Lock-Free Contention Adapting Search Trees

Chris Blythe, Nick Cunningham, and Jeff Hoskins

November 30, 2018

Abstract

The prevalence of larger and larger datasets necessitates modifications to the means by which these data are stored and accessed. To efficiently increase the throughput of database operations under heavy load, the problem of data contention must be addressed. Windblad *et al.* did so by developing a lock-free binary tree that dynamically adjusts the distribution of the data within the search tree. In this paper, we seek to implement the basic structure and functionality of their lock-free contention adapting search tree (LFCAT). The throughput of our implementation will be measured under varying concurrency and operational loads.

LFCAT Code Repository: `https://github.com/ucfblythe/COP6616-LFCAT/`
LFCAT STM Code Repository: `https://github.com/ucfblythe/COP6616-LFCATSTM/`

1 Introduction

As the prevalence of digital technology in modern society has steadily increased over the past decade, so too has the quantity of information that has been produced. The efficient acquisition, processing, and utilization of these massive unstructured data sets has become a crucial area of research. Cloud processing of this "Big Data" has become incredibly common as different industries rely on insights from analyzing this data to make long-term business decisions in real-time. Unfortunately, traditional relational database models have trouble adapting to the variety and scale of such large sets of data as they require a fixed predefined structure, complex fail-over schemes, and costly partitioning to handle the increased capacity [1]. Alternatively, modern NoSQL databases attempt to provide scalability, performance, availability, and flexibility [1] as a means of coping with variety, throughput, and amount of data with which modern database systems must handle.

One innovative implementation of a NoSQL database is known as the key-value store model. This model is quite flexible as it simplifies the storage problem to a key used for indexing within the structure, and a value or a set of values to be stored as a generic blob. This simplicity means that it is easy to perform single-item operations including contains, insert, and remove as well as multi-item operations including range query which returns all values whose keys are within a specified range. However, the problem of data contention, that is multiple operations seeking to utilize the same data, is still a limiting factor for the performance of the database especially as the number of cores in a system increases.

2 Contention Adaptation

To better understand the problem of contention in the key-value model, consider a binary tree holding a set of keys in the leaf nodes. When a static synchronization strategy with a fixed granularity is used to overcome contention within the tree, either single-key operations such as insert, remove, and contains will perform well or multi-key operations such as range query will, but not both. If a fine-grain synchronization strategy is used, single-key operations will perform well since they will experience contention infrequently, but multi-key operation performance will suffer since the operations must perform a large amount of overhead to access each node in the query. Conversely, if a coarse-grain synchronization strategy is used, multi-key operations will perform well since fewer places in the tree require synchronization, but single-key operations will perform worse since more contention will occur even if the single-key operations are being performed on different nodes that are logically unrelated. Attempting to fine-tune the granularity of synchronization cannot account for both cases efficiently. To do so would require *a priori* knowledge of the level of contention and the distribution of the operations to be performed. Worse, if the amount of contention within the tree changes over time or different parts of the tree experience differing amounts of contention, static approaches cannot account for these types of variations.

A dynamic synchronization strategy that adapts to the amount of contention that is occurring within a part of the tree at a given time can outperform static synchronization strategies. Leaf nodes that experience a high amount of contention from many single-key operations colliding at a node are split into multiple nodes with a new route node connecting the two together to reduce contention. Leaf nodes that experience low amounts of contention from use predominately with multi-key operations and few single-key operations are merged with neighboring nodes to reduce synchronization overhead when multi-key operations operations are performed.

In a 2015 paper, Sagonas and Winblad describe a lock-based contention adaptive tree which adjusts the synchronization granularity based on the heuristics of the leaf nodes in the tree [2]. When a leaf node experiences contention above a set threshold, the set of keys held in that node is split in half with each half being placed in a new leaf node. Thus, the level of contention on each of the new leaf nodes is lower than the original node. Additionally, when adjacent nodes experience contention below a set threshold, the keys held in these nodes are combined into a single set and are placed in a new leaf node. Recently, Sagonas, Winblad, and Jonsson sought to further improve performance of these trees by removing the need for locks [3]. This lock-free contention adapting search-tree (LFCAT) is what we seek to implement in this paper.

3 Related Works

Due to the high demand for research in this field several efficient data structures that provide concurrent range query support already exist. One such structure with similarities to the LFCAT structure is the lock-free k-ary search tree [4]. The k-ary tree provides an unbalanced search tree containing k keys in immutable nodes. Queries in the k-ary search tree are done using a read-validate method. If relevant nodes in the structure are modified during or after a read then the validate process fails and the operation must be retried. The k-ary tree shows the shortcomings of a fixed granularity approach, as varying types of operations at a higher scale lead to consistently high conflict in the fixed structure and require continuous retries. Another method is known as the KiWi data structure [5], wherein nodes are versioned by update operations based on a global counter. With high enough concurrency between operations this global counter is likely to be a high source of contention that does not scale with the number or types of operations applied. These methods offer no adaptations depending on the current level of contention found in the structure. The same authors of the LFCAT implementation originally explored the concept of contention adaptation in a lock based implementation [2]. This lock-based implementation uses mutable sequential structures to store items, which limits the amount of memory required for update operations. However, the coarse synchronization of standard locks created a sequential bottleneck that limited the potential concurrency of operations over multiple nodes.

4 Data Structure Overview

4.1 Node Types

The LFCAT data structure is defined by a number of specialized nodes: route nodes, base nodes, join-main nodes, join-neighbor nodes, and range nodes. Each node has its own purpose and descriptive state that collectively describe the full state of the LFCAT. Route nodes are used to define the path from the root node to a given leaf node. Like a standard binary search tree, the route nodes each contain a key that is used to properly order the tree. Key indexes are used to traverse route nodes until a leaf node is found. The leaf nodes are defined as base nodes, and they contain the actual dataset stored as sets of key-value pairs. The three remaining node types are used to implement the split, join, and range query methods which are described below.

The structure of the dataset held in each base node is not constrained to be of any particular type. Because the storage mechanism is independent of the LFCAT protocols, the datasets can be any container that holds the key-value pairs. While the datasets are not the main focus of this paper, we should note that Sagonas, Winblad, and Jonsson used a simple treap structure in their paper largely due to their efficiency in distributing values thus providing $O(\log n)$ lookup times. We sought to preserve the order of the keys in each dataset, and opted to use an AVL tree in our implementation because AVL trees are guaranteed to be balanced whereas treaps are randomly ordered.

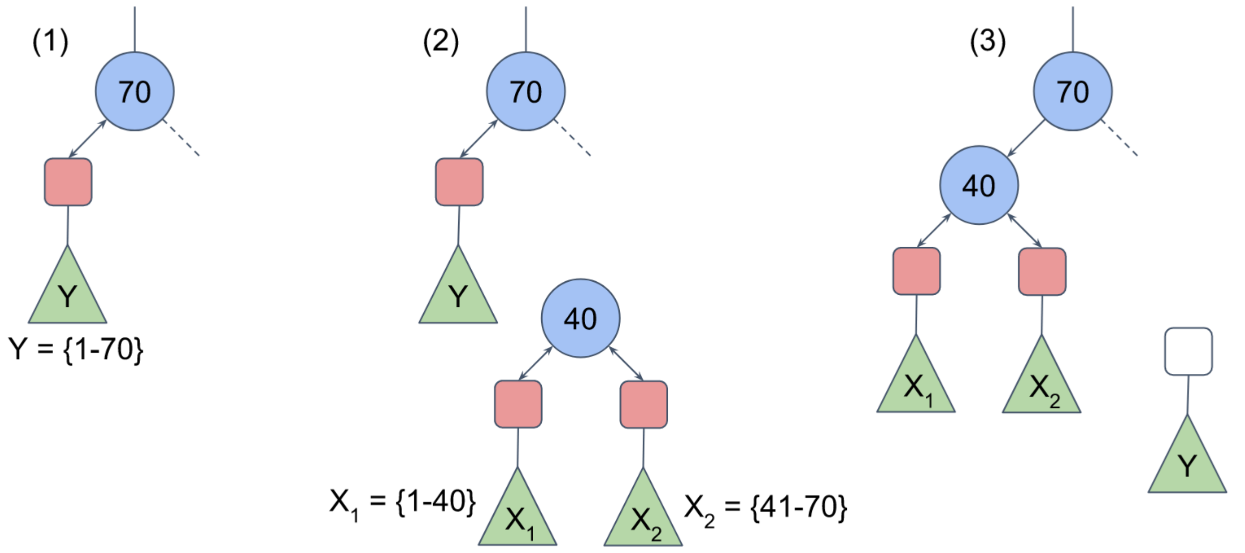

Figure 1: Diagram of split adaptation with base nodes in red, route nodes in blue, and AVL trees in green. A local route node is created with a key equal to the median value of the original AVL tree. New base nodes are created, each containing half of the original AVL tree, which are assigned as the children of the local route node. A CAS is performed to replace the old base node with the new route node, leaving the original base node to be garbage collected.

4.2 High Contention Adaptation

Updates to any node in the LFCAT require that a local node be created, then modified, and swapped with the original node using a Compare-And-Swap (CAS) operation. The CAS is the linearization point of each operation as it is the moment when the local modification become visible in the LFCAT. A failed CAS operation indicates that the original node is experiencing contention. If the original node is a base node, its heuristic value is increased and the update is attempted again. Repeated failures of single-key operations, like insert and remove, will drive up the contention heuristic, indicating that the target node contains a frequently accessed set of keys and that the base node should be split in order to reduce the contention.

When a base node is split, each half of the original AVL tree is assigned to its own local base node. A local route node is created containing the split key of the original AVL tree, and the local base nodes are set as the route node's children. The update to the LFCAT structure takes effect when the old base node is swapped with the new route node using CAS. This process is shown in Figure 1. By splitting the base node, the contention on this set of keys is lowered, prompting fewer CAS failures, thus increasing the performance of the LFCAT for single-key operations.

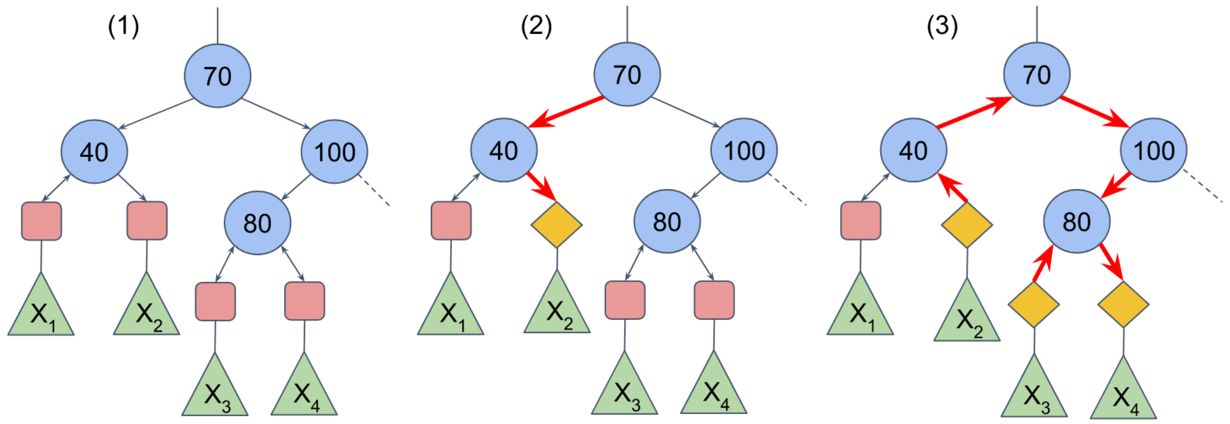

Figure 2: Diagram of range query. Yellow diamonds are range nodes that denote that they are currently involved in a range query operation. All base nodes in range are found and replaced with range nodes via a depth-first traversal from the lowest node in range to the highest. Once all nodes in range have been marked, a new AVL tree consisting of the union of the trees on the marked nodes is returned.

4.3 Range Queries

Another common operation required of key-value store models is the range query, which returns as set of key-value pairs within a specified key range. Range queries are difficult to optimize because the number of key-value pairs that will be returned is not known in advance. Given a search range, the range query operation traverses the structure, saving the nodes encountered in stacks, and marking the relevant nodes. Figure 2 shows an example of a range query traversal. The operation begins by traversing the LFCAT to find the base node containing the lowest key within the specified range. The base node and its parent are then placed in a stack. These nodes are marked as being a part of the range query by replacing their nodes with range node copies via CAS. Starting from the base node with the lowest key in the range, the range query operation checks the next leftmost node that has yet to be visited. This is repeated until a base node containing a key that matches or exceeds the upper limit of the range is reached. The stack of verified nodes is then traversed in order to join each AVL tree into one result set. Other threads see a range query is complete when the result set is populated. This is the linearization point of the range query.

4.4 Low Contention Adaptation

Unlike the individual node access of the insert and remove operations, range query performance is adversely affected by sparsely distributed keys. Collecting keys spread over multiple nodes requires more work for the range query operation because each node within the range of the query must be replaced with a range node using CAS. The more nodes a range query contains, the more nodes that must be swapped thereby increasing the overhead. When a query encounters a range that contains multiple base nodes, the contention heuristic for those nodes is lowered. Like the single-key operations, the range query also compares each node's heuristic to the contention thresholds and adapts if needed. If there are fewer CAS

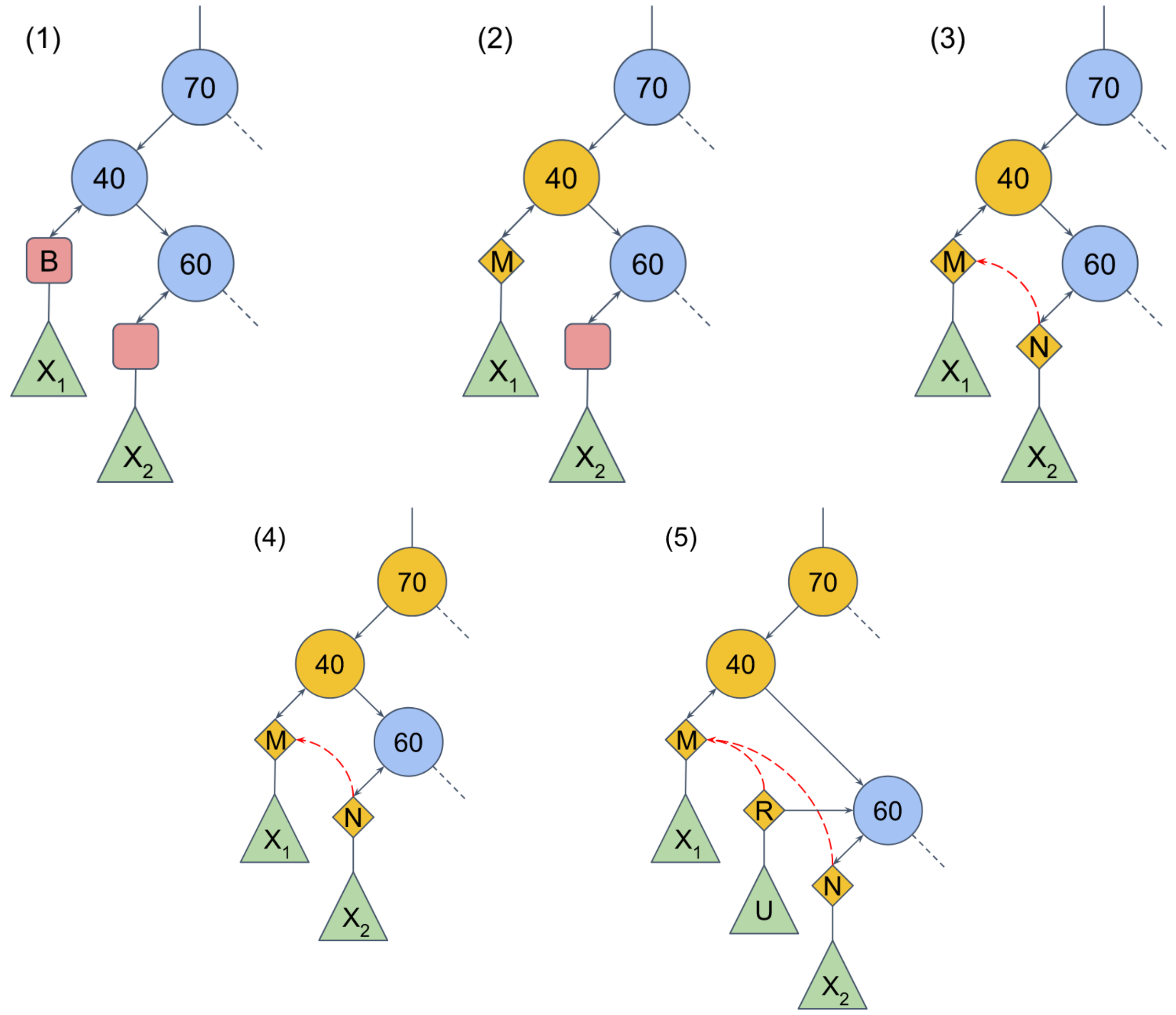

Figure 3: Diagram of secure join. (1) Node B will be merged with its leftmost neighbor. (2) B is replaced with a join-main node M and its parent is marked. (3) The leftmost neighbor is found and replaced with a join-neighbor node N. (4) The grandparent of M is marked. (5) The replacement node R is prepared. Up to this point, an operation can abort the join. The join operation continues from here via complete join shown in Figure 4.

failures than there are multi-node ranges, the contention heuristic will eventually drop low enough to call for the joining of one or more base nodes. By performing this adaptation, future range queries will not have to swap as many nodes, thereby increasing the throughput. Performance of single-key operations can also be improved if little or no contention is experienced by making the path to a node shorter.

For low contention adaptation, two base nodes are merged via the join operation, resulting in a coarser granularity for the key distribution. When a join operation is performed on a base node, the process is split into two parts labeled as secure join and complete join. Secure join, as diagrammed in Figure 3, starts by swapping the base node with a join-main node using CAS. Next, the leftmost neighbor (i.e. the next base node in key order) is found via a depth-first traversal and swapped with a join-neighbor node, again using CAS. These two join nodes indicate to other threads that a join operation is in progress and that these nodes may not be replaced until the join is complete or aborted.

If these CAS operations are successful then the algorithm must also notify the grandparent route node of the target, since it will point to the newly joined base node once the join operation is complete. This notification takes place through the CAS of an atomic reference variable in the grandparent node, with the newly created join-main node. This CAS can only succeed if no other node has marked the grandparent node for a join operation. Once the grandparent is successfully notified, another CAS operation takes place where one of the join-main node's atomic references is swapped to point to a new base node that stores the combined AVL tree of the join-main and join-neighbor nodes. Up to this point any other operation could intercede and mark the join-main node as 'aborted' since any operation has higher priority than node maintenance.

Once the join-main node's reference has been changed from a static 'preparation' node to a local joined node, the join operation is ready to be finalized via complete join as shown in Figure 4. From this point on, other operations will attempt to help complete the join operation rather than simply aborting it. To complete the join operation the parent route of the target node is logically marked as invalid, and any existing grandparent is made to point to the replacing structure. If the join-neighbor node was a direct sibling of the join-main node, then the replacement of the parent route node is simply the newly created base node containing the joined data. However, if the join-neighbor node is the result of a deeper sub-node structure then the parent node will be replaced by the top Route node that leads to that join-neighbor node from the targets parent. The join-main node is no longer a logical part of the LFCAT structure, but to describe it's logical removal to other operations the join must logically mark the node as completed by atomically setting one of it's references to a static 'done' node.

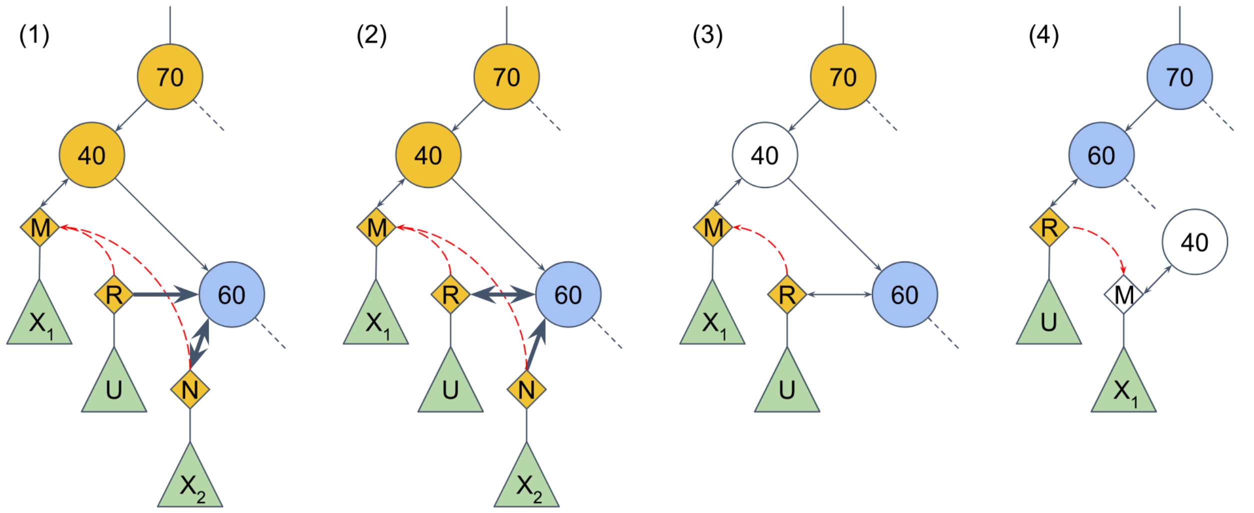

Figure 4: Diagram of complete join. (1) is a copy of (5) from Figure 3 with the relevant references emphasized. (2) performs a CAS on the left child node of route node 60 to point from the old node N to the new replacement node R. (3) Node N is now removed from the tree. Route node 40 is logically marked as invalid. (4) A CAS is performed to replace the reference to route node 40 in the grandparent node with a reference to route node 60. Route node 40 and node M are garbage collected, and the join operation is now complete.

5 Implementation

Our implementation follows along with the basic concepts of the LFCAT structure as described in [3]. However, unlike the C based pseudo-code provided by the authors, our implementation attempts to provide an object-oriented alternative written in Java. Using polymorphic objects that implement generic interfaces allows us the flexibility to create a more dynamic storage structure than that of the provided pseudo-code. Java was selected as the implementation language largely due to it's abstracted object reference model and automated garbage collection. By removing the need for manual memory management techniques, like object pooling, we simplify the overall structure and can better focus on the algorithm implementation. Of course, this simplification will likely result in lower performance as we do not have complete control over how, or when, memory gets collected nor can our algorithm recycle memory locations that have already been allocated. A link to the implementation is included at the end of the abstract. The repository contains the LFCAT implementation and the STM implementation using Deuce-STM.

The most basic building block of our LFCAT structure is a Node class with abstract methods that all other node types must implement. The Node class keeps track of the current parent and usage statistics as well as ensuring that all other nodes can report whether they are: in a replaceable state, in need of adaptation, or in need of assistance. The actual implementation of these functions is left to the node types as each type has different behaviors under different conditions. Base nodes inherit from node, but also contain an AVL tree member variable where in the actual key-value pairs will be stored. This type of object oriented paradigm is used across our structure to provide a separation of concerns over the different node types that leads to increased legibility, ease of use, and possible extensions in future work.

Our implementation also differs from the original LFCAT structure in that we have introduced the size of the AVL tree as a consideration in the statistics used for node adaptation. Our implementation requires any update operation to make a full copy of the AVL tree in a given node before attempting a CAS with the new local node. This copy is expensive as it increases linearly with the number of nodes in the AVL tree. As the leaf node gets larger, this cost becomes prohibitively expensive as most of the time of an update is spent on copying the AVL tree before being updated. This cost is specially apparent when update operations fail, as they will need to recopy the AVL tree from the new node in order to attempt their update again. By introducing the tree size as a factor in the node statistic we are forcing a balanced LFCAT structure that still allows for contention adaptations while distributing the load across nodes.

In the early stages of our implementation we found that update operations from multiple threads at one time could easily conflict and enter a race condition cycle of continuously attempting the same operations. The original LFCAT implementation should address this through the high contention adaption and by allowing threads that fail the update the chance to help other threads with ongoing operations. Allowing the failed threads to provide continuing progress as well as a pseudo back-off so that failed threads do not all reattempt at the same time. However, we found that under a high amount of contention the update operations would often not be able to complete the split adaptation before another thread invalidated the split node. Also, update operations cannot assist each other, which means that if multiple threads with updates fail because of another update operation they will all reattempt their updates at the same time. To help alleviate this contention we introduced an elimination back-off to our update operations in the LFCAT structure. If an operation fails and cannot assist another node then it must wait for a small period of randomly selected time before attempting the operation again. If the operation continues to fail the guaranteed amount of wait time is exponentially increased.

6 Testing

Our LFCAT implementation was tested under different concurrency and contention loads. Varying the concurrency levels was achieved by using 1, 2, 4, 8, 16, and 32 threads for each of the following operation distributions. Each scenario consists of randomly selected/ordered operations that still conform to the distribution being tested. A real-world distribution consisted of 80% contains, 9% insert, 9% remove, and 2% range queries. A high-contention distribution consisted of 59.9% contains, 20% insert, 20% remove, and 0.1% range queries. A low-contention distribution consisted of 70% contains, 5% insert, 5% remove, and 20% range queries. A high-conflict distribution consisted of 0% contains, 40% insert, 40% remove, and 20% range queries. Each experiment consisted of 1,000,000 total operations, wherein the throughput (operations per μs) was measured as a function of the thread count for each of the distributions. To further simulate a real world environment each test scenario is run

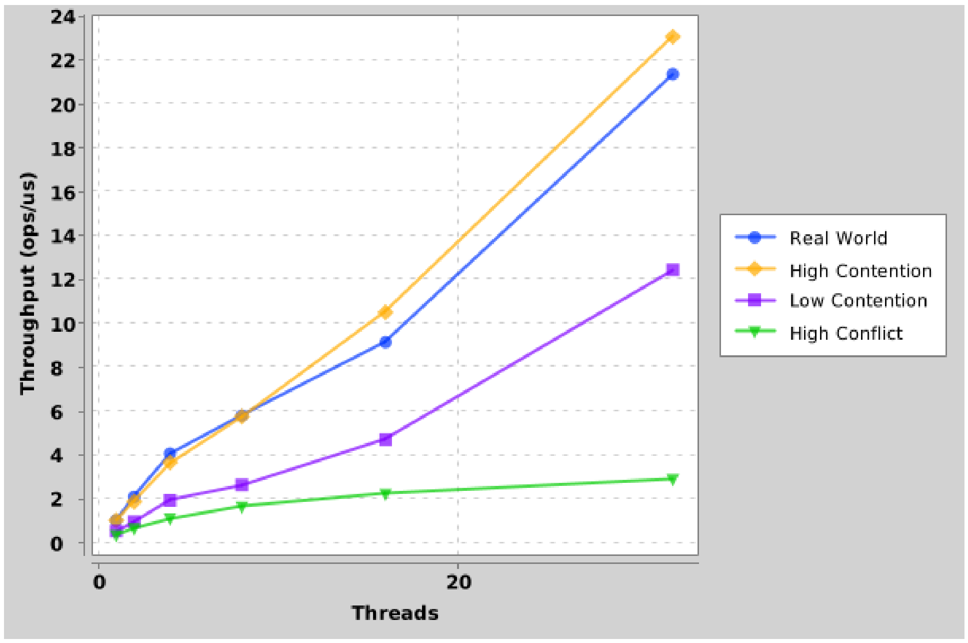

Figure 5: Performance of the LFCAT for 1, 2, 4, 8, 16, and 32 threads for various operational loads.

on a pre-populated LFCAT consisting of up to 500,000 entries spread out across the structure.

Two additional experiments were performed each using 16 threads and only a single operation distribution. One varied the range query sizes with a distribution of 10% insert, 10% remove, and 80% range queries and the other varied the maximum size of the AVL trees while using the real-world distribution. Both of these experiments were used to test the limitations on the adaptability of the LFCAT data structure. All tests were performed on an Intel i7-8700K with 6 cores and 12 threads. The system has 12 MB of L3 cache and 16 GB of RAM.

7 Results

7.1 LFCAT Performance

Figure 5 shows the performance of our LFCAT implementation under the various loads described above. The high conflict load performs the worst since this load receives the most contention. All loads are ordered based on the percentage of range queries with higher percentages resulting in lower throughput. This is due to the large amounts of contention that occur when a range query is being performed. All nodes in the range must be marked and cannot be updated until the range operation is complete. Since the range query has the potential of hitting a large number of nodes, this effectively places a barrier that other operations must wait on before proceeding. The throughput of the LFCAT scales well with increasing number of threads even as context switching occurs when more than 12 threads are utilized.

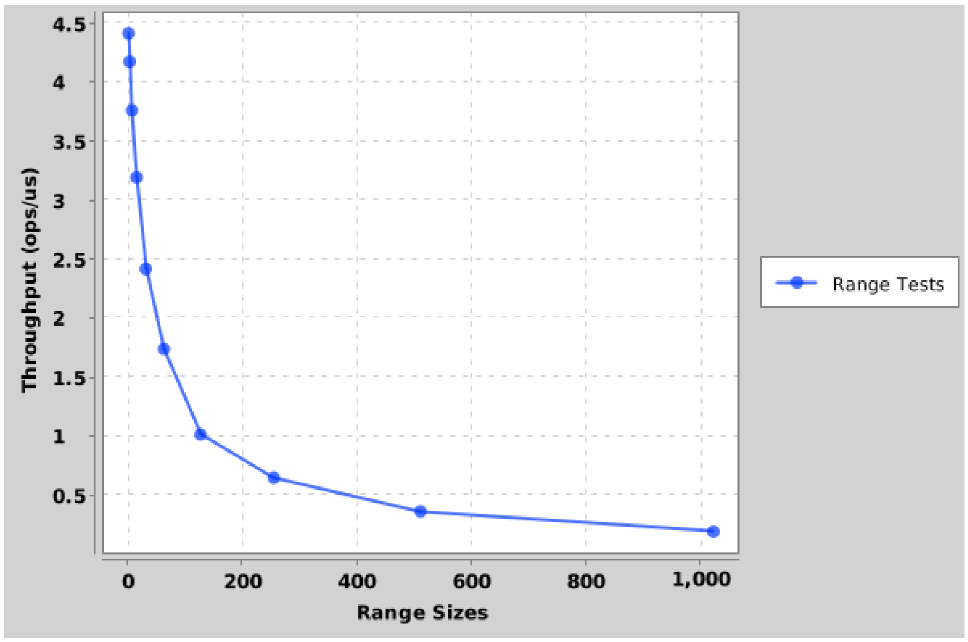

Figure 6: LFCAT throughput as a function of maximum range size. 10% insert, 10% remove, and 80% range queries using 16 threads. Tested at range sizes of 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024

The performance of our LFCAT implementation performs slightly worse than the original author's implementation under similar conditions and loads. These differences in performance can be attributed to the use of Java in our implementation instead of C in the original paper. Our implementation sacrifices better performance for automatic memory management via garbage collection which simplified implementation and ensured no memory leaks occurred. Also, Java ensures the ABA problem is avoided in the LFCAT algorithm since an allocated object in the memory pool cannot be reused while any thread maintains a reference to the original object.

7.2 Varying Range Query Sizes

The size of range queries has a significant impact on the throughput of the LFCAT. Figure 6 shows the performance when the range size is varied from 1 to 1024 in powers of 2 increases. The throughput decreases monotonically with increasing range size. As the range increases, more nodes must be marked and no insertions, deletions, or other range queries can occur in the region that has been marked until the current range query completes. This is to ensure the operations are linearizable with one another. Since no other operations within the marked range can proceed until the range operation completes, this significantly increases the contention each operation conflicts with as the range size increases. Another simpler factor that affects the performance is that all range operations, regardless of the size, count as a single operation for the throughput so as the range increases, the amount of time for each range operation increases proportionally.

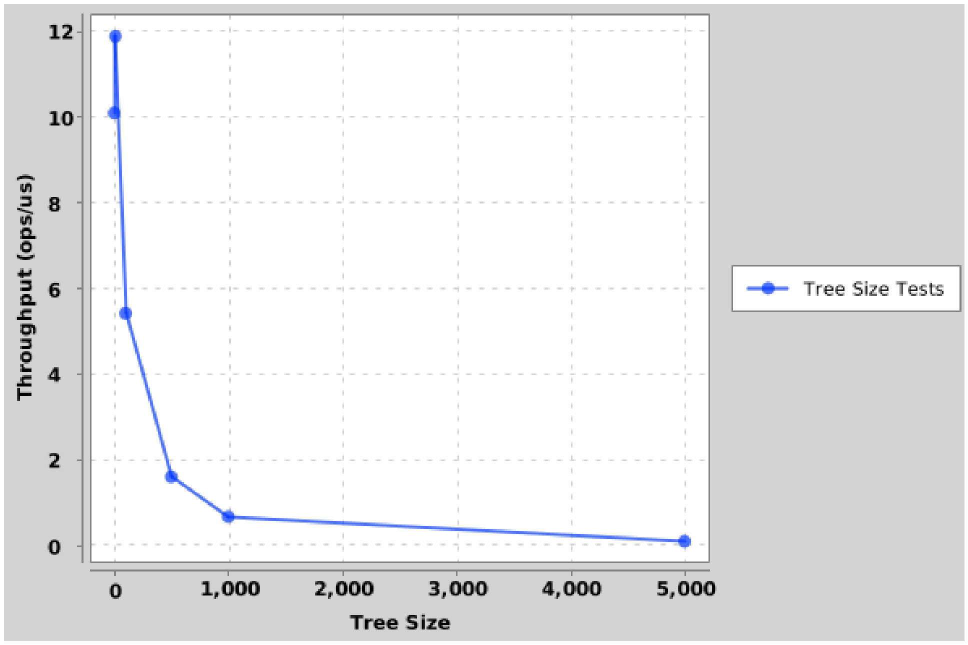

Figure 7: LFCAT throughput as a function of maximum AVL tree size. Tested on maximum tree sizes of 10, 100, 500, 1000, and 5000 under real-world load distribution using 16 threads.

7.3 Limiting AVL Tree Size

Another experiment was run which limited the maximum AVL tree size stored in the leaf nodes of the LFCAT. This test was performed because a significant degradation in throughput was observed when the AVL tree sizes were allowed to increase unbounded. Figure 7 quantifies this observation. Maximum AVL tree sizes of 10, 100, 500, 1000, and 5000 were tested. A peak is observed at a max AVL tree size of 100 then throughput quickly degrades from there. Additional tests could pinpoint where the optimum is achieved, but the optimum would vary with the load distributions so this analysis was not performed.

The reason the throughput degrades as the AVL tree size increases is because the AVL tree must be copied each time a leaf node is updated. In their implementation, the authors use a treap and claim an update could be performed in $O(\log n)$ time by only updating the nodes along the path from the root to the key being updated, but we could not see how this could be performed in place in either the treap or AVL tree without the need to copy the tree first. if multiple threads attempted to update the same AVL tree in a given leaf node, this could lead to a race condition and potential data corruption. The re-balancing of the AVL tree does not adversely affect the performance as this operation can be performed efficiently in $O(\log n)$ time.

7.4 STM Performance

Figure 8 shows the performance of a comparable binary search tree implemented using the Deuce Software Transactional Memory Library (Deuce-STM). Our Deuce-STM imple-

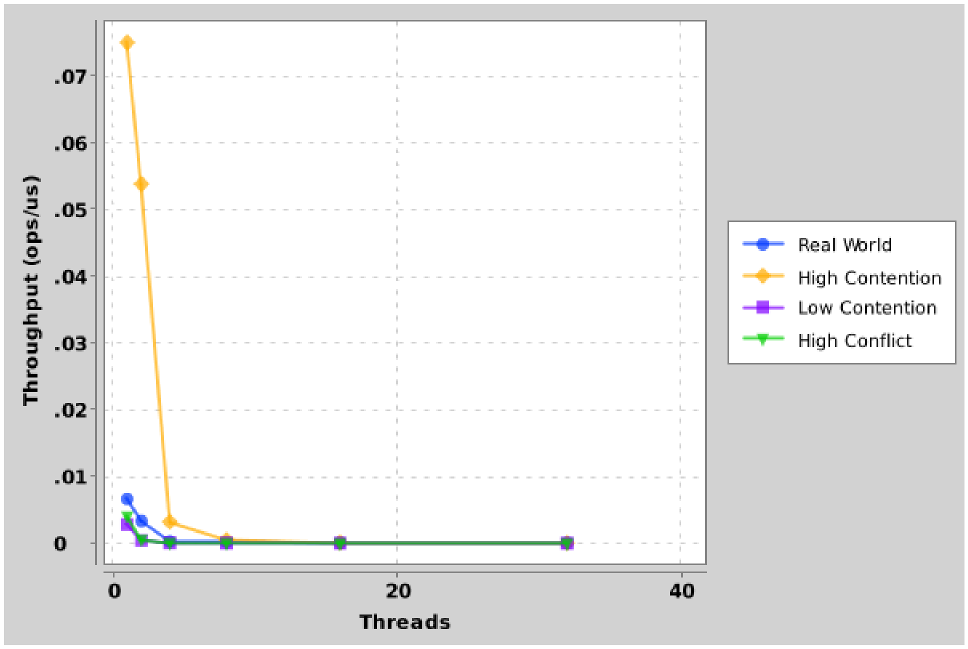

Figure 8: Performance of the Deuce-STM binary search tree for 1, 2, 4, 8, 16, and 32 threads for various operational loads.

mentation is essentially a binary search tree with insert, remove, contains, and range query support. It does not provide any of the contention adaptation methods described above, but rather lets the operations resolve as STM transactions. The algorithm performs best using a single thread and degrades as more threads are used.

The library does not efficiently perform synchronization between threads. As the number of threads increases, more operations overlap and must be restarted. Since the operation of updating and replacing an AVL tree in the leaf nodes is expensive, having to restart, re-copy, update, and re-balance the new tree any time a conflict is encountered tanks the performance. This is especially problematic for range queries which have a higher chance of conflicting with other operations and are the most expensive operation to restart. As more range queries are performed, the chances of a collision increase and more operations must be restarted, degrading performance. Increasing the transaction size would further reduce the performance because more conflicts would occur and these conflicts would be more expensive to restart.

Overall, STM is not a good approach for the LFCAT algorithm. The LFCAT is highly optimized for the task at hand and the STM library cannot automatically optimize the algorithm better than human efforts. Additionally, the contention adaptation presented in this paper is not well suited for STM libraries because split and join are specifically designed to minimize the contention dynamically whereas the Deuce-STM library cannot perform such adaptations, limiting the potential performance from the start.

8 Conclusion

This report presents an overview of the Lock-Free Contention Adaptive Search Tree along with an implementation in Java. The code is accessible at the repository linked in the abstract. The performance of our implementation was tested under varying operational loads and levels of concurrency. Our implementation comes close to the performance reported in the original paper. The differences can be attributed to the use of Java instead of C. Our implementation sacrifices the better performance for automatic memory management which simplified implementation. Overall, the LFCAT presents an efficient algorithm that performs well under both low-contention and high-contention conditions and can scale efficiently with additional processors. By adapting the structure of the tree to the local spatial and temporal contention, the LFCAT performs on par with comparable algorithms designed for specific scenarios and can significantly outperform those same algorithms for scenarios that are not optimized for. The LFCAT is an efficient solution to bridge the gap between fine-grain and course-grain synchronization approaches.

9 Future Work

Additional performance improvements could be achieved by optimizing the split and join thresholds and the amounts by which the statistic value is changed when low- and high-contention conditions occur. Other studies could investigate more complicated adaptation schemes that keep a larger history of contention adaptations that have recently occurred. For example, if the previous five adaptations in a region of the LFCAT have been splits, then on the sixth split adaptation the node could be preemptively split into a binary tree with four leaf nodes in an attempt to predict the next necessary adaptation.

References

[1] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[2] Konstantinos Sagonas and Kjell Winblad. Contention adapting search trees. pages 215–224, 06 2015.

[3] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. pages 121–132, 07 2018.

[4] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *OPODIS*, 2011.

[5] 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. *SIGPLAN Not.*, 52(8):357–369, January 2017.