the allocated memory with the version's data and timestamp, sets the next pointer to the current head of the version chain, and finally updates the in-place version head in the record's metadata.

Garbage collection. OneShotGC leverages the partitioned layout of the version storage to free unreadable and obsolete versions in one shot, thereby avoiding the traversal of long version chains. Figure 6 shows the algorithm for OneShotGC (function *oneShotGC* on line 12). Garbage collection is initiated by every transaction when it deregisters from the version storage by triggering the GC manager, which first locks the partition for blocking any new transactions getting assigned while GC is in process. Effectively, OneShotGC treats every partition as a single version with a given timestamp, corresponding to the maximum timestamp of the versions included in the partition. Accordingly, garbage collection will be triggered when there is no active transaction in the system that will need to access the partition. Partition-level garbage collection is implemented as a simple

pointer reset. Every new memory allocation is performed from the beginning of the partition. The partition's tag is also incremented by one to invalidate all references to versions contained within the corresponding partition in order to prevent illegal memory accesses.

In OneShotGC, as the GC takes place at partition-level granularity, hence, the complexity of the GC process itself is $O(m)$ for m partitions, that is, when there the entire partition is unreadable, GC takes place in one shot, thereby, fully eliminating version traversals during the GC process.

5 CHAIN CONSOLIDATION

Snapshot isolation requires the existence of at least one snapshot for each transaction, that is, one version of each tuple in the database. Usually, DBMS employ GC approaches that reclaim only based on the global minimum, that is, reclaiming versions only from the tail of the version list which is older than the oldest active transaction. Such an approach can cause the version chains to grow abruptly long in the presence of long-running transactions with concurrent writers, causing two main problems: the size of transient memory allocations grows unnecessarily large even though the maximum number of active versions should be the number of active transactions in the system, and secondly, long-running transactions would require to traverse long versions chains in order to access appropriate snapshot version.

Chain consolidation is an optimization for transactional garbage collection algorithms. In addition to reclaiming the obsolete versions from the tail, it also reclaims globally unreadable versions in the middle of version chains, essentially removing all versions that are not required by any active transaction in the system. Consolidation helps reduce traversal lengths for long-running transactions and reclaims transient memory timely, keeping the transient storage cost minimum. However, chain consolidation requires traversing the active version chains to identify the versions that can be reclaimed or merged and then update the chain links.

SAP HANA's interval-based GC [13] and Steam GC's eager pruning of obsolete versions (EPO) [2] performs chain consolidation at the version level, collecting in-middle unreadable versions. In Steam GC with EPO, GC and consolidation are interspersed with transactions. Essentially, Steam GC with EPO checks and prunes obsolete versions during every update of the tuple, that is, whenever the version chain is extended by a new version, ensuring that the version chain will never grow to more versions than the number of active transactions, as well as having no obsolete versions. Instead, SAP HANA's interval-based GC is performed by a background thread, triggered periodically.Version-level consolidation minimizes the chain length, reducing chain traversals for readers, as well as, reducing unnecessary transient storage allocations. However, consolidation incurs the cost of additional version traversals, pruning and re-linking version chains, and reclaiming the memory of pruned versions. OneShotGC builds upon temporally-partitioned versioned storage, as described in Section 4. OneShotGC allows any optimization for reducing chain traversals, including eager pruning of obsolete versions like Steam or SAP HANA or having a skip-list instead of a linked list [9]. However, OneShotGC defers the actual memory reclamation until the partition containing the version is collected as a whole.

OneShotGC enables consolidation at the partition-level, instead of version-level. Partition-level consolidation is required when the GC manager reclaims a partition containing versions that are not the oldest ones in the chain. Accordingly, the gap created in some version chains needs to be bridged to allow transactions to traverse the version chain, transparently. Figure 7 exemplifies the partition-level consolidation case. Suppose that the GC manager frees the whole partition P_2, increasing its tag from 1 to 2. Then, version 5 of P_3 will lose the link to version 4 in P_2 and version 2 will be unreachable from version 5 and so will version 1, despite being readable by some active transaction.

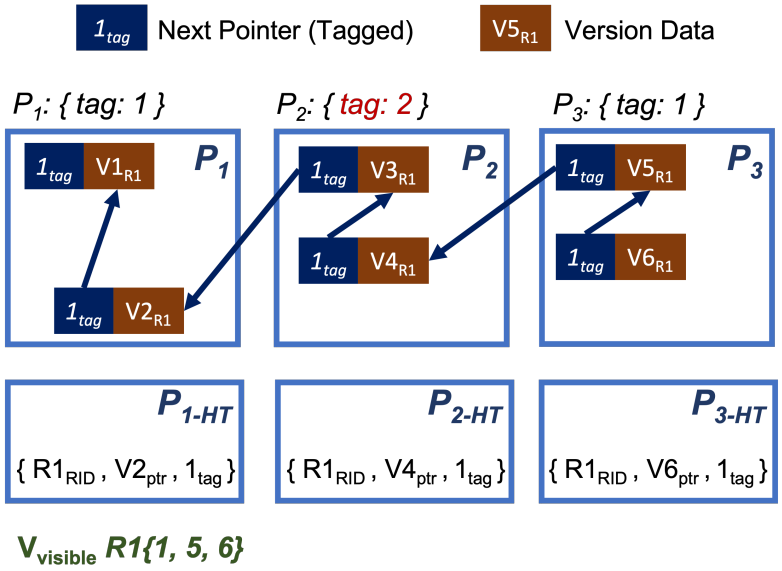


Fig. 7. Delta storage with partition-level consolidation ($V_{visible}$ are the only readable versions by active transactions)

OneShotGC bridges the chains by introducing an additional data structure that includes the most recent version of every record included in each partition. We implement this data structure in our system as a hash table, denoted as P_{i-HT} in Figure 7. Every entry in the hash table includes the record identifier, the partition tag when the entry was created, and a reference to the version entry in the partition. When the GC manager frees a partition P_i, it increases the partition tag, thus rendering all tagged pointers to the freed versions invalid, while also clearing the P_{i-HT} table. We recall that a tagged pointer encapsulates the partition, the tag, and the memory address of a version.

In the following, we explain the consolidation steps in different phases of the transactional workload execution.

Version chain traversal. The simplest case in version chain traversal is for all versions to be stored in the same partition. However, this is typically the exception rather than the norm, and cross-partition traversal is often required. If the next version is stored in a different partition, say P_i, then the transaction traversing the chain will check whether the tag of P_i is the same as the one included in the tagged pointer. If this is not the case, the transaction will probe every P_{i-HT} in reverse order to find the next version in the list. Probing hash table P_{i-HT} until finding the next readable version is required since it is not known which intermediate versions have been removed.

Figure 7 shows an example where consolidation is required. Suppose that a transaction needs to access $V1_{R1}$ based on its timestamp. Traversal will start from the most recent version, that is $V6_{R1}$. After moving to $V5$, the transaction will then check the validity of the tagged pointer to $V4$. Since the tag of P_2 has become 2, the transaction will check P_1, where it will find the next available version, that is $V2$, which in turn will lead to $V1$.

Figure 8 illustrates a more detailed example where a transaction $txn3$ requires a cross-partition traversal to access the corresponding readable version $V2$. As shown in the figure, there are six delta partitions, where four partitions (P1, P2, P3, P6) have not been garbage collected yet while

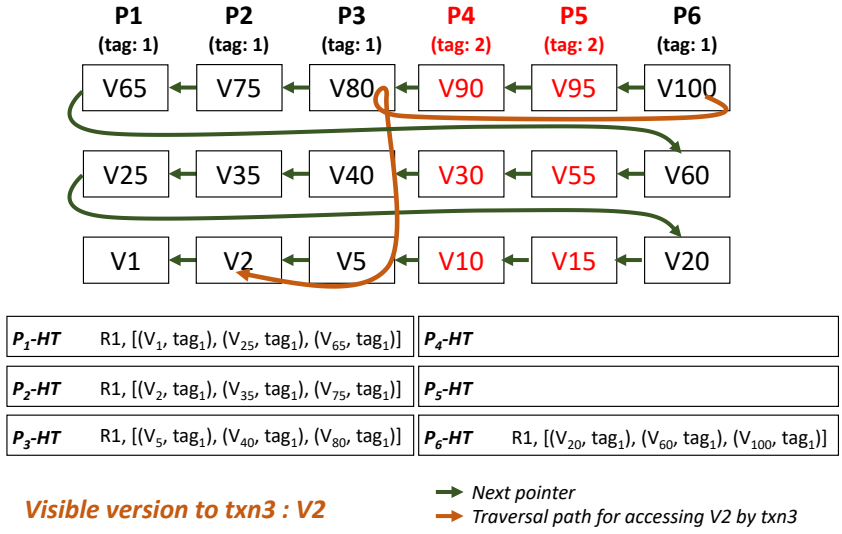


Fig. 8. Version traversal with cross-partition traversal: P4 and P5 are consolidated, hence $txn3$ traverses chain, starting from P6, then P3 and P2.

P4 and P5 have been reclaimed, and hence the version chain for record $R1$ had been implicitly consolidated. We show three wraparounds in the version list to demonstrate that the versions list spans across all the delta partitions multiple times. For simplicity, we show that each partition contains only one version of $R1$. Suppose that $txn3$ requires access to a version of the record $R1$, thereby requiring cross-partition traversal. The version traversal starts at the most recent version, $V100$ in P6. Then, for the next version, $txn3$ observes that the next pointer of $V100$ is invalid. It probes the hash tables P_{5-HT} and P_{4-HT} for intermediate version list heads for record $R1$. Following on, the transaction will probe P_{3-HT} to get a list of intermediate version list heads, from which the transaction will select the minimum invisible version, skipping over the invisible part of the list, and from thereon, will access $V5$, and finally the readable version $V2$.

As in the default transaction assignment policy, a number of transactions are assigned incrementally to each partition in a round-robin fashion. The probing of the hash tables occurs in reverse order to find the next valid and visible version in the list (P5, then P4, and P3 in the example shown in Figure 8). Probing each partition causes additional accesses in the traversal path. However, each partition is expected to contain a chunk of the version chain (one or more versions), skipping many obsolete versions in the chain and amortizing the cost of additional probes.

Record update. When a new version V_a of a record is created, then it is inserted in the persistent storage, shifting the already existing one, say V_b, to the version storage. The GC manager provides the partition where the shifted version should be stored, say P_i, and V_a installs a tagged pointer to V_b that is now placed in the version storage. GC manager also updates P_i if the version V_b is older then the oldest version in P_i. Knowing the oldest version timestamp for each partition is essential for consolidation eligibility during GC. Then, the transaction checks whether the next version in the chain, pointed by V_b, is stored in another partition than P_i, say P_j. If this is the case, then it updates the hash table of P_j to include the most recent version of the chain in that partition, which is the version pointed to by V_b.

Garbage collection. Enabling partition level consolidation adds another step to OneShotGC's algo-

```
1  void DeltaPartition::oneShotGC_with_consolidation(){
2      int64_t expected= 0;
3      // Lock the partition for active readers by setting to negative large number to denote active
           ↪ GC
4      if(n_active_txn.compare_exchange_strong(expected, INT64_MIN)){
5          bool doGc = false;
6          if(global_minActiveTxn() > this->max_active_txn){ // try full GC
7              doGc = false;
8          } else{
9              // consolidate if partition is unreadable by any active transaction
10             bool consolidate = true;
11             for(auto txn: activeTransactions){
12                 if(txn <= max_active_txn && txn >= min_version_ts){
13                     consolidate = false;
14                     break;
15                 }
16             }
17             if(consolidate)
18                 doGc = true;
19         }
20         if(doGc){
21             tag++;
22             this->reset();
23             this->hashTable.clear();
24             min_version_ts = INT64_MAX;
25         }
26         // unlock partition for active writers.
27         this->n_active_txn = 0;
28     }
29 }
```

Fig. 9. Algorithm for OneShotGC GC with consolidation

rithm: check for interval exclusion. Figure 9 shows the algorithm for OneShotGC with consolidation. The main difference is that, when a partition becomes inactive, that is, no active transaction is assigned to a partition, it first tries to do a full GC (line 6) given all versions created in that partition are obsolete, or if the full GC condition fails, algorithms check that if the versions contained in the partition are readable by any active transaction in the system (line 11-16), if not, then they are safe to be reclaimed in one shot. Once the partition is deemed as GC-able, the GC manager proceeds (lines 20-25) by incrementing the partition's tag, clearing the corresponding hash table, resetting the oldest version timestamp, and finally, resetting the memory allocation pointer. As described before, all version pointers are tagged, hence, any partition crossing chain would be rerouted through the partition's hashtable, which contains the pointer to the most recent version for the corresponding partition.

Trade-offs. The consolidation of the version chain reduces the chain length, and hence the number of random accesses required to traverse the whole chain. However, this comes at the cost of additional indirection for cross-partition accesses of readers. Further, writers have to insert the reference to the most recent version of the partition in the corresponding table. Nevertheless, if there was no consolidation, then a single long-running transaction can lead to long version

chains, causing long version traversals for readers and unnecessary growth of version storage. Furthermore, OneShotGC gives up GC precision causing deferred GC of unreadable versions within a partition until the whole partition is unreadable by any active transaction. However, OneShotGC eliminates random access version traversal during the GC process, including version identification, list rearrangement, and memory deallocations.

6 TUNING ONE-SHOT GC

OneShotGC improves garbage collection performance by exploiting the temporal properties of the workload. Temporality-aware version storage enables OneShotGC to scale by grouping the versions with the same temporal properties, instead of the traditional version maintenance used in existing GC approaches. Effectively, it groups versions that are expected to become unreachable at the same time by storing them in the same partition. OneShotGC physically partitions the version storage into temporally grouped partitions and then tracks version liveness at the partition level. During GC, logical un-linking happens automatically as a by-product of employing tagged pointers, while physical memory reclamation occurs in a single pointer-swap for each partition, that is, in one shot. OneShotGC decouples the versioned storage design and governing policies and thereby enables the system to modify transaction assignment policy, the number of partitions, and the size of each partition at runtime without blocking the concurrent transaction processing.

The main challenge in tuning OneShotGC is grouping the versions that are expected to be garbage collected together, as the version lifetimes are not dependent on the transaction that created it, but on the transaction which may or may not access it. However, grouping versions based on their readability by concurrent transactions requires the writer to account for all the active concurrent transactions in the system, which in turn becomes the scalability bottleneck for in-memory OLTP. On the contrary, in disk-based MVCC [9], the OLTP engine gets the second opportunity to rearrange versions based on expected lifetimes.

Workloads that have transactions with similar profiles do not require many partitions. Specifically, our experiments show that in traditional transactional workloads, having even as few as two partitions can be adequate to leverage OneShotGC. The reason is that short-lived transactions start and finish at about the same time. Therefore, their execution converges to a batch model, which is clearly the best fit for OneShotGC. In our implementation, we use a default naive policy that sets the initial number of partitions to two, with the partition size of 10GB each, and then assign a number of transactions to a partition incrementally in a round-robin fashion. However, real-world workloads typically include transactions with different profiles, where some transactions may take longer than expected, resulting in a larger temporal window.

As a default adaptive policy, accounting for most workloads, the system may set a default number of partitions (minimum two) and a default partition size. Default partition sizing can be based on transactional profiles, accounting for the size of versions created by each transaction type multiplied by the provisioned concurrency of the system so that a partition can serve transactions without expansion. For transaction-to-partition assignment policy, the system abstracts the delta-partitions as a queue of infinite size. Then, it assigns a number of transactions to each delta-partition incrementally and expands the queue as needed. GC'ed partitions are inserted back into the queue for reuse, automatically adjusting the number of partitions and their size based on the workload. Suppose that we set the number of transactions per partition to one. In this case, the configuration will converge to transaction-level GC, similar to SteamGC, where each partition will act as transaction-private storage and be reclaimed when the entire transaction falls behind the global minimum.

7 EXPERIMENTAL EVALUATION

This section includes the results of our experimental evaluation. First, we describe the hardware that we used to execute our experiments, then some essential details of our software, and finally the benchmark that we used to derive our workload. Finally, we present the results of our analysis, categorizing them with a focus on each aspect of the proposed approach, including scalability, tuning parameters, and consolidation under mixed workloads.

7.1 Hardware & Software setup

Hardware. All the experiments were conducted on a server equipped with 2x14-core Intel Xeon Gold 6132 processor (32-KB L1I + 32-KB L1D cache, 1024-KB L2 cache, and 19.25-MB LLC) clocked at 2.60 GHz, with Hyper Threads, summing to a total of 56 logical threads (28 physical-threads + 28 hyper-threads), and 1.5-TB of DRAM, running Ubuntu 20.04 operating system.
Software. To have a fair comparison, we implemented all approaches into the prototype HTAP system Proteus [19, 20]. OLTP engine employs MV2PL concurrency control protocol with snapshot isolation. For OLTP multi-versioning, the system uses delta storage with newest-to-oldest ordering, and copies the entire record as the version which can be extended to use attribute-level before-images as versions, however, is orthogonal to the contributions of OneShotGC. OLTP engine also maintains an index, implemented using cuckoo hashing [17]. The index contains the transactional timestamps, tagged delta pointer, and logical record pointer in the form of RowIDs (RID). The storage layout for OLTP is columnar. Workers are started and pinned on each available physical thread and all transactional workers generate and execute transactions in a closed loop, simulating a full transactional queue. Unless stated otherwise, we do not use hyper-threads to remove performance artifacts of micro-architecture interference in hyper-threads. Before every experiment, we execute a warm-up phase, and then we report the system's steady-state latency and throughput.

Baseline. We compare OneShotGC with Steam GC [2] as the baseline. For a fair comparison, we implemented Steam GC in Proteus, and to remove any physical memory management overheads and analyze only the effects of version creation, access, and removal, we create a thread-local memory allocator. Each memory allocator, on start, reserves and faults 4GB of memory, expands if required, and serves memory requests from the transaction-level memory allocator in the case of Steam GC configuration.

Benchmark. We performed experimental evaluation using two benchmarks, TPC-C and YCSB and extended them to simulate mixed-workload, having an OLAP-style long-running transaction.

TPC-C is an industry-standard benchmark for evaluating transactional systems, replicating a workload of retail order processing. TPC-C has five transaction types: `NewOrder`, `Payment`, `Delivery`, `OrderStatus`, and `StockLevel`. In a full mix, each transaction's type is selected at random from the aforementioned types, with a probability of 45%, 43%, 4%, 4%, and 4%, respectively. We execute two mixes of TPC-C: the first one contains `NewOrder` and `Payment` only, with the ratio of 51% and 49%, respectively, simulating a write-intensive workload, whereas the second mix contains all five transaction types.We assign one warehouse to each transaction worker, which generates and executes the transaction in a closed-loop for the specified duration.

YCSB is a key-value style benchmark designed for testing and analyzing the scalability of transactional workload in a system. We implement a YCSB-style workload in our system, using a single table with 10 columns of 8-bytes each. We set the total number of records as $1000 * num\_workers$ to have similar record access and write distribution across workers. Each transaction composes of 10 operations, and then, we show both, 50% and 100% write ratio, using read/write for stressing version subsystem while write-only for stressing version creation and GC subsystem, respectively.

To simulate *mixed workload*, we extend the aforementioned benchmarks to include an OLAP-style query, which scans and aggregates integer columns. Using a column scan-and-aggregate query, we simulate a long-running reader, which reads values according to the assigned transaction timestamp and may block GC, given the long-running nature compared to transactions executed in parallel. Specifically, for YCSB, we scan-and-aggregate first N column, while for TPC-C, we aggregate, 1 to 4 columns of `TPC-C Stock` relation, that are, `s_quantity`, `s_ytd`, `s_order_cnt` and `s_remote_cnt`, reason being that these columns are also update intensive under `NewOrder` transaction, therefore, causing the long-running query to conflict with concurrent transactions, and thereby, reading appropriate snapshot from versioned storage.

7.2 Scalability

In this section, we analyze the scalability of OneShotGC and compare it with Steam. In experiments with OneShotGC, we use two delta partitions, 10GB each, allocated as 5GB per NUMA socket, and assign 1024 transactions to each delta in sequence.

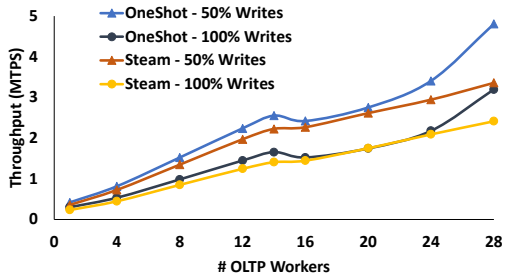
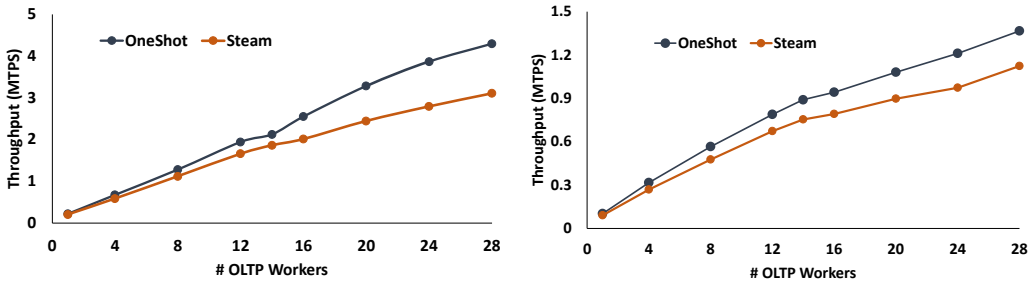


Fig. 10. Scalability of OneShotGC with YCSB workloads

YCSB. Figure 10 evaluates YCSB scalability having 50% and 100% write ratio and reports throughput corresponding to the workload and number of OLTP workers. Steam scales with a change after 14 threads as the transactional engine spans two NUMA sockets and pays off cross-NUMA loads for acquiring minimum active transactions from each thread-local transaction table after each transaction end. OneShotGC also pays a performance penalty after crossing the socket boundary but stabilizes and scales again at 24 and 28 threads; the performance penalty in crossing the NUMA boundary is due to the fact that given the YCSB access distribution and the unbalanced number of threads across sockets, workers from the second socket execute the majority of transactions on data from the first socket. Unlike Steam, OneShotGC does not perform GC at the end of each transaction, thereby the cost of acquiring minimum active transaction is spread across transactions, and only a single worker performs GC, that is, when the number of active transactions on any given delta partition becomes zero, and the partition is unreadable by any other active transaction in the system. This simulates transactions with a similar profile; when the transactions are assigned to the second delta partition, the first delta partition becomes obsolete and can be garbage collected altogether. Overall, with 28 threads, we see 30% and 25% gains for read-write and write-only workloads over Steam.

TPC-C. Figure 11(a) evaluates performance scalability with TPC-C workload, having `NewOrder` and `Payment` transaction mix. We observe a similar behavior as with YCSB, and both approaches pay a cross-socket load penalty after crossing socket boundaries. However, OneShotGC scales better across socket boundary as only a single thread, the last worker to deregister from the corresponding delta partition, performs GC after the partition becomes unreadable under SI guarantees. Compared



(a) TPC-C (`NewOrder` and `Payment` Txn)

(b) TPC-C (All transactions)

Fig. 11. Scalability of OneShotGC with TPC-C workloads

to Steam, which has a traversal cost upon GC, OneShotGC does not traverse version chains during GC; however, it does pay in atomically registering and de-registering transactions from delta partitions, once per transaction, and thereby, the performance gain increases with the number of threads.

Figure 11(b) shows OneShotGC's scalability when we include all five transaction types of the TPC-C workload. Specifically, we repeat the experiment described in the previous paragraph, but this time we include all the transaction types of the workload. For both OneShotGC and Steam, the scalability is similar to the one of figure 11(a). Traversal-based GC approaches, e.g., Steam, incur overhead for traversing the versions during GC, which increases as more transactions are active in the system. In contrast, OneShotGC does not require version traversal during the GC process and performs garbage collection of an entire delta-storage partition in one shot – incurring minimal overhead. As a result, even though the TPC-C workload contains different write-intensive and read-only transactions, OneShotGC scales better.

Summary. OneShotGC registers and de-registers each transaction to a delta partition, introducing two additional atomic operations per transaction. However, this cost is amortized by eliminating costly version traversal during the GC process. Note that, during GC, the version chain has to be latched, which causes tail latencies in the GC process for traversal-based GC mechanisms. Essentially, with an update-intensive workload, such as in figure 11(a), creating a large number of versions, OneShotGC amortizes the bookkeeping cost while having zero traversals during the actual GC process.

Furthermore, acquiring the global minimum transaction timestamp for the GC pre-condition, OneShotGC pays once when the partition has zero active transactions, while for SteamGC without EPO enabled, on every transaction end, each worker has to acquire the global minimum and then perform GC for obsolete versions of committed transactions, costing housekeeping on every transaction end. The main advantage of OneShotGC over traversal-based GC approaches is that OneShotGC does not require access to version chains during GC. Even in interval-based GC approaches, if the version is not pruned during the consolidation, it must access the versions to unlink and then reclaim the memory either at the transaction level or version level.

7.3 Transaction Size

Transaction size, that is, the number of operations per transaction, plays a vital role in understanding the benefits of OneShotGC. Compared to traversal-based GC approaches, OneShotGC eliminates individual version traversal and reclamation, and garbage collects them as a whole partition, in

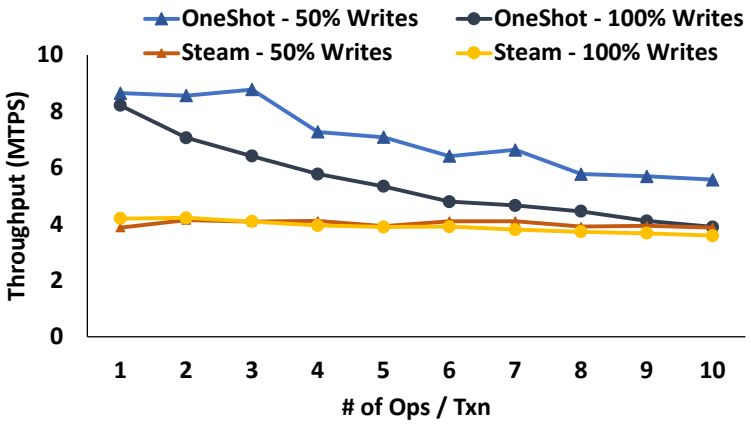


Fig. 12. Effect of transaction size & GC approach in transaction processing

one shot. For large transaction sizes, the majority of time is spent in concurrency control and transaction processing itself; however, for smaller transactions, for example, single attribute update, the GC mechanism may impose housekeeping overheads.

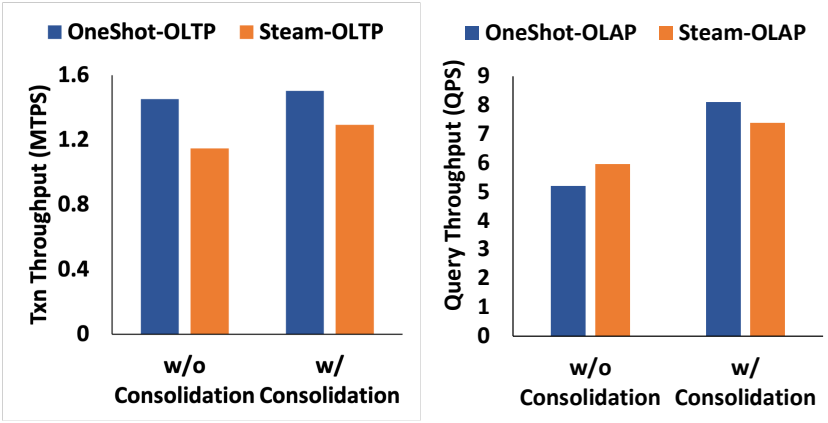
In active GC approaches, which interleave transaction processing with GC, smaller transactions may cost more than the transaction processing itself. On the other hand, for passive GC approaches, bigger transaction costs more, as it accumulates more versions, and that need to be traversed and collected. In contrast to traversal-based GC approaches, OneShotGC strikes the balance between the work done in the GC cycle, and additional cost per transaction. In OneShotGC, every transaction, on begin has to register itself with the assigned delta partition, and de-registers itself at the end of the transaction. This bookkeeping process of OneShotGC causes atomic addition and subtraction on delta partitions, shared by the transactions in the same partition. However, in contrast to the batch transaction model, OneShotGC does not enforce batch boundaries, and thus, transactions of different latencies can execute concurrently without blocking each other.

Figure 12 analyzes the effect of transaction size on GC approach. We use YCSB workload, having a single integer (8-byte) column, with a varying number of operations with 28 transactional threads, and compare OneShotGC with Steam GC, which is an active GC, traversal-based approach. We observe that Steam has a constant overhead of acquiring the global minimum active transaction after each transaction, which overshadows the cost of the transaction if the number of operations is small. In the case of OneShotGC, the design does not impose any overhead per transaction except for registration and de-registration from delta partitions. This allows OneShotGC to make performance predictable with respect to work requested within each transaction.

Summary. OneShotGC amortizes the cost of per-transaction bookkeeping and unlike active GC approaches, does pose additional overhead for smaller transactions, and for larger transactions, avoids unnecessary traversals.

7.4 Chain Consolidation in Mixed Workloads

In this section, we experimentally analyze the benefit and overheads of using partition consolidation. specifically, the impact of consolidation on transaction processing, and the benefit under simulated mixed workloads which delays the regular GC process.



(a) Transactional throughout

(b) Analytical throughput

Fig. 13. Performance under mixed workload with consolidation

To simulate a mixed workload, we employ CH-BenCHmark [4] style workload, where we run TPC-C as the transactional workload on 13 worker threads while reserving 1 worker which simulates analytical query, scanning, and aggregating a full column. Having a long-running reader in the system has two impacts: 1) Blocks regular GC process and causes long-tail version chains as being the oldest transaction in the system. 2) Performance penalty for both, R/W transaction, and long-running reader, as the reader latches the version chain for the duration it reads the appropriate record version, which in this case without consolidation, will be after traversing several versions given the concurrent transactions also updating the same tuples.

Figure 13 shows the evaluation results with and without consolidation for both, OneShotGC and Steam. In OneShotGC, we employ partition consolidation, described in section 5, while Steam employs EPO (eager pruning of obsolete versions) [2].

OneShotGC transactional throughput does not drop in the given workload, as after the first partition crossing pointer, a record is unlatched and is available for any concurrent transaction to update, while in Steam, the chain is latched until the reader finds the appropriate readable version. OneShotGC is slower than Steam in the case where there is no consolidation as the version traversal requires checking tag validity, which causes another operation in the traversal process. The analytical throughput of both approaches is comparable, given the chain traversal is reduced with the help of consolidation.

Summary. Partition consolidation for OneShotGC relies on transaction spread over delta partitions, and uses the same principle as vanilla OneShotGC, that is, GC in one shot, but also allows in-middle partition GC, which consolidates version chains across delta partitions. Partition consolidation has a benefit over traversal-based consolidation, such as Steam's EPO or HANA's interval GC, that it does not require traversal of version chains that were not pruned during consolidation, while also saving additional consolidation work per pruned version chain. Furthermore, for the case where version chains grow abruptly large within the partition, OneShotGC does support Steam's EPO, that is, on version creation, unlink and prune obsolete versions from the version chains.

7.5 Tuning OneShotGC

In this section, we analyze the effect of different configurations of OneShotGC by tuning the number of delta partitions, the size of each partition, and transaction assignment. We execute the

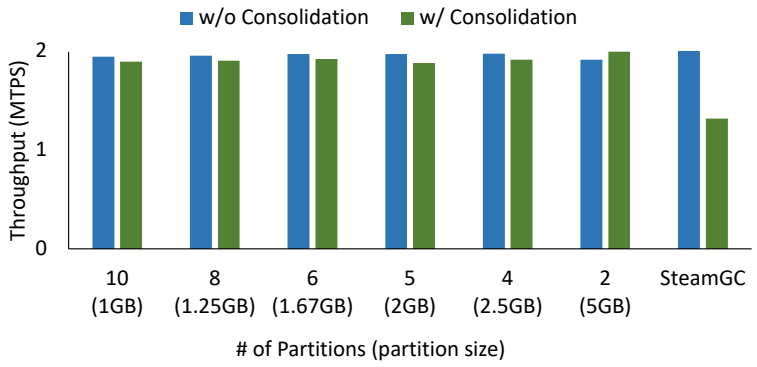


Fig. 14. Effect of scaling number of partitions

TPC-C workload with 14 worker threads, that is, a single NUMA-socket to isolate any effects from cross-socket accesses.

Figure 14 shows the experiment where we scale the number of partitions while keeping the total size of delta storage constant, that is, 10GB, and scale the number of partitions from 2 to 8, dividing the total size accordingly. For the transaction-assignment policy, we assign 16 transactions to a partition, in sequence, and then, we compare partition-level consolidation with version-level consolidation, that is, EPO. Through this experiment, we observe that EPO suffers from performance penalty given the traversal and consolidation process on each version creation request. EPO benefits in keeping the length of version chain size as the number of active transactions in the system but at the cost of the extra traversal and amount of work, interspersed across transactions.

In the case of scaling the number of partitions in version storage, we see almost no effect when there is no consolidation, while minimal variation in the case of consolidating is enabled. This variation is caused by the extra consolidation step in the OneShotGC algorithm, which is triggered whenever the number of transactions registered to a partition becomes zero.

Summary. Transactional workload consists of operations of short-lived similar operations, and hence, allow OneShotGC to exploit the temporality in the garbage collection phase. Moreover, we see that, with only two partitions, the storage size to store versions of a batch of transactions is enough to maintain the maximum throughput of the system. However, in the case of mixed workload, where a single reader can block the GC process, more partitions are desired to increase the spread and allow in-middle partition consolidation.

8 RELATED WORK

Disk-based MVCC. Disk-based MVCC employs hierarchical storage, similar to main data storage with a buffer pool, having recent or hot versions in memory while older versions are flushed to disk. In disk-based MVCC, whenever a version is flushed to a layer below, from in-row to intermediate buffers, then to persistent storage, it provides the system yet another opportunity to prune and reorganize version storage on the basis of currently active transactions. vDriver [8] builds on the principle of Single In-row Remaining Off-row (SIRO) versioning, keeping the first version in-row within the data page while flushing older versions through the hierarchy of caches and clusters based on version's age and access pattern. In-row versions are an attractive optimization for disk-based DBMS for providing one-version locality for the reader as I/O becomes the bottleneck. In-row versions provide locality for the next version in disk-based DBMS as I/O is the main bottleneck,

however, with in-memory DBMS, in-row versioning would cost constant storage overhead which overshadows the gains of having in-row versions compared to disk-based versioning. OneShotGC focuses on the design of delta storage for off-row versions, eliminating random-access traversal and scans of versions during the GC process itself. Any optimizations for in-row versions, as well as, flushing off-row versions to disk can be clubbed with OneShotGC to get the best of both worlds.

Generic Systems GC. Programming language and system runtimes (e.g. Java, .NET Framework) provide managed memory management [3, 7]. Under the hood, the memory manager tracks the liveness of the objects by tracking the reachability of all dynamically allocated objects, and once the object becomes unreachable, the system cleans and reclaims all the memory of such dead objects [15, 23, 24]. Generational GC [21] is the most common approach to classify memory regions based on the age of allocated objects, and then optimizes the GC algorithm [24] accordingly which includes periodicity, concurrent or stop-the-world, etc. Examples of generational GC include Java's G1GC and CMS garbage collectors.

Generic GC approaches cannot assume the lifecycle of allocated objects, hence, requires continuously tracking reachability of each object. Further, may also trigger copying and moving objects to optimize based on the object's age and reduce memory fragmentation. On the contrary, in MVCC-based DBMS, the lifecycle of objects in the transient storage follows the transactional semantics, that is, the oldest active transaction defines the temporary object's reachability. Moreover, in delta storage-based MVCC, DBMS pre-segregates persistent and transient storage, essentially explicitly classifying the expected object generations, and hence does not require object copying and moving across regions.

9 CONCLUSION

We design a version storage design that partitions the versions based on their temporal property, that is, timestamps, and allow transparent cross-partition version traversal through tagged pointers. Based on temporality-aware storage design, we devise a novel OneShot garbage collection algorithm that exploits the temporality of versioned storage in MVCC-based DBMS and completely eliminates version traversal during garbage collection by reclaiming obsolete versions in single pointer-swap, that is, in one shot. Further, we enable version consolidation at the partition level, eliminating version traversal and validity checks per version, and consolidate entire partitions as a whole, reducing version chain length, for efficient traversals under mixed workloads. Finally, we experimentally analyze our OneShotGC, showing the scalability and feasibility of the design under different workload conditions, and provide insights on the cost and benefits of using OneShotGC over traversal-based versioned storage and maintenance design and algorithms.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, the shepherd, and all the members of DIAS (EPFL) Laboratory for their valuable feedback. This work was partially funded by the FNS project "Efficient Real-time Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

REFERENCES

[1] [n. d.]. Peloton. https://pelotondb.io/.

[2] 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. https://doi.org/10.14778/3364324.3364328

[3] Rodrigo Bruno and Paulo Ferreira. 2018. A study on garbage collection algorithms for big data environments. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–35.

[4] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on*

Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011, Goetz Graefe and Kenneth Salem (Eds.). ACM, 8. https://doi.org/10.1145/1988842.1988850

[5] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[6] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[7] Richard Jones and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., USA.

[8] Jong-Bin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived Transactions Made Less Harmful. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 495–510. https://doi.org/10.1145/3318464.3389714

[9] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 938–950. https://doi.org/10.1145/3448016.3452783

[10] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1675–1687. https://doi.org/10.1145/2882903.2882905

[11] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. 2017. Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 49–60. https://doi.org/10.5441/002/edbt.2017.06

[12] 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. https://doi.org/10.14778/2095686.2095689

[13] 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 *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1307–1318. https://doi.org/10.1145/2882903.2903734

[14] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf

[15] Yossi Levanoni and Erez Petrank. 2006. An on-the-fly reference-counting garbage collector for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (2006), 1–69.

[16] Liang Li, Gang Wu, Guoren Wang, and Ye Yuan. 2019. Accelerating Hybrid Transactional/Analytical Processing Using Consistent Dual-Snapshot. In *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Guoliang Li, Jun Yang, João Gama, Juggapong Natwichai, and Yongxin Tong (Eds.), Vol. 11446. Springer, 52–69. https://doi.org/10.1007/978-3-030-18576-3_4

[17] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel (Eds.). ACM, 27:1–27:14. https://doi.org/10.1145/2592798.2592820

[18] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[19] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 International Conference on Management of Data,*

SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2043–2054. https://doi.org/10.1145/3318464.3389783

[20] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[21] David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices* 19, 5 (1984), 157–167.

[22] 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. https://doi.org/10.14778/3067421.3067427

[23] Hirotaka Yamamoto, Kenjiro Taura, and Akinori Yonezawa. 1998. Comparing reference counting and global mark-and-sweep on parallel computers. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, 205–218.

[24] Benjamin Zorn. 1990. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. 87–98.

Received April 2022; revised July 2022; accepted August 2022