EEMARQ: Efficient Lock-Free Range Queries with Memory Reclamation

Gali Sheffi ✉
Department of Computer Science, Technion, Haifa, Israel

Pedro Ramalhete ✉
Cisco Systems, Zürich, Switzerland

Erez Petrank ✉
Department of Computer Science, Technion, Haifa, Israel

Abstract

Multi-Version Concurrency Control (MVCC) is a common mechanism for achieving linearizable range queries in database systems and concurrent data-structures. The core idea is to keep previous versions of nodes to serve range queries, while still providing atomic reads and updates. Existing concurrent data-structure implementations, that support linearizable range queries, are either slow, use locks, or rely on blocking reclamation schemes. We present EEMARQ, the first scheme that uses MVCC with lock-free memory reclamation to obtain a fully lock-free data-structure supporting linearizable inserts, deletes, contains, and range queries. Evaluation shows that EEMARQ outperforms existing solutions across most workloads, with lower space overhead and while providing full lock freedom.

2012 ACM Subject Classification Software and its engineering → Memory management; Theory of computation → Concurrency

Keywords and phrases safe memory reclamation, lock-freedom, snapshot, concurrency, range query

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.5

Related Version *Full Version*: `https://arxiv.org/abs/2210.17086`

Supplementary Material *Software (Source Code)*: `https://github.com/galisheffi/EEMARQ`
archived at `swh:1:dir:4e0ee7f41e81d100392b41c7af384a7896ee9760`

Funding This work was supported by the Israel Science Foundation Grant No. 1102/21.

1 Introduction

Online Analytical Processing (OLAP) transactions are typically long and may read data from a large subset of the records in a database [51, 56]. As such, analytical workloads pose a significant challenge in the design and implementation of efficient concurrency controls for database management systems (DBMS). Two-Phase Locking (2PL) [64] is sometimes used, but locking each record before it is read implies a high synchronization cost and, moreover, the inability to modify these records over long periods. Another way to deal with OLAP queries is to use Optimistic Concurrency Controls [25], where the records are not locked, but they need to be validated at commit time to guarantee serializability [46]. If during the time that the analytical transaction executes, there is any modification to one of these records, the analytical query will have to abort and restart. Aborting can prevent long read-only queries from ever completing.

DBMS designers typically address these obstacles using Multi-Version Concurrency Control (MVCC). MVCC's core idea is to keep previous versions of a record, allowing transactions to read data from a fixed point in time. For managing the older versions, each record is associated with its list of older records. Each version record contains a copy of an older version, and its respective time stamp, indicating its commit time. Each update of the record's values adds a new version record to the top of the version list, and every read of an older version is done by traversing the version list, until the relevant timestamp is reached.

© Gali Sheffi, Pedro Ramalhete, and Erez Petrank;
licensed under Creative Commons License CC-BY 4.0
26th International Conference on Principles of Distributed Systems (OPODIS 2022).
Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 5; pp. 5:1–5:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keeping the version lists relatively short is fundamental for high performance and low memory footprint [11, 19, 33, 34, 36]. However, the version lists should be carefully pruned, as a missing version record can be harmful to an ongoing range query. In the Database setting, the common approach is to garbage collect old versions once the version list lengths exceed a certain threshold [11, 36, 52]. During garbage collection, the entire database is scanned for record instances with versions that are not required for future scanners. However, garbage collecting old versions with update-intensive workloads considerably slows down the entire system. An alternative approach is to use transactions and associate each scan operation with an undo-log [6, 41]. But this requires allocating memory for storing the undo logs and a safe memory reclamation scheme to recycle log entries.

In contrast to the DBMS approach, many concurrent in-memory data-structures implementations do not provide an MVCC mechanism, and simply give up on range queries. Many map-based data-structures provide linearizable [29] insertions, deletions and membership queries of single items. Most data-structures that do provide range queries are blocking. They either use blocking MVCC mechanisms [3, 12, 43], or rely on the garbage collector of managed programming languages [7, 22, 50, 67] which is blocking. It may seem like lock-free data-structures can simply employ a lock-free reclamation scheme with an MVCC mechanism to obtain full lock-freedom, but interestingly this poses a whole new challenge.

Safe manual reclamation (SMR) [40, 55, 58, 61, 66] algorithms rely on *retire()* invocations by the program, announcing that a certain object has been unlinked from a data-structure. The task of the SMR mechanism is to decide which retired objects can be safely reclaimed, making their memory space available for re-allocation. Most SMR techniques [40, 55, 61, 66] heavily rely on the fact that retired objects are no longer reachable for threads that concurrently traverse the data structure. Typically, objects are retired when deleted from the data-structure. However, when using version lists, if we retire objects when they are deleted from the current version of the data-structure, they would still be reachable by range queries via the version list links. Therefore, it is not safe to recycle old objects with existing memory reclamation schemes. Epoch-based reclamation (EBR) [15, 24] is an exception to the rule because it only requires that an operation does not access nodes that were retired before the operation started. Namely, an existing range query prohibits reclamation of any deleted node, and subsequent range queries do not access these nodes, so they can be deleted safely. Therefore, EBR can be used without making any MVCC-specific adjustments, and indeed EBR is used in many in-memory solutions [3, 43, 65]. However, EBR is not robust [58, 61, 66]. I.e., a slow executing thread may prevent the reclamation of an unbounded number of retired objects, which may affect performance, and theoretically block all new allocations.

One lock-free memory reclamation scheme that can be adopted to provide lock-free support for MVCC is the VBR optimistic memory reclamation scheme [58, 59]. VBR allows a program to access reclaimed space, but it raises a warning when accessed data has been re-allocated. This allows the program to retry the access with refreshed data. The main advantages of VBR are that it is lock-free, it is fast, and it has very low memory footprint. Any retired object can be immediately reclaimed safely. The main drawback[1] is that reclaimed memory cannot be returned to the operating system, but must be kept for subsequent node allocations, as program threads may still access reclaimed nodes. Similarly to other schemes, the correctness of VBR depends on the inability of operations to visit previously retired

[1] VBR also necessitates type-preservation. However, this does not constitute a problem in our setting, as all allocated memory objects are of the same type. For more details, see Section 3.

objects. In data structures that do not use multi versions, the deleting thread adequately retires a node after disconnecting it from the data structure. But in the MVCC setting, disconnected nodes remain connected via the version list, foiling correctness of VBR.

In this paper we modify VBR to work correctly in the MVCC setting. It turns out that for the specific case of old nodes in the version list, correctness can be obtained. VBR keeps a slow-ticking epoch clock and it maintains a birth epoch field for each node. Interestingly, this birth epoch can be used to tell whether a retired node in the version list has been re-allocated. As we show in this paper, an invariant of nodes in the version list is that they have non-increasing birth-epoch numbers. Moreover, if one of the nodes in the version list is re-allocated, then this node must foil the invariant. Therefore, when a range query traverses a version list to locate the version to use for its traversal, it can easily detect a node that has been re-allocated and whose data is irrelevant. When a range query detects such re-allocation, it restarts. As shown in the evaluation, restarting happens infrequently with VBR and the obtained performance is the best among existing schemes in the literature. Using the modified VBR in the MVCC setting, a thread can delete a node and retire it after disconnecting it from the data structure (and while it is still reachable via version list pointers).

Two recent papers [43, 65] presented efficient MVCC-based key-value stores. The main new idea is to track and keep a version list of modified fields only and not of entire nodes. For many data-structures, all fields are immutable except for one or two pointer fields. For such data-structures, it is enough to keep a version list of pointers only. The first paper proposed a lock-free mechanism, based on *versioned CAS objects* (vCAS) [65], and the second proposed a blocking mechanism, based on *Bundle objects* (Bundles) [43]. While copying one field instead of the entire node reduces the space overhead, the resulting indirection is harmful for performance: to dereference a pointer during a traversal, one must first move to the top node of the version list, and then use its pointer to continue with the traversal. As we show in Section 4, this indirection suffers from high overheads in read-intensive workloads. Bundles ameliorate this overhead by caching the most recent pointer in the node to allow quick access for traversals that do not use older versions. However, range queries still need to dereference twice as many references during a traversal. The vCAS approach presents a more complicated optimization that completely eliminates indirection (which we further discuss in Section 3). However, its applicability depends on assumptions on the original data-structure that many data structures do not satisfy. Therefore, it is unclear how it can be integrated into general state-of-the-art concurrent data-structures. In terms of memory reclamation, both schemes use the EBR technique (that may block in theory, due to allocation heap exhaustion). This means that even vCAS does not provide a lock-free range query mechanism. A subsequent paper on MVCC-specific garbage collectors [8], provides a robust memory reclamation method for collecting the version records, but it does not deal with the actual data-structure nodes.

In this paper we present EEMARQ (End-to-End lock-free MAp with Range Queries), a design for a lock-free in-memory map with MVCC and a robust lock-free memory management scheme. EEMARQ provides linearizable and high-performance inserts, deletes, searches, and range queries.

The design starts by applying vCAS to a linked-list. The linked-list is simple, and so it allows applying vCAS easily. However, the linked-list does not satisfy the optimization assumptions required by [65], and so a (non-trivial) extension is required to fit the linked-list. In the unoptimized variant, there is a version node for each of the updates (insert or delete) and this node is used to route the scans in the adequate timestamp. But this is costly, because

traversals end up accessing twice the number of the original nodes. A natural extension is to associate the list nodes with the data that is originally kept in the version nodes. We augment the linked-list construction to allow the optimization of [65] for it. They use a clever mapping from the original version nodes to existing list nodes, that allows moving the data from version nodes to list nodes, and then elide the version nodes. Now traversals need no extra memory accesses to version nodes. This method is explained in Section 3.2.

Second, we extend VBR by adding support for reachable retired nodes on the version list. The extended VBR allows keeping retired reachable version nodes in the data structure (which the original VBR forbids) while maintaining high performance, lock-freedom, and robustness.

Finally, we deal with the inefficiency of a linked-list by adding a fast indexing to the linked-list nodes. A fast index can be obtained from a binary search tree or a skip list. But the advantage we get from separating the linked-list from the indexing mechanism is that we do not have to maintain versions for the index (e.g., for the binary search tree or the skip list), but only for the underlying linked-list. This separation between the design of the versioned linked-list and the non-versioned index, simplifies each of the sub-designs, and also obtains high performance, because operations on the index do not need to maintain versions. Previous work uses this separation idea between the lower level of the skip list or leaves of the tree from the rest for various goals (e.g., [43, 69]).

The combination of all these three ideas, i.e., optimized versioned linked-list, extended VBR, and independent indexing, yields a highly performant data structure design with range queries. Evaluation shows that EEMARQ outperforms both vCAS and Bundles (while providing full lock-freedom).

2 Related Work

Existing work on linearizable range queries in the shared-memory setting includes many solutions which are not based on MVCC techniques. Some data-structure interfaces originally include a tailor-made range query operation. E.g., there are trees [12, 13, 22, 67], hash tries [54], queues [44, 45, 53], skip lists [5] and graphs [30] with a built-in range query mechanism. Other and more general solutions execute range queries by explicitly taking a snapshot of the whole data-structure, followed by collecting the set of keys in the given range [1, 4]. The Snapcollector [50] forces the cooperation of all executing threads while a certain thread is scanning the data-structure. Despite being lock-free and general, the Snapcollector's memory and performance overheads are high. The Snapcollector was enhanced to support range queries that do not force a snapshot of the whole data-structure [16]. However, this solution still suffers from major time and space overheads.

Another way to implement range queries is to use transactional memory [23, 31, 48, 49]. Transactions can either be implemented in software or in hardware, allowing range queries to take effect atomically. Although transactions may seem as ideal candidates for long queries, software transactions incur high performance overheads and hardware transactions frequently abort when accessing big memory chunks. Read-log-update (RLU) [38] borrows software transaction techniques and extends read-copy-update (RCU) [39] to support multiple updates. RLU yields relatively simple implementations, but its integration involves re-designing the entire data-structure. In addition, similarly to RCU, it suffers from high overheads in write-extensive workloads.

Arbel-Raviv and Brown exploited the EBR manual reclamation to support range queries [3]. Their technique uses EBR's global epoch clock for associating data-structure items with their insertion and deletion timestamps. EEMARQ uses a similar technique for associating

data-structure modifications with respective timestamps. They also took advantage of EBR's retire-lists, in order to locate nodes that were deleted during a range query scan. However, while their solution indeed avoids extensive scanning helping when deleting an item from the data-structure (as imposed by the Snapcollector [50]), it may still impose significant performance overheads (as shown in Section 4). EEMARQ minimizes these overheads by keeping a reachable path to deleted nodes (for more details, see Section 3.2).

Multi-Version Concurrency Control. MVCC easily provides isolation between concurrent reads and updates. I.e., range queries can work on a consistent view of the data, while not interfering with update operations. This powerful technique is widely used in commercial systems [19, 21], as well as in research-oriented DBMS [32, 52], in-memory shared environments [43, 65] and transactional memory [23, 31, 35, 48, 49]. MVCC has been investigated in the DBMS setting for the last four decades, both from a theoretical [9, 10, 34, 47, 68] and a practical [19, 32, 36, 41, 52] point of view. A lot of effort has been put in designing MVCC implementations and addressing the unwanted side-effects of long version lists. This issue is crucial both for accelerating version list scans and for reducing the contention between updates and garbage collection. Accelerating DBMS version list scans (independently of the constant need to prune them) has been investigated in [34]. Most DBMS-related work that focuses on this issue, tries to minimize the problem by eagerly collecting unnecessary versions [2, 11, 21, 33, 37, 41, 52].

Safe Memory Reclamation. Most existing in-memory environments that enable linearizable range queries, avoid managing their memory, by relying on automatic (and blocking) garbage collection of old versions [7, 22, 50, 67]. Solutions that do manually manage their allocated memory [3, 43, 65], use EBR for safe reclamation. In EBR, a shared epoch counter is incremented periodically, and upon each operation invocation, the threads announce their observed epoch. The epoch clock can be advanced when no executing thread has an announcement with a previous epoch. During reclamation, only nodes that had been retired at least two epochs ago are reclaimed. EBR is safe for MVCC because a running scan prevents the advance of the epoch clock, and also the reclamation of any node in the data-structure that was not deleted before the scan.

The MVCC-oriented garbage collector from [8] incorporates reference counting (RC) [17, 18, 27], in order to maintain lock-freedom while safely reclaiming old versions. Any object can be immediately reclaimed once its reference count reaches zero, without the need in invoking explicit *retire()* calls. While RC simplifies reclamation, it incurs high performance overheads and does not guarantee a tight bound on unreclaimed garbage (i.e., it is not robust).

Other reclamation schemes were also considered when designing EEMARQ. NBR [61] was one of the strongest candidates, as it is fast and lock-free (under some hardware assumptions). However, it is not clear whether NBR can be integrated into a skip list implementation (which serves as one of our fast indexes). Pointer–based reclamation methods (e.g., Hazard Pointers [40]) allow threads to protect specific objects (i.e., temporarily prevent their reclamation), by announcing their future access to these objects, or publishing an announcement indicating the protection of a bigger set of objects (e.g., Hazard Eras [55], Interval-Based Reclamation [66], Margin Pointers [62]). Although these schemes are robust (as opposed to EBR), it is unclear how they can be used in MVCC environments. They require that reaching a reclaimed node from a protected one would be impossible (even if the protected node is already retired). I.e., they require an explicit unlinking of old versions before retiring them. Besides the obvious performance overheads, it may affect robustness

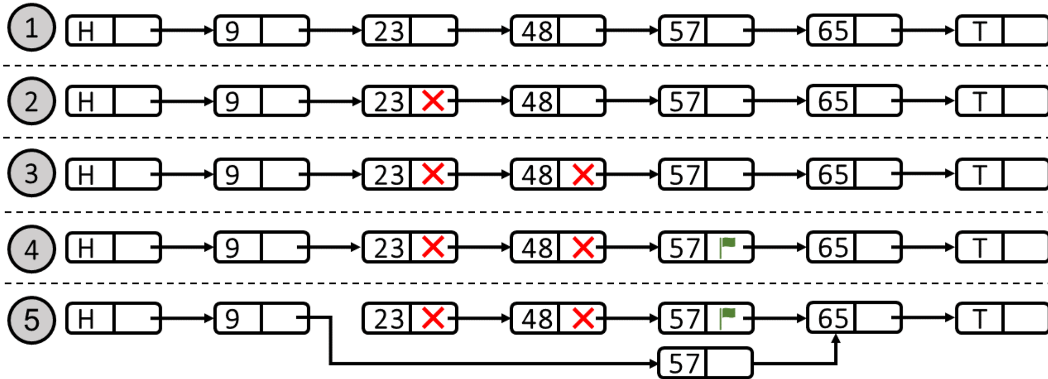


Figure 1 Removing nodes 23 and 48 from the linked-list. At stages 1–4, the list logically contains 5 nodes and at stage 5, it logically contains 3 nodes. The logical deletions of nodes 23 and 48 are executed via marking them (stages 2 and 3, respectively), flagging node 57 (stage 4) and inserting a new 57 representative instead of the three of them (stage 5). Nodes 23, 48, and the flagged 57 are then retired.

(as very old versions would not be reclaimed). The garbage collector from [8] uses Hazard Eras for unlinking old versions (to be eventually collected using an RC-based strategy), but it has not been evaluated in practice.

3 The Algorithm

In this Section we present EEMARQ's design. We start by introducing a new lock-free linearizable linked-list implementation in Section 3.1. The list implementation is based on Harris's lock-free linked-list [26], and includes the standard *insert()*, *remove()* and *contains()* operations. In Section 3.2 we explain how to add a linearizable and efficient range query operation. We describe the integration of the designated robust SMR algorithm in Section 3.3, and explain how to improve performance by adding an external index in Section 3.4. A full linearizability and lock-freedom proof for our implementation (including the range queries mechanism and the SMR and fast indexing integration) appears in the full version of this paper [60].

As discussed in [65], node-associated version lists introduce an extra level of indirection per node access. Methods that use designated version objects for recording updates, suffer from high overheads, especially in read-intensive workloads. For avoiding this level of indirection, we introduce a new variant of Harris's linked-list. In our new variant, there is no need to store any update-related data in designated version records, since it can be stored directly inside nodes, in a well-defined manner. Associating each node with a single data-structure update (i.e., an insertion or a removal) is challenging. Typically, an insert operation includes a single update, physically inserting a node into the list. A remove operation involves a marking of the target node's *next* pointer (serving as its logical deletion, and the operation's linearization point in many existing implementations) and a following physical removal from the list. In other words, each node may be associated with multiple list updates. Since the target node's physical deletion is not the linearization point of any operation, there is no need to record this update. However, each node may still be associated with either one or two updates throughout an execution (i.e., its logical insertion and deletion).

In our linked-list implementation, some new node is inserted into the list during every physical update of the list (either a physical insertion or deletion), which obviously yields the desirable association between nodes and data-structure updates (each node is associated with the update that involved its insertion into the list). The node inserted during a physical insertion is simply the inserted node, logically inserted into the list. The node inserted during a physical removal is a designated new node that replaces the deleted node's successor, and is physically inserted together with the physical removal of the deleted node [2].

Figure 1 shows an example for inserting a new node during deletion. This list illustration shows the list layout throughout the deletion procedure of two nodes. At the first stage, the list contains five nodes, ordered by their keys (together with the *head* and *tail* sentinels). Then, some thread marks node 23 for deletion. At this point, before physically removing it, some other thread marks node 48 for deletion. I.e., both nodes must be physically unlinked together, as successive marked nodes. In order to remove the marked nodes, node 57 is flagged, as it is the successor of the last marked node in the sequence. Node 57 is flagged for making sure that its *next* pointer does not change. Finally, all three nodes are physically unlinked from the list, together with the physical insertion of a new node, representing the old flagged one. I.e., although all three nodes were physically removed from the list, node 57 was not logically removed, as it was replaced by a new node with the same key. After the physical deletion at stage 5, the three nodes are retired (for more details, see Section 3.3). As opposed to Harris's implementation, the linearization point of both deletions is this physical removal, which atomically inserts the new representative into the list. Our mapping from modifications to nodes, maps this deletion to the new node, inserted at stage 5 (which represents the deletion of both nodes). In Section 3.2 we explain how this mapping is used for executing range queries.

3.1 The Linked-List Implementation

Our linked-list implementation, together with the list node class, is presented in Algorithm 1. The simple pointer access methods implementation (e.g., *mark()* in line 23 and *getRef()* in line 40) appear in Appendix A. In a similar way to Harris's list, the API includes the *insert()*, *remove()* and *contains()* operations [3]. The *insert()* operation (lines 7–17) receives a key and a value. If there already exists a node with the given key in the list, it returns its value (line 11). Otherwise, it adds a new node with the given key and value to the list, and returns a designated NO_VAL answer (line 17). The *remove()* operation (lines 18–25) receives a key. If there exists a node with the given key in the list, it removes it and returns its value (line 25). Otherwise, it returns NO_VAL (line 22). The *contains()* operation (lines 26–30) receives a key. If there exists a node with the given key in the list, it returns its value (line 30). Otherwise, it returns NO_VAL (line 29).

All three API operations use the *find()* auxiliary method (lines 31-58), which receives a key and returns pointers to two nodes, *pred* and *curr* (line 58). As in Harris's implementation, it is guaranteed that at some point during the method execution, both nodes are consecutive reachable nodes in the list, *pred*'s key is strictly smaller than the input key, and *curr*'s key is equal or bigger than the given key. I.e., if *curr*'s key is strictly bigger than the input key, it is guaranteed that there is no node with the given input key in the list at this point. The method traverses the list, starting from the *head* sentinel node (line 33),

[2] When multiple nodes are physically removed together, it replaces the successor of the last node in the sequence of deleted nodes.

[3] The *rangeQuery()* operation is added in Section 3.2. In addition, lines marked in blue in Algorithm 1 can be ignored at this point. They will also be discussed in Section 3.2

Algorithm 1 Our Linked-List Implementation.

```
1:  class Node
2:      Long ts
3:      K key
4:      V value
5:      Node* next
6:      Node* prior
7:  procedure INSERT(key, val)
8:      while (true) do
9:          pred, curr ← FIND(key)
10:         if (curr → key == key)
11:             return curr → val
12:         n := alloc(key, val, ⊥)
13:         n → next := curr
14:         n → prior := curr
15:         if (CAS(&pred → next, curr, n))
16:             CAS(&n → ts, ⊥, getTS())
17:             return NO_VAL
18: procedure REMOVE(key)
19:     while (true) do
20:         pred, curr ← FIND(key)
21:         if (curr → key ≠ key)
22:             return NO_VAL
23:         if (!mark(curr)) continue
24:         FIND(key)                ▷ physical deletion
25:         return curr → val
26: procedure CONTAINS(key)
27:     pred, curr ← FIND(key)
28:     if (curr → key ≠ key)
29:         return NO_VAL
30:     else return curr → val
31: procedure FIND(key)
32:     retry:
33:     pred := head
34:     pNext := pred → next
35:     curr := getRef(pNext)
36:     while (true) do
37:         while(isMarkdOrFlagged(curr →
    next))
38:             if (!getRef(curr → next))
39:                 break
40:             curr := getRef(curr → next)
41:         if (curr → key ≥ key)  break
42:         pred := curr
43:         pNext := pred → next
44:         if (isMarkdOrFlagged(pNext))
45:             goto retry
46:         curr := getRef(pNext)
47:     CAS(&pred → ts, ⊥, getTS())
48:     if (pNext ≠ curr)
49:         if (!TRIM(pred, getRef(pNext)))
50:             goto retry
51:         pNext := pred → next
52:         if (isMarkdOrFlagged(pNext))
53:             goto retry
54:         curr := pNext
55:         if ( isMarkedOrFlagged(curr →
                next) ∨ curr → key < key)
56:             goto retry
57:     CAS(&curr → ts, ⊥, getTS())
58:     return pred, curr
59: procedure TRIM(pred, victim)
60:     curr := victim
61:     while (isMarked(curr → next)) do
62:         curr := getRef(curr → next)
63:     CAS(&curr → ts, ⊥, getTS())
64:     if  (!flag(curr)  ∧  !isFlagged(curr  →
    next))
65:         return false
66:     succ := getRef(curr → next)
67:     if (succ) CAS(&succ → ts, ⊥, getTS())
68:     newCurr := alloc(curr → key, curr → val,
    ⊥)
69:     newCurr → next := succ
70:     newCurr → prior := victim
71:     if   (CAS(&pred   →   next,   victim,
    newCurr))
72:         CAS(&newCurr → ts, ⊥, getTS())
73:         return true
74:     return false
```

and until it gets to an unmarked and unflagged node with a key which is at least the input key (line 41). Recall that the two output variables are guaranteed to have been reachable, adjacent, unmarked and not flagged at some point during the method execution. Therefore, as long as the current traversed node is either marked or flagged (checked in line 37), the traversal continues, regardless of the current key (lines 37-40). Once the traversal terminates (either in line 38 or 41), if the current two nodes, saved in the *pred* and *curr* variables, are adjacent (the condition checked in line 48 does not hold), then the method returns them in line 58. Otherwise, similarly to the original implementation, the method is also in charge of physically removing marked nodes from the list.

As we are going to discuss next, our physical removal procedure, as depicted in Figure 1, is slightly different from the original one [26]. Physical deletions are executed via the *trim()* auxiliary method (lines 59–74). Although nodes are still marked for deletion in our implementation (line 23), their successful marking does not serve as the removal linearization point. I.e., reachable marked nodes are still considered as list members. The *trim()* method receives two nodes as its input parameters, *pred* and *victim*. *victim* is the physical removal candidate, and is assumed to already be marked. *pred* is assumed to be *victim*'s predecessor in the list, and to be neither marked nor flagged. As depicted in Figure 1, consecutive marked nodes are removed together. Therefore, the method traverses the list, starting from *victim*, for locating the first node which is not marked (lines 61–62). When such a node is found, the method tries to flag its *next* pointer, for freezing it until the removal procedure is done. In general, pointers are marked and flagged using their two least significant bits (which are practically redundant when reading node address aligned to a word). Both marked and flagged pointers are immutable, and a pointer cannot be both marked and flagged. Therefore, the flagging trial in line 64 fails if *curr*'s *next* pointer is either marked or flagged. If the flagging trial is unsuccessful, and not because some other thread has already flagged *curr*'s *next* pointer, the method returns in line 65. Otherwise, a new node is created in order to replace the flagged one (lines 68–70). Note that since this node's *next* pointer is flagged (i.e., immutable), it is guaranteed that the new node points to the original one's current successor. The actual trimming is executed in line 71. If the compare-and-swap (CAS) is successful, then the sequence of marked nodes, together with the single flagged one (at the end of the sequence), are atomically removed from the list, together with the insertion of the new copy of the flagged node (the new copy is neither flagged nor marked).

As the physical removal is necessary for linearizing the removal (as will be further discussed in Section 3.2), a remover must physically remove the deleted node before it returns from a *remove()* call. The marking of a node in line 23 only determines the remover's identity and announces its intention to delete the marked node. Therefore, the remover must additionally ensure that the node is indeed unlinked, by calling the *find()* method in line 24. In the full version of this paper [60] we formally prove that the list implementation, presented in Algorithm 1, is linearizable and lock-free.

3.2 Adding Range Queries

Given Algorithm 1, adding a linearizable range queries mechanism is relatively straight forward. We use a method which is similar to the vCAS technique [65]. As discussed in Section 1, the vCAS scheme introduces an extra level of indirection for the linked-list per node access. Indeed, we show in Section 4 that the vCAS implementation suffers from high overheads. The original vCAS paper provides a technique for avoiding this level of indirection. The suggested optimization relies on the following (very specific) assumption: a certain node can be the third input parameter to a successful CAS operation only once throughout the entire execution. That successful CAS is considered as the *recording* of this node, and the property is referred to as *recorded-once* in [65].

Although the recorded-once property yields a linearizable solution, which reduces memory and time overheads, this assumption does not hold in the presence of physical deletions, as they usually set the deleted node's predecessor to point to the deleted node's successor [26], or to another, already reachable node [14, 42], and then, this reachable node is recorded more than once. This makes the suggested technique inapplicable to Harris's linked-list [26] and most other concurrent data-structures (e.g., [12, 28, 42]). The original vCAS paper implemented a recorded-once binary search tree, based on [20], which we compare against in

Section 4. We extend the recorded-once condition and make it fit for the linked-list and other data-structures. We claim that associating each node with the data of a single data-structure update (as provided by our list) is enough for avoiding indirection in this setting. Given such an association, there is no need to store update-related data in designated version records, since it can be stored directly inside nodes, in a well-defined manner.

First, in a similar way to [3, 43, 65], we add a shared clock, for associating each node with a timestamp. The shared clock is read and updated using the *getTS()* and *fetchAddTS()* methods, respectively (see Appendix A). The shared clock is incremented whenever a range query is executed (e.g., see line 2 in Algorithm 2), and is read before setting a new node's timestamp (e.g, see lines 16, 47, 57, 63, 67 and 72 in Algorithm 1). Next, we change the nodes layout (see our node class description in Algorithm 1). On top of the standard fields (i.e., key, value and *next* pointer), we add two extra fields to each node. The first field is the node's timestamp (denoted as *ts*), representing its insertion into the list. Nodes' *ts* fields are always initialized with a special $\perp$ value (see lines 12 and 68 in Algorithm 1), to be given an actual timestamp after being inserted into the list. The second field, *prior*, points to the previous successor of this node's first predecessor in the list (its predecessor when being inserted into the list). Both fields are set once and then remain immutable. E.g., consider the new node, inserted into the list at stage 5 in Figure 1. Its *prior* field points to the node whose key is 23, as this is the former successor of the node whose key is 9, which is the first predecessor of the newly inserted node. By its specification, once the *prior* field is set (see line 14 and 70 in Algorithm 1), it is immutable. These two new fields are not used during the list operations from Algorithm 1, but we do specify their proper initialization, in order to support linearizable range queries. Moreover (and in a similar way to [3, 65]), list inserts and deletes are linearized during the execution of the *getTS()* method, as follows. Let n be the node inserted into the list in line 15, during a successful *insert()* operation. n's timestamp is set at some point, not later than the CAS in line 16 (it may be updated earlier, by a different thread). The *getTS()* invocation that precedes the successful update of n's timestamp is the operation's linearization point. In a similar way, consider a successful *remove()* operation. The removed node is unlinked from the list during a successful *trim()* execution. Let *newCurr* be the node successfully inserted into the list in line 71, during this successful *trim()* execution. *newCurr*'s timestamp is set at some point, not later than the CAS in line 72 (it may be updated earlier, by a different thread). The *getTS()* invocation that precedes the successful update of *newCurr*'s timestamp is the operation's linearization point.

Our range queries mechanism is presented in Algorithm 2. The *rangeQuery()* operation receives three input parameters (see line 1): the lowest and highest keys in the range, and an output array for returning the actual keys and associated values in the range. In addition to filling this array, it also returns its accumulated size in the *count* variable. The operation starts by fetching and incrementing the global timestamp counter (line 2), which serves as the range query's linearization point. I.e., the former timestamp is the one associated with the range query. This way, the range query is indeed linearized between its invocation and response, along with guaranteeing that the respective view is immutable during the operation (as new updates will be associated with the new timestamp).

After incrementing the global timestamp counter, the operation uses the *find()* auxiliary method in order to locate the first node in range (lines 4–12). As opposed to the vCAS mechanism [65], and in a similar way to the Bundles mechanism [43], we observe that until the traversal reaches the target range, there is no need to take timestamps into consideration. This observation is crucial for performance, as there is no need to traverse nodes via the

Algorithm 2 The Range Queries Mechanism.

```
1:  procedure RANGEQUERY(low, high, *arr)
2:      ts := fetchAddTS()
3:      currKey := low
4:      while (true) do
5:          pred, curr ← FIND(currKey)
6:          currKey := pred → key
7:          while (pred → ts > ts) do
8:              pred := pred → prior
9:          if (pred → key ≤ low)
10:             curr := pred
11:             break
12:         ts := getTS() - 1
13:     while (curr → key < low) do
14:         succ := getRef(curr → next)
15:         CAS(&succ → ts, ⊥, getTS())
16:         while (succ → ts > ts) do
17:             succ := succ → prior
18:         curr := succ
19:     count := 0
20:     while (curr → key ≤ high) do
21:         arr[count] → key := curr → key
22:         arr[count] → value := curr → value
23:         count := count + 1
24:         succ := getRef(curr → next)
25:         CAS(&succ → ts, ⊥, getTS())
26:         while (succ → ts > ts) do
27:             succ := succ → prior
28:         curr := succ
29:     return count
```

prior fields (which produce longer traversals in practice). In addition, it enables using the fast index (described in Section 3.4) for enhancing the search. During each loop iteration, we first find a node with a key which is smaller than the lowest key in the range (saved as the *pred* variable in line 5). Then, we optimistically try to find a relatively close node, following *prior* pointers, until we get to a small enough timestamp (lines 7–8). Since this search may result in a node with a bigger key (e.g., see line 14 in Algorithm 1), the next iteration sends a smaller key as input to the *find()* execution in line 5. Note that in the worst case scenario, the loop in lines 4–12 stops after the *find()* execution in line 5 outputs the *head* sentinel node (as its timestamp is necessarily smaller than ts). Therefore, it never runs infinitely. The purpose of updating *ts* in line 12 will be clarified in Section 3.3, as it is related to the VBR mechanism. Note that in any case, this update does not foil correctness, since it is still guaranteed that the range query is linearized between the operation's invocation and response.

When *pred* has a key which is smaller than the range lower bound, the operation moves on to the next step (the loop breaks in line 11). At this point, the traversal continues according to the respective timestamp[4], until getting to a node with a key which is at least the range lower bound (lines 13–18). Once a node with a big enough key is found, the traversal continues in lines 20–28. At this stage, the *count* output variable and the output array are updated according to the data accumulated during the range traversal. Finally, the *count* output variable, indicating the total number of keys in range, is returned in line 29. Note that throughout the traversals in lines 16–17 and lines 26–27, there is no need to update *succ*'s timestamp (as done in lines 15 and 25), since it serves as a node's *prior* and thus, is guaranteed to already have an updated timestamp (for more details, see the full version of this paper [60]).

[4] In the full version of this paper [60] we prove that a node's successor at timestamp T can be found by starting from its current successor and then following *prior* references until reaching a node with a timestamp which is not greater than T.

3.3 Adding A Safe Memory Reclamation Mechanism

Before integrating our list with a manual memory reclamation mechanism, we must first install *retire()* invocations, for announcing that a node's memory space is available for re-allocation. Naturally, nodes are retired after unsuccessful insertions, or after they are unlinked from the list. I.e., n is retired if the CAS in line 15 is unsuccessful, *newCurr* is retired if the CAS in line 71 is unsuccessful, and upon a successful trimming in line 71, the unlinked nodes are retired, starting from *victim*. The last retired node is *curr*, which is replaced by its new representative in the list, *newCurr*. Note that we do not handle physical removals of *prior* links. Handling them is unnecessary, and might cause significant overheads, both to the list operations and to the reclamation procedure. Therefore, retired nodes are still reachable from the list head during retirement: *newCurr*'s *prior* field points to *victim*, making all of the unlinked nodes reachable via this pointer (and their *next* pointers).

To add a safe memory reclamation mechanism to our list, we use an improved variant of Version Based Reclamation (VBR) [58]. VBR cannot be integrated as is. Similarly to most safe memory reclamation techniques, it assumes that retired objects are not reachable via the data-structure links. This assumption is crucial to the correctness of VBR, as retired objects may be immediately reclaimed. In addition, VBR uses a slow ticking epoch clock, and ensures that the clock ticks at least once between the retirement and future re-allocation of the same node. During execution, the operating threads constantly check that the global epoch clock has not changed. Upon a clock tick, they conservatively treat all data read from shared memory as stale, and move control to an adequate previous point in the code in order to read a fresh value. As long as the clock does not tick, threads may continue executing without worrying about use-after-free issues. The intuition is that if a node is accessed during a certain epoch, then it must have been reachable during this epoch. I.e., even if this node has already been retired, its retirement was during the current epoch, which means that it has not been re-allocated yet (as the clock has not ticked yet).

Our list implementation poses a new challenge in this context. Suppose that the current epoch is E, and that a certain node, n, is currently in the list (i.e., it has not been unlinked using the *trim()* method yet). In addition, suppose that n's *prior* field points to another node, m, that has been retired during an earlier epoch. Then m may be reclaimed and re-allocated during E. A traversing thread may access n's *prior* field during E, without getting any indication to the fact that the referenced node is a stale value. Another problem, which does not affect correctness, but may cause frequent thread starvations, is that the global epoch clock is likely to tick during a long range query. In the original VBR scheme, a clock tick forces the executing thread to start its traversal from scratch, even if it has not encountered any reclaimed node in practice.

In order to overcome the above problems, we made some small adjustments to the original VBR scheme. First, we kept the global epoch clock of VBR and the timestamp clock of the range queries separated. We separated the two, as VBR works best with a (very) slow ticking clock. Read-intensive workloads (in which range queries dominate the execution) incur high overheads when combining the two clocks. The separation of the two independent clocks helps overcome the potential aborts. The second step was to modify the nodes' layout (The VBR-integrated node layout appears in the full version of this paper [60]). Recall that the VBR scheme adds a *birth epoch* to every node, along with a version per mutable field. Non-pointer mutable fields are associated with the node's birth epoch, and pointers are associated with a version which is the maximum between the birth epoch of the node and the birth epoch of its successor. Our list nodes have two mutable fields, their timestamp *ts* (changes only once), and their *next* pointer. The *prior* pointers are immutable. Accordingly, we

associated the node's timestamp with its VBR-integrated birth epoch, serving as its version (there was no need to add an extra *ts* version), and added a designated *next* pointer version. Writes to the mutable fields are handled exactly as in the original VBR scheme. Accordingly, upon allocation, a node's timestamp is initialized to ⊥, along with the current VBR epoch as its associated birth epoch. When the timestamp is updated (see line 16, 47, 57, 63, 67 and 72 in Algorithm 1, or lines 15 and 25 in Algorithm 2), the birth epoch (also serving as the timestamp's version) does not change, as the two fields are accessed together, via a wide-compare-and-swap (WCAS) instruction. Similarly, *next* pointers are associated with the maximum between the two respective birth epochs, and are also updated using WCAS.

Reads are handled in a different manner from the original VBR, as the problems we mentioned above must be treated with special care. The original VBR repeatedly reads the global epoch in order to make sure that it has not changed. In our extended VBR variant, it is read once. After reading the global epoch, and as long as the executing thread does not encounter a birth epoch or a version which is bigger then this epoch, it may continue executing its code. The motivation behind this behavior is that even if a certain node in the system has meanwhile been reclaimed, this node does not pose a problem as long as the current thread does not encounter it. Therefore, traversing threads follow three guidelines: (1) a node's birth epoch is read again after each read of another field, (2) after dereferencing a *next* pointer, the reader additionally makes sure that the successor's birth epoch is not greater than the pointer's version, and (3) after dereferencing a *prior* pointer (which is not associated with a version), the reader additionally makes sure that the successor's birth epoch is not greater than the predecessor's birth epoch. If any of these conditions does not hold, then the reader needs to proceed according to the original VBR's protocol. In the full version of this paper [60] we prove that these three guidelines are sufficient for maintaining correctness. Upon an epoch change, the original VBR enforces a rollback to a predefined checkpoint in the code. Accordingly, we install code checkpoints. Whenever a check that our guidelines impose fails, the executing thread rolls-back to the respective checkpoint. Checkpoints are installed in the beginning of each API operation (i.e., *insert()*, *remove()*, *contains()* and *rangeQuery()*). Another checkpoint is installed after a successful marking in line 23 of Algorithm 1, as the identity of the marking thread affects linearizability (and therefore, a rollback to the beginning of the operation would foil linearizability). Note that a successful insertion in line 15 does not force a checkpoint (although it affects linearizability, by setting the inserter identity), as it is not followed by any reads of potentially reclaimed memory.

Another issue that needs to be dealt with is the guarantee that life-cycles of nodes, allocated from the same memory address, do not overlap. The original VBR scheme does so by associating each node with a retire epoch. A node's retire epoch is set upon retirement. During re-allocation, if the current global epoch is equal to the node's retire epoch, then the global epoch is incremented before re-allocation. We chose to optimize over the original VBR, discarding the retire epoch field, as it adds an extra field per allocated node. Instead, each retire list is associated with the epoch, recorded once it is full (right before it is returned to the global pool of nodes). Upon pulling such a list from the global pool, if its associated epoch is equal to the current one, then the global epoch is incremented.

Finally, consider the following scenario. Suppose that a thread T_1 is running a range query, the current epoch is E and the current global timestamp is t. Next, suppose that another thread, T_2, reclaims a node n that has been retired during E, and that is relevant for T_1's range query. Starting from this point, whenever T_1 accesses the newly allocated node, it rolls back and starts its traversal from scratch (as the new node has a birth epoch which is

greater then its predecessor through the *prior* pointer, foiling guideline 3). As long as T_1's *ts* variable does not change, T_1 will infinitely get to the new allocated node and then roll-back to the beginning. We reduce the probability of such scenarios in practice, by updating the *ts* variable in line 12 of Algorithm 2. We further ensure that the current global timestamp is up-to-date by incrementing it upon each re-allocation, if necessary.

3.4 Adding A Fast Index

Our linked-list implementation encapsulates the key-value pairs and enables the timestamps mechanism. However, when key ranges are large, the linked-list does not perform as good as other concurrent data-structures. It forces a linear traversal per operation, as opposed to skip lists [24, 28] and binary search trees [14, 42]. We observe that the index links in such data-structures (e.g., the links connecting the upper levels in a skip list or the inner levels in a tree) are only required for fast access. The actual data exists only in the lowest level of the skip list (or the leaves of an external tree). Therefore, we allow a simple integration of an external index, enabling fast access instead of long traversals. The index should provide an *insert(key, node)* operation, receiving a key and a node pointer as its associated value. It additionally should provide a *remove(key)* operation. Finally, instead of providing a *contains(key)* operation, it should provide a *findPred(key)* operation, receiving a key and returning a pointer to the node associated with some key which is smaller than the given one.

The *findPred(key)* can be naively implemented by calling the data-structure search method (there usually exists such method. E.g., [26, 28, 42]) with a smaller key as input. I.e., it is possible to search for key minus 2 or minus 10. Obviously, this does not guarantee that a suitable node will indeed be returned. However, if the selected smaller key is not small enough, it is possible to start a new trial and search for a smaller key. Our experiments showed that limiting the number of such trials per search to a small constant (e.g., 5 in our experiments) is negligible in terms of performance, and is usually enough for locating a relevant node. In addition, the index is used only for fast access, so correctness is not affected even if all trials fail. Specifically, the *findPred(key)* operation can be easily implemented for a skip list, using its built-in search auxiliary method [24, 28] (as it returns a predecessor with a smaller key). Examples for using a skip list as a fast index have already been introduced for linearizable data-structures [57] and in the transactional memory setting [63]. The *findPred(key)* operation can also be implemented for some binary search trees, by traversing the left child, instead of the right child, at some point during the search path. We applied this method when implementing our tree index, based on Natarajan and Mittal's BST [42].

Algorithm 3 Starting a new traversal using the index.

```
1:  currKey := key
2:  attempts := MAX_ATTEMPTS
3:  while (attempts ≠ 0) do
4:      attempts−−
5:      pred := index → findPred(currKey)
6:      predTS := pred → timestamp
7:      predNext := pred → next
8:      predKey := pred → key
9:      if (pred → birth > currEpoch)
10:         rollback
11:     else if (predKey ≥ key ∨
                  predTS == ⊥)
12:         continue
13:     else if (isMarkedOrFlagged(
                  predNext))
14:         currKey := predKey
15:     else break
16: if (attempts == 0)
17:     pred := head
18:     predNext := pred → next
```

We update the fast index as follows: New Nodes are inserted into the index after being inserted into the list (i.e., right before the *insert()* operation returns in line 17 of Algorithm 1). Nodes are removed from the index after being removed from the list, and right before being retired (see Section 3.3). Note that the *curr* node, replaced by *newCurr* via the CAS in line 71, should be removed from the index, followed by an insertion of *newCurr*. In our implemented index, we have implemented an *update()* operation instead. This operation receives as input a node reference, and uses it to replace a node with the same key in the index[5]. In case the external index does not provide an *update()* operation, this also may be executed via the standard *remove()* operation, followed by a respective *insert()* operation. The index is read only during the *find()* auxiliary method. Instead of starting each list traversal from the *head* sentinel node (see line 33 in Algorithm 1), the traversing thread tries to shorten the traversal by accessing the fast index. I.e., instead of executing the code in lines 33-34 of Algorithm 1, each thread executes the code from Algorithm 3. It starts by initializing the searched key to the input key (received as input in line 31 of Algorithm 1). Then, after finding the alleged predecessor, using the *findPred()* operation (line 5), if its birth epoch is bigger than the last recorded one, the executing thread rolls-back to its last recorded checkpoint (see Section 3.3). Otherwise, if *pred*'s key is not smaller than the given input key, or its timestamp is not initialized yet, the thread starts another trial, with the same key. Otherwise, if *pred* is either marked or flagged (line 13), the thread starts another trial, with a smaller key. Otherwise, it is guaranteed that *pred* and *predNext* hold a valid node and its (unmarked and unflagged) *next* pointer, respectively, and the thread may start its list traversal from line 35 of Algorithm 1. Note that the code presented in Algorithm 3 always terminates, as in the worst case scenario, the loop breaks after a predefined number of attempts.

4 Evaluation

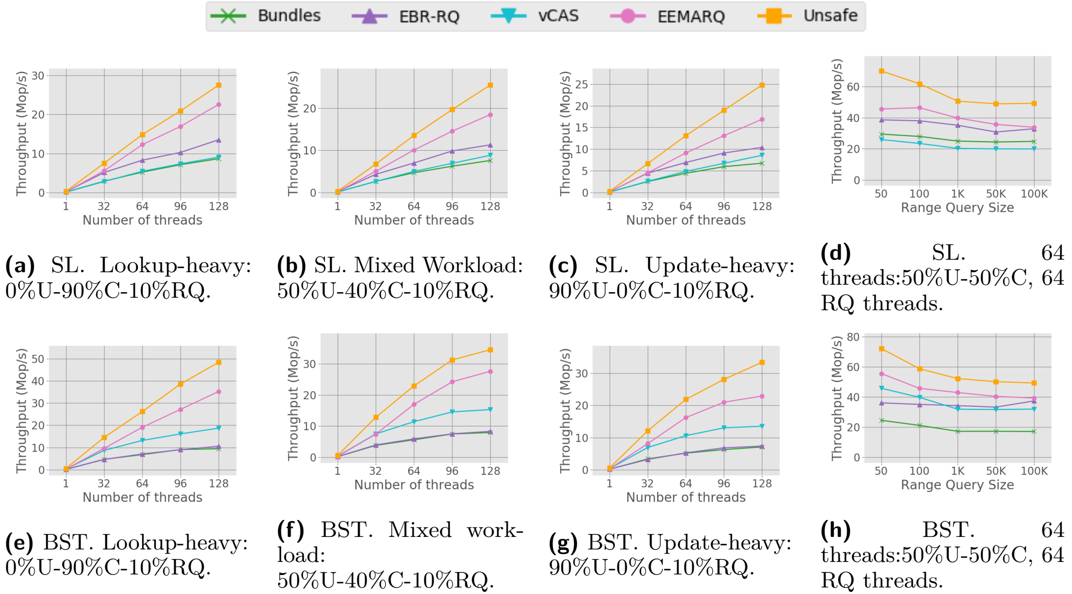
For evaluating throughput of EEMARQ, we implemented[6] the linked-list presented in Section 3[7], including the extended VBR variant, as described in Section 3.3. In addition, we implemented the lock-free skip list from [24] and the lock-free BST from [42]. Both the skip list and the tree were used on top of the linked list, and served as fast indexes, as described in Section 3.4. Deleted nodes from both indexes were manually reclaimed, using the original VBR scheme, according to the integration guidelines from [58] (without the adjustments described in Section 3.3). Each data-structure had its own objects pool (as VBR forces type preservation). Retire lists had 64 entries. Since VBR allows the immediate reclamation of retired nodes, retire lists were reclaimed as a whole every time they contained 64 nodes. I.e., at most 8192 (64 retired nodes X 128 threads) objects were over-provisioned per data-structure at any given moment.

We compared EEMARQ against four competitors, all using epoch-based reclamation. EBR-RQ is the lock-free epoch-based range queries technique by Arbel-Raviv and Brown [3], vCAS is the lock-free technique by Wei et al. [65], Bundles is the lock-based bundled references technique by Nelson et al. [43], and Unsafe uses a naive non-linearizable scan of the nodes in the range without synchronizing with concurrent updates (used as our baseline). We

[5] The implemented *update()* operation also takes the node's birth epoch into account, and does not replace a node with a reclaimed node or with a node with a smaller birth epoch

[6] The code is available here.

[7] For avoiding unnecessary accesses to the global timestamps clock, the *ts* field updates from Algoritm 1 and Algorithm 2 were executed only after the *ts* field was read, and only if it was still equal to $\perp$.



(a) SL. Lookup-heavy: 0%U-90%C-10%RQ.
(b) SL. Mixed Workload: 50%U-40%C-10%RQ.
(c) SL. Update-heavy: 90%U-0%C-10%RQ.
(d) SL. 64 threads:50%U-50%C, 64 RQ threads.
(e) BST. Lookup-heavy: 0%U-90%C-10%RQ.
(f) BST. Mixed workload: 50%U-40%C-10%RQ.
(g) BST. Update-heavy: 90%U-0%C-10%RQ.
(h) BST. 64 threads:50%U-50%C, 64 RQ threads.

Figure 2 Throughput evaluation under various workloads for the skip list (2a–2d) and the tree (2e–2h). The key range is 1M. In Figures 2a-2c and 2e-2g, the range query size is 1000. Y axis: throughput in million operations per second. X axis: #threads in Figures 2a–2c and 2e–2g, and range query size in Figures 2d and 2h.

did not compare EEMARQ against RLU [38], as its mechanism is not linearizable, and it was also shown to be slower than our competitors [43]. For EBR-RQ and Bundles, we used the implementation provided by the authors. The Unsafe code was provided by the EBR-RQ authors. The vCAS authors provided a vCAS-based lock-free BST, including their optimization for avoiding indirection (see Section 3.2 for more details). Since there does not exist any respective skip list implementation, we implemented a vCAS-based skip list according to the guidelines from [65]. The vCAS-based skip list was not optimized, as the optimization technique, suggested in [65], does not fit to this data-structure. For our competitors' memory reclamation, we used the original implementations, provided by [3, 43, 65], without any code or object pools usage changes. In particular, memory was not returned to the operating system. I.e., all implementations used pre-allocated object pools [58, 61] for reclaiming memory.

Setup. We conducted our experiments on a machine running Linux (Ubuntu 20.04.4), equipped with 2 Intel Xeon Gold 6338 2.0GHz processors. Each processor had 32 cores, each capable of running 2 hyper-threads to a total of 128 threads overall. The machine used 256GB RAM, an L1 data cache of 3MB and an L1 instruction cache of 2MB, an L2 unified cache of 80MB, and an L3 unified cache of 96MB. The code was written in C++ and compiled using the GCC compiler version 9.4.0 with -std=c++11 -O3 -mcx16. Each test was a fixed-time micro benchmark in which threads randomly call the *insert()*, *remove()*, *contains()* and *rangeQuery()* operations according to different workload profiles. We ran the experiments with a range of 1M keys. Each execution started by pre-filling the data-structure to half of its range size, and lasted 10 seconds (longer experiments showed similar results). Each experiment was executed 10 times, and the average throughput across all executions

is reported. Figure 2 shows the skip list and tree scalability under various workloads. The updates (half *insert()* and half *remove()*), *contains()* and *rangeQuery()* percentiles appear under each graph. Figures 2a-2c and 2e-2g show the skip list and tree scalability as a function of the number of executing threads, under a variety of workloads. All queries have a fixed range of 1K keys (following [65]). Figures 2d and 2h show the effect of varying range query size on the skip list and tree performance. In these experiments, 64 threads perform 25% *insert()*, 25% *remove()* and 50% *contains()*, and 64 threads perform range queries only. In the full version of this paper [60] we present the respective range queries and updates throughput for Figures 2d and 2h, along with additional results for other workloads.

Discussion. The EEMARQ skip list surpasses the next best algorithm, EBR-RQ, by up to 65% in the lookup-heavy workload (Figure 2a), by up to 50% in the mixed workload (Figure 2b), and by up to 70% in the update-heavy workload (Figure 2c). The EEMARQ tree surpasses its next best algorithm, vCAS, by up to 75% in the lookup-heavy workload (Figure 2e), by up to 65% in the mixed workload (Figure 2f), and by up to 70% in the update-heavy workload (Figure 2g).

The results show that avoiding indirection is crucial to performance. In particular, EEMARQ outperforms its competitors in the lookup-heavy workloads (Figures 2a and 2e), in which memory is never reclaimed. I.e., its range query mechanism, which completely avoids traversing separate version nodes, has a significant advantage under such workloads. It can also be seen when examining EEMARQ's competitors. While the vCAS-based tree (which avoids indirection) is EEMARQ's next best competitor, the vCAS-based skip list (that involves the traversal of designated version nodes) is the weakest among the skip list implementations. In addition, the Bundles technique, which employs such a level of indirection when range queries are executed, also performs worse than most competitors, under most workloads.

Under update-dominated workloads, EEMARQ is faster also thanks to its efficient memory reclamation method. While all other algorithms use EBR as their memory reclamation scheme, EEMARQ enjoys VBR's inherent locality and fast reclamation process. Moreover, EEMARQ avoids the original VBR's frequent accesses to the global epoch clock, as described in Section 3.3. Although using VBR forces rollbacks when accessing reclaimed nodes, our fast index mechanism allows a fast retry, which makes the impact of rollbacks small. Indeed, experiments with longer retire lists (i.e., fewer rollbacks) showed similar results. This is clearly shown in Figures 2d and 2h: EEMARQ outperforms its competitors when the query ranges are big (10% of the data-structure range). I.e., possible frequent rollbacks do not prevent EEMARQ from outperforming all other competitors.

5 Conclusion

We presented EEMARQ, a design for a lock-free data-structure that supports linearizable inserts, deletes, contains, and range queries. Our design starts from a linked-list, which is easier to use with MVCC for fast range queries. We add lock-free memory reclamation to obtain full lock-freedom. Finally, we facilitate an easy integration of a fast external index to speed up the execution, while still providing full linearizability and lock-freedom. As the external index does not require version maintenance, it can remain simple and fast. We implemented the design with a skip list and a binary search tree as two possible fast indexes, and evaluated their performance against state-of-the-art solutions. Evaluation shows that EEMARQ outperforms existing solutions across read-intensive and update-intensive

workloads, and for varying range query sizes. In addition, EEMARQ's memory footprint is relatively low, thanks to its tailored reclamation scheme, enabling the immediate reclamation of deleted objects. EEMARQ is the only technique that provides lock-freedom, as other existing methods use blocking memory reclamation schemes.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
2. Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. Constant time recovery in azure sql database. *Proceedings of the VLDB Endowment*, 12(12):2143–2154, 2019.
3. Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.
4. Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 336–343, 2008.
5. Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 299–308, 2013.
6. Daniel Bartholomew. *MariaDB cookbook*. Packt Publishing Ltd, 2014.
7. Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.
8. Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. *arXiv preprint arXiv:2108.02775*, 2021.
9. Philip A Bernstein and Nathan Goodman. Concurrency control algorithms for multiversion database systems. In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 209–215, 1982.
10. Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.
11. Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory mvcc systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.
12. Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.
13. Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.
14. Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 329–342, 2014.
15. Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
16. Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–10, 2017.
17. Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

18 David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

19 Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

20 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

21 Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

22 Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

23 Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and scalable multi-version software transactional memory. *ACM SIGPLAN Notices*, 46(8):179–188, 2011.

24 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

25 Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

26 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

27 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

28 Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

29 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

30 Nikolaos D Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

31 Idit Keidar and Dmitri Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 150–165. Springer, 2015.

32 Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

33 Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Long-lived transactions made less harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 495–510, 2020.

34 Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Rethink the scan in mvcc databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 938–950, 2021.

35 Priyanka Kumar, Sathya Peri, and K Vidyasankar. A timestamp based multi-version stm algorithm. In *International Conference on Distributed Computing and Networking*, pages 212–226. Springer, 2014.

36 Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in sap hana. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1307–1318, 2016.

37 Li Lu and Michael L Scott. Generic multiversion stm. In *International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.

38 Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183, 2015.

39 Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

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

41 AB MySQL. Mysql, 2001.

42 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

43 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022.

44 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 239–248. IEEE, 2015.

45 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks, and Games*, pages 358–369. Springer, 2015.

46 Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

47 Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 9(1):89–99, 1984.

48 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: Selective multi-versioning stm. In *International Symposium on Distributed Computing*, pages 125–140. Springer, 2011.

49 Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25, 2010.

50 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *International Symposium on Distributed Computing*, pages 224–238. Springer, 2013.

51 Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2, 2009.

52 Behandelt PostgreSQL. Postgresql. *Web resource: http://www. PostgreSQL. org/about*, 1996.

53 Aleksandar Prokopec. Snapqueue: lock-free queue with constant time snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 1–12, 2015.

54 Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

55 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

56 G Satyanarayana Reddy, Rallabandi Srinivasu, M Poorna Chander Rao, and Srikanth Reddy Rikkula. Data warehousing, data mining, olap and oltp technologies are essential elements to support decision-making process in industries. *International Journal on Computer Science and Engineering*, 2(9):2865–2873, 2010.

57 Gali Sheffi, Guy Golan-Gueta, and Erez Petrank. A scalable linearizable multi-index table. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 200–211. IEEE, 2018.

58 Gali Sheffi, Maurice Herlihy, and Erez Petrank. VBR: Version Based Reclamation. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.DISC.2021.35`.

59 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–445, 2021.

60 Gali Sheffi, Pedro Ramalhete, and Erez Petrank. Eemarq: Efficient lock-free range queries with memory reclamation. *arXiv preprint arXiv:2210.17086*, 2022.

61 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

62 Daniel Solomon and Adam Morrison. Efficiently reclaiming memory in concurrent search data structures while bounding wasted memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 191–204, 2021.

63 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. *ACM SIGPLAN Notices*, 51(6):682–696, 2016.

64 Alexander Thomasian and In Kyung Ryu. Performance analysis of two-phase locking. *IEEE Transactions on Software Engineering*, 17(5):386, 1991.

65 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

66 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.

67 Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–38, 2021.

68 Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

69 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

A Auxiliary Methods and Initialization

As described in Section 3.1 and Algorithm 1 and 2, the linked-list represents a map of key-value pairs. Each pair is represented by a node, consisting of five fields: the mutable *ts* field represents its associated timestamp, the immutable *key* and *value* fields represent the pair's key and value, respectively, the mutable *next* field holds a pointer to the node's successor in the list, and the immutable *prior* field points to the previous successor of its first predecessor in the list.

The list is initialized as follows: The global timestamps clock is initialized to 2, and the $\perp$ constant, representing an uninitialized timestamp, is set to 1. The list has a single entry point, which is a pointer to the *head* sentinel node. *head*'s key is the minimal key in the key range (denoted as $-\infty$). Its timestamp is set to the initial system timestamp (2 in our implementation) and its next pointer points to the *tail* sentinel node. *tail*'s key is

the maximal key in the key range (denoted as ∞), its timestamp is equal to *head*'s, and its next pointer points to null. Both *prior* fields point to null, as *head* has no predecessor, and *tail*'s predecessor (which is *head*) has no previous successor. After initialization, the list is considered as empty (the sentinel nodes do not represent map items).

Algorithm 4 Our Auxiliary Methods Implementation.

```
1: getTS()
2:     return globalTS.load()
3: fetchAddTS()
4:     return globalTS.fetch&add()
5: MARK_MASK := 0x1
6: FLAG_MASK := 0x2
7: AUX_MASK := 0x3
8: isMarked(ptr)
9:     if (ptr ∧ MARK_MASK) return true
10:    return false
11: isFlagged(ptr)
12:    if (ptr ∧ FLAG_MASK) return true
13:    return false
14: isMarkedOrFlagged(ptr)
15:    if (ptr ∧ AUX_MASK) return true
16:    return false
17: getRef(ptr)
18:    return ptr ∧ ¬ AUX_MASK
19: mark(node)
20:    ptr := node → next
21:    if (isMarkedOrFlagged(ptr)) return false
22:    markedPtr := ptr ∨ MARK_MASK
23:    return CAS(&node → next, ptr, markedPtr)
24: flag(node)
25:    ptr := node → next
26:    if (isMarkedOrFlagged(ptr)) return false
27:    flaggeddPtr := ptr ∨ FLAG_MASK
28:    return CAS(&node → next, ptr, flaggeddPtr)
```

The pseudo code for the global timestamps clock and node pointers access auxiliary methods appears in Algorithm 4. The *getTS()* method (lines 1-2) is used to atomically read the global timestamps clock, and the *fetchAddTS()* method (lines 3-4) is used to atomically update it.

We use the pointer's two least significant bits for encapsulating the mark and flag bits (see lines 5-7). The *isMarked()* (lines 8-10), *isFlagged()* (lines 11-13), and *isMarkedOrFlagged()* (lines 14-16) methods receive a pointer and return an answer using the relevant bit mask. The *getRef()* method (lines 17-18) receives a (potentially marked or flagged) pointer and returns the actual reference, ignoring the mark and flag bits. The *mark()* (lines 19-23) and *flag()* (lines 24-28) methods receive a node reference (the input pointer is assumed to be unmarked and unflagged), dereference it, and mark or flag the node's next pointer (respectively), assuming it is neither marked nor flagged.