Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data

Kjell Winblad[1]

1 **Department of Information Technology, Uppsala University, Sweden**
kjell.winblad@it.uu.se

Abstract

The need for scalable concurrent ordered set data structures with linearizable range query support is increasing due to the rise of multicore computers, data processing platforms and in-memory databases. This paper presents a new concurrent ordered set with linearizable range query support. The new data structure is based on the contention adapting search tree and an immutable data structure. Experimental results show that the new data structure is as much as three times faster compared to related data structures. The data structure scales well due to its ability to adapt the sizes of its immutable parts to the contention level and the sizes of the range queries.

1998 ACM Subject Classification D.2.8 Performance measures, E.1 Trees, H.2.4 Concurrency

Keywords and phrases linearizability, concurrent data structures, treap

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

The use of concurrent ordered set data structures[1] with support for linearizable[2] range queries[3] is increasing as multicores are becoming more readily available and due to the rise of big scale data processing platforms and in-memory databases such as Google's F1 [22] and Yahoo's Flurry [1]. Both of these require set data structures with fast updates[4] to store incoming data while concurrently serving (typically large) linearizable range queries for analytics [3]. Although there are many concurrent set data structures (e.g. [21, 12, 16]) and ordered set data structures (e.g. [8, 13, 6]), there are only a few concurrent data structures with efficient linearizable range queries [5, 2, 17, 19, 7, 3].

This paper proposes a new concurrent ordered set data structure that internally makes use of an immutable data structure. The difference between an immutable data structure and its mutable counterpart is that the immutable data structure's update operations do not modify the given data structure instance in-place but instead return a new version, leaving the input instance intact. For many data structures, e.g. binary search trees, the operations of the immutable version are asymptotically as efficient as in its mutable counterpart [14]. As an example, the insert operation of an immutable balanced binary search tree only needs to make a copy of the nodes on the path to the inserted node, which only consists of $\mathcal{O}(\log n)$ nodes, where n is the number of items that are stored in the search tree.

[1] A concurrent ordered set data structure represents a set of items that can be manipulated concurrently by several threads and where an item consists of an ordered key and optionally some additional data.
[2] A linearizable operation appears to happen instantly between the operation's invocation and return [10].
[3] A range query operation returns all items with keys within the given range (specified by two keys).
[4] An update operation is an insert operation or a remove operation where the former inserts an item (replacing an existing item if one with an equal key already exists) and the latter removes an item with the given key if such an item exists.

2017 Imperial College Computing Student Workshop (ICCSW 2017).
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One can derive a concurrent ordered set data structure with linearizable range query support from a single mutable reference to an immutable data structure [9]. A lookup or a range query simply performs the operation in the referenced immutable data structure. An update operation repeatedly tries to update the reference using an atomic compare-and-swap operation [9] until the update succeeds. Unfortunately, this coarse-grained approach does not scale when concurrent updates are common due to the scalability bottleneck that exists in the updating of the shared reference. Instead, several data structures [5, 2] use immutable parts that can store a fixed number of items to shorten the time range queries need to spend reading shared mutable data. This fine-grained approach can be efficient when it is possible to fine-tune the size of the data structure's immutable parts to fit the sizes of the range queries (the number of items within the range) and the contention level. However, the fine-grained approach does not work well when the access pattern is unknown or differs in different parts of the data structure.

The *main contribution* of this paper is a new concurrent ordered set data structure with linearizable range query support that solves the problems with the coarse-grained and fine-grained approaches described above by dynamically changing the sizes of its immutable parts to fit the workload at hand. The new data structure is based on the contention adapting search tree (CA tree) [18, 19] and an immutable data structure. Previous results [19] show that CA trees using mutable data structures provide good scalability in scenarios with short range queries. However, previous CA tree variants' scalability for large range queries is limited as their range queries lock out other threads from large portions of the data structure for a time period whose length is proportional to the number of items with keys in the given range. The new CA tree variant eliminates this problem by utilizing an immutable data structure. As is shown in this work, the new CA tree variant's ability to reduce the lock holding times does not only make its scalability substantially better compared to the old CA tree variants but also much better than the other recently proposed data structures with linearizable range query support.

This paper starts with a high-level description of CA trees (Section 2). The new CA tree variant and its implementation are described in Sections 3 and 4. Analytical and experimental comparisons with related data structures are given in Sections 5 and 6. The paper finishes with a conclusion (Section 7).

2 High-Level Description of Contention Adapting Search Trees

CA trees are structured as depicted in Figure 1. The items that are stored in a CA tree are located in sequential data structures under the base nodes (see Figure 1). To efficiently find a specific item in a CA tree, the search is directed by the keys in the routing nodes. All items stored under the left branch of a routing node have keys that are less than the key of the routing node and all items stored under the right branch have keys that are greater than or equal to the key in the routing node. The sequential data structures are protected from concurrent accesses by locks in the base nodes. A base node lock has a statistics counter which is incremented when a thread needs to wait to acquire the lock and decremented when no waiting is required. If the statistics counter in a base node reaches a certain threshold, the items stored in the base node are split between two new

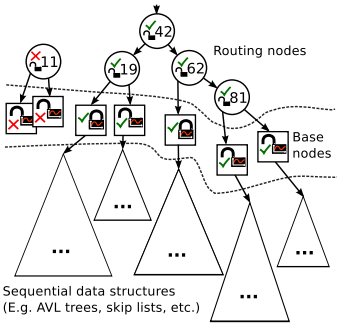
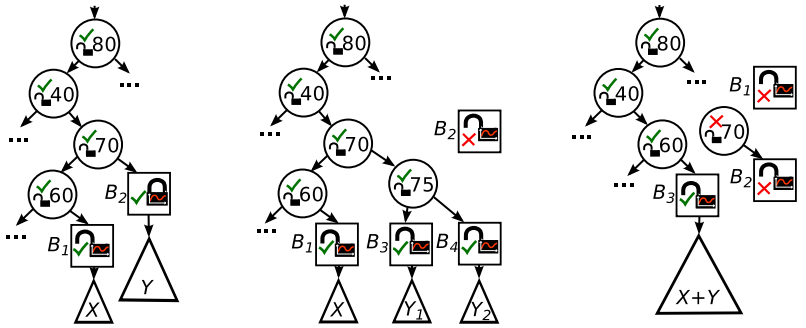


Figure 1 The structure of a CA tree. Numbers denote keys.



(a) Initial CA tree **(b)** CA tree after a split **(c)** CA tree after a join

Figure 2 Effect of the split and join operations on the CA tree of Figure 2a.

base nodes to reduce the contention; see Figure 2a and Figure 2b. In the reverse direction, if the statistics counter in a base node reaches the threshold for low contention adaption, the items in the base node and a neighbor base node are joined into one new base node; see Figure 2a and Figure 2c. Base nodes also have a valid flag (depicted by ✓ and ✗) which is used to indicate if a base node is in the CA tree or if it has been removed. Operations that end up in an invalid (✗) base node need to retry the search until they end up in a valid (✓) base node. Routing nodes also have a valid flag and a lock that are only used rarely during the low contention optimizing join. Range queries are performed in CA trees by first finding and locking the base node containing the first key in the range and then traversing and locking subsequent base nodes until a base node containing a key which is equal to or greater than the largest key in the range is found.

An optimization that has been shown to greatly enhance performance of read operations (lookup and range query) is to let read operations optimistically attempt to do their operation without writing to shared memory [18, 19]. This can be done by using a *sequence lock* [11] as base node lock. A sequence lock has an operation to read a sequence number from the lock. If a thread gets the same even sequence number from two calls of this operation, then the sequence lock guarantees that the lock has not been acquired between the two calls. An optimistic attempt of a read operation first scans the sequence numbers and checks the valid flags of the base nodes that the operation needs to read data from, and then performs the operation, after which the sequence numbers from the locks have to be checked again to make sure that the sequence numbers match the previously read sequence numbers. If the optimistic attempt fails, the operation is done by acquiring the base node locks in read-mode (several read-mode lock holders can hold the lock at the same time).

The reader is referred to the earlier papers on CA trees [18, 19] for a detailed description including pseudo-code and arguments that their operations provide linearizability, deadlock freedom, and livelock freedom.

3 CA Tree Optimization Enabled by Immutable Data Structures

By using a mutable reference to an immutable data structure as a CA tree's sequential data structure, it is straightforward to reduce the amount of time that read operations spend on reading shared mutable data. Assuming that the CA tree's sequential data structure component is implemented with a mutable reference to an immutable data structure, a lookup or range query operation only needs to copy the values of the references that are needed

by the operation while traversing shared mutable data. The immutable data structures referenced by the copied references are then traversed after the base node locks have been unlocked (or after the second sequence number scan). Especially for range queries, this optimization can give a large reduction of the amount of time which is spent on reading shared mutable data as the time that range queries need to spend on traversing the sequential data structures is at least linear in the number of items in the range. With the optimization, range queries may only need to traverse a few base nodes (one in the best case) while reading shared mutable data even when the number of items in the range is large.

It is straightforward to see that this optimization does not jeopardize correctness as the result of a read operation would be the same if the traversal of the sequential data structures happened instantly at the linearization point (due to the immutability of the data structures referenced by the copied references).

4 The Implementation of the Optimized CA Tree

To experimentally evaluate the optimization described in the previous section, a CA tree using a mutable reference to an immutable treap [20] as its sequential data structure has been implemented in Java. A treap is a self-balancing binary search tree with expected time complexity of $\mathcal{O}(\log n)$ for insert, remove and lookup and an expected time complexity of $\mathcal{O}(\log n + r)$ for range queries, where n is the number of items stored in the data structure and r is the number of items in the range. The treap also has efficient split and join operations [20] which is important for the CA tree's low and high contention adaptions [18]. To facilitate cache friendly range queries, the treap implementation stores all items in fat leaf nodes containing arrays that can store up to 64 items.

One heuristic, that CA trees use to reduce the time that future similar range queries need to spend on traversing base nodes, is to decrement the contention statistics counters in the locks of base nodes needed by a range query, if more than one base node is needed; cf. [19]. With the optimization described in the previous section in place, the portion of a range query that is spent on traversing shared mutable data is even more affected by the number of base nodes that the range query needs to access than without the optimization. The reason for this is that the optimization moves the traversal of the sequential data structures from within the period that is spent on reading shared mutable data to after this period. It therefore makes sense to decrement the contention statistics counters with a larger value, when the optimization is used, as the potential benefit for similar range queries is larger than without the optimization. Indeed, experiments show that changing the heuristic to decrement by the value 100 instead of the value one (which is used by the old CA tree implementations) gives significantly better performance in scenarios with large range queries. Except for the change of the value used to decrement the statistics counters in the described heuristic, the immutable treap version of the CA tree has the same constants and thresholds for low and high contention adaptions as described in the previous work [19].

5 Related Work

The CA tree is the only one of the previously proposed approaches for linearizable range queries [4, 5, 2, 17, 7, 3] that dynamically changes the synchronization granularity to optimize for the conflicting requirements of range queries of different sizes and single-key operations [19].

The *SnapTree* by Bronson *et al.* [4] has an efficient linearizable clone operation that returns a copy of the data structure from which a range query operation can easily be derived.

SnapTree's clone operation waits for active update operations and forces subsequent update operations to copy nodes lazily before node modifications so that the copy is not modified. The behavior of *SnapTree* resembles the behavior of the data structure implemented from a single mutable reference to an immutable data structure (that is discussed in Section 1) when range queries are common. The *SnapTree* can thus serve as an example of the coarse-grained approach for doing range queries.

The lock-free k-ary search tree is an unbalanced external search tree with up to k keys stored in every node [6]. Range queries in k-ary search trees are performed by doing a read scan and a validation scan of the immutable leaf nodes containing items in the range [5]. The range query operation needs to retry if the validation scan fails. The k-ary is an example of the fine-grained approach discussed in Section 1. Another example of this fine-grained approach based on software transactional memory is the Leaplist [2].

Chatterjee has proposed a general method for doing range queries in lock-free ordered set data structures [7] based on the work by Erez and Shahar [15]. Unfortunately, the scalability of Chatterjee's method suffers from the global sequential hot spot in the list of range-collector objects that all range queries have to write to in the worst case.

The *KiWi* data structure by Basin *et al.* [3] supports wait-free range queries and lookup operations as well as lock-free update operations. Update operations help range queries by storing multiple versions of inserted items when it is needed for the range queries. Similarly to Robertson's data structure [17], *KiWi*'s range queries atomically increment a global version counter which is used by update operations to decide if storing an additional version is necessary. *KiWi*'s global version number counter is bound to become a scalability bottleneck with a high enough level of parallelism. Similarly to the treap based CA tree, *KiWi* tries to improve cache locality by storing items in arrays that can store up to k items.

A fundamental difference between the other efficient methods for range queries in ordered set data structures and the optimized CA tree is the time spent by range queries reading shared mutable data and thus the time in which conflicts with update operations can happen. In the other methods [5, 2, 17, 19, 7, 3], this time is at least linear in the number of items inside the range while the optimized CA tree can do much better as is described in Section 3.

6 Evaluation

We now experimentally evaluate the optimized CA tree implementation using the immutable treap described in Section 4 (*Im-Tr-CA*). *Im-Tr-CA* will be compared to the recently proposed methods for doing linearizable range queries in ordered sets: *SnapTree* [4], k-ary [5], Chatterjee's method applied to a lock-free skiplist [7] (*ChatterjeeSL*), *KiWi* [3] and the two CA tree variants that use a mutable skiplist with fat nodes (*SL-CA*) and a mutable AVL tree (*AVL-CA*) as sequential data structures [19]. The lock-free skiplist, called ConcurrentSkipListMap, from the Java library that only supports range queries that are not linearizable (*NonAtomicSL*) is also included in the comparison. All data structures were provided by their respective authors and are implemented in Java. The maximum number of items in the nodes (k) is set to 64 for k-ary, *Im-Tr-CA* and *SL-CA* as this value has previously been shown to give good results [5]. *KiWi*'s constants are set as described in the *KiWi* paper [3].

The benchmarks were run on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz each with eight cores and hyperthreading, giving a total of 32 actual and 64 logical cores), turbo boost turned off, 128GB of RAM, running Linux 3.16.0-4-amd64 and Oracle JVM 1.8.0_131 (with the JVM flags -Xmx8g -Xms8g -XX:+UseCondCardMark -server -d64). Each data point comes from the average of three measurements runs of 10 seconds each that

were preceded by 3 warm up runs, also of 10 seconds each. The purpose of the warm up runs is to give the just-in-time compiler enough time to compile the code. Error bars showing the minimum and maximum measurements are displayed when they are large enough to be seen.

The keys for the operations lookup, insert and remove as well as the starting keys for range queries are randomly generated from a range of size S. The data structure is pre-filled before the start of each benchmark run by performing $S/2$ random insert operations. In all experiments presented in the main part of this paper $S = 10^6$. The interested reader can find results for $S = 10^5$ and $S = 10^7$ in the appendix of this paper. Range queries calculate the sum of the items in the range and the number of items in the range. As a sanity check, the average number of items that are traversed per range query is calculated and checked against the expected value.

The Random Operations Benchmark This benchmark measures throughput of a mix of operations performed by N threads. In all captions, benchmark scenarios are described by strings of the form w:A% r:B% q:C%-R, meaning that the benchmark performs $(A/2)$% insert, $(A/2)$% remove, B% lookup operations and C% range queries of maximum range size R. The range sizes are randomly set to values between 1 and R.

Figure 3 shows the results from three scenarios with increasing range sizes. In the scenario with small range queries of maximum size 10 (Figure 3a), the best performing data structures (the CA trees and k-ary) are almost indistinguishable. *ChatterjeeSL* and *KiWi* that have a global scalability bottleneck, as explained in the previous section, both scale worse in this scenario. *Im-Tr-CA* scales well in this scenario due to its ability to adapt the sizes of its immutable parts but does not get much benefit from its quick traversal of shared mutable data as conflicts between threads are rare for all data structures in this scenario.

Conflicts are still relatively rare in the scenario with range queries of maximum size 1000 (Figure 3b). The top performing data structures in this scenario (*Im-Tr-CA*, *SL-CA*, *KiWi* and k-ary) are those with cache locality friendly nodes that store several items in arrays. However, *Im-Tr-CA* and *SL-CA*, that adapt their synchronization granularity to the scenario at hand, outperform *KiWi* and k-ary with a wide margin in this scenario.

The scenario that has large range queries of maximum size 100000 (Figure 3c) shows the distinguishing feature of *Im-Tr-CA* that outperforms all the other data structures with a large margin. Conflicts between range queries and update operations are very likely with

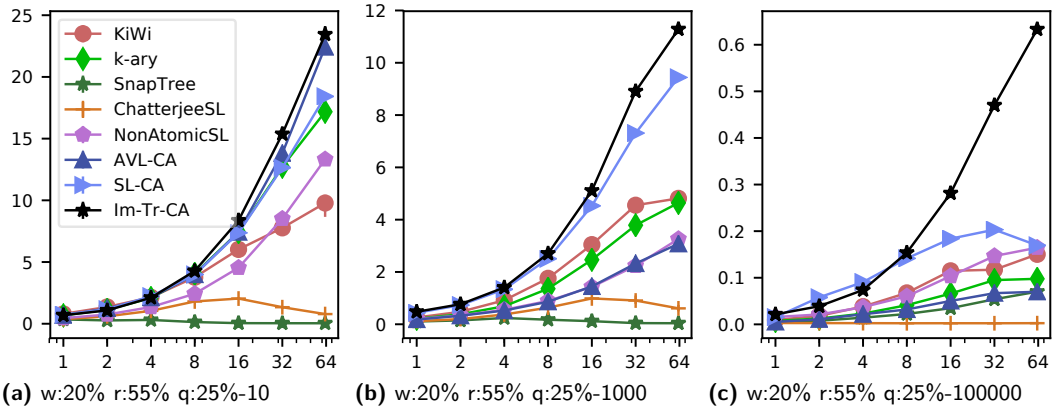
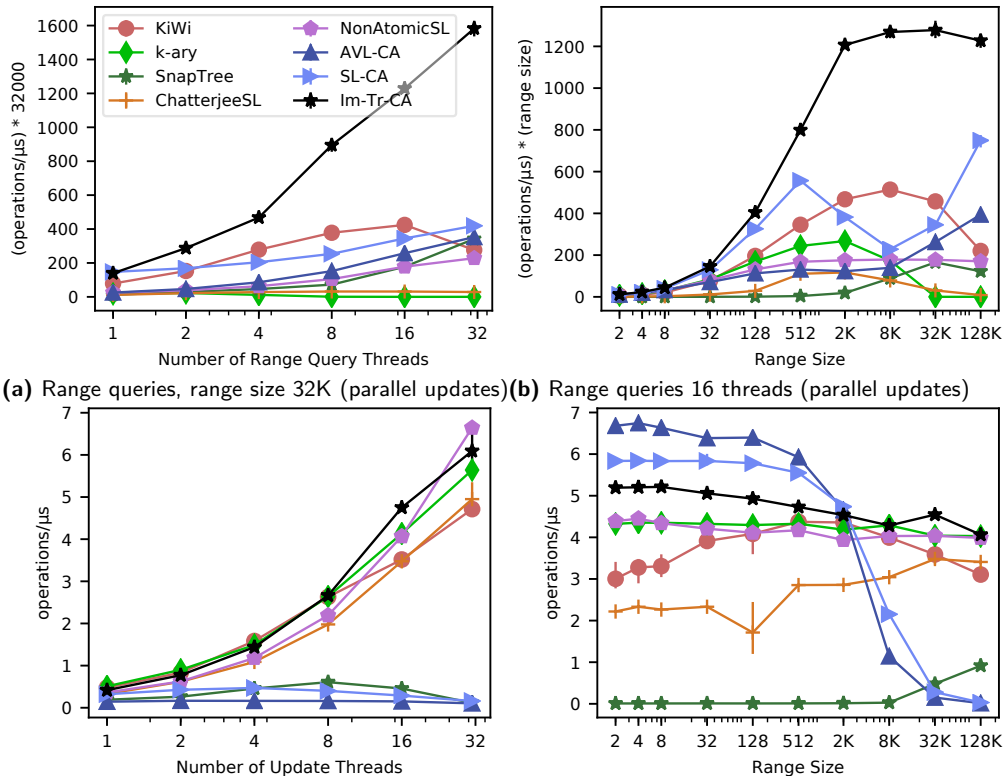


Figure 3 Throughput (operations/μs) on the y-axis and thread count on the x-axis.



(a) Range queries, range size 32K (parallel updates) **(b)** Range queries 16 threads (parallel updates)

(c) Updates (parallel range queries, range size 32K) **(d)** Updates 16 threads (parallel range queries)

Figure 4 Results for benchmark with separate threads doing updates and range queries.

these large range queries but the conflicts are significantly less costly in *Im-Tr-CA* due to its short critical sections, as is explained in Section 3.

Separate Threads for Range Queries and Updates Most of the time is spent doing range queries when large range queries are used in the random operations benchmark. Thus, another benchmark is needed to measure the data structures' ability to handle large range queries concurrently with frequent update operations. To this end, we use a similar benchmark to the one developed by the *KiWi* authors. This benchmark is motivated by large scale applications that require quick updates of a data set while other threads do large linearizable range queries concurrently (for analytics) [3]. In this benchmark, half the threads do update operations (insert and remove with equal probability) while the other half do range queries with a range of fixed size. The throughput for updates is presented separately from the range query throughput so that one can study the performance of these operations separately. Note that in the graphs that show the range query throughput, the number of operations per μs multiplied by the range query size is shown on the y-axis to make the graphs more readable.

Figure 4a and Figure 4c show the results of this benchmark with a range query size of 32K and with varying thread counts. In Figure 4b and Figure 4d, the thread count is fixed to 32 (16 updaters and 16 threads doing range queries) and the x-axis shows varying range query sizes. First of all, *Im-Tr-CA* with its short range query critical section is overall the fastest data structure in the scenarios. *KiWi* is the second most performant data structure in the scenarios with range query sizes larger than 2000. The bumpy performance of *CA-SL* in

Figure 4b can be explained by the fact that the range queries acquire the base node locks in read-mode which enables concurrent range queries to bypass waiting update operations and take over the lock. *CA-SL* thus provides good throughput for range queries with a range size of 512 because conflicts are rare and with a range size of 128K as "conflicts" with other range queries that have already acquired the relevant base nodes are common. *ChatterjeeSL*'s and *KiWi*'s update operations need to read the RangeCollector list (*ChatterjeeSL*) or the global version number (*KiWi*) which are updated by range queries. The more frequent updates of these global objects with smaller range queries can explain the slight partial upward trend that exists for *ChatterjeeSL* and *KiWi* in Figure 4d. *k*-ary's range queries are starved by update operations in the scenarios with large range queries. The *SnapTree*'s operations are slow in most scenarios due to its coarse-grained approach for doing range queries, but the *SnapTree*'s performance is better in scenarios with larger and less frequent range queries.

Table 1 shows the average number of base nodes and the number of base nodes traversed per range query in *Im-Tr-CA* after the benchmark runs of the scenarios displayed in Figure 4b and Figure 4d. It is evident from the table data that *Im-Tr-CA*'s range queries spend a very short time traversing shared mutable data even for large range queries (e.g. approximately the time it takes to traverse 13 base nodes in the case with range queries of size 32K). After the base nodes have been traversed, the collected immutable data can be traversed without any need to care about other threads and without disturbing other threads.

Table 1 Statistics for *Im-Tr-CA* in the scenarios displayed in Figure 4b and Figure 4d.

Range Size	2	4	8	32	128	512	2K	8K	32K	128K
# base nodes	2.5K	2.1K	1.7K	1.0K	590	390	310	310	390	430
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.0	1.1	1.2	1.6	3.5	13	56

7 Conclusion

A new CA tree variant that makes use of an immutable data structure has been presented. The advantage of the new CA tree variant over the CA tree variants that use mutable data structures as the sequential data structure component is that the new variant drastically reduces the time period in which conflicts between large range queries and other operations can happen. Compared to all other data structures with linearizable range query support, the CA trees have the advantage that they dynamically adapt the synchronization granularity to fit the workload at hand. The experimental comparison shows that the presented implementation's quick traversal of shared mutable data and cache friendly design makes the implementation outperform the best of the other data structures with a wide margin in scenarios with large range queries. Furthermore, the new CA tree variant also performs better or close to the best of the other data structures in scenarios with small range queries due to its ability to dynamically change its synchronization granularity. As future work, we plan to design and evaluate a lock-free CA tree variant. A lock-free CA tree variant could potentially give even better performance as it could avoid priority inversions and other lock related problems.

Acknowledgments Vincent Gramoli gave me the idea of looking into immutable data structures in combination with the CA tree. Amelie Lind, Martin Viklund, Stephan Brandauer, Andreas Löscher, Elias Castegren and Konstantinos Sagonas helped me improve the language. This work was supported by the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center).

A Source Code

The source code for the CA tree implementations and the benchmarks can be found online (`https://www.it.uu.se/research/group/languages/software/im_tr_ca`).

B Results with Other Set Sizes

Results corresponding to the results in figures 3 and 4 but with smaller set sizes (the key range size $S = 10^5$) can be found in figures 5 and 6. The corresponding results for larger set sizes ($S = 10^7$) can be found in figures 7 and 8. The statistics corresponding to the statistics in Table 1 but with the smaller and larger set sizes can be found in tables 2 and 3.

In the cases with the smallest set size ($S = 10^5$), the ranges of size 32K and 128K span 32% and 100% of the set represented by the data structures; see Figure 6. *Im-Tr-CA*'s range queries lock out update operations from the portion of the set that is covered by the range query. Even though this only happens for a short period of time as Table 2 shows, it still has a negative effect on update operations as Figure 6d shows. This is compensated by *Im-Tr-CA*'s excellent performance for range queries in these scenarios as Figure 6b shows.

In the cases with the largest set size ($S = 10^7$), the ranges span a smaller part of the sets represented by the data structures which explains why many of the other data structures are closer to *Im-Tr-CA* in these scenarios; see figures 7 and 8.

References

1 Flurry analytics. `https://developer.yahoo.com/flurry/docs/analytics/`. Accessed: 2017-07-26.

2 Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM.

3 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. KiWi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 357–369, New York, NY, USA, 2017. ACM.

4 Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

5 Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems: 16th International Conference, OPODIS 2012. Proceedings*, pages 31–45. Springer, 2012.

6 Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems: 15th International Conference, OPODIS 2011. Proceedings*, pages 207–221. Springer, 2011.

7 Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM.

8 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge Computer Laboratory, 2004.

9 M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPoPP '90, pages 197–206, New York, NY, USA, 1990. ACM.

10 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

11 Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, 2005.

12 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, New York, NY, USA, 2002. ACM.

13 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

14 Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

15 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, pages 224–238, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

16 Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, New York, NY, USA, 2012. ACM.

17 Callum Robertson. Implementing contention-friendly range queries in non-blocking key-value stores. Bachelor thesis, The University of Sydney, November 2014.

18 Konstantinos Sagonas and Kjell Winblad. Contention adapting search trees. In *14th International Symposium on Parallel and Distributed Computing*, ISPDC, pages 215–224. IEEE, 2015.

19 Konstantinos Sagonas and Kjell Winblad. Efficient support for range queries and range updates using contention adapting search trees. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC*, volume 9519 of *LNCS*, pages 37–53. Springer, 2016.

20 R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, Oct 1996.

21 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, May 2006.

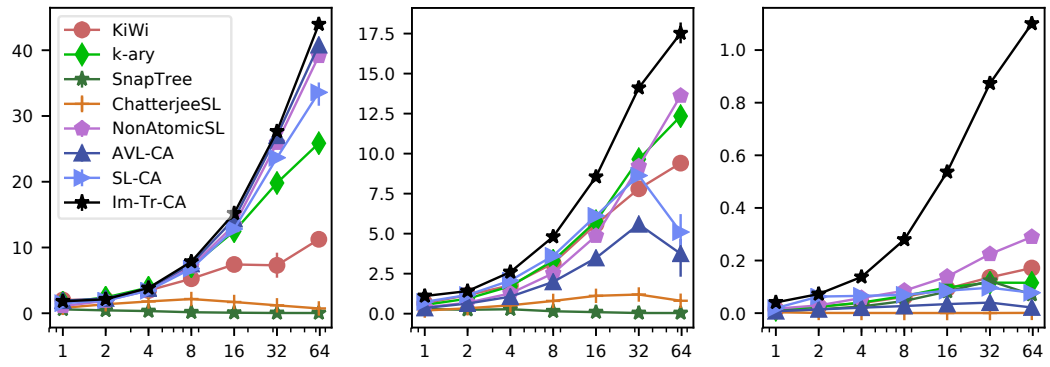
22 Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, August 2013.

Table 2 Statistics for *Im-Tr-CA* in the scenarios displayed in Figure 6b and Figure 6d.

Range Size	2	4	8	32	128	512	2K	8K	32K	128K
# base nodes	890	720	570	330	190	120	97	130	90	96
$\frac{\text{\# traversed base nodes}}{\text{\# range queries}}$	1.0	1.0	1.1	1.1	1.2	1.6	3.0	11	27	98

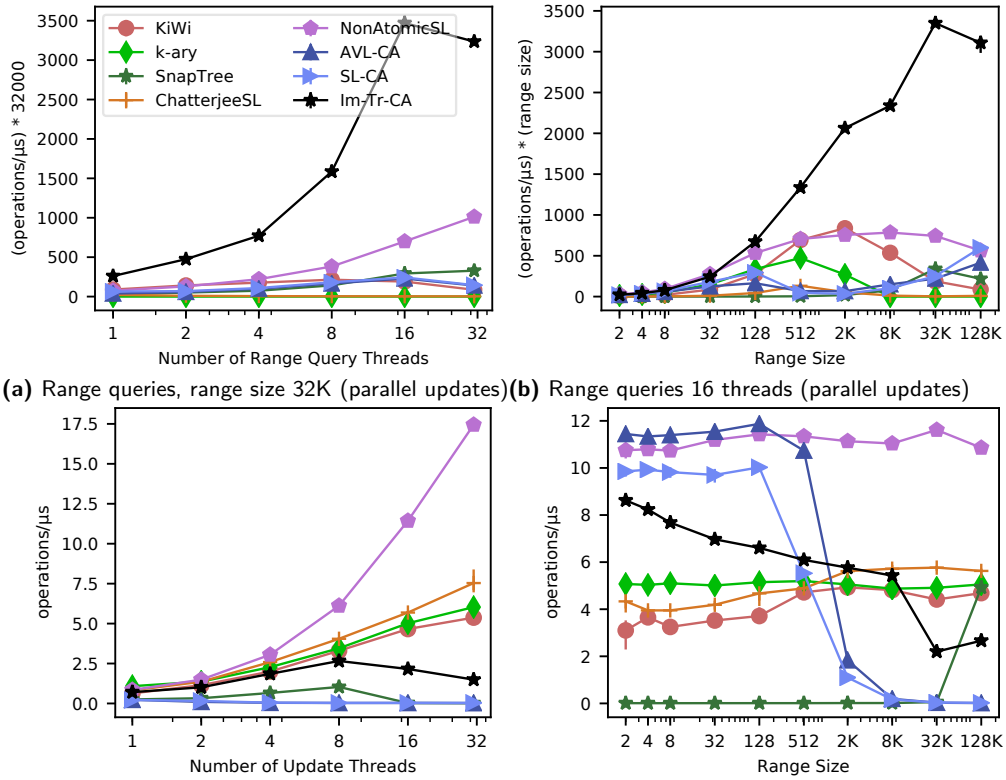
Table 3 Statistics for *Im-Tr-CA* in the scenarios displayed in Figure 8b and Figure 8d.

Range Size	2	4	8	32	128	512	2K	8K	32K	128K
# base nodes	6.0K	5.3K	4.5K	2.9K	1.8K	1.2K	900	860	1.0K	1.1K
$\frac{\text{\# traversed base nodes}}{\text{\# range queries}}$	1.0	1.0	1.0	1.0	1.0	1.1	1.2	1.7	4.2	15



(a) w:20% r:55% q:25%-10 **(b)** w:20% r:55% q:25%-1000 **(c)** w:20% r:55% q:25%-100000

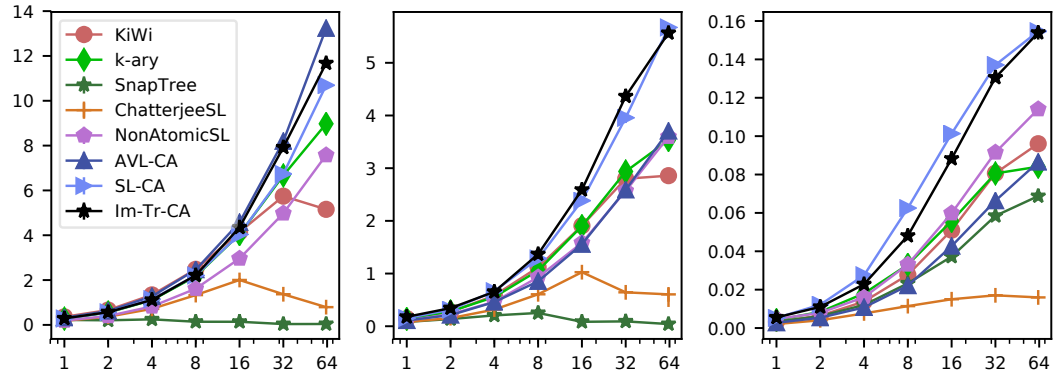
Figure 5 Results with key range of size 10^5 which corresponds to a set size of approximately 5×10^4. Throughput (operations/μs) on the y-axis and thread count on the x-axis.



(a) Range queries, range size 32K (parallel updates) **(b)** Range queries 16 threads (parallel updates)

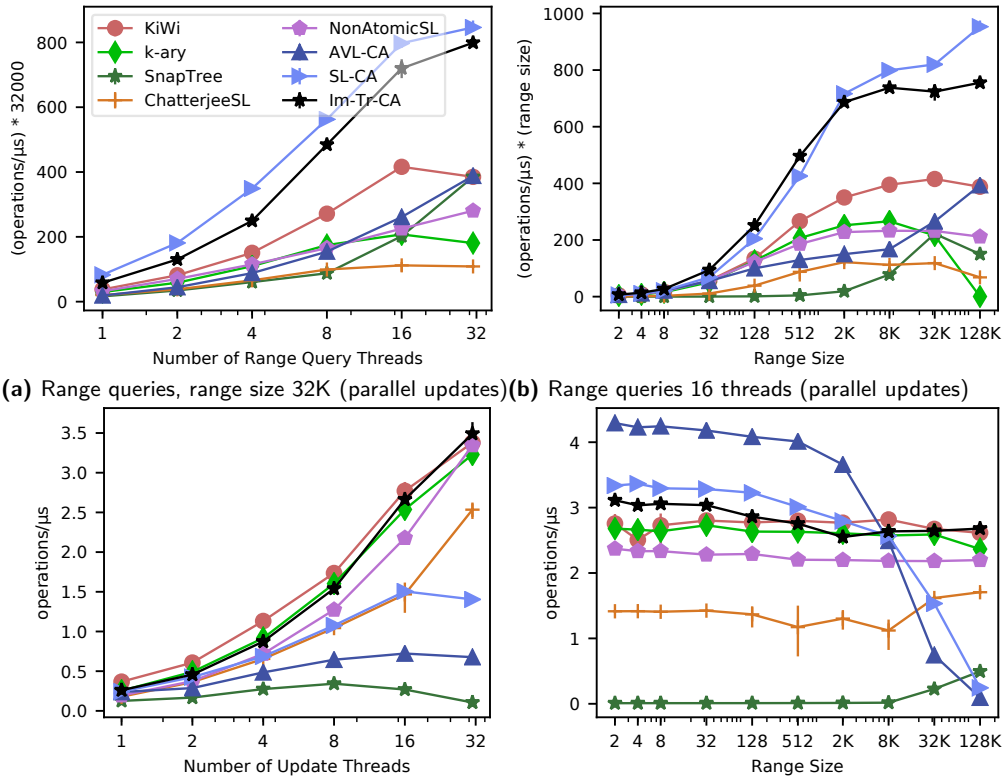
(c) Updates (parallel range queries, range size 32K) **(d)** Updates 16 threads (parallel range queries)

Figure 6 Results for benchmark with separate threads doing updates and range queries with key range of size 10^5 which corresponds to a set size of approximately 5×10^4.



(a) w:20% r:55% q:25%-10 **(b)** w:20% r:55% q:25%-1000 **(c)** w:20% r:55% q:25%-100000

Figure 7 Results with key range of size 10^7 which corresponds to a set size of approximately 5×10^6. Throughput (operations/μs) on the y-axis and thread count on the x-axis.



(a) Range queries, range size 32K (parallel updates) **(b)** Range queries 16 threads (parallel updates)

(c) Updates (parallel range queries, range size 32K) **(d)** Updates 16 threads (parallel range queries)

Figure 8 Results for benchmark with separate threads doing updates and range queries with key range of size 10^7 which corresponds to a set size of approximately 5×10^6.