Bundling Linked Data Structures for Linearizable Range Queries

Jacob Nelson-Slivon
Lehigh University, USA
jjn217@lehigh.edu

Ahmed Hassan
Lehigh University, USA
ahh319@lehigh.edu

Roberto Palmieri
Lehigh University, USA
palmieri@lehigh.edu

Abstract

We present bundled references, a new building block to provide linearizable range query operations for highly concurrent lock-based linked data structures. Bundled references allow range queries to traverse a path through the data structure that is consistent with the target atomic snapshot. We demonstrate our technique with three data structures: a linked list, skip list, and a binary search tree. Our evaluation reveals that in mixed workloads, our design can improve upon the state-of-the-art techniques by 1.2x-1.8x for a skip list and 1.3x-3.7x for a binary search tree. We also integrate our bundled data structure into the DBx1000 in-memory database, yielding up to 40% gain over the same competitors.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Information systems** → **Data structures**.

Keywords: Concurrent Data Structures, Range Queries, Fine-grain Synchronization

1 Introduction

Iterating over a collection of elements to return those that fall within a contiguous range (also known as a *range query* operation) is an essential feature for data repositories. In addition to database management systems, which historically deploy support for range queries (through predicate reads or writes), recent key-value stores (e.g., RocksDB [18] and others [17, 23, 30, 35, 45]) enrich their traditional APIs to include range query operations.

With the high-core-count era in full swing, providing high-performance range query operations that execute concurrently with modifications is challenging. On the one hand, ensuring strong correctness guarantees of range queries, such as linearizablity [27], requires that they observe a consistent snapshot of the collection regardless of any concurrent update that may take place. On the other hand, since range queries are naturally read-only operations, burdening them with synchronization steps to achieve strong correctness guarantees may significantly deteriorate their performance.

In this paper we propose *bundled references*[1], a new building block to design lock-based linearizable concurrent linked data structures optimized to scale up performance of range query operations executing concurrently with update operations. The core innovation behind bundled reference lies in adapting the design principle of Multi Version Concurrency Controls (MVCC) [20, 50] to lock-based linked data structures, and improving it by eliminating the overhead of determining the appropriate version to be returned. Bundled references achieve that by augmenting each link in a data structure with a record of its previous values, each of which is tagged with a timestamp reflecting the point in (logical) time when the operation that generated that link occurred. In other words, we associate timestamps to references connecting data structure elements instead of the nodes themselves.

The bundled reference building block enables the following characteristics of the data structure:

- Range query operations establish their snapshot just before reaching the first element in the range, which helps reduce interference from ongoing and subsequent update operations on the range;
- Data structure traversals, including those of contains and update operations, may proceed uninstrumented until reaching the desired element(s).
- Well-known memory reclamation techniques, such as EBR [21], can be easily integrated into the bundled references to reclaim data structure elements, which reduces the space overhead of bundling.

We demonstrate the efficacy of bundling by applying it to three widely-used lock-based ordered Set implementations, namely a linked list, a skip list, and a binary search tree (BST). While the linked list is a convenient data structure to illustrate the details of our design that favors range query operations, the skip list and the BST are high-performance data structures widely used in systems (such as database indexes) where predicate reads are predominant. In these new data structure we augment the existing links with bundled references to provide linearizable range queries.

[1]This work builds on an initial design appeared in [38].

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508412

In a nutshell, the history of a link between nodes (called a *bundle*) is consistently updated every time a successful modification to the data structure occurs, and its entries are labeled according to a global timestamp. Data structure traversals, including those of range query and contains operations, do not need to use bundles until they reach their target range or element. Then, range queries read the global timestamp and follow a path made of the latest links marked with a timestamp lower than (or equal to) the previously read timestamp. Contains operations also use bundles after the non-instrumented traversal to consistently read the target element, but they do not need to access the global timestamp.

Bundling follows the trend of providing abstractions and techniques to support range query operations on highly concurrent data structures [4, 33, 36, 43, 46]. Among them, the most relevant competitors include read-log-update (RLU)[36], the versioned CAS (vCAS) approach [49], and a solution based on epoch-based reclamation (EBR-RQ) [4]. We contrast their functionalities in the related work section and their performance in the evaluation section.

In summary, analyzing the performance results we found that bundling achieves a more consistent performance profile across different configurations than RLU and EBR-RQ, whose design choices can lead them to prefer specific workloads. Concerning mixed workloads, bundling offers up to 1.8x improvement over EBR-RQ, and up to 3.7x improvement over RLU. Compared with vCAS, bundling's performance prevails in all linked list experiments (reaching up to 2x improvement). Also, bundling outperforms vCAS in both skip list and BST for read-dominant cases, reaching up to 1.5x improvement. In write-dominated workloads, vCAS equals bundling in skip list, and outperforms it in BST at high thread count. We also integrate our bundled skip list and BST as indexes in the DBx1000 in-memory database and test them using the TPC-C benchmark. We find that bundling provides up to 1.4x better performance than the next best competitor at high thread count.

2 Related Work

Linearizable range queries. Existing work has focused on providing range queries through highly-specific data structure implementations [5, 6, 9, 10, 44, 47]. While recognizing their effectiveness, their tight dependency on the data structure characteristics makes them difficult to extend to other structures, even if manually. The literature is also rich with effective concurrent data structure designs that lack range query support and cannot leverage the above data structure specific solutions to perform range queries. This motivates generalized solutions, which achieve linearizable range queries by applying the same technique to various data structures [4, 36, 38, 43].

Read-log-update (RLU) [36] is a technique in which writing threads keep a local log of updated objects, along with the logical timestamp when the update takes effect. When no reader requires the original version, the log is committed. It extends read-copy-update (RCU) [37] to support multiple object updates. Range queries using RLU are linearized at the beginning of their execution, after reading a global timestamp and fixing their view of the data structure. However, in RLU, updates block while there are ongoing RLU protected operations, as it only commits its changes after guaranteeing no operation will access the old version. Bundling minimizes write overhead because new entries are added while deferring the removal of outdated ones.

Snapcollector [43] logs changes made to the data structure during an operation's lifetime so that concurrent updates are observed. A range query first announces its intention to snapshot the data structure by posting a reference to an object responsible for collecting updates. It traverses as it would in a sequential setting, then checks a report of concurrent changes it may have missed. The primary difference with respect to RLU is that range queries are linearized at the end of the operation, after disabling further reports.

Although the construction of Snapcollector is wait-free, this method may lead a range query to observe reports of changes that were already witnessed during its traversal. Creating and announcing reports penalizes operation performance; not to mention the memory overhead required to maintain these reports. We experimentally verify that the cost of these characteristics is high. With our bundling approach, a range query visits nodes in the range only once to produce its view of the data structure and is linearized when it reads the global timestamp right before entering the range. An extension of Snapcollector enables snapshotting only a range of the data structure instead of all elements [12]. However, this approach continues to suffer many of the same pitfalls as the original design. In addition to these, concurrent range queries with overlapping key ranges are disallowed.

Arbel-Raviv and Brown [4] build upon epoch-based memory reclamation (EBR) to provide linearizable range queries. In this method, range query traversals leverage a global timestamp to determine if nodes, annotated with a timestamp by update operations, belong in their snapshot. In order to preserve linearizability, remove operations announce their intention to delete a node before physically removing them and adding them to the list of to-be-deleted nodes, or limbo list. Range queries scan the data structure, the announced deletions, and limbo list to determine which nodes to include in their view, potentially resulting in a situation where nodes are observed multiple times. The design also prioritizes update-mostly workloads, since range queries' timestamp updates conflict. Our bundling approach enhances performance of range queries by allowing them to traverse only the nodes in the range without needing to validate its snapshot.

Bundling, like the competitors included thus far, focuses solely on supporting linearizable range queries. Other techniques target linearizable bulk update operations, which mutate several data structure elements atomically [46]. These methodologies are not optimized specifically for range queries and therefore are not included in our comparisons.

MVCC. Multi-version concurrency control (MVCC), widely used in database management systems, relies on timestamps to coordinate concurrent accesses. Many different implementations exist [7, 8, 32, 34, 40]; all rely on a multiversioned data repository where each shared object stores a list of versions, and each version is tagged with a creation timestamp. Transactions then read the versions of objects that are consistent with their execution.

The de facto standard for version storage in MVCC systems is to maintain a list of versions (*version list*) for each object that is probed during a read [50], with innovations targeting this particular aspect. One example, Cicada [34], uses a similar idea of installing `PENDING` version for every written object as the first step in its validation phase. The main difference in bundling is that pending entries only exist for a short duration surrounding the linearization point. X-Engine [29] and KiWi [6] use version lists for objects, rather than links like in bundling, which cause them to visit nodes not belonging to their snapshot. Multi-versioned Software Transactional Memory [20] applies MVCC on in-memory transactions. However, it inherently requires update transactions to instrument all accesses (and validate them) while bundling exploits data structure specific semantics to avoid false conflicts and improve performance.

Persistent data structures. The bundled reference abstraction is similar in spirit to the concept of *fat nodes* in *persistent data structures* [16]. In principle, persistent data structures are those which maintain all previous versions of the data structure. Bundling aims at providing efficient linearizable range queries in highly concurrent workloads, while persistent data structures are commonly used in functional programming languages to maintain theoretical requirements regarding object immutablility [28, 41] and algorithms requiring reference to previous state [13, 48].

vCAS. vCAS [49] implements range queries for lock-free data structures while retaining the non-blocking guarantee of data structures through their so called vCAS objects. Like bundling, each vCAS object maintains a versioned list of values but for CAS objects. Operations can query the state of this vCAS object at a certain snapshot using a *versioned read* interface. Since CAS objects in lock-free data structures are often the links between data structure nodes, vCAS enables traversals to observe links at a given snapshot, which can then be leveraged to implement range queries.

Applying the vCAS technique to lock-free data structures is straightforward since it only requires replacing CAS objects with vCAS objects. On the other hand, the requirements when applying the vCAS technique to lock-based data structures are unclear since there is no simple rule to determine which components should be replaced by vCAS objects. In principle, there is a mismatch between the target abstraction of vCAS and its application to lock-based data structures. In contrast, bundling is designed specifically to represent references and more precisely captures the meaning of versioning in the context of linked data structures. In our evaluation, we demonstrate vCAS can be ported to lock-based data structures by using vCAS objects in place of both pointer(s) and metadata (e.g., flags for logical deletion), but not locks. Despite that, it is still not clear whether vCAS is broadly applicable to lock-based data structures.

The replacement strategy required for vCAS illuminates another important difference. Bundling augments references in the data structure and keeps the original links unchanged, while vCAS replaces them. Because of this, bundling provides flexibility in how operations traverse the data structure, either using bundles or the original links, which is a fundamental enabler for our optimized traversals. For vCAS, the result of tightly coupling the versioned CAS object to the underlying field is that all operations must pass through the vCAS API, which leads to additional costs for vCAS when traversals are long (see Section 6). Whether or not our optimization can be applied in the case of vCAS is an open question, but it would likely include non-trivial changes to the current abstraction. Similarly, making bundling lock-free is not an insignificant challenge and potentially requires replacing locks with multi-CAS operations to atomically update each link along with its augmented bundle. Unifying the two is an intriguing future direction.

Optimized Traversals. Among other publications on optimized traversals [2, 3, 14, 15, 24–26], two recent works are worth mentioning. Like their predecessors, both capture the notion of data structure traversals and their logical isolation from an operation's "real" work. In the first, Freidman et al. [22] define a Traversal Data Structure as a lock-free data structure satisfying specific properties that allow operations to be split into two phases. These phases are the Traversal phase and the Critical phase, of which only the latter requires instrumentation for making a data structure consistently persistent in non-volatile memory. This resembles our notions of pre-range and enter-range, but is applied specifically in the context of lock-free data structures and non-volatile memory. Similarly, Feldman et al. [19] construct a proof model that is centered around proving the correctness of optimistic traversals. The model combines single-step compatibility with the forepassed condition to demonstrate that, for a given operation, a node has been reachable at some point during the traversal. An interesting extension would be to understand the applicability of this model when range queries are added to the data structure APIs. Due to versioning, the current conditions may not fully capture the requirements to prove that range queries are linearizable.

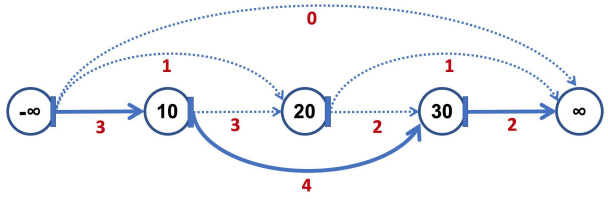

Figure 1. An example of using bundled references in a linked list. The path made of solid lines represents the state of the linked list after all update operations take place. Edges are labeled with their respective timestamps.

3 The Bundle Building Block

The principal idea behind *bundling* is the maintenance of a historical record of physical changes to the data structure so that range queries can traverse a consistent snapshot. As detailed below, the idiosyncrasy of bundling is that this historical record stores *links* between data structure elements that are used by range query operations to rebuild the exact composition of a range at a given (logical) time.

Update operations are totally ordered using a global timestamp, named `globalTs`, which is incremented every time a modification to the data structure takes place .

Every link in the data structure is backed by a *bundle*, implemented as a list of *bundle entries* (Listing 1). Each bundle entry logs the value of the link and the value of `globalTs` at the time the link was added to the bundle. Whenever an update operation reaches its linearization point, meaning when it is guaranteed to complete, it prepends a bundle entry consisting of the newest value of the link and the value of the global timestamp. Because of this, the head of the bundle always reflects the link's latest value.

It is worth noting that other solutions [4, 49] also use a global timestamp but assign the responsibility of incrementing it to range query operations. We experimentally verify (see our evaluation section) that incrementing the global timestamp upon completion of range query or update operation is irrelevant in terms of performance for bundling. Our decision of letting update operations increment `globalTs` simplifies traversals with bundles since it allows each bundle entry to be tagged with a unique timestamp.

Since each link's history is preserved through the bundles, range queries simply need to reach the range, read the global timestamp, and traverse range of the linked data structure using the newest values no larger than the observed global timestamp. This design is inherently advantageous when pruning bundle entries. In fact, a bundle entry may be removed (or recycled) if an entry is no longer the newest in the bundle and no range query needs it.

Figure 1 shows an example on how bundles are deployed in a linked list. As shown in the figure, the next pointer of each node is replaced by a bundle object that encapsulates the history of this next pointer. The figure shows the state of the linked list and its bundles after the following sequence of operations (on an empty linked list): `insert(20)`, `insert(30)`, `insert(10)`, `remove(20)`.

To understand how this state is generated, we assume that the list is initialized with a single bundle reference whose timestamp is "0" (the initial value of `globalTs`), which connects its head and tail sentinel nodes. Inserting `20` does not replace this reference. Instead, it creates a new entry in the head's bundle with timestamp "1" pointing to the newly inserted node as well as an entry with the same timestamp in this new node pointing to the tail node. Similarly, inserting `30` and `10` adds new bundle entries with timestamps "2" and "3", respectively. The last operation that removes `20` also does not replace any reference. Instead, it creates a new bundle entry in `10`'s bundle (with timestamp "4") that points to `30`, which reflects physically deleting `20` by updating the predecessor's next pointer.

Now assume that different range queries start at different times concurrently with those update operations. For clarity, we name a range query R_i if it reads the value i of `globalTs`, and for simplicity we assume its range matches the entire key range. Regardless of the times at which the different nodes are traversed, each range query is always able to traverse the proper snapshot of the list that reflects the time it started. For example, R_0 will skip any links in the range added after it started because all of them have timestamp greater than "0". Also, R_3 will observe `20` even if it reaches `10` after `20` is deleted. This is because in that case it will use the bundle entry whose timestamp is "3", which points to `20`.

The solid lines in the figure represent the most recent state of the linked list. Different insights can be inferred from this solid path. First, the references in this path are those with the largest timestamp in each bundle. This guarantees that any operation (including range queries) that starts after this steady state observes the most recent abstract state of the list. Second, once the reference with timestamp "4" is created, `20` becomes no longer reachable by any operation that will start later, because this operation will observe a timestamp greater than (or equal to) "4". Thus, unreachable elements can be concurrently reclaimed.

3.1 Bundle Structure

Bundling a data structure entails augmenting its links with a bundle to produce a bundled reference. Importantly, bundling does not replace links. For example, in Listing 2, each node in a linked list has a `next` field and an associated `bundle` field. A design invariant that we maintain to support this behavior is that the value of a link is equal to the newest entry in its corresponding bundle. Later, we describe how duplicating the newest link can be leveraged to improve performance when bundles are not needed to be traversed.

As shown in Listing 1, a *bundle* is a collection of entries. Each bundle entry must contain a pointer value, `ptr`, and the timestamp associated with this value, `ts`.

Bundles are accessed using a `DereferenceBundle` API, which returns the value of the bundled reference that *satisfies* a given timestamp. We say an entry *satisfies* timestamp ts if it is the newest entry in the bundle when the global timestamp equaled ts, which is the bundle entry with the largest timestamp that is less than or equal to ts.

Listing 1. Bundle.

```
1 timestamp_t globalTs;
2 class BundleEntry {
3     Node * ptr;
4     timestamp_t ts;
5     BundleEntry * next;
6 }
7 class Bundle {
8     BundleEntry * head;
9 }
```

Listing 2. Linked List Node.

```
1 class Node {
2     key_t key;
3     val_t val;
4     lock_t lock;
5     bool deleted;
6     // Bundled reference.
7     Node * next;
8     Bundle bundle;
9 }
```

In order to perform a traversal, each bundled data structure must implement two functions to determine the next node in the structure. The first is `GetNext`, which uses the original links; the second is `GetNextFromBundle`, which uses `DereferenceBundle` internally. More details about these two functions will follow.

We implement our bundles as a list of entries, sorted by their timestamp, but note that its implementation is orthogonal to the bundling framework as long as it supports the `DereferenceBundle` API. So far we have assumed that a bundle may hold infinite entries. We address memory reclamation in Section 5.

3.2 Bundles and Update Operations

Generally speaking, an update operation has two phases. The operation first traverses the data structure to reach the desired location where the operation should take place, then performs the necessary changes. In bundling, only places where pointers are changed must be supplemented with operations on bundles; the operation's traversal procedure remains unaltered.

The goal for updates is to reflect changes, observable at the operation's completion, so that range queries can see a consistent view of the data structure. This is performed by book-ending an update's original critical section, which is defined at some point after all necessary locks are held, with `PrepareBundles` and `FinalizeBundles`, resulting in a three step process:

1. `PrepareBundles` (Algorithm 1) inserts in the bundles of updated nodes a new entry in a *pending* state (Line 7). Then, `globalTs` is atomically fetched and incremented and its new value is returned (Line 9).
2. Next, the operation's original critical section is executed, making the update visible to other operations.
3. Lastly, `FinalizeBundles` (Algorithm 2) sets the linearization timestamp of the pending entries by annotating them with the newly incremented timestamp, before releasing locks.

Algorithm 1: PrepareBundles

```
   Input: The set of bundles to update, bundles; the bundles' new values, ptrs
1  begin
2      for (b,p) in (bundles, ptrs) do
3          newEntry ← new BundleEntry(ptr)    // Pending entry
4          while true do
5              newEntry.next ← expected ← bundle.head
6              while bundle.head.ts = PENDING_TS do {}
7              if AtomicCompareAndSwap(&bundle.head, expected,
                 newEntry) then break
8
9      return AtomicFetchAndAdd(&globalTs, 1) + 1
```

Algorithm 2: FinalizeBundles

```
   Input: A bundle to prepare, bundle; the linearization timestamp, ts
1  begin
2      for b in bundles do
3          b.ts ← ts
```

The `PrepareBundles` step is crucial in the correctness of bundling. As detailed in Section 3.3, when a read operation traverses a node using a bundle, it waits until any pending entry is finalized to guarantee that it does not miss a concurrent update that should be included in its linearizable snapshot. Additionally, concurrent updates attempting to add their own pending entry must block until the ongoing update is finalized (Algorithm 1, Line 6). This is done so that concurrent updates to the same bundle are properly ordered by timestamp. It is also possible to address this problem by assuming that all nodes whose bundles will change are locked. We choose not to do so to make our design independent of data structure specific optimizations (see Section 4.1). Also note that incrementing `globalTs` must occur before any newly inserted nodes are reachable otherwise the linearizability of range queries may be compromised.

3.3 Bundles and Range Queries

We consider range queries to have the following three phases:

1. `pre-range`: the traversal of data structure nodes until the start of the range.
2. `enter-range`: the traversal from a node just outside the range to the first node falling within the range.
3. `collect-range`: the traversal of the range, during which the result set is constructed.

Algorithm 3 outlines a range query for a bundled data structure, reflecting these steps.

Pre-range. The `pre-range` traversal is performed to locate the node immediately preceding the range (Lines 4-8). Note that `GetNext` is a data structure specific procedure that returns the next node in a traversal toward the range *without using bundles*. For example, in the case of a binary search tree this would be the appropriate child link, not its corresponding bundle. Using an uninstrumented traversal reduces overhead and improves performance by avoiding the costs

associated with accessing bundles, as well as blocking on pending entries. Once the predecessor to the range is located (Line 8), the `enter-range` phase is executed.

Enter-range. In sequential algorithms, the `enter-range` phase is trivial, as it is simply a dereference. However, for bundling it represents a critical moment that must be treated carefully since it is here that we switch to traversing a linearizable snapshot through bundles.

Algorithm 3: RangeQuery

```
Input: Inclusive range to query, [low, high]; sentinel node and entry point
       to the data structure, root
Output: Set of nodes in the given range
1  begin
2      while true do
3          pred ← root                                          // Pre-range
4          curr ← pred.GetNext(low, high)
5          while curr ≠ nullptr do
6              pred ← curr
7              curr ← curr.GetNext(low, high)
8              if curr.key is in the range then break
9          ts ← globalTs                                        // Enter-range
10         curr ← pred.GetNextFromBundle(low, high, ts)
11         while curr is not in the range do
12             curr ← curr.GetNextFromBundle(low, high, ts)
13         return CollectRange(curr, low, high, ts)   // Collect-range
```

Algorithm 4: DereferenceBundle

```
Input: A bundle to dereference, bundle; the linearization timestamp, ts
1  begin
2      entry ← bundle.head
3      while entry.ts = PENDING_TS do {}   // Maybe wait on update
4      while entry.ts > ts do entry ← entry.next
5      return entry.ptr
```

To do so, a range query first fixes its linearizable snapshot of the data structure by reading `globalTs` into a local variable *ts* (Line 9). This ensures that all future updates that increment the timestamp after the range query executes Line 9 are ignored. Next, in Lines 10-12, the `enter-range` phase executes a loop of `GetNextFromBundle` to reach the first node in the range. This loop is needed to handle the cases in which nodes are inserted/deleted after the `pre-range` phase ends and before `globalTs` is read.

Internally, `GetNextFromBundle` uses `DereferenceBundle` (Algorithm 4) to follow links. Given a bundle and a timestamp *ts*, the function `DereferenceBundle` works as follows. The call first reads the head of the bundle and waits until the entry is not pending. Next, it scans the bundle for the entry that satisfies *ts* (i.e., the first entry whose timestamp is less than or equal to *ts*), returning the address of the node the entry points to. Since it is written only by a single in-progress update, pending timestamps provide a low-contention linearization mechanism.

As we mentioned before, blocking while the most recent entry is pending is a necessary step to ensure that both range query and contains operations are correctly linearized along with update operations. Because of this step, read operations observe updates only after the changes have become visible and the bundles are finalized.

It is worth noting that the transition from the `pre-range` phase to the `enter-range` phase is always consistent. After the `pre-range` phase, a range query holds a reference to the predecessor of its range that was traversed to using normal links. This node either has a pending bundle entry or a finalized one, since updates only make a node reachable after preparing the bundles and incrementing the global timestamp. Because a range query reads `globalTs` only after reaching the node, it is guaranteed to find an entry satisfying its timestamp by calling `GetNextFromBundle`, even if it must wait for the concurrent update to finalize it first.

Collect-range. After the range is reached in the `enter-range` phase, the `collect-range` phase is performed by calling the data structure specific function `CollectRange`. This function explores the data structure using bundles and the timestamp from Line 9 to generate the result set, returning the empty set if no nodes exist in the range. As was the case for `GetNextFromBundle`, `CollectRange` uses `DereferenceBundle` to traverse the data structure and collect the result set.

A critical invariant that guarantees the correctness of bundling is that the `collect-range` phase always finds its required path through the bundles. This is trivial if both deleted nodes and outdated bundle entries are never reclaimed because the entire history of each reference is recorded. Section 5 shows how to preserve this invariant when they are reclaimed.

3.4 Bundles and Contains Operations

Contains operations must observe the same history as range queries in order to guarantee linearizability. The correctness anomaly that would arise if contains operations were to execute entirely without bundles is equivalent to the one reported in [1, 31, 42], in which two update operations operating on different elements are observed in different order by concurrent reads.

Our solution is to treat the contains operation as a single-key range query. There is no additional overhead during the `pre-range` phase because the traversal is uninstrumented. Also, the `collect-range` phase will return the correct result since both the low and high keys are the same. Like range queries, the `enter-range` phase preserves correctness because it enforces contains to be blocked if they reach a pending entry that is not finalized by a concurrent update.

One drawback of treating contains as a special case of range query is that reading `globalTs` to set the linearization snapshot of a contains can increase contention on `globalTs`, significantly. To eliminate this overhead, we observe that a contains operation *does not* need to fix its linearization point by reading `globalTs`. Its linearization point can be delayed until dereferencing the last bundle in its traversal. To implement that, contains calls `DereferenceBundle` with

an infinitely large timestamp at Lines 10 and 12 instead of reading `globalTs` at Line 9. In Section 6, we assess the performance impact of this optimization.

3.5 Correctness

The correctness proof can be found in the companion technical report [39], but we include a brief intuition here.

The linearization point of an update operation is defined as the moment it increments `globalTs`; on the other hand, range queries are linearized when they read `globalTs` to set their snapshot. Considering that bundle entries are initialized in a pending state, this guarantees that range queries will observe all updates linearized before they read `globalTs`.

The linearization point of contains is less trivial. Briefly, it is dictated by the linearization point of the update operation that inserted the bundle entry corresponding to the last dereference made in the `enter-range` phase. If this update increments `globalTS` concurrently with the contains, the contains is linearized immediately after the increment; otherwise, the contains is linearized at its start.

3.6 Applying Bundling

Designing a bundled data structure is not meant to be an automated process; it requires selecting a concurrent linked data structure and engineering the deployment of the bundle abstraction. That said, all the bundled data structures in this paper share common design principles. First, they are all linked data structures and have a sentinel node to start from.

Second, they are lock-based and all locks are acquired before modifications take place, and released after. This is better known as two-phase locking (2PL) [8]. Because the data structures follow 2PL, all bundle entries are finalized atomically for a given update operation, which makes reasoning about the correctness of the resulting bundled data structures easier. In turn, this allows us to illustrate our technique with varying degrees of complexity (e.g., multiple children in the case of the binary search tree). Note that this does not preclude other locking strategies, but care should be taken to ensure that bundles are updated appropriately.

Finally, all of the data structures we apply bundling to implement non-blocking traversals that avoid checking locks. Although bundling makes reads blocking, it retains the optimized traversals during the `pre-range` phase and only synchronizes with updates (via the bundles) during the phases `enter-range` and `collect-range`. This is possible because we augment the existing links instead of replacing them.

4 Bundled Data Structures

We now describe how to apply bundling to the highly-concurrent lazy sorted linked list [25]. We also apply bundling to more practical data structures, such as the lazy skip list [26] and the Citrus unbalanced binary search tree [3].

4.1 Bundled Linked List

Listing 2 provides a full definition of member variables of the linked list node. Note that the only additional field compared to the original algorithm is the bundle.

Algorithm 5: Insert operation of Bundled Linked List

```
   Input: key, val
1  begin
2      while true do
3          pred, curr ← Traverse(key)
4          Lock(pred)
5          if ValidateLinks(pred, curr) then
6              if curr.key == key then
7                  return false
8              newNode ← new Node(key, val, curr)
9              bundles ← (newNode.bundle, pred.bundle)
10             ptrs ← (curr, newNode)
11             ts ←PrepareBundles(bundles, ptrs)
12             pred.next ← newNode
13             FinalizeBundles(bundles, ts)
14             Unlock(pred)
15             return true
16         Unlock(pred)
```

Insert Operation. Initially, insert operations (Algorithm 5) traverse the data structure to determine where the new node must be added. After locking the predecessor, both current and predecessor nodes are validated by checking that they are not logically deleted and that no node was inserted between them. If validation succeeds and the key does not already exist, then a new node with its next pointer set to the successor is created. Otherwise, the nodes are unlocked and the operation restarts.

The above follow the same procedure that the original linked list (without bundling) would use. To illustrate how bundling applies, we highlight the bundling-specific lines blue. In case of successful validation, the next thing is to perform the three steps described in Section 3.2 to linearize an update operation in a bundled data structure: installing pending bundle entries and incrementing the global timestamp (Line 11), performing the original critical section (Line 12), and finalizing the bundles (Line 13). For an insertion, the bundles of the newly added node and its predecessor must be modified to reflect their new values and the timestamp of the operation. Then, locks are released.

Note that in Algorithm 5 we employ an optimization where only the predecessor is locked by insert operations (Line 4). In [25], it is proven that this optimization preserves linearizability. However, it also reveals a subtle but important corner case that motivates the need for updates to wait for pending bundles to be finalized (Line 6 of Algorithm 1). Because the current node is not locked, it is possible that a concurrent update operation successfully locks the new node after it is reachable and before its bundles are finalized by the inserting operation. This nefarious case is protected by first waiting for the ongoing insertion to finish to ensure the bundle remains ordered (see Section 3.2).

In Algorithm 5, the `PrepareBundles` step is invoked only after the locks are held. Although this nesting synchronization might look redundant, it is important to guarantee safety in the aforementioned case. Fusing the two is not out of the question, but would require careful consideration regarding both updates and range queries. Informally, a range query could first read the timestamp then check the lock, waiting until it is released by the update, akin to the pending entry in the original algorithm. We mention this point to illustrate that bundling is adaptable, but we do not include a design since it tightly couples our approach to the underlying data structure and lengthens the critical section during which a read may have to wait.

Remove operations. Remove operations follow a similar pattern by first traversing to the appropriate location, locking the nodes of interest, validating them (restarting if validation fails), marking the target node as deleted, unlinking it if its key matches the target key, then finally unlocking the nodes and returning. Here, the original critical section includes the logical deletion and the unlinking of the deleted node, which is therefore surrounded with the required bundle maintenance by performing the operation via calls to `PrepareBundles` and `FinalizeBundles`.

Importantly, range query operations do not need to check if a node is logically deleted because they are linearized upon reading the global timestamp and all bundle entries point to nodes that existed in the data structure at the corresponding timestamp. The removal of a node is observed through the finalization of the predecessor's bundle if their linearization point occurs after the remove operation increments `globalTs`. If the range query's linearization point falls before the remove, then the newly inserted entry in the predecessor's bundle will not satisfy the observed `globalTs` value and an older entry will be traversed instead.

Similarly, contains do not check for logical deletions because they are linearized according to the last entry traversed during the `enter-range` phase and will be linearized after a node is physically unlinked, or before it is logically deleted.

The removed node's bundle is also updated and points to the head of the list. This is to protect against a rare case resulting from the `pre-range` phase of a concurrent range query returning the removed node. In this case, if the range query reads `globalTs` (i.e., is linearized) after a sequence of deletions of all keys starting from this node through some nodes in the range, the range query will mistakenly include the deleted nodes. Adding a bundle entry in the removed node solves this problem by redirecting the range query to the head node and allowing the `enter-range` phase to traverse from the beginning using the bundle entries matching the `globalTs` value it reads. As shown in Section 6, this scenario rarely occurs and has no impact on performance.

Read operations. We now analyze the functions required by contains and range queries: `GetNext`, `GetNextFromBundle`, and `CollectRange`. To do so, we describe the execution of a range query and then discuss the changes necessary to support contains operations.

When executing a range query on the linked list, the `pre-range` phase consists of scanning from the head until the node immediately preceding the range (or target key) is found using `GetNext`, which simply returns `next`. The operation then traverses using the bundles until finding the first node in the range using `GetNextFromBundle`. This function uses a single call to `DereferenceBundle` to return the node that satisfies `ts` in the bundle of the current node. While the current node *curr* is in the range, `CollectRange` adds *curr* to the result set then updates it to point to the next node in the snapshot calling `DereferenceBundle` on `bundle`. Because of the construction of a linked list, the operation terminates once the range is exceeded. Together, these functions are used by Algorithm 3 to perform linearizable read operations.

Recall from Section 3.4 that contains operations follow the same general strategy as range queries, but use a infinitely large timestamp to traverse the newest bundle entry at each node. Hence, the value stored at Line 9 of Algorithm 3 can be replaced with the maximum timestamp.

4.2 Bundled Skiplist

The second data structure where we apply bundling is the lazy skip list [26], whose design is similar to the bundled linked list. In the following, we highlight the differences between the two designs.

The first difference is that skip list consists of a bottom *data layer* where data resides, and a set of *index layers* to accelerate traversal. Hence, given a target key, a regular traversal returns a set of (`pred`, `curr`) pairs at both the index and data layers. If the target key exists in the data structure, then it also returns the highest level (`levelFound`) at which the node was found. A naive approach to bundling this skip list would be to replace all links with bundled references, including the index layers. However, recall that for read operations the use of bundles is delayed to improve performance. Therefore it is sufficient to only bundle references at the data layer, leaving the index layers as is.

Second, because update operations manipulate multiple links per node, they are linearized using logical flags. Specifically, insert operations set a `fullyLinked` flag in the new node after the links of all its `pred` nodes are updated to point to it. Setting this flag is the linearization point of insert operations in the original lazy skip list. As required by bundling, the original critical section that sets the predecessors' links and marks the node as logically inserted is book-ended by the preparation (Algorithm 1) and finalization (Algorithm 2) of the bundles for the predecessor and the new node, similarly to the bundled linked list.

Similar to inserts use of `fullyLinked`, remove operations are linearized in the original lazy skip list by a logical deletion, handled in the bundled skip list as follows. First, the

bundles of the predecessor and the removed node are prepared using Algorithm 1. Similar to the lazy list, the removed node's bundle will point to the head of the list. Next, the critical section marks the node as logically deleted then updates the references of the predecessor nodes in the index and data layers, to physically unlink the node. Finally, the bundles are finalized with Algorithm 2, allowing read operations to observe the change.

Finally, to support linearizable read operations, the skip list defines the required functions as follows. `GetNext` leverages the index layers to find the node whose next node at the data layer is greater than or equal to the target key by examining. Similar to the bundled linked list, `GetNextFromBundle` returns the bundle entry satisfying *ts*, but this will only traverse the data layer. `CollectRange` then scans the data layer, using bundles, to collects the range query's result set.

4.3 Bundled Binary Search Tree

For our bundled tree, we reference the Citrus unbalanced binary search tree [3], which leverages RCU and lazy fine-grained locking to synchronize update operations while supporting optimistic traversals. We modify it by replacing each child link of the search tree with a bundled reference.

Citrus implements a traversal enclosed in a critical section protected by RCU's read lock. This protects concurrent updates from overwriting nodes required by the traversal. After the traversal, if the current node matches the target, the operation returns a reference to it (named `curr`), a reference to its parent (named `pred`) and the direction of the child from `pred`. Otherwise, the node is not found and the return value of `curr` is null.

Because the Citrus tree is unbalanced, insertions are straightforward and always insert a leaf node. Otherwise adhering to the original tree algorithm, insertions are linearized by first preparing the bundle of the new node and of `pred`, corresponding to the child updated and incrementing the global timestamp with `PrepareBundles`, then setting the appropriate child, and finalizing the bundles by calling `FinalizeBundles`. Lastly, the insert operation unlocks `pred` and returns.

The more interesting case is a remove operation, which should address three potential situations, assuming that the target node `curr` is found and will be removed. In the first case, `curr` has no children. To remove `curr` the child of `pred` that pointed to `curr` is updated along with its bundle. In the second case, `curr` has exactly one child. In this scenario, the only child of `curr` replaces `curr` as the child of `pred`. Again, the bundle corresponding to `pred`'s child is also updated accordingly. The last, and more subtle, case is when `curr` has two children, in which we must replace the removed node with its successor (the left-most node in its right subtree). In all cases, both of the removed node's bundles are updated to point to the root to avoid the divergent path scenario described in Section 4.1.

In this last case, both `curr`'s successor and its parent are locked. Then, following RCU's methodology, a copy of the successor node is created and initialized in a locked state with its children set to `curr`'s children. The effect of this behavior is that possibly six bundles must be modified during `PrepareBundles` and `FinalizeBundles` to reflect the new physical state after the operation takes effect. The required changes are: 1) `pred`'s left or right bundle is modified with a reference to the copy of `curr`'s successor, 2) both bundles in the copy of `curr`'s successor must point to the respective children of `curr`, and 3) both of `curr`'s bundles must lead to the root of the tree to avoid the divergent path scenario described in Section 4.1. Optionally, if the parent of `curr`'s successor is not `curr` itself, then this node's bundle is updated to point to the successor's right-hand branch, since the successor is being moved. In all cases, the remove operation is linearized at the moment the child in `pred` is changed, making the update visible.

Read operations differ slightly from the bundled linked list and skip list implementations. For trees, unlike lists, the node found during the `pre-range` phase is not necessarily a node whose key is lower than the lower bound of the range. Instead, it is the first node discovered through a traversal whose child is in the range. Its child is the root of the subtree that includes all nodes belonging to the range. More concretely, `GetNext` examines the current node and either follow the left or right child depending on whether the node is greater than or less than the range, respectively. Similar to before, the node reached by the optimistic traversal of the `pre-range` phase may not be the correct entry point to the range, and subsequent `enter-range` phase using bundles is needed. `GetNextFromBundle` parallels `GetNext`, but uses bundles instead of child pointers.

As in [4], `CollectRange` follows a depth-first traversal using bundles. A stack is used to keep the set of to-be-visited-nodes to help traverse the subtree rooted at the node returned by the `enter-range` phase. `CollectRange` initializes the stack with the first node reached in the range. It then iteratively pops a node from the stack and checks whether its key is lower than, within, or greater than the range. Next, it adds the node's corresponding children to the stack according to this check. If the node is within the range it adds the node to the result set. Finally, it pops the next node from the stack and performs the above procedure again. Note that, as previously discussed, contains will follow the same general pattern as range queries but will use an infinitely large timestamp to traverse the newest entry at each link.

5 Memory Reclamation

We rely on epoch-base memory reclamation (EBR) [21] to cleanup physically removed nodes and no longer needed bundle entries because, as already assessed by [4], quiescent state memory reclamation [37] (a generalized form of EBR)

mirrors the need for a range query to observe a snapshot of the data structure. To avoid overrunning memory with bundles, we employ a background cleanup mechanism to recycle outdated bundle entries. This is accomplished by tracking the oldest active range query and only retiring bundle entries that are no longer needed by any active operation.

EBR Overview EBR guarantees that unreachable objects are freed by maintaining a collection of references to recently retired objects and an epoch counter to identify. It operates under the assumption that object are not accessed outside of the scope of an operation (i.e., during quiescence). EBR monitors the epoch observed by each thread and the objects retired during each epoch. The epoch is only incremented after all active threads have announced that they have observed the current epoch value. When a new epoch is started, any objects retired two epochs prior can be safely freed. We use a variant of EBR, called DEBRA [11], that stores per-thread limbo lists which also reduces contention on shared resources by recording removed nodes locally for each thread.

Freeing Data Structure Nodes. EBR guarantees that no node is freed while concurrent range queries (as well as any concurrent primitive operation) may access it; and, bundling guarantees that no range query that starts after physically removing a node will traverse to this node. As an example, consider the two following operations: *i*) a range query, R, whose range includes node x; and *ii*) a removal operation, U_t, which is linearized at time t and removes x. If R is concurrent with U_t, then EBR will guarantee that x is not freed since R was not in a quiescent state and a grace period has not passed. In this case, R may safely traverse to x based on its observed timestamp, without concern that the node may be freed. On the other hand, if R starts after U_t, then trivially x will never be referenced by R and is safe to be reclaimed since R observed a timestamp greater than or equal to t.

Freeing Bundle Entries. Bundle entries are reclaimed in two cases. The first, trivial, case is that bundle entries are reclaimed when a node is reclaimed. The second case is more subtle. After a node is freed, there may still exist references to it (in other nodes' bundles) that are no longer necessary and should be freed. Bundle entries that have a timestamp older than the oldest active range query can be reclaimed only if there also exists a more recent bundle entry that satisfies the oldest range query. This cleanup process may be performed during operations themselves or, as we implement, delegated to a background thread.

To keep track of active range queries, we augment the global metadata with `activeRqTsArray`, which is an array of timestamps that maintains their respective linearization timestamp. During cleanup, this array is scanned and the oldest timestamp is used to retire outdated bundle entries. Once a node is retired, EBR becomes responsible for reclaiming its memory in the proper epoch.

Reading the global timestamp and setting the corresponding slot in `activeRqTsArray` must happen atomically to ensure that a snapshot of the array does not miss a range query that has read the global timestamp but not yet announced its value. This is achieved by first setting the slot to a pending state, similar to they way we protect bundle entries, which blocks the cleanup procedure until the range query announces its linearization timestamp. Second, the cleanup thread has to be protected by EBR as well.

Since contains operations use the most recent bundle entry at each node, they do not need to announce themselves in `activeRqTsArray` since we guarantee that the background cleanup thread never retires the newest node. Note that a contains operation, as well as all other operations, still announces the EBR epoch it observes when it starts as usual. Accordingly, even if the entry it holds becomes no longer the most recent entry and is retired, it will not be reclaimed until the contains operation finishes.

6 Evaluation

We evaluate bundling by integrating our design into an existing benchmark [4] and comparing with the following competitors: *EBR-RQ* [4] is the lock-free variant of Arbel-Raviv and Brown's technique based on epoch-based reclamation; *RLU* [36] implements an RCU-like synchronization mechanism for concurrent writers; *vCAS* [49] is our porting of the target data structures to use vCAS objects; *Unsafe* is an implementation of each data structure for which range queries are uninstrumented (i.e., non-linearizable).

It should be noted that although the infrastructure supporting range queries in EBR-RQ is lock-free, the data structures still require locking. We ignore the locking variant of EBR-RQ because it consistently performs worse at all configurations. For the same reason, we do not include Snapcollector [43] in our plots. Also, RLU is not included in the results for skip list because no implementation is available.

Since no procedural conversion for lock-based data structures to a vCAS implementation exists, we follow their guideline by replacing with a vCAS object every pointer and metadata necessary for linearization, except for the locks. During updates, a new version is installed in the vCAS object whenever the field changes by using the vCAS API. Since the mutable fields are vCAS objects and all accesses to those fields are instrumented, operations are linearizable. Although it is not clear whether this porting can be generalized to any lock-based data structure, we believe it is sufficient to contrast performance with our approach.

In all experiments, we added another version of Bundle, named *Bundle-RQ*, which moves the responsibility of updating `globalTs` to range queries (similar to [49] and [4]). A general observation is that there is no significant difference in performance between Bundle and Bundle-RQ throughout all experiments. Incrementing the timestamp on updates has

as good or better performance, except in the 100% update workload where it is slightly worse. As a result, our decision to increment the timestamp on updates is not a dominating factor in performance. The timestamp itself does pose a scalability bottleneck, but this is observed across all competitors since they all use logical timestamps to represent versions. Because of this, we avoid discussing the details of this version for brevity and we focus on commenting Bundle's performance in the rest of the evaluation. We also evaluated a version in which the contains operation reads `globalTs` instead of calling `DereferenceBundle` with an infinitely large timestamp. However, this version was constantly 10%-20% worse in light update workloads than our optimized Bundle version, due to the increasing cache contention on `globalTs`. Accordingly, we excluded it from the figures for clarity.

Finally, recollect that removals in our bundled data structures insert entries in the bundles of removed nodes pointing to the head (or root for the Citrus tree). Our experiments reveal no impact on the overall performance since traversals of this kind occur at most four times for every 1 million operations, across all data structures and workloads.

All code is written in C++ and compiled with `-std=c++11 -O3 -mcx16`. Tests are performed on a machine with four Xeon Platinum 8160 processors, with a total of 192 hyper-threaded cores and a 138 MB L3 cache, running Ubuntu 20.04. All competitors leverage epoch-based memory reclamation to manage memory. EBR-RQ accesses the epoch-based memory reclamation internals to provide linearizable range queries. All other competitors have strategies that are orthogonal to how memory is managed. While memory is reclaimed, each approach avoids freeing memory to the operating system to eliminate performance dependence on the implementation of the `free` system call. The results for bundling include the overhead for tracking the oldest active range query for cleaning up outdated bundle entries.

6.1 Bundled Data Structure Performance

In each of the following experiments, threads execute a given mix of update, contains, and range query operations, with target keys procured uniformly. The data structure is initialized with half of the keys in the key range; all updates are evenly split between inserts and removes stabilizing the data structure size at half-full. Workloads are reported as $U - C - RQ$, where U is the percentage of updates, C is the percentage of contains and RQ is the percentage of range queries. All reported results are an average of three runs of three seconds each, except where noted. The key range of each data structure is as follows: the lazy list is 10,000 and the skip list and Citrus tree are both 1,000,000. Except where noted, ranges are 50 keys long with starting points uniformly generated from the set of possible keys.

Traversal Instrumentation. Traversal-dominant data structures (e.g., linear data structures like a linked list) offer insight into the overhead required by each algorithm for supporting range queries. The most obvious consequence of the above observation is that the performance of strategies that require every access to be instrumented suffer compared to those that support uninstrumented traversals. Figure 2 shows the total throughput of operations on a lazy list as a function of the number of worker threads at both 10% updates and 90% updates with 10% range queries.

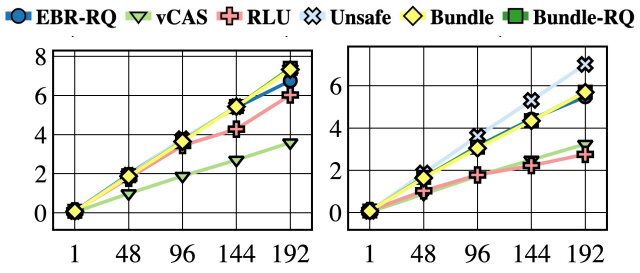

Figure 2. Throughput (Mops/s) of a lazy list for 10-80-10 and 90-0-10 workloads while varying the number of threads.

At 10% updates, Bundle avoids paying a per-node cost for all operations by performing at least a portion of traversals using regular links. Only after landing in the (proximity of the) target range (or key) Bundle starts incurring an overhead. Although internally EBR-RQ uses double-compare single-swap (DCSS), which may add overhead per-dereference, its design is such that a single pointer can be used to reference both the descriptor and the next node. RLU's overhead is concentrated in updates because they wait for ongoing reads; this cost becomes apparent at higher thread-counts. In contrast to the other competitors, vCAS requires an additional dereference per-node to maintain linearizability, leading to less than half the throughput of bundling.

At 90% update rates, the drawbacks of other competitors start to become visible. For example, RLU's heavy update synchronization causes its performance to fall below that of vCAS. Bundle and EBR-RQ maintain higher performance because of their low instrumentation costs.

Varying Workload Mix. Figure 3 reports the throughput (Mops/s) of different workload mixes as a function of thread count for both the skip list (Figures 3a-3f) and Citrus tree (Figures 3g-3l). Except for the two rightmost columns, the range query percentage is 10%, while the update and contains percentages vary. Results for 2% and 50% range query operations can be found in the companion technical report [39].

Our first general observation is that our bundled data structures perform best compared to RLU and EBR-RQ when the workload is mixed. In the four left-most columns, Bundle matches or surpasses the performance of the next best strategy, in most cases. Under these mixed workloads, Bundle achieves maximum speedups compared to EBR-RQ of 1.8x in the skip list (Figure 3a) and 3.7x in the Citrus tree (Figure 3g). Against RLU, bundling can reach 1.8x better performance (Figure 3i). For both, the best speedups are achieved at 192

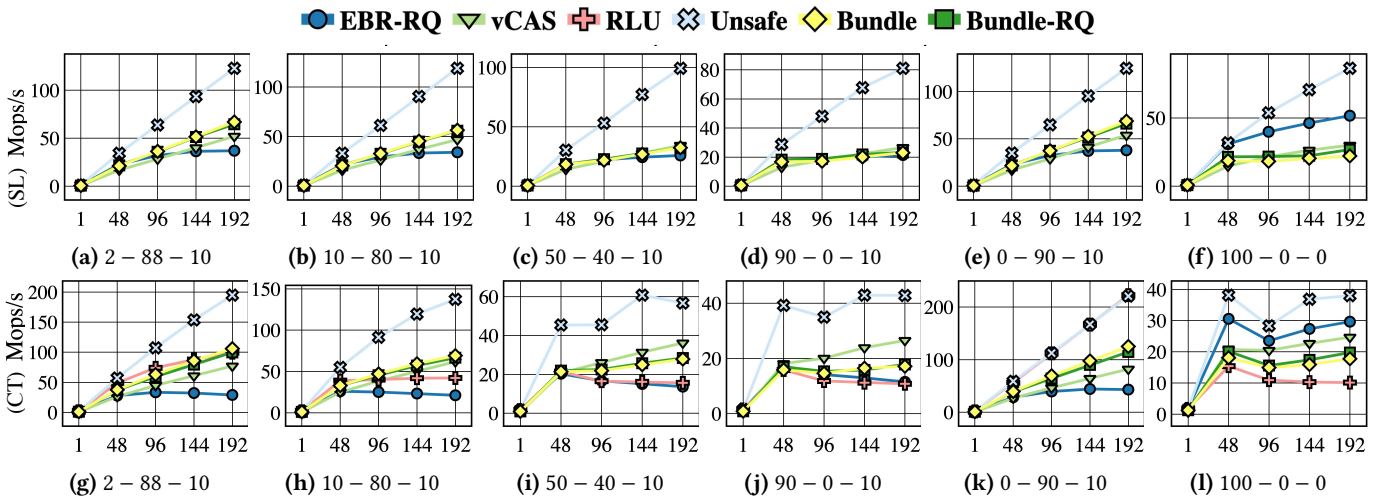

Figure 3. Throughput (Mops/s) under various workload configurations for the skip list (SL) (a) – (f) and Citrus tree (CT) (g) – (l), while varying the number of threads on the x-axis. Workloads are written as $U - C - RQ$, corresponding to the percentages of update (U), contains (C) and range queries (RQ).

threads. Overall, the mixed configurations represent a wide class of workloads; whereas, the two rightmost columns represent corner case workloads and are discussed separately, later. Although we do not test an RLU implementation of the skip list, based on our observations of the Citrus tree results, we expect it would offer similar performance to Bundle for lower update percentages.

In more detail, Bundle performs better than EBR-RQ in low-update workloads by avoiding high-cost range queries. Bundling localizes overhead by waiting for pending entries, limiting the impact of concurrent updates on range query performance. In contrast, EBR-RQ's design allows that possibly many additional nodes be visited in the limbo lists to finalize a snapshot. For the skip list, even at 2% updates, we measure EBR-RQ traversing an average of 141 additional nodes *per-range query* at 48 threads and up to 575 nodes at 192 threads. The above numbers are nearly identical for both the lazy list and Citrus tree, indicating that this behavior is independent of data structure.

The case is similar for the Citrus tree, where Bundle outperforms EBR-RQ by an average 1.2x across mixed workloads at 48 threads, except 90% where it is 10% worse. At higher thread counts, bundling has the clear advantage. For all workloads at 48 threads, Bundle does not perform better than RLU. At a high number of threads, Bundle is never worse. For example, when more updates are present in the workload (Figures 3i and 3j), Bundle outperforms RLU (1.8x and 1.6x) at 192 threads. Interestingly, at high update workloads the Citrus tree does not scale well beyond one NUMA zone, regardless of the presence of range queries.

Compared to vCAS, Bundle performs up to 1.2x-1.5x better in low update workloads (i.e., 0%, 2% and 10% updates) across all thread configurations due to its lightweight traversal. In update-intensive workloads (i.e., 50% and 90% updates), Bundle and vCAS have very similar performance for the skip list, within 5%. This is because their update operations have similar execution patterns (i.e., adding a new version to a lists) and the underlying data structure synchronization starts impacting performance. In the Citrus tree, vCAS demonstrates better performance at 50% and 90% updates and high thread count: as much as 1.5x. This is because operations in vCAS are able to help other concurrent operations, while Bundle must wait for them to finish. Exploring whether bundling can exploit a similar behavior is an interesting future work.

To better understand the (extreme) cases in which Bundle performance is surpassed by either RLU or EBR-RQ, we note that RLU and EBR-RQ prefer workloads at opposing ends of the configurations spectrum. In the read-only setting (Figure 3k), RLU performs well because there are no synchronization costs, and achieves performance nearly equivalent to Unsafe in the Citrus tree. We also evaluate the case of range query only workload and we obserse identical trends as the read-only case (Figures 3f and 3l).

Despite its impressive performance , recall that even a low percentage of updates is enough to cause RLU to collapse (see Figure 3g), namely from the synchronization enforced by writers (i.e., RLU-sync). Hence, when a workload is primarily updates (Figures 3i, 3j, and 3l), RLU is worst. On the other hand, EBR-RQ range queries increment a global epoch counter and validate their snapshot, which leads to worse performance in read-dominated workloads compared to update-intensive ones. In fact, at update-only workloads (the rightmost column), EBR-RQ behaves nearly as well as Unsafe, while for read-only (the second-to-last column) workloads it is the worst competitor.

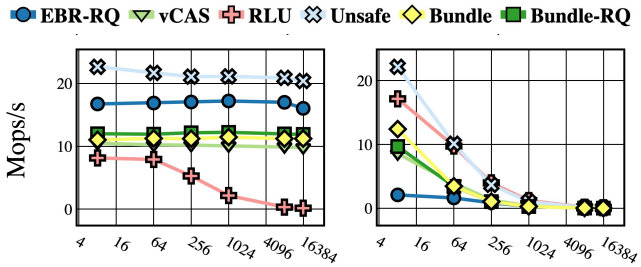

Figure 4. Update (left) and range query (right) throughput for 24 update-only threads and 24 range-query-only threads while varying the range query size.

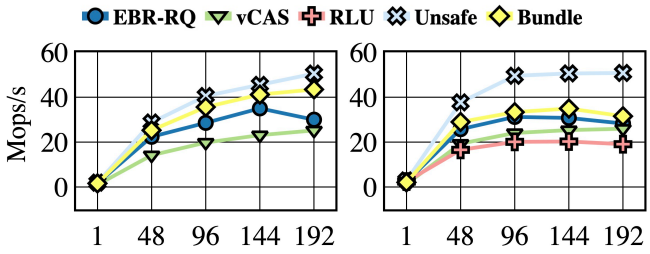

Figure 5. Throughput (Mops/s) of index operations in DBx1000 running the TPC-C benchmark at different thread counts. Skip list (left) and Citrus tree (right). There is no RLU-based skip list.

The performance of both Bundle and vCAS in those extreme cases demonstrates that they properly handle this trade-off, with Bundle outperforming in the read-only cases due to its lightweight traversal.

Varying Range Query Size. Figure 4 shows the throughput (Mops/s) of update and range query operations on a Citrus tree when a single NUMA zone is partitioned such that half of the hyperthreaded cores (i.e., 24) are running 100% updates and the other half are running 100% range queries, while varying the range query length.

Our first observation is that for Bundle, EBR-RQ, and vCAS, the performance of update operations is not negatively impacted by the length of range queries, while for RLU this is not the case. In the former three strategies, update operations are not blocked by range queries, whereas in the latter they must synchronize with ongoing read operations. The result is that for RLU, longer running range queries have a detrimental impact on update performance.

Considering range query performance, EBR-RQ throughput is effectively flat across all range query sizes. This is attributed to the required checking of limbo-lists and the helping mechanism to support DCSS. The result is that it dominates the cost of range queries and even short ones perform poorly. On the other hand, Bundle, vCAS, and RLU demonstrate much better performance for short range query operations (i.e., >10x over EBR-RQ). The disparity is primarily due to the cost of EBR-RQ range queries depending on the number of concurrent update operations, while Bundle, vCAS, and RLU only incur local synchronization costs.

6.2 Database Integration Performance

We evaluate the application level performance of bundling by running the TPC-C benchmark in DBx1000 [51], an in-memory database system, integrated with our bundled skip list and Citrus tree. The data structures implement the database indexes and the number of warehouses is equal to the number of threads. Threads access other warehouses with a 10% probability.

We use the `NEW_ORDER` (50%), `PAYMENT` (45%) and `DELIVERY` (5%) transaction profiles. The `DELIVERY` profile is particularly interesting since its logic includes a range query over the index representing the new order table, ordered by `order_id`, with the goal of selecting the oldest order to be delivered. Next, the order is deleted to prevent subsequent `DELIVERY` transactions from delivered the same order again. In our experiments, the range query selects the oldest order in the last 100 orders. A `PAYMENT` transaction performs a range query on the customer index to look up a customer by name with 60% probability and then updates the total income according to the amount on the payment. `NEW_ORDER` modifies multiple tables and updates their indexes accordingly, including the new order index.

We report the total index throughput measured across the system when the skip list and Citrus tree is used as an index (Figure 5). Note that we elide a comparison against the baseline DBx1000 index since it is a hashmap and does not support range queries. Our results demonstrate that bundling outperforms all competitors regardless of the number of threads used. Summarizing our findings, Bundle is on average, across all thread configurations, only 13% worse than Unsafe for the skip list but 29.6% for the Citrus tree. In comparison to the next best competitor (EBR-RQ), Bundle achieves 1.4x and 1.1x better performance for the skip list and Citrus tree, respectively.

7 Conclusion

We presented three concurrent linked data structure implementations deploying a novel building block, called bundled references, to enable range linearizable query support. Bundling data structure links with our building block shows that the coexistence of range query and update operations does not forgo achieving high-performance, even in lock-based data structures.

Acknowledgments

Authors would like to thank the shepherd and the anonymous reviewers for all the constructive comments that increased the quality of the final version of the paper. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1814974 and CNS-2045976.

References

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[2] Yehuda Afek, Hillel Avni, and Nir Shavit. 2011. Towards consistency oblivious programming. In *International Conference On Principles Of Distributed Systems*. Springer, 65–79.

[3] Maya Arbel and Hagit Attiya. 2014. Concurrent updates with RCU: search tree as an example. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. ACM, 196–205. https://doi.org/10.1145/2611462.2611471

[4] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 14–27.

[5] Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: lessons learned in designing tm-supported range queries. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. ACM, 299–308. https://doi.org/10.1145/2484239.2484254

[6] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. 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, Austin, TX, USA, February 4-8, 2017*. ACM, 357–369. http://dl.acm.org/citation.cfm?id=3018761

[7] Naama Ben-David, Guy E Blelloch, Yihan Sun, and Yuanhao Wei. 2019. Multiversion concurrency with bounded delay and precise garbage collection. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 241–252.

[8] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[9] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*. ACM, 257–268. https://doi.org/10.1145/1693453.1693488

[10] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking k-ary Search Trees. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7702)*. Springer, 31–45. https://doi.org/10.1007/978-3-642-35476-2_3

[11] Trevor Alexander Brown. 2015. 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, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. 261–270.

[12] Bapi Chatterjee. 2017. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*. 1–10.

[13] Shu-Yao Chien, Vassilis J Tsotras, and Carlo Zaniolo. 2001. Efficient management of multiversion documents by object referencing. In *VLDB*, Vol. 1. 291–300.

[14] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A contention-friendly binary search tree. In *European Conference on Parallel Processing*. Springer, 229–240.

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[16] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 109–121.

[17] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: a distributed, searchable key-value store. In *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese (Eds.). ACM, 25–36. https://doi.org/10.1145/2342356.2342360

[18] Facebook. 2020. RocksDB: a persistent key-value store for fast storage environments. https://rocksdb.org/.

[19] Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 128 (nov 2020), 29 pages. https://doi.org/10.1145/3428196

[20] Sergio Miguel Fernandes and João P. Cachopo. 2011. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, Calin Cascaval and Pen-Chung Yew (Eds.). ACM, 179–188.

[21] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge, UK.

[22] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[23] Google. 2019. LevelDB:. https://github.com/google/leveldb.

[24] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. On Developing Optimistic Transactional Lazy Set. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8878)*. Springer, 437–452. https://doi.org/10.1007/978-3-319-14472-6_29

[25] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. 2005. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3974)*. Springer, 3–16. https://doi.org/10.1007/11795490_3

[26] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4474)*. Springer, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11

[27] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[28] Rich Hickey. 2008. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. 1–1.

[29] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[30] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John K. Ousterhout. 2016. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *USENIX Annual Technical Conference*. USENIX Association, 57–70.

[31] Masoomeh Javidi Kishi, Sebastiano Peluso, Henry F Korth, and Roberto Palmieri. 2019. SSS: Scalable Key-value Store with External Consistent and Abort-free Read-only Transactions. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 589–600.

[32] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *arXiv preprint arXiv:1201.0228* (2011).

[33] Doug Lea. 2005. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.* 58, 3 (2005), 293–309. https://doi.org/10.1016/j.scico.2005.03.007

[34] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[35] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 183–196. https://doi.org/10.1145/2168836.2168855

[36] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 168–183.

[37] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[38] Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. 2021. POSTER: Bundled references: an abstraction for highly-concurrent linearizable range queries. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 448–450. https://doi.org/10.1145/3437801.3441614

[39] Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. 2022. *Technical Report: Bundling Linked Data Structures for Linearizable Range Queries*. Technical Report. https://arxiv.org/abs/2201.00874

[40] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[41] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

[42] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. 2015. GMU: Genuine Multiversion Update-serializable Partial Data Replication. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (2015), 2911–2925.

[43] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8205)*. Springer, 224–238. https://doi.org/10.1007/978-3-642-41527-2_16

[44] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. ACM, 151–160. https://doi.org/10.1145/2145816.2145836

[45] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *SOSP*. ACM, 497–514.

[46] Matthew Rodriguez and Michael Spear. 2020. Optimizing Linearizable Bulk Operations on Data Structures. In *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*, José Nelson Amaral, Lizy Kurian John, and Xipeng Shen (Eds.). ACM, 24:1–24:10. https://doi.org/10.1145/3404397.3404414

[47] Konstantinos Sagonas and Kjell Winblad. 2015. Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9519)*. Springer, 37–53. https://doi.org/10.1007/978-3-319-29778-1_3

[48] Neil Sarnak and Robert E Tarjan. 1986. Planar point location using persistent search trees. *Commun. ACM* 29, 7 (1986), 669–679.

[49] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 31–46. https://doi.org/10.1145/3437801.3441602

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

[51] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511

A Artifact Description

Here we give a high-level overview of the artifact supporting this paper. For more detailed instructions on how to configure, run and plot the results, we refer the reader to the comprehensive README file included in the root directory of the source code (https://zenodo.org/record/5790039). Our implementation graciously builds upon the existing benchmark found in [4].

A.1 Requirements

The project is written primarily in C++ and was compiled with GNU Make using the GCC compiler with the compilation flags `-std=c++11 -mcx16 -O3`. The two primary dependencies for the project are the `jemalloc` memory allocation library and the `libnuma` NUMA policy library. The plotting scripts are written in Python and rely on a handful of libraries. For example, we use the Plotly plotting library to generate interactive plots. A full list of dependencies can be found in the README. Note that NUMA must be enabled to run properly and it is recommended to run on a machine with at least 48 threads to reproduce our results.

Docker. For convenience, we also include a Dockerfile that handles setting up the environment. However, if using Docker, keep in mind that performance may suffer and the results may differ from those presented in the paper.

A.2 Code Organization

The root directory contains implementations of all range query techniques, data structures incorporating them, and the benchmarks used for our evaluation. The following is a list of notable directories that pertain to our design and evaluation.

Relevant directories:

- `bundle` (bundled reference implementation)
- `bundle_lazylist` (bundled lazylist implementation)
- `bundle_skiplist_lock` (bundled skip list implementation)
- `bundle_citrus` (bundled Citrus tree implementation)
- `vcas_lazylist` (vCAS-based lazylist)
- `vcas_skiplist_lock` (vCAS-based skip list)
- `vcas_citrus` (vCAS-based Citrus tree)
- `rq` (range query provider implementations)
- `microbench` (microbenchmark code)
- `macrobench` (macrobenchmark code)

Our primary contributions are located in the directories prefixed with `bundle`, which implement the bundled references used when bundling data structures, as well as our bundled versions of the lazylist, skip list and Citrus tree. The same naming strategy is used for the RLU- and vCAS-based approaches. The `rq` directory contains implementations of range query providers, including the one used by bundling, which defines the interaction with the global timestamp.

The `microbench` and `macrobench` directories contain the code used for our evaluation. The microbenchmark (i.e., `microbench`) consists of two experiments designed to understand the implications of design choices between competitors, corresponding to Figure 2, Figure 3 and Figure 4 in the paper. The first experiment tests various update rates while fixing the range query size to 50 and the range query rate to 10%. Results of this experiment are shown in the first two of the aforementioned figures. The second experiment measures performance of a workload with 50% updates and 50% range queries, while adjusting the range query size, and corresponds to Figure 4. This experiment uses dedicated threads to perform each operation type to highlight the effects of updates on range queries, and vice versa. Finally, the macrobenchmark (i.e., `macrobench`) integrates the data structures as indexes in the DBx1000 in-memory database. For both, data is saved to the benchmark's `data` sub-directory, which is then processed by the respective `make_csv.sh` scripts and plotted using `plot.py` (located in the root directory).

A.3 Initial Configuration

Configuring the above benchmarks relies on six parameters that must be adjusted according to the testbed, prior to compilation. The following can be found in the `config.mk` file located in the root directory:

- `allocator`, the allocator used by the benchmarks (blank represents the system allocator)
- `maxthreads`, the max number of concurrent threads
- `maxthreads_powerof2`, `maxthreads` rounded up to a power of 2
- `threadincrement`, the thread step size
- `cpu_freq_ghz`, processor frequency for timing
- `pinning_policy`, the thread affinity mapping

Note that the maximum number of threads cannot exceed the maximum number of threads in the testbed. See the Configuration Tips section of the README for additional information and examples.

A.4 Microbenchmark

For the following section we assume that the current working directory is `microbench`. To build the benchmark, use the command: `make -j lazylist skiplistlock citrus rlu`. Although each data structure binary can be run independently to test specific configurations (see the README), there are scripts to generate a full suite of configurations automatically. Once compiled, the experimental configurations presented in this paper can be run using `./runscript.sh`.

Internally it calls `./experiment_list_generate.sh`, which configures the range query techniques, data structures, and sizes to test and defines the experiments shown in the paper.

The `./runscript.sh` script then reads the set of experimental configurations generated and starts executing them. The number of trials and length of each can be adjusted in the `./runscript.sh` file. The last important configuration file in the microbenchmark is `supported.inc`, which determines which combination of data structure and size configurations are tested.

A file called `summary.txt` is created in the `./data` that contains an overview for the entire run. Individual results are stored in sub-directories whose path is `./data/<experiment>/<data_structure>`. If there are any errors or warnings during execution they will be reported in the `./warnings.txt` file. We describe how to generate plots from the results in Section A.6.

A.5 Macrobenchmark

The macrobenchmark demonstrates the performance of competitors by replacing the index of the in-memory database system called DBx1000 [51]. We assume for the remainder of this section that the working directory is `./macrobench`.

To build this portion of the project, run the `./compile.sh` script. Once complete, the macrobenchmark can then be executed with the `./runscript.sh` script. This script is preconfigured to execute the TPC-C benchmark using a workload mix consisting of 50% `NEW_ORDER`, 45% `PAYMENT` and 5% `DELIVERY` transactions. Similar to the microbench, all results are saved in the `./data` sub-directory.

A.6 Plotting Results

In order to generate plots, we first translate the results produced by the respective `./runscript.sh` scripts into CSV files for ingestion by our plotting script. This is handled automatically by the `plot.py` Python script, which calls the `make_csv.sh` script within each benchmark directory.

After the microbenchmark and macrobenchmark have been executed successfully, plots can be generated using `python plot.py --save_plots` and including either the `--microbench` or `--macrobench` command line flag. This will first generate the requisite CSV file in the corresponding data sub-directory and then create plots from the data contained in the CSV files. The experimental configuration is automatically loaded, but we note that other command line flags can override their values, if necessary. Figures are saved as interactive HTML files in the `./figures` directory, located in the root directory of the project. The plots can be viewed using a browser.

A.7 Expected Output Files

Compiling, running and generating plots produces a number of output files. Here, we provide a summary of the expected output after all of the above steps have been executed. Note that we use angle brackets to indicate that the contents are repeated for the range query techniques (i.e., `rq`) and data structures (i.e., `ds`). Although other files are generated by the compilation phases (e.g., object files), we do not list them here but instead focus on those of most importance to the user. We assume that the current directory is the project's root directory.

```
./
├── microbench/
│   ├── data/
│   │   ├── summary.txt
│   │   ├── rq_sizes/
│   │   │   └── <rq>/
│   │   │       └── *.out
│   │   └── workloads/
│   │       └── <rq>/
│   │           └── *.out
│   └── <machine_name>.<ds>.<rq>.out
├── macrobench/
│   ├── data/rq_tpcc/
│   │   ├── summary.txt
│   │   └── *.out.txt
│   └── bin/<machine_name>/rundb_TPCC_IDX_<ds>_<rq>.out
└── figures/
    ├── microbench/
    │   ├── rq_sizes/
    │   │   └── <ds>/
    │   │       └── *.html
    │   └── workloads/
    │       └── <ds>/
    │           └── *.html
    └── macrobench/
        └── <ds>/
            └── *.html
```