Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA

Juchang Lee [#]
juc.lee@sap.com

Hyungyu Shin [‡]
hgshin@dblab.postech.ac.kr

Chang Gyoo Park [#]
chang.gyoo.park@sap.com

Seongyun Ko [‡]
syko@dblab.postech.ac.kr

Jaeyun Noh [#]
jaeyun.noh@sap.com

Yongjae Chuh [#]
yongjae.chuh@sap.com

Wolfgang Stephan [†]
wolfgang.stephan@sap.com

Wook-Shin Han [‡][*]
wshan@dblab.postech.ac.kr

[#]SAP Labs Korea, Korea
[‡]Pohang University of Science and Technology (POSTECH), Korea
[†]SAP SE, Germany

ABSTRACT

While multi-version concurrency control (MVCC) supports fast and robust performance in in-memory, relational databases, it has the potential problem of a growing number of versions over time due to obsolete versions. Although a few TB of main memory is available for enterprise machines, the memory resource should be used carefully for economic and practical reasons. Thus, in order to maintain the necessary number of versions in MVCC, versions which will no longer be used need to be deleted. This process is called garbage collection. MVCC uses the concept of *visibility* to define garbage. A set of versions for each record is first identified as candidate if their version timestamps are lower than the minimum value of snapshot timestamps of active snapshots in the system. All such candidates, except the one which has the maximum version timestamp, are safely reclaimed as garbage versions. In mixed OLTP and OLAP workloads, the typical garbage collector may not effectively reclaim record versions. In these workloads, OLTP applications generate a high volume of new versions, while long-lived queries or transactions in OLAP applications often block garbage collection, since we need to compare the version timestamp of each record version with the snapshot timestamp of the oldest, long-lived snapshot. Thus, these workloads typically cause the in-memory version space to grow. Additionally, the increasing version chains of records over time may also increase the traversal cost for them. In this paper, we present an efficient and effective garbage collector called HybridGC in SAP HANA. HybridGC integrates three novel concepts of garbage collection: timestamp-based group garbage collection, table garbage collection, and interval garbage collection. Through experiments using mixed OLTP and OLAP workloads, we show that HybridGC effectively and efficiently collects garbage versions with negligible overhead.

Keywords

Garbage collection; multi-version concurrency control; SAP HANA

1. INTRODUCTION

Commercial database management systems (DBMSs) including SAP HANA employ multi-version concurrency control (MVCC) due to robust and better performance for various workloads [10]. In MVCC, updates (including deletes) by a transaction to a record generate new versions rather than updating the existing record in place, and thus, a series of versions are maintained for each record.

SAP HANA supports snapshot isolation [3], which is a popular MVCC protocol where a transaction can read a snapshot of committed versions, i.e., a snapshot of the versions that were created by committed transactions. There are two variants of snapshot isolation supported in SAP HANA [19]: statement-level snapshot isolation (Stmt-SI) and transaction-level snapshot isolation (Trans-SI). In Stmt-SI, which is the default isolation level of SAP HANA and widely used by the vast majority of enterprise applications of SAP, all reads logically occur at the beginning of the statement. In Trans-SI, all reads logically occur at the beginning of the transaction. In Stmt-SI, each statement has its own snapshot associated with a new snapshot timestamp, while, in Trans-SI, each transaction has its own snapshot with a new snapshot timestamp.

While MVCC supports fast and robust performance, it has the potential problem of a growing number of versions over time due to obsolete versions. Although a few TB of main memory is available for enterprise machines, the memory resource should be used carefully for economic and practical reasons [2]. Thus, in order to maintain the necessary number of versions in MVCC, versions which will no longer be used need to be deleted. This process is called *garbage collection*.

[*]corresponding author

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2903734

MVCC uses the concept of *visibility* to define garbage. Specifically, in typical commercial DBMSs supporting MVCC, a set of versions for each record is first identified as a candidate if their version timestamps are lower than the minimum value (a.k.a. the global minimum timestamp) of snapshot timestamps of active snapshots in the system. All such candidates, except the one which has the maximum version timestamp, are safely reclaimed as garbage versions. One can guarantee that such record versions will no longer be visible to any active snapshots. Figure 1 shows how the typical garbage collector identifies garbage versions. First, the global minimum timestamp is set to 3. Thus, the conventional garbage collector identifies v_{11} *only* as garbage, since its version timestamp (=1) is lower than 3, and there exists v_{12} whose version timestamp 2 is also lower than 3. However, notice that both v_{13} and v_{14} are invisible to any active transaction, and thus, they should be subject to being "garbage collected."

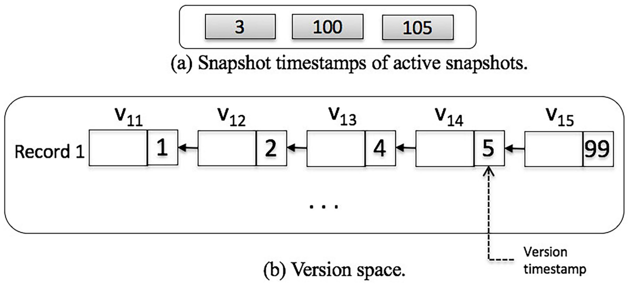

Figure 1: An example of record versions.

In mixed OLTP and OLAP workloads, the typical garbage collector may not effectively reclaim record versions. In these workloads, OLTP applications generate a high volume of new versions, while long-lived queries (under Stmt-SI) or transactions (under Trans-SI) in OLAP applications often block garbage collection, since we need to compare the version timestamp of each record version with the snapshot timestamp of the oldest, long-lived snapshot. Thus, these workloads typically cause the in-memory version space to grow. Additionally, the increasing version chains of records over time may also increase the traversal cost for them. We also observe other types of garbage collection blockers in real customer systems. Long-duration cursors or Trans-SI transactions due to either application logic or developers' mistakes can easily block garbage collection. For example, we collected statistics on cursor durations from a real ERP system. Out of 408,664 distinct queries executed in the system, the life times of six cursors were more than one hour!

Figure 2 shows the impact of the long-lived snapshot when there exists no optimized garbage collector. This real screen shot is taken from the HANA system load view, which visualizes the status of key performance indicators in the system over time. Especially, the light blue color, marked as "Active Commit ID Range", denote the difference between the last CID value and the minimum global snapshot timestamp value. If the value increases, this means that there is at least one long-live snapshot in the system. The number of record versions, marked as "Active Versions" in blue color, keeps increasing over time. As a result, the system's memory consumption, marked as "Used Memory" with green color, keeps growing.

Conventional workarounds for this version space overflow

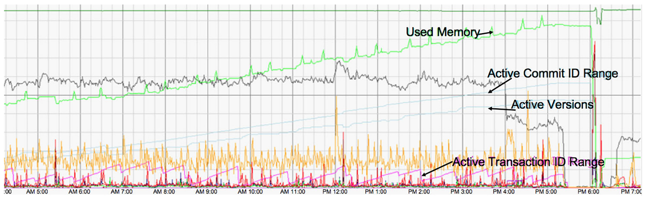

Figure 2: A real example of version space overflow problem.

problem are as follows. 1) The system flushes old versions out to disk. 2) The system closes problematic cursors or Trans-SI transactions by force and returns errors to clients. This is implemented in SAP HANA, especially to handle application developers' mistakes. 3) The system implicitly closes cursors earlier than the explicit cursor close request; as soon as the query results are materialized or the system can guarantee that there is no more request to access the version space, the system implicitly closes cursors. This workaround is also supported in SAP HANA for a limited set of cases.

In order to collect more garbage versions, unlike the traditional garbage collector which uses the global minimum timestamp, we propose an *interval garbage collector* that uses *visible intervals* among consecutive timestamps of each record version. Here, the visible interval $[s, e)$ for a record v represents a set of snapshot timestamps to which v can be visible. The traditional garbage collector uses the global minimum timestamp and is thus unable to collect record versions which are invisible to any active or future snapshots. For example, v_{13}, and v_{14} are invisible to any active transactions. Visible intervals for the record 1 of Figure 1 are {[1, 2), [2, 4), [4, 5), [5, 99), [99, ∞)}. Consider the interval for v_{13}, [4, 5). No active snapshot timestamp is included in this interval. Thus, v_{13} is invisible to any active snapshot and can be reclaimed as garbage. Similarly, v_{14} whose visible interval is [5, 99) is also reclaimed as garbage if we use this interval-based decision.

Garbage collection may incur non-negligible overhead for normal transaction processing. In order to efficiently collect garbage, we propose the concept of the *group garbage version* collector. Typically, a set of versions generated by the same transaction can form a group, and thus, the garbage collector checks whether this group can be reclaimed as a whole. The group garbage collector can also be implemented using either the timestamp-based decision or the interval-based decision.

Figure 3 shows the taxonomy of 4-quadrants of garbage collectors containing two dimensions. Along the dimension of garbage granularity, a garbage version can be either a single record version or a group version. Along the dimension of comparison unit, there exist timestamp-based and interval garbage collectors. Thus, we can have four types of garbage collectors (ST, SI, GT, and GI). Note that existing garbage collectors belong to ST.

In addition to granularity and comparison unit, we can exploit semantic optimization for efficiently collecting garbage versions. Under Stmt-SI or some special cases of Trans-SI (e.g. pre-compiled stored procedures), one can reclaim more garbage versions by identifying the tables which are not relevant to the currently long-running snapshots. In this paper,

		Comparison unit	
		Timestamp	Interval
Granularity	Single version	ST	SI
	Group version	GT	GI

Figure 3: The taxonomy of garbage collectors.

this type of garbage collector is called the *table garbage collector* (TG).

In order to execute effective and efficient garbage collection, we propose the novel concept of *hybrid garbage collector* which executes GT, TG, and SI in this order. GT is a light-weight garbage collector since it examines group garbage versions only. TG is also effective under Stmt-SI and relatively lighter than SI. Although the SI can reclaim more garbage versions, it can be heavier than the others. Thus, we propose an efficient algorithm for SI to identify garbage versions using visible intervals.

In summary, this paper makes the following key contributions.

- We propose the novel concept of *interval garbage collection*. For this, we formally model the garbage collection problem as the consecutive interval intersection problem. Then, we provide an efficient merge-based solution to this problem.
- We propose the novel concept of *group garbage collection*. This granular garbage collection enables us to efficiently determine a set of record versions as garbage. Furthermore, this concept can be applied to typical, minimum snapshot timestamp-based garbage collection as well as interval garbage collection.
- We propose the novel concept of *table garbage collection*. This exploits the semantic information that versions in tables which are irrelevant to current snapshots can be reclaimed at any time.
- We present a hybrid garbage collector which combines three different flavors of garbage collectors. Thus, we can effectively and efficiently collect garbage versions with negligible performance overhead.
- We implement the hybrid garbage collector in SAP HANA and perform extensive experiments using a modified TPC-C benchmark which includes long-running queries. Experimental results confirm that the hybrid garbage collector effectively collects garbage versions and considerably reduces the latency of long-running OLAP queries.

The rest of this paper is organized as follows. Section 2 reviews some background information in SAP HANA. In Section 3, we propose the concept of the interval garbage collector. Section 4 presents the concepts of the group garbage version, table garbage collection, and the hybrid garbage collection for SAP HANA. In Section 5, we present the results of performance evaluations, and Section 6 reviews related work. Section 7 concludes the paper.

2. PRELIMINARIES

In this section, we first explain the architecture of SAP HANA. We then explain how SAP HANA manages the version space.

2.1 SAP HANA In-memory Database

SAP HANA is an ACID-compliant and in-memory relational DBMS, designed to efficiently handle both of OLTP and OLAP workloads together in a single system [16,18,19]. SAP HANA has both an in-memory column store and an in-memory row store for high-performance OLAP and high-performance OLTP, respectively.

In order to seamlessly integrate the column store and row store from transaction and query processing perspectives, the unified transaction manager and the unified query processor are built on top of the two different stores. For example, the unified transaction manager provides durability based on logging and checkpointing to a common persistency, and it also provides snapshot isolation based on the common commit timestamp.

For supporting snapshot isolation, the transactions across the row store and the column store are centrally controlled by the unified transaction manager although each store has its own version space layout. While the garbage collection scheme proposed in this paper can also be applied to the column store, the rest of this paper is described based on its implementation at the row store for convenience of the description.

2.2 Version Management in HANA RowStore

The in-memory area in the SAP HANA row store consists of the table space and the version space. Since the row store implements the snapshot isolation based on MVCC, the newly added version is appended to the version space, instead of performing the in-place update directly to the table space. Later on, by garbage collection, the added data is moved to the table space once it is certain that there is no potential reader to the original data maintained in the table space. In this way, the table space maintains the oldest versions of the existing records while the version space maintains newer record versions until they are reclaimed or their record images are copied to the table space by a garbage collector.

In particular, on every INSERT/UPDATE/DELETE operation, a *record version* (or a *version entry*—we use the terms interchangeably) is created at the version space. A single record version consists of a version header and its payload. The version header stores the creator's operation type, the changed record's identifier (RID), the table ID which the record belongs to, and a few pointers to maintain version chains. The payload stores the new record image for the UPDATE operation. For the INSERT operation, the new record image is directly inserted into the table space since there is no older image for the particular record.

In the version space, the record versions having the same RID are linked with each other in the latest-first order. Differently from organizing the version entries in the oldest-first order, the *latest-first* ordering in the version chain can reduce the cost of traversing version chains for future queries and transactions. This is more likely to access more recent versions by the nature of snapshot isolation. The pointer to the latest record version of each version chain is maintained by a central hash table, called *RID hash table*, for

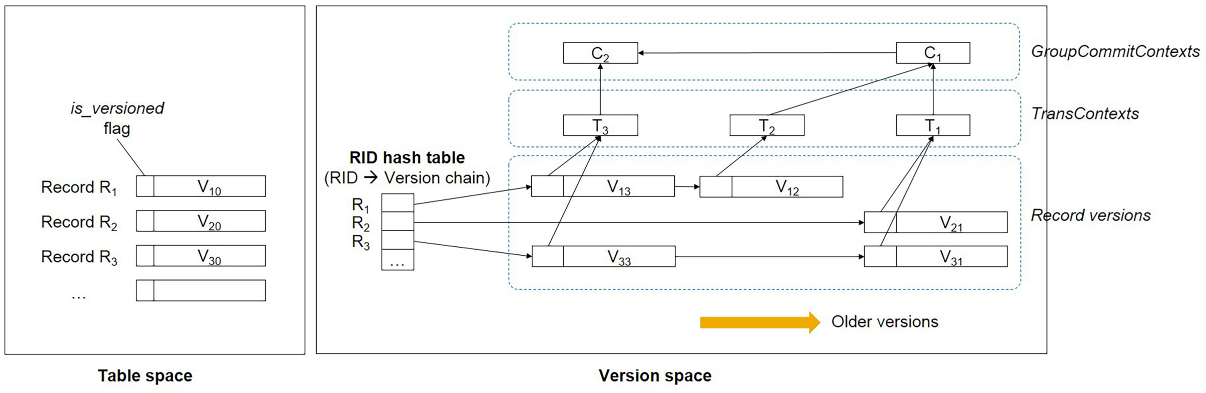

Figure 4: Version management in the HANA row store.

fast RID-to-version-chain lookup. In order to reduce the cost of checking whether a particular record has its own version chain, each record in the table space stores the so-called *is_versioned* flag. This flag is set only when the record in the table space has additional record versions in the version space. If this flag is not set, looking up the hash table can be skipped.

The RID hash table is implemented as a chained hash structure. Basically, it is an array of hash buckets, each of which stores the pointer to the corresponding version chain. However, if multiple version chains are associated to a single hash bucket, the hash bucket creates additional linked list to maintain the pointers to the multiple version chains. As the result, if a single hash bucket contains 10 version chains, then the query accessing the hash bucket may encounter 5 additional pointer traversals in average and 10 additional pointer traversals in the worst case just to find the corresponding version chain. Since every pointer traversal in this additional chain may lead to hardware-level cache misses, this cost can significantly affect the overall performance, especially for short-running queries and transactions where the version space access cost takes a non-trivial portion of its execution time. Remark that the row store is currently implemented using the RID hash table, considering that an efficient garbage collector can reduce the probability of the hash collisions. The detailed experimental analysis on the performance impact of such hash collisions is given in Section 5.

The record versions created by the same transaction are associated as a group by pointing to the same data object, called *TransContext*. When a transaction issues a write operation for the first time, it creates a TransContext object. All the next record versions created by the transaction point to the same TransContext object. The TransContext object points to NULL until the corresponding transaction starts to commit. On transaction commit, once the set of write transactions to be committed together is decided by the *group commit* logic, another object called *GroupCommitContext* is created, and thus the TransContext objects for all concurrent transactions committing together point to the same GroupCommitContext object. After that, if the commit ID (CID) is decided for the group commit operation, the CID is written to the GroupCommitContext object, and the CID becomes immediately visible to all related TransContext objects and version entries which share the same CID value. This atomic, indirect CID assignment scheme is more efficient than the approach of directly copying the CID values to the record versions. The indirect CID assignment is done in an atomic way without relying on any latch or lock, and thus is efficient.

The expected penalty in the atomic indirect CID assignment is that the accessing the CID value of a record version may have to follow an additional pointer, but this cost can be minimized by asynchronously propagating the CID value from the GroupCommitContext object to the individual record versions. For efficient backward CID propagation, backward links from the GroupCommitContext object to individual record version are also maintained.

Figure 4 illustrates the version space management scheme in the row store. We assume the records R_1, R_2, and R_3 are already in a user table. The transaction T_1 generates the record versions V_{21} and V_{31} and commits with creating TransContext T_1. The transaction T_2 generates the record versions V_{12} and commits with creating TransContext T_2. Since T_1 and T_2 are committed together by the same group commit operation, they share the same GroupCommitContext C_1. Then, the transaction T_3 generates the record version V_{13} and V_{33} and commits with creating TransContext T_3 and GroupCommitContext C_2.

3. VARIANTS OF GARBAGE COLLECTOR

In the paper, we classify garbage collectors into four variants according to the comparison unit and the granularity.

3.1 Timestamp vs Interval Garbage Collector

Conventional, timestamp-based garbage collectors identify a version record by comparing its timestamp with active snapshot timestamps. Specifically, a set of versions for each record is first identified as a candidate if their version timestamps are lower than the global minimum timestamp in the system. All such candidates, except the one which has the maximum version timestamp, are safely reclaimed as garbage versions. Such record versions will no longer be visible to any active snapshots.

Before we explain interval garbage collectors, we formally model the interval-based garbage collection as the consecutive interval intersection problem. First, we define some terminology and introduce the consecutive intersection problem. We then propose an efficient merge-based solution to this problem.

Consider an integer t and an ordered sequence S of integers. Without loss of generality, we assume that S always contains a number which is larger than or equal to any t. Then, the *least greater number* (LGN) for t with respect to S is defined as the smallest number in S such that the number is greater than or equal to t. We denote the least greater number by LGN(t,S). Suppose that t=10, and S = [1, 4, 6, 8, 12, 14]. Then, LGN(t,S) = $min\{12, 14\}$ = 12. If t =15, LGN(t,S) = ∞.

Definition 1 (Consecutive interval intersection). *Given two ordered sequences of integers, S and T, find the subset $T_\cap$ of T satisfying the following condition.*

$$T_\cap = \{t | t \in T, \mathrm{LGN}(t+1, T) \leq \mathrm{LGN}(t, S)\} \square \quad (1)$$

Consider S = [90, 92, 95, 96, 99] and T = [91, 93, 94, 95, 98]. We can compute LGN($t+1$, T) and LGN(t,S) for each t. Finally, we can compute $T_\cap = \{93, 94\}$.

Suppose that S is an ordered sequence of snapshot timestamps, and T is an ordered sequence of record versions for a record. Then, the elements in $T_\cap$ are identified as garbage. Here, $[t, \mathrm{LGN}(t+1, T))$ is called *visible interval* for t.

Now, we explain how to compute $T_\cap$. The naive way to compute $T_\cap$ uses nested loops. That is, for each record version t, we perform a set intersection operation for every snapshot timestamp in S. Then, the time complexity is $O(|T| \times |S|)$.

In order to minimize the garbage collection overhead, we now propose a merge-based solution. Algorithm 1 shows a merge-based garbage collector. It computes $T_\cap$ in Eq. (1) in $O(|T| + |S|)$. We denote the i-th element of T by $T[i]$. In order to merge two ordered sequences, we maintain two index variables, i and j. For each element $T[i]$, we move j until $S[j] \geq T[i]$ (Lines 3 $\sim$ 4). Then, $S[j]$ should be LGN($T[i]$, S). If $S[j] \geq T[i+1]$, then $T[i]$ is identified as garbage. Otherwise, we skip $T[i]$ by incrementing i since it is not garbage.

Algorithm 1 MergeBasedGC(S, T)

```
Input: Two ordered sequence of integers $S$, $T$.
Output: $T_\cap$.
 1: $i \leftarrow 0$, $j \leftarrow 0$
 2: while $i < |T|$-1 do
 3:     if $S[j] < T[i]$ then
 4:         $j \leftarrow j + 1$
 5:     else if $T[i+1] \leq S[j]$ then
            /*$T[i+1]$ represents LGN($T[i] + 1, T$).*/
 6:         $T_\cap \leftarrow T_\cap \cup T[i]$
 7:         $i \leftarrow i + 1$
 8:     else
 9:         $i \leftarrow i + 1$
10:     end if
11: end while
12: return $T_\cap$
```

3.2 Single vs Group Garbage Collector

The granularity for garbage collectors can be either a single record version or a group of record versions. All record versions generated by transactions belonging to the same GroupCommitContext have the same version timestamp. Thus, if we group record versions by their timestamp, we can efficiently check whether a (version) group is reclaimed as garbage. That is, we can directly apply the timestamp-based garbage collector to both the single version and the version group. Such garbage collectors are classified as ST and GT, respectively.

Now, we explain how we can apply the interval garbage collector to the version group. For this purpose, we propose the novel concept of *immediate successor subgroup*.

The immediate successor subgroup in a group G_i consists of record versions in G_i which have the immediate successors in the next group G_{i+1}. Then, we can apply the interval garbage collector to an ordered sequence of immediate successor subgroups. In this scheme, we need an efficient, systematic mechanism to move a record version in G_i to form its immediate successor subgroup, which is beyond of the scope of our paper and would be an interesting future topic of research.

Figure 5 shows an ordered sequence of k groups. Each group G_i has one immediate successor subgroup denoted by sg_i. Note that the version v_1 belongs to sg_1 since its immediate successor is in G_2, while the version v_2 does not. However, if we generate the immediate successor of v_2 and insert the successor into G_2, then v_2 belongs to sg_1. Note that sg_k does not contain any record version since G_k is the last group in the ordered sequence.

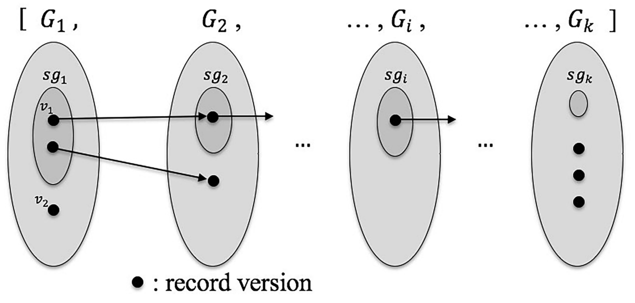

Figure 5: An ordered sequence of k groups containing immediate successor subgroup.

Now, we can apply the interval garbage collector to both the single version and the version group. Such garbage collectors are classified as SI and GI, respectively. In the next section, we explain how these garbage collectors are implemented in SAP HANA.

4. IMPLEMENTATION IN SAP HANA

This section describes how the proposed interval garbage collector and the group garbage collector are implemented in SAP HANA together with the additional practical optimizations. Specifically, we describe GT, SI, and the table garbage collectors implemented in SAP HANA.

4.1 Global Group Garbage Collector

GT is implemented as the global garbage collector in SAP HANA, considering the following two optimization strategies: (1) efficiently select the minimum global snapshot timestamp value and (2) efficiently detect a group of garbage versions.

As the first optimization strategy, the so-called *global snapshot timestamp tracker* (global STS tracker) is maintained globally in the system. The global STS tracker is an ordered list of reference-counted snapshot timestamp values. When a snapshot starts, it acquires its snapshot timestamp value from the transaction manager according to the definition of snapshot isolation. If the acquired snapshot timestamp value is already in the tracker, we simply increment the reference count of the corresponding snapshot timestamp object by one. Otherwise, a new snapshot timestamp object is inserted into the tracker. Then, if the snapshot associated with a query or a transaction finishes its processing, the reference count of the corresponding snapshot timestamp decrements by one. When the reference count becomes zero, the corresponding snapshot object is deleted from the global STS tracker. Since each active snapshot maintains a pointer to the corresponding snapshot timestamp object in the global STS tracker as shown in Figure 6, we can directly access the snapshot timestamp object without scanning the entire list in the tracker. Note that the snapshot timestamp values in the global STS tracker are maintained in increasing order. When the global garbage collector needs to find the global minimum snapshot timestamp value at a certain time point, accessing the head of the list without scanning the entire tracker is sufficient.

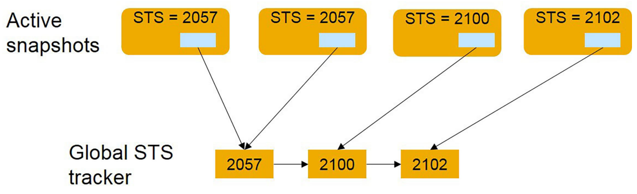

Figure 6: Global Snapshot Timestamp Tracker.

As the second optimization strategy, we exploit the GroupCommitContext and the TransContext objects, described in Section 2. Since (1) record versions created by the same transaction are associated with the same TransContext object, and (2) the transactions which are committed together and share the same CID value are associated as a group with the same GroupCommitContext object, we can efficiently find the group of record versions that share the same CID by just accessing the GroupCommitContext object. The GroupCommitContext objects are also maintained by an ordered list of their CID values as shown in Figure 7. Then, the global group garbage collector can identify a set of garbage versions as a group by just checking the GroupCommitContext object list without traversing the individual record versions. If the global garbage collector encounters a GroupCommitContext object which has a larger CID value than the minimum global snapshot timestamp, then the global garbage collector finishes its iteration of the GroupCommitContext object list. After the group garbage collector finishes the identification of groups of garbage versions, the corresponding record versions, TransContext objects, and GroupCommitContext objects are physically deleted in the background.

4.2 Interval Garbage Collector

For the simplicity of implementation in SAP HANA, the group garbage collector uses only the timestamp-based decision, while the single garbage collector uses only the interval-based decision. The interval garbage collector consists of

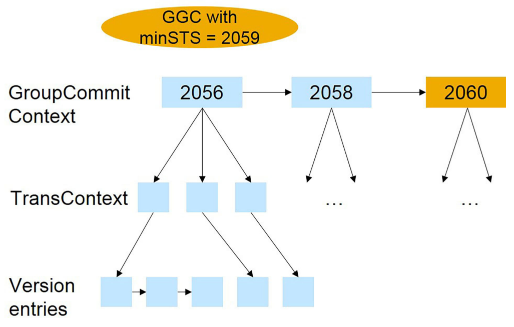

Figure 7: Global Group Garbage Collection.

the following four steps. (1) The full set of active snapshot timestamps is retrieved by scanning the whole snapshot timestamp values maintained by the global STS tracker. Let's denote the ordered set of active snapshot timestamp values by S (following the same notation at Definition 1). (2) At the second step, the interval garbage collector scans the existing GroupCommitContext objects and finds GroupCommitContext objects whose CID values are larger than the minimum value of S and smaller than the maximum value of S. We denote the set of the identified GroupCommitContext objects by P. (3) At the third step, by iterating GroupCommitContext objects in P in the highest-CID-first order, the interval garbage collector accesses the record versions reachable from the GroupCommitContext objects. (4) Finally, for each reachable record version, the interval garbage collector identifies (and reclaim, if exists) garbage versions in its version chain, following Algorithm 1. The CID values in the version chain compose T in Algorithm 1, and the interval garbage collector reclaims the record versions whose CID values exist in $T_{\cap}$.

Instread of accessing version chains starting from the available GroupCommitObject list as described above, it is an alternative implementation option to reach to the version chains starting from the RID hash table, which is more useful when we need to logically partition the version space to execute the interval garbage collector by multiple threads in parallel.

4.3 Table GC: Semantic Optimization

In addition to the interval garbage collector and group garbage collector, we present another optimization dimension for garbage collection by exploiting semantic information of executed snapshots. In this optimized garbage collection, called *table garbage collection* (TG), TG leverages the target tables that will be accessed by a given snapshot a priori. In general, it is not possible to predict the complete set of the tables to be accessed by the snapshot at the time when a snapshot timestamp is assigned to it. However, under Stmt-SI, which is the default isolation mode in SAP HANA and widely used in SAP enterprise applications, the complete set of the accessed tables within that snapshot can be retrieved by just accessing its compiled query plan. Even under Trans-SI, in certain scenarios (e.g. pre-compiled stored procedure execution), it is possible to identify the tables to be accessed within the snapshot a priori. Therefore, it is possible to restrict the negative impact of long-live snapshots only to their relevant tables.

If a table garbage collector is invoked, it performs the following three steps: (1) discovering long-lived snapshots and their scopes, (2) moving snapshot timestamp objects from the global STS tracker to relevant, per-table STS trackers, (3) reclaiming versions by traversing the version space based on the per-table minimum snapshot timestamps. In the first step, it checks whether there is any long-lived (determined based on the pre-defined time threshold) snapshot in the system by accessing the system monitor which keeps track of every active snapshot's status If any long-live snapshot is found, then it checks again whether the complete set of the target tables of the detected long-lived snapshots is already determined. In the second step, the snapshot timestamp of the classified snapshot is copied to one or more relevant *per-table snapshot timestamp trackers* (per-table STS trackers), each of which maintains only the timestamps of the snapshots relevant to the given table. Then, the snapshot timestamp of such snapshot is removed from the global STS tracker. Finally, the table garbage collector decides the minimum snapshot timestamps from the available per-table STS trackers and then, based on the per-table minimum snapshot timestamp, the table garbage collector is able to reclaim versions which could not be reclaimed by the global garbage collector alone. During this visibility check, depending on the table ID stored in the record version, it is determined which minimum snapshot timestamp is compared with the record version's CID.

In the example of Figure 8, suppose that the snapshots S_1 and S_2 are identified as long-lived snapshots. Assume that their scopes are Table 1 and Table 2, respectively. For the record versions which are not relevant to Table 1 and Table 2, 2100 is used as the minimum snapshot timestamp. For the record versions relevant to Table 1 or Table 2, 2057 is used as the minimum snapshot timestamp.

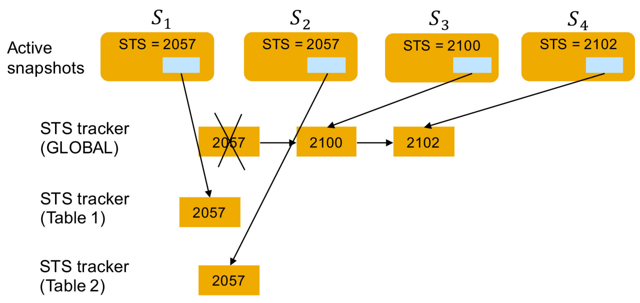

Figure 8: Per-table snapshot timestamp trackers for table garbage collection.

Although it is also possible to apply this semantic optimization to a finer-granular object such as partitions (by additionally leveraging the partition-pruning results of the queries), only the table-level semantic optimization is implemented in SAP HANA for now. In practice, this table garbage collection is used more widely even for Trans-SI, particularly for internal transactions, because there is an API that application programmers can explicitly declare which tables will be accessed by the transactions. If the transaction tries to access a non-declared table object, an error is reported to the transaction. In addition to the internal transaction case, if a Trans-SI transaction is used in a pre-compiled stored procedure, it is also possible to identify the tables to be accessed within the snapshot a priori.

For mixed OLTP and OLAP workload, the table garbage collector is particularly useful in a database system such as SAP HANA which uses separate row and column stores. By the table garbage collection, the long-lived OLAP queries running on column store tables do not affect the garbage collection of the other row store tables which deals with a large number of versions rapidly created by the OLTP workloads.

4.4 HybridGC: Putting It Altogether

Finally, we propose a hybrid garbage collector which combines the global group garbage collector, the table garbage collector, and the interval garbage collector. The three garbage collectors are invoked independently, having its own invocation period. When the table garbage collector or the interval garbage collector is invoked, it internally executes the global group garbage collector first. For the remaining versions after the implicitly invoked group garbage collection, the table garbage collector or the interval garbage collector tries to reclaim them.

Figure 9 shows three regions of the version space by the three different version garbage collectors in HybridGC. The set of versions marked by the region A is reclaimed at once by the global group garbage collector. The region B is inspected by the table garbage collector, while the region C is inspected by the interval garbage collector. While the interval-garbage collector eventually scans all existing version chains which are available when the interval garbage collector starts, the table garbage collector scans only a subset of the GroupCommitContext objects whose commit timestamps are less than the minimum value of the global STS tracker. Note that the global garbage collector and the interval garbage collector need to be slightly changed when they are used together with the table garbage collector because the table garbage collector could move some snapshot timestamp values from the global STS tracker to the per-table STS trackers. That is, the global garbage collector should calculate its global minimum snapshot timestamp by considering not only the global STS tracker but also the existing per-table STS trackers. The interval garbage collector also needs to consider existing per-table STS trackers as well as the global STS tracker. To deal with the situation where there are too many per-table STS trackers, the union of the global STS tracker and the available per-table STS trackers is pre-materialized and maintained separately to fast extract the minimum value for the global garbage collector or the entire snapshot timestamp set for the interval garbage collector.

All the proposed garbage collectors can be easily executed in parallel by multiple threads. For example, by logically partitioning the GroupCommitContext object list or by logically partitioning the version space based on table ID or RID.

5. EXPERIMENTS

We evaluate the performance of three different garbage collectors, which are implemented in SAP HANA row store The garbage collectors considered are as follows: (1) GT, a global group garbage collector which uses the timestamp-based decision; (2) GT+TG, a garbage collector that uses the combination of GT + TG; (3) HG(=HybridGC in Sec-

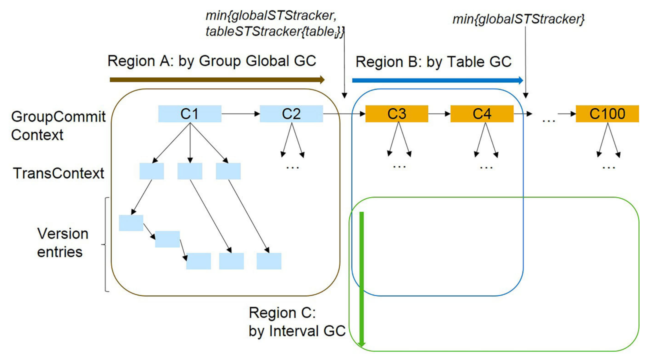

Figure 9: Hybrid garbage collection.

tion 4.4), the hybrid garbage collector which executes the three garbage collectors GT+TG+SI in this order.

Our main objective is to show that the hybrid garbage collector 1) effectively collects garbage and 2) efficiently works with negligible overhead. The detailed goals of the experiment are as follows:

- HG reclaims the version space more effectively than the two alternatives (Section 5.2).
- HG improves the OLTP performance especially for the SAP HANA row store by reducing the number of hash collisions in the version space (Section 5.3).
- HG improves query performance during incremental query processing (Section 5.4).
- HG effectively collects garbage in the presence of Trans-SI transactions, while the other alternatives do not. (Section 5.5).
- The overhead of HG is negligible (Section 5.6).

5.1 Experiment Setup

We used the TPC-C benchmark with 100 warehouses with the following minor modifications: 1) In order to avoid network performance effect, the TPC-C logic was embedded inside the database server using HANA SQLScript, which is the stored procedure supported by SAP HANA. 2) In order to see the behaviors of the garbage collectors while the record versions are generated rapidly, we allocated a dedicated worker thread for each warehouse and let the thread access the home warehouse only. Note that this modification does not simplify the work of garbage collectors because the worker thread information or the database partition information is not exploited by the garbage collectors in this experiment. 3) In order to emulate mixed OLTP and OLAP workloads, we added an emulated OLAP workload by using a long-duration cursor (under Stmt-SI) or long-duration transactions (under Trans-SI).

All TPC-C transactions were executed under the Stmt-SI mode except the experiment in Section 5.5. In all the experiments, the garbage collectors are triggered periodically with 1 second for GT, 3 seconds for TG, and 10 seconds for SI. The impact of the garbage collection period is discussed in Section 5.6. In order to measure how effective garbage collectors are, we use the number of record versions. We use the throughput (=committed statements/sec) for measuring efficiency of each garbage collector.

All the experiments were performed on a single 4-socket machine having 1TB of main memory and 60 physical CPU cores in total (120 logical cores with hyper-threading).

5.2 Impact on Version Space Size

To see the impact of a long-duration cursor to the version space size, we executed a simple scan query on the STOCK table without closing its cursor until the TPC-C benchmark finishes (each worker thread terminates after executing a fixed number of iterations in TPC-C). We measured the number of record versions maintained by each garbage collector.

Figure 10 shows the number of record versions existing in the version space over time. While the number of record versions keeps growing for GT and GT+TG, the number of record versions in HG remained almost constant in spite of the long-duration cursor. This indicates that SI effectively collects garbage versions while the other two do not in the presence of the long-lived query. GT+TG reclaims more record versions than GT alone because the long-duration cursor is defined only for the table STOCK in this experiment.

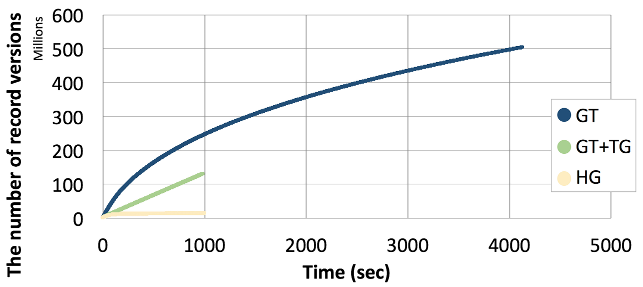

Figure 10: The number of record versions with a long-duration cursor.

Remark that the version growth ratio in GT decreases over time while GT+TG shows almost linear growth in terms of the number of the record versions. This was because the version generation speed from the TPC-C benchmark decreased as the number of the hash collisions at the version space increases. This issue will be discussed in detail in Section 5.3. Similarly, the graphs of HG and GT+TG terminate earlier than GT because the given number of TPC-C iterations at HG or GT+TG finishes earlier than GT.

Figure 11 shows the accumulated number of record versions reclaimed by each of GT, TG and SI when HG is used in the experiment of Figure 10. Due to the long-duration cursor, GT did not reclaim any record version but TG and SI respectively reclaimed 379 million versions and 118 million versions during the 1000 seconds of the TPC-C run time. The number of reclaimed versions by SI in Figure 11 is equivalent to the number of remaining versions at GT+TG in Figure 10 and this indicates that HG could reclaim more versions than GT+TG by the work of SI.

5.3 Impact on OLTP Performance

In order to see the impact of the long-duration cursor to the OLTP performance, we measured the throughput of the

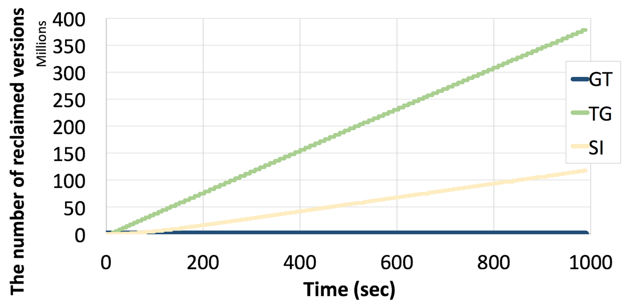

Figure 11: The accumulated number of reclaimed versions per garbage collector under HG.

TPC-C transactions in the same experiment with the one in Section 5.2.

Figure 12 shows the throughput of the TPC-C transactions in terms of the number of executed statements per second. The overall performance using GT alone dropped over time while the long-duration cursor is available for the table STOCK. This is due to the characteristics of the chained hash implementation in SAP HANA row store. As explained in Section 2, if the number of the version chains in the version space increases, the chained hash need to maintain additional hash entries in a separate linked list, which incurs the additional navigation cost to find the corresponding version chain for a given RID.

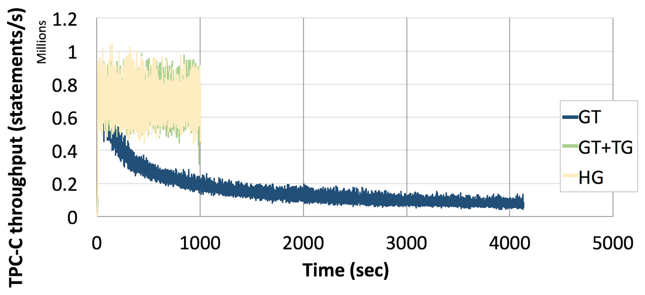

Figure 12: TPC-C throughput with a long-duration cursor.

To confirm the correlation between the TPC-C throughput drop and the hash collision ratio, we additionally measured the hash collision ratio during the same experiment with the one in Figure 13. The hash collision ratio is defined as the number of the version chains assigned to a single hash bucket on average. For example, the hash collision ratio of 10 means that 10 version chains are assigned to a single bucket in the hash table on average. The TPC-C transactions are relatively short, and thus, the increase in the hash collision ratio affected the overall performance significantly. Note that, in GT+TG, the number of the record versions keeps increasing as shown in Figure 10, but it does not lead to the increase of the collision ratio at the version space hash because TPC-C generates only the UPDATE workload to the STOCK table and the UPDATE record version does not create a new version chain.

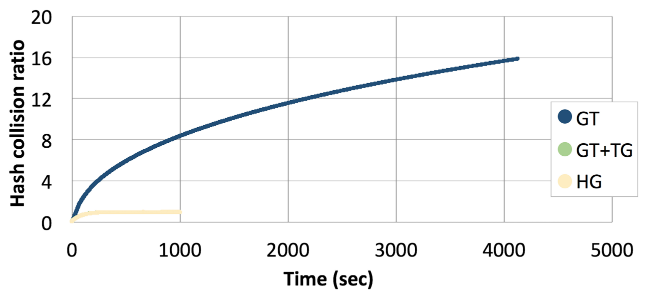

Figure 13: Hash collision ratio in the experiment of Figure 12.

5.4 Impact to Incremental Query Processing Performance

In this section, we check whether the garbage collectors affect the performance of the incremental query processing. In order to emulate incremental query processing, the long-duration cursor incrementally and periodically fetches its query results to the client. While the query is supposed to return 5 million result records in total and eventually, 10 thousands of the result records are returned to the client at a time by the FETCH call of the returned cursor. The next FETCH operation is called after 5 seconds of sleep time, emulating the application-side processing logic for the received chunk of the result records.

Figure 14 shows the latency of the individual FETCH operations of a cursor over time. HG showed almost constant latency even though FETCH operations are repeated within a cursor while the latency increased over time in GT or GT+TG. It is expected that two performance penalties are involved in case of GT and GT+TG: (1) the hash collision in the RID hash table and (2) the cost of traversing the record versions within a single version chain. Unlike OLTP-style workloads which access more recent record versions (the time gap between the snapshot timestamp assignment operation and the version access operation is relatively short), the second cost could become more prominent for long-running, incremental query processing because it should access older record versions as its own snapshot timestamp becomes relatively older.

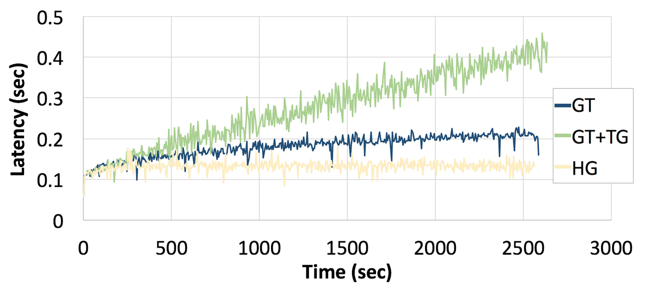

Figure 14: Latency of individual FETCH operations in a cursor.

To verify our analysis, we also measured the number of the record versions traversed by a single FETCH operation. Figure 15 shows as a similar trend as Figure 14. This confirms that the latency of an individual FETCH operation is mostly affected by the number of record versions traversed for the FETCH operation.

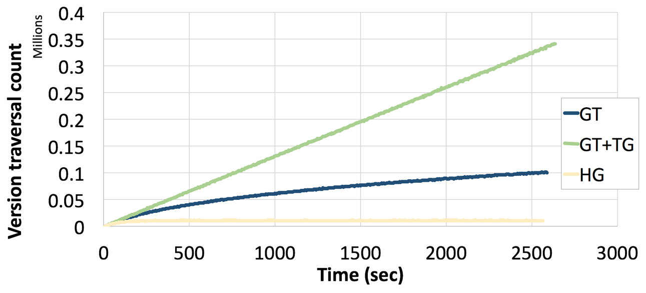

Figure 15: The number of the record versions traversed by individual FETCH operations in a cursor.

By the similar reason, not only for the incremental query processing, we can expect the same performance impact for the queries executed within Trans-SI transactions because the snapshot timestamp is assigned at the time of the transaction start but the queries can be executed later with the assigned snapshot timestamp. This will be further discussed in Section 5.5.

5.5 Impact of Trans-SI Transactions

So far, we have run the experiments only under Stmt-SI. To see the impact of Trans-SI transaction, instead of using the long-duration cursor as in the previous experiments, we ran the following Trans-SI transaction repeatedly: (1) start a Trans-SI transaction, (2) sleep 10 minutes (emulating other query processing or application logic inside the transaction boundary), (3) execute the simple scan-based query for the STOCK table, (4) commit the transaction.

Figure 16 shows the latency of the query executed within the Trans-SI transaction. TG did not show any gain compared with GT since TG was unable to predict the target tables that the Trans-SI transaction would access, as explained in Section 4. On the contrary, SI can effectively collect garbage versions regardless of whether GT is blocked by Stmt-SI cursors or Trans-SI transactions. As a result, HG performs the best, showing the shortest query latency among the three.

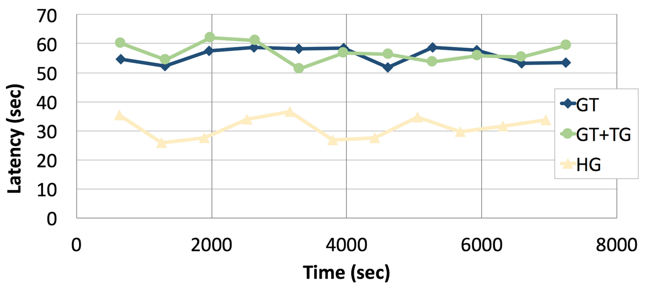

Figure 16: The latency of the queries executed in the Trans-SI transactions.

Figure 17 shows the number of record versions in the version space. There are three saw-like plots in the figure. This experiment confirms that HG maintains the smallest number of record versions in the version space. Note that the number of the record versions drops sharply almost periodically (approximately, every 10 minutes(=every 600 seconds)) since the global snapshot timestamp is reclaimed from the global snapshot timestamp tracker at the transaction end time under Trans-SI.

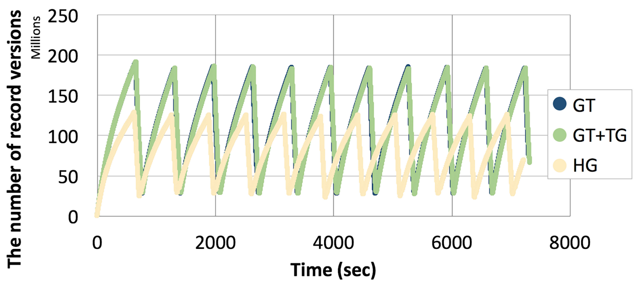

Figure 17: The number of record versions.

5.6 Overhead of Garbage Collection

Finally, to check the amount of overhead incurred by HG, we measure the TPC-C transaction throughput while varying the invocation period of the three different garbage collectors. Figure 18 and Figure 19 show the TPC-C transaction throughput in terms of the executed statements per second with increasing the garbage collectors' invocation period from 1 second, which is the minimum value configurable in SAP HANA. Note that the period of TG was varied by fixing the GT's period as 1 second under the configuration GT+TG, while the period of SI was varied by fixing the GT's period to 1 second and the TG's periods to 3 seconds under the configuration HG. Figure 18 represents experiments without any long-duration snapshot and Figure 19 represents the experiment with the long-duration cursor for the table STOCK.

In Figure 18, when there is no long-duration snapshot, varying the period of SI or TG did not affect the TPC-C throughput because most of versions are reclaimed by GT which is running by 1 second regardless of the period of SI or TG. When the periods of SI, TG, and GG are fixed to 1 second (as shown in the left-most point of each graph), HG showed about 0.8% of performance overhead compared to GT and GG+TG showed about 0.3% of overhead. This is because, compared to GT, SI or TG additionally checks whether there is any long-duration cursor or whether there is any intermediate record versions that can be reclaimed by them. Remark that, as the triggering period of GT increases, the overall performance sharply dropped due to the hash collision issue already discussed in Section 5.3.

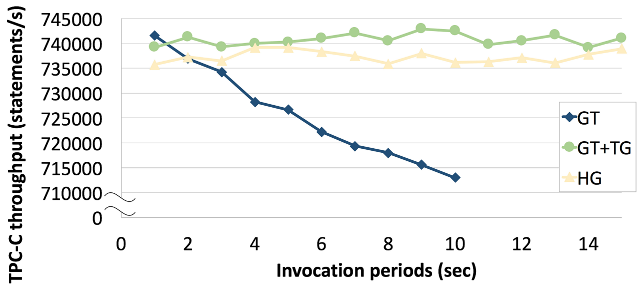

Figure 18: TPC-C throughput with varying the garbage collectors' invocation periods (without any long-duration snapshot).

In Figure 19, when there is a long-duration cursor on the table STOCK, GT almost failed to reclaim. Thus, the throughput under GT remained almost constant regardless of its invocation period. If TG were called less frequently, then the performance under GT+TG would eventually converge to the performance of GT. This is why the throughput under GT+TG slowly decreased over time. Under HG, the TPC-C throughput was almost insensitive to its invocation period.

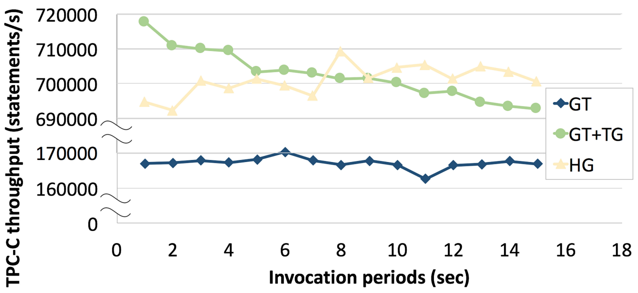

Figure 19: TPC-C throughput with varying the garbage collectors' invocation periods (with a long-duration cursor)

6. RELATED WORK

We first review existing garbage collection schemes in multi-version concurrency control. Then, we explain the fundamental difference between garbage collector for MVCC and those for programming languages. We finally explain the current implementation status and future plan for garbage collection in SAP HANA.

6.1 Garbage Collection in MVCC Database Systems

HyPer [15], one of the most recent in-memory database implementations, updates data in-place, i.e., the latest record image is directly applied into a table space. It also stores versions as before-image deltas in undo buffers which can be used for transaction rollback as well as for reconstructing a needed record version. HyPer also provides for light-weight lock-free garbage collection of versions within a transaction's undo log buffers comparing each transaction's commit timestamp with the oldest transactionID. This, however, is a variant of GT where each group corresponds to only a single transaction.

In Hekaton [6], there is no distinction between the table space and the version space. All the record versions are maintained at a single space. The versions of the same record are linked with each other but in the oldest-first order according to the figure in the paper (differently from HANA row store or Hyper). Regarding garbage collection, the paper describes "any version whose end timestamp is less than the current oldest active transaction in the system is not visible to any transaction and can be safely discarded." This indicates that their garbage collector belongs to ST. In addition to its background garbage collection process, Hekaton uses a cooperative mechanism [6] where the threads running the transaction workload can try to reclaim garbage versions when the thread traverses version chains. Compared to the latest-first version chain maintenance, it seems that this cooperative garbage collection mechanism could make more sense in their oldest-first version chain maintenance scheme because there is a higher probability of encountering garbage versions until the right version entry is found during the query processing.

Silo [20] uses a variant of single-version concurrency control which supports read-only snapshot transactions by additionally maintaining per-epoch snapshot versions. Because versions in a snapshot are claimed only when its epoch precedes the oldest transaction's epoch. This can be regarded as a variant of GT where each group corresponds to a set of transactions within an epoch.

Loesing et al. [12] propose a shared-data architecture in a distributed environment. They also use load-link and store-conditional primitives to implement efficient MVCC. The paper mentions potential growth of garbage records. However, again, [12] also uses the minimum snapshot timestamp. Thus, their garbage scheme also belongs to ST.

To the best of our knowledge, most of well-known DBMSs employing the multi-version concurrency control rely on the global garbage collection. It appears that the need and the concept of SI are already known to some implementors [9, 17]. However, the algorithm available in [17] is more expensive than the merge-based SI presented in this paper because [17] requires linear search on the active snapshot timestamps whenever it checks the visibility of a given version. [9] describes the need of SI but we could not find any further information about its implementation.

6.2 Garbage Collection under Programming Language Runtime

In the area of programming language runtime, there have been a number of research studies seeking to achieve lock-free access to shared objects by creating multiple versions of them if the object is being read by one or more threads. The created old versions can be reclaimed only after all the pre-existing readers finish their access to the old versions. That is, in this context, the garbage versions are defined as versions that are not referred to from anywhere or by the term of the "reachability" [6]. For example, in order to identify not-referenced object versions, [1, 5, 21] maintain *reference counters* to individual versions. [14] proposed the so-called *hazard pointers* to bookkeep the set of active references instead of using the reference counters. [7, 13] suggested to decide the reachability to an object version by checking whether all the active threads have reached a *quiescence point* [13] or a sufficient number of *epochs* [7].

Differently from the garbage collection in programming language runtime, the garbage collection under the database multi-version concurrency control relies on the concept of *visibility* rather than *reachability* because it must consider the set of versions which can potentially be accessed by the active statements or transactions in the future, not only the versions currently being accessed. Thus, the detailed mechanism is inherently different.

6.3 SAP HANA In-Memory Row Store

The MVCC implementation of SAP HANA row store was inherited from P*TIME [4, 8] since P*TIME is used as one of the key components of SAP HANA. As SAP HANA keeps evolving into a platform to handle both OLTP and OLAP workloads together, its garbage collection scheme has been revisited and optimized as presented in this paper. Among the proposed garbage collection optimizations, GT and TG have already been available in the product versions [11] while SI was only available in a pre-production version at the time of writing this paper. For HANA column store, TG is already incorporated and the other optimizations are also to be applied.

7. CONCLUSION

In this paper, we presented an efficient and effective garbage collector called HybridGC in SAP HANA. We empirically showed that HybridGC effectively reclaimed garbage versions while incurring negligible overhead.

We first proposed the novel notion of interval-based garbage collection by formally modelling the garbage collection problem as the consecutive interval intersection problem. We then proposed an efficient, merge-based solution to this problem. We next proposed the novel notion of group garbage collection, which enables us to efficiently determine a set of record versions as garbage. We also explained how this granular garbage collection can be applied to the timestamp-based garbage collector as well as the interval-based garbage collector. We next proposed the novel notion of table garbage collection, which exploits semantic information such that versions in tables irrelevant to current active snapshots can be reclaimed at any time. We finally proposed HybridGC which combines these three different flavours of garbage collectors.

Through experiments using mixed OLTP and OLAP workloads, we showed that HybridGC effectively and efficiently collected garbage versions. Overall, we believe that our comprehensive study for garbage collection lays a foundation for future research in various MVCC systems.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge Sang Kyun Cha, who lead the development of P*TIME and SAP HANA row store and thus provided the foundation for the research presented in this paper. The authors would like to appreciate Sangyong Hwang, Ki Hong Kim, and Sung Heun Wi who implemented the initial version of MVCC of P*TIME, which became the basis of this research. The authors would like to thank the anonymous reviewers for their valuable comments.

9. REFERENCES

[1] S. Ashwin, P. Roy, S. Seshadri, A. Silberschatz, and S. Sudarshan. Garbage collection in object oriented databases using transactional cyclic reference counting. In *VLDB Conference*, pages 366–375, 1997.

[2] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *ICDE Conference*, pages 1246–1252, 2015.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[4] S. K. Cha and C. Song. P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *VLDB Conference*, pages 1033–1044, 2004.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254. ACM, 2013.

[7] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[8] S. Y. Hwang, K. H. Kim, S. H. Wi, and S. K. Cha. Dual access to concurrent data in a database management system. US Patent 7930274, issued in 2009.

[9] InnoDB. A feature request to InnoDB. https://bugs.mysql.com/bug.php?id=74919.

[10] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB Conference*, pages 298–309, 2011.

[11] J. Lee, C. G. Park, Y. Chuh, J. Noh, and M. Muehle. Version garbage collection using snapshot lists. US Patent 13/750,204, issued in 2015.

[12] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD Conference*, pages 663–676, 2015.

[13] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[15] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD Conference*, pages 677–689, 2015.

[16] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD Conference*, pages 1–2, 2009.

[17] RocksDB. A code snippet from RocksDB. https://github.com/facebook/rocksdb/blob/master/db/compaction_iterator.cc#L204.

[18] SAP. SAP HANA In-memory Platform. http://go.sap.com/solution/in-memory-platform.html.

[19] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD Conference*, pages 731–742, 2012.

[20] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[21] J. D. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.