A Specialized B-Tree data structures for Datalog Evaluation in Java

Louise Adolfsson
E17, Lund University, Sweden
lo6806ad-s@student.lu.se

Simon Tenggren
C16, Lund University, Sweden
si6187te-s@student.lu.se

Abstract

Today there is a high demand for Datalog engines to be able to handle large amounts of data quickly. By implementing a B-Tree in Java there can be significant improvements to its efficiency. In this paper we present and evaluate the Java implementation of a specialized B-Tree data structure. Our B-Tree handles parallel insertions and dense data. It uses a hint system for faster traversals in the B-Tree and has a optimistic read-write lock that allows for it to handle several threads at once efficiently. The B-Tree outperforms the TreeSet greatly and is just slightly better than the Concurrent SkipList Set in some cases. The B-Tree shows potential as a replacement for Java's data structures from the results of the benchmarks.

1 Introduction

Datalog is a declarative logic language powered by many different engines. Datalog is used in the context of database queries, static program analysis, network analysis and more. Because of these common use cases for the language, it requires highly efficient parallel data structures that can handle a very large amount of insertions and read operations concurrently. This due to the importance of achieving high performance in said engines with very large amount of data. Soufflé [1] is one of these engines which translate Datalog into highly optimized parallel C++ code. It has implemented new specialized data structures for this task. One of which is the Specialized B-Tree [8]. This data structure is available open-source under the Universal Permissive License. [2, 3] Currently, to the best of our knowledge, there are no open-source implementations of this data structure in Java. A Java implementation of this data structure would allow for current Datalog engines powered by Java, such as MetaDL[6], to improve their performance significantly. The paper [8] describes the performance improvement of the specialized data structure compared to previously used C++ alternatives as very significant. For example the B-Tree showed significant improvements in parallel insertion performance of up to 59 times compared to the C++ hashset during multi threaded benchmarks.[8]

Our Java implementation of the specialized B-Tree data structure is implemented with help from the C++ implementation as reference. The implementation is evaluated in the context of the Java Virtual Machine against currently used alternatives on sequential performance, parallel performance, and memory usage.

In short the contributions of this paper are the following:

- Describe the implementation of our specialized B-Tree data structure, with a focus on how it differs from the C++ implementation.
- Evaluate our implemented data structure and compare it to the currently used alternatives found in the Java Runtime Library, in the context of the Java Virtual Machine (JVM).

2 Parallel Datalog

Having Datalog handle several threads at once can improve the efficiency immensely. This is due the semi-naïve evaluation strategy employed by Datalog engines, allowing for evaluation to be performed in parallel without resulting in redundant computations.

2.1 B-Tree

A B-Tree of order k is a self-balancing search tree in which all the elements in each of the nodes are stored in sorted arrays of size k, see figure 1. The constraints on the B-Tree are the following:

1. A node may have up to $k + 1$ child nodes.
2. The root node has at least two children if it is not a leaf node.
3. An inner node with m children has at least $\lceil \frac{m}{2} \rceil$ elements.
4. An inner node with m children contains $m-1$ elements.
5. All leaf nodes appear at the same level in the structure.

There are three types of nodes in a B-Tree: leaf nodes, inner nodes, and the root node. All of which can change their type after a number of insertion operations are performed on the tree. For example, the root node starts of as a leaf node that can contain up to k different elements. When it is full it splits and births two new leaf nodes and itself changes its type to an inner node, see the top node in figure 1. Leaf nodes are the most basic nodes in the tree, see the bottom nodes in figure 1, they only contains the sorted array of elements. Inner nodes are nodes inside the tree which have an array of pointers to a number of other nodes (children), leaf nodes or other inner nodes, see the root node (top node) in figure 1. They also

Course paper, EDAN70, Lund University, Sweden
January 12, 2021.

carry a sorted array of elements. The elements in the nodes are arranged such that the left-most child node contains the elements that are less (defined by the comparator-function) than those in the first element in the parent. The second child node contains those elements that are greater than the first element, but less than the second element. This is continued for the amount of keys in the inner node, e.g. a node with two elements contain three children, like in figure 1. The first child contains elements that are lesser than all of the elements in the parent, see the leftmost bottom node in figure 1. The second child's elements are greater than the first element in the parent, but lesser than the second element in the parent, see the middle bottom node in figure 1. And finally the third child in which the elements are greater than those in the parent node, see the rightmost bottom node in figure 1.

An important part of the B-Tree that can affect its performance is the order of the tree, that is, the amount of elements that can be stored in each node. If the order of the B-Tree is increased it grows wider compared to a lower order. This reduces the height of the B-Tree and thus reduces the length of the traversal.

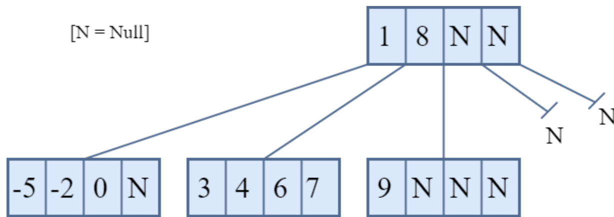

Figure 1. A B-Tree with several elements inserted.

2.2 Specialization of the B-Tree

For the task of Datalog evaluation several specialized optimizations are done to the data structure. These are the hint-system and the Optimistic-Read-Write locking mechanism. The hint-system allows for insertions and searches in the tree to be sped up, by caching previously visited nodes. The Optimistic-Read-Write lock is a system specially designed for the distinct phases of reading or writing during Datalog evaluation.

2.2.1 Optimistic Read-Write Lock

To handle parallel insertion, non-locking synchronization is put in place, the Optimistic read-write lock. The lock is inspired by the Linux Kernels heavy-read-seldom-write lock called Seqlocks.[9]. In a Seqlock, a reader thread keeps a version number (lease) of an atomic variable when it goes through the data structure. When the reader has read its data, it validates that data is still correct by making sure the lease number is the same as the lock's version number and that the version is not odd. If the lease cannot be validated the reader thread restarts its read operation. The writer waits until the version number is an even number then increments the version number to an odd number which ensures that all reader threads are invalidated if they try to perform a read operation while a concurrent write is ongoing. When the writer is done writing, it increments the version number to a even number and the reader threads can start trying to read again and other writer threads can start writing if there are any. Soufllé has made some changes to the lock and how it is used in the data structure. Each node in the tree has its own lock, and the root node has a dedicated lock because virtually each operation will need to access the root node. Threads in the B-Tree all start off as reader-threads and try to upgrade once they decide that a write is necessary. And thus can perform its tree traversal concurrently alongside with other threads until it has reached its destination node and wants to upgrade itself to a write thread.

2.2.2 Hints

The hint mechanism is a way to speed up searches in the tree, either when searching for where to place an element or when trying to find an element. The hint is a cache of the latest node that was reached either through an insertion or search in the tree. Whenever the next search or insertion operation is done on the tree the cached node is tested if the inserted or searched for element is covered by that node. If it is covered, the tree traversal up until that node can be skipped. If the element isn't covered by that node a normal tree traversal is done and the hint is updated. The hints also reduce the amount of nodes being locked. This is because when skipping some of the path in the tree traversal the nodes that otherwise would be traversed through are skipped, and thus won't be locked.

2.3 Differences between Java and the C++ reference

There exists some differences between our Java implementation and the C++ reference from Soufflé. Most of these are due the different features available in the respective languages. The C++ implementation made use of features not available in Java such as pointers, `sizeof`-operator, generic arrays, operator overloading, and how inheritance functions differently in C++.

C++ has some optimizations that cannot be achieved in Java. Theses are first and foremost operations utilizing the possibility of determining the sizes of objects at compile time using the `sizeof`-operator. It can therefore determine the size of the nodes and make optimizations concerning the order in the tree to optimize the number of keys per node and thus also the number of children per inner node. In Java you cannot easily determine the size of your objects at compile time and thus this optimization is not implemented. Instead the minimum order of the tree is set to three and the user can set it to be larger than that if they wish.

In C++ there is support for generic arrays, which Java does not allow. Instead a Java `Object` array is used with the size set to the maximum number of keys as a node is created. These arrays do not grow in size during execution and thus no resizing is done which would we be costly. However the arrays in C++ compared to `Object`-arrays have no overhead from range checking operations that are done with each access or insertion in the `Object`-array. Since the `Object`-arrays are objects in of themselves and not continuous memory, which causes another level of abstraction. Instead the elements are stored as pointers to other elements that might be allocated in different parts of the heap, which doesn't utilize spatial locality and can cause severe detriment to performance, especially when trying to find the correct position to traverse the tree or finding the correct insertion point in the `Object`-array. The B-Tree implements the `SortedSet` generic interface, which allows the user to supply any non-primitive data type as the type of key to be stored in the B-Tree.

3 Evaluation

We compare our B-Tree to the existing standard Java data structures that are suitable for use in Datalog implementations today. We look at:

- The performance of parallel insertion, membership test and iteration over the full structure.
- Memory usage

And compare the data structure against:

- Java's TreeSet (wrapped with Collections.synchronizedSet() where appropriate)
- Java's ConcurrentSkipListSet.

There are a lot of difficulties trying to test the performance of Java applications. There are several things that are non-deterministic during run time, mainly [5]:

- JIT Compilation
- Optimization in the VM
- Thread scheduling
- Garbage Collection
- Various System Effects

The benchmarks are run for two different stages of execution, start-up and steady-state. Start-up is during the start of execution on the JVM, here the JIT-compilation has not yet affected the performance to great degree, and thus has other bottlenecks such as class-loading. During steady-state the JIT compiler has performed several optimizations of the parts of the code that is deemed "hot", i.e. the most heavily used parts of the code, when the execution has reached steady-state it is deemed that the JIT compiler has performed most if not all of the possible optimizations and performance cannot be increased further.[5] As such they are naturally measured in different ways.

To test the B-Tree performance we developed a small C framework that creates VMs and run Java benchmarks of several operations on the B-Tree implementation and the currently used Java alternatives. Measurements are taken by measuring the wall clock time that it takes to execute the benchmarks.

The methodology of measuring both the start-up and steady-state performance of the benchmarks follows the steps specified in the article "Statistically Rigorous Java Performance Evaluation". [5]

3.1 Start-up performance

A good measurement of start-up performance is to see how long it takes to execute a short-running Java application, the shorter the better. As well as the time for class loading. The steps to measuring the start-up performance are the following:

1. Measure the execution time over several VM invocations. Every invocation are running one benchmark iteration. At least 30 measurements are taken.
2. Create a confidence interval for a given confidence level. (In our case 95%)

3.2 Steady-state

Steady-state performance measurement is to see how fast a long-running Java application can be executed. The faster the better. However steady-state performance have two issues to consider[5, 10]:

- The time it takes an application to reach the steady-state can vary between applications. In some cases it may take very long time to reach the steady-state.
- The steady-state performance can vary between VM invocations, due to different methods may be optimized at different levels across the different invocations.

To handle there issues we are

1. Running several VM invocations where each have multiple benchmark iterations run.
2. For every VM invocation determining the iteration where steady-state performance is reached.
3. Calculating the mean of the iterations where steady-state performance is reached.
4. Computing a suitable confidence interval for a given confidence level from the calculated means from each VM invocation. (95%)

All benchmarks have the following parts in common:

- Utilizes the same comparator-function.
- Inserts the same elements.
- In the case of random insertions or searches the same seed is used for the random number generator.
- The elements inserted are all long-tuples of size two or ten.

Benchmark	start-up	steady-state
Parallel In-Order Insertion	X	X
Parallel In-Order Search (Full-Scan)	X	X
Parallel Randomized Insertion	X	X
Parallel Randomized Search (Full-Scan)	X	X
Sequential In-Order Insertion	X	
Sequential Randomized Insertion	X	
Sequential In-Order Search (Full-Scan)	X	X
Sequential Randomized Search (Full-Scan)	X	

Table 1. Benchmarks ran for all the datastructures.

- Multithreaded benchmarks are run with six dedicated threads for operations on the data structures.

Due to non-deterministic elements such as thread scheduling the elements might appear in different threads in different benchmarks and iterations of those benchmarks when using the random engine. If each thread would have their own random engine, it is not possible to ensure that the same elements are inserted during each of the benchmarks iterations, since thread scheduling would give each thread a non-deterministic amount of time for each thread if there are more threads than cores on the machine. The B-Tree is evaluated at four different orders (number of elements per node) the default value of 3 as well as 8, 16, and 32.

4 Results

The benchmarks that has been run for the data structures can be seen in table 1. We also measured the heap usage of 10 thousand, 100 thousand, and 1 million insertions on the data structures. The benchmarks were run on consumer grade computer with the following specifications:

- CPU: AMD Ryzen 7 3700X 3.6GHz
- RAM: 16GB DDR4 3200Mhz
- JVM Initial Heap size of 256MB and a maximum heap size of 1GB, according to the default heap size settings of the JVM. [4]

4.1 Steady-State Performance

The B-Tree performed very well during the steady-state performance tests and outperformed the TreeSet in all tests. The SkipList however remains just slightly ahead in some instances. 20 million tuples with two elements each were used in the search tests and 24 million tuples with the same amount of elements for the insertion tests, see figures 2, 3, 4, 5. There were also test made with a little over 25 million (exactly 25 165 824) bigger tuples of size 10, see figure 7, 8, 9, 10. These test were mostly done with only the SkipList and the B-Tree of order 32 since the other tests from steady-state has proven uninteresting since they are greatly outperformed by both the SkipList and the B-Tree of order 32. When the tuple size gets larger the cost of comparisons in the data structure gets higher on average. The data structures that then perform more comparisons will then see a large performance hit than the those who perform fewer.

It is in the in-order searches and insertions, see figure 3, 7, we see the power of the implemented hints. When each thread has their own dedicated hints they can frequently reuse the same node they just traversed to, and with an increasing order, the chance of a hit increases since each node covers more nodes, and contains more keys. However when the order is higher the cost of traversing the tree is also a lot less, since the tree is wider instead adding more height.

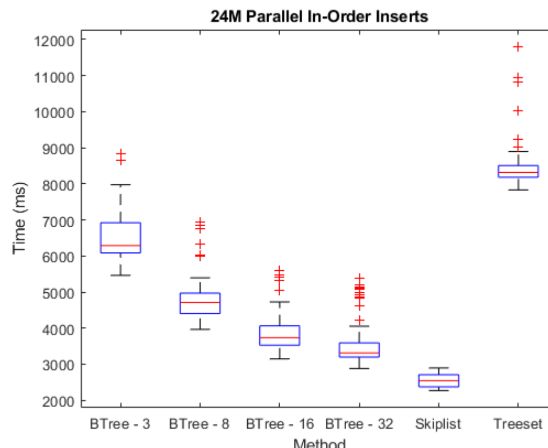

Figure 2. Parallel insertion with 24M tuples of size 2 with the different data structures during steady-state performance.

4.2 Start-up Performance

In start-up performance the B-Tree did fairly poorly. Both the TreeSet and the SkipList outperformed the B-Tree but the SkipList only did so slightly see figures 11,12. However start-up performance is not crucial to the B-Tree since it is specialized to handle large amounts of data, and thus the applications integrating the B-Tree is concerned with how well it performs during steady-state.

5 Memory Usage

Memory usage was measured by examining the heap usage for the data structures after ten thousand, one hundred thousand, and one million insertions in the different data structures. The heap usage is measured by using the Java library function `java.lang.Runtime.freeMemory()` which approximates the amount of free memory before and after the insertions are performed. The SkipList and TreeSet used

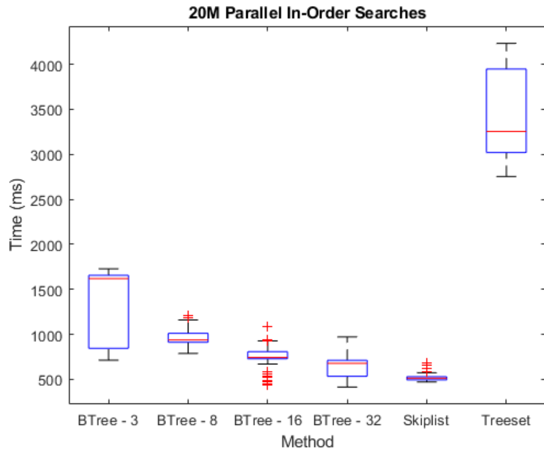

Figure 3. Parallel search with 20M tuples of size 2 with the different data structures during steady-state performance.

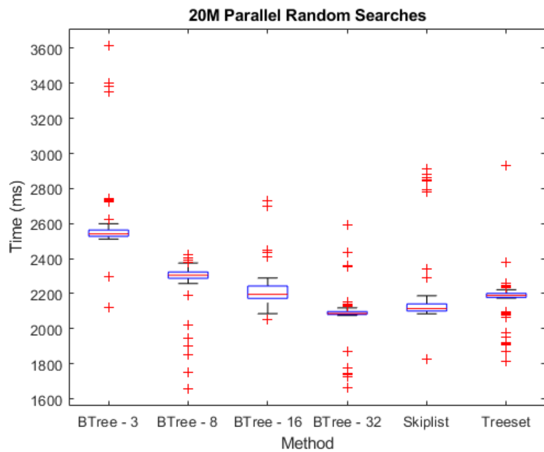

Figure 4. Parallel search with 20M tuples of size 2 with the different data structures during steady-state performance.

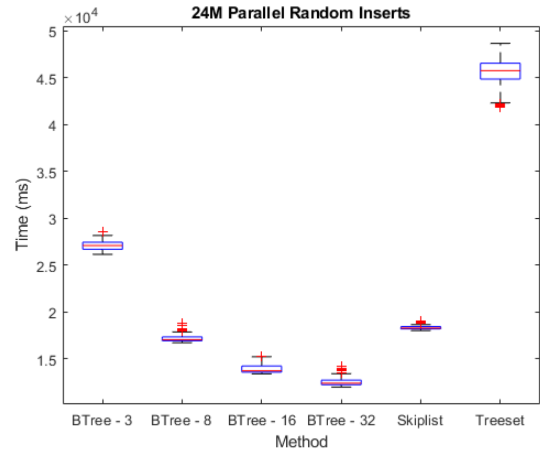

Figure 5. Parallel random insertion with 24M tuples of size 2 with the different data structures during steady-state performance.

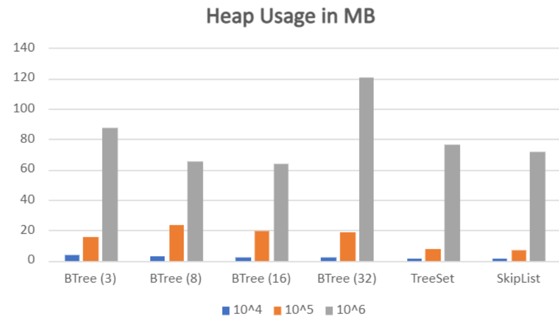

Figure 6. Heap Usage in Megabytes for the different data structures during steady-state performance.

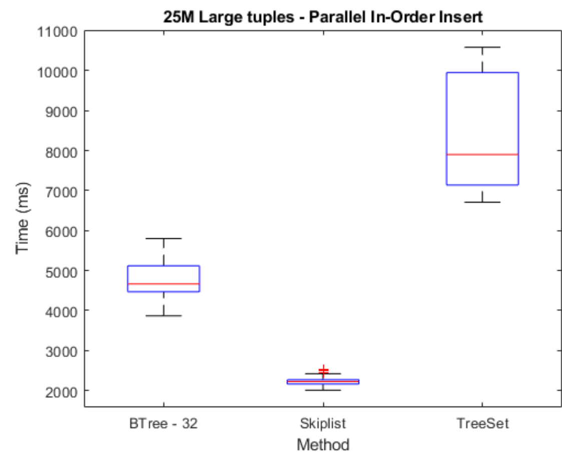

Figure 7. Parallel in-order insertion with 25M tuples of size 10 with the different data structures during steady-state performance.

almost the same amount of memory at all numbers of insertions. While the B-Tree of orders 8 and 16 used less for one million insertions, but at order 32 the heap usage was significantly larger than both the TreeSet and SkipList.

5.1 A Quick Summary of Skip Lists

The SkipList performed very well in the steady-state benchmarks usually performing equal to the order 32 B-Tree or better in the In-Order tests. A SkipList is a probabilistic data structure usually implemented as a sorted linked-list. The SkipList has several different levels of "express-lanes" where there are pointers to different parts of the lists, where the higher level has the largest "gaps" and allows to skip more of the list. The bottom layer is the actual sorted linked-list. The data structure is probabilistic because the pointers in the levels that can be used to skip forward in the list are created

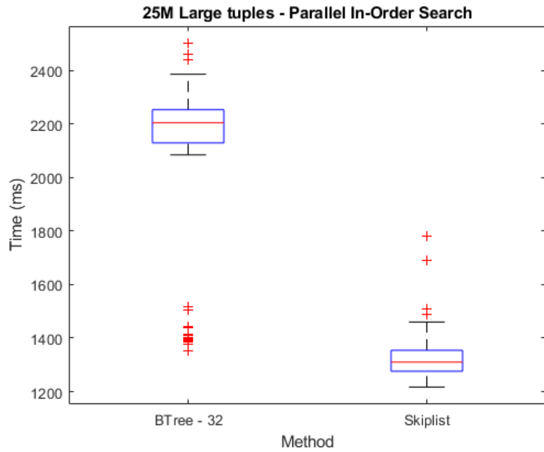

Figure 8. Parallel search with 25M tuples of size 10 with the different data structures during steady-state performance.

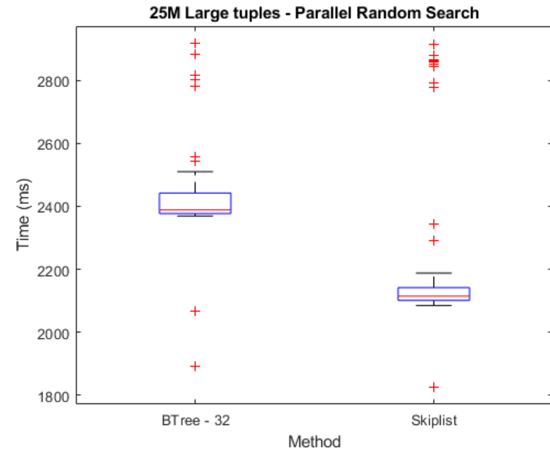

Figure 10. Parallel random search with 25M tuples of size 10 with the different data structures during steady-state performance.

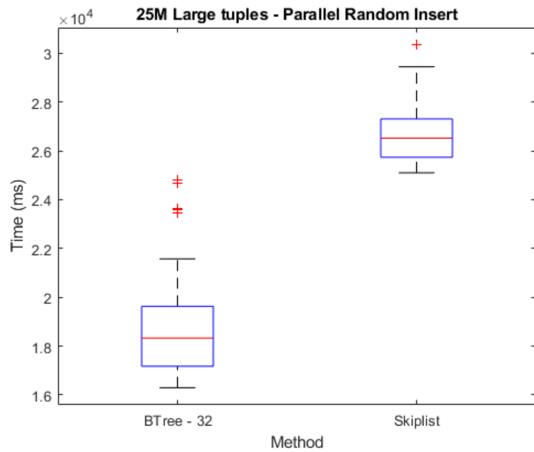

Figure 9. Parallel random insertion with 25M tuples of size 10 with the different data structures during steady-state performance.

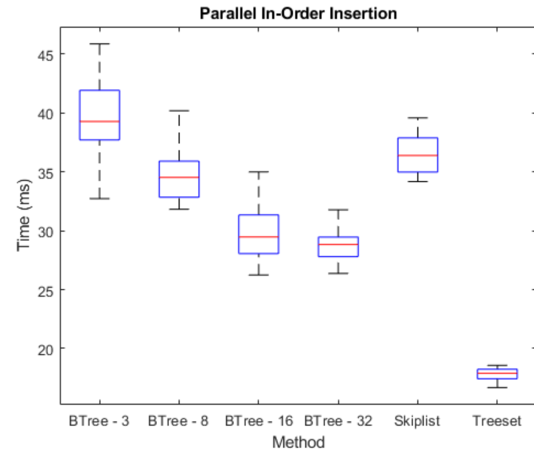

Figure 11. Parallel insertions in the different data structures during start-up performance.

with a set probability. The SkipList has amortized insertion and search complexity of `O(log(n))`, which is the same as the Tree-Set and B-Tree. However it uses much less space comparatively making it great alternative compared to the regularly used Tree sets.

When inserting an element into a SkipList the higher levels are traversed through first, until the inserted element is not larger than the next element in the lane or next node in the lane points to null. When this occurs the next lower level lane is used and the process is repeated. This explains the great performance in the in-order insertion for the SkipList. Since the elements are inserted in order, the list can skip until the end of the list very fast using the higher level express-lanes and very seldom use the lower-level lanes. However in the random insertions these lanes aren't as useful, since the element can appear somewhere in the beginning of the list and the higher level express lanes cannot be utilized by jumping a large gap forward in the list.

6 Discussion

The results showed that the B-Tree outperformed the TreeSet in steady-state performance while being a contender to the concurrent SkipList, outperforming it during random parallel insertion but doing slightly worse in the in-order tests. When concerning the start-up performance the B-Tree was outperformed by both. As stated above, the multi threading only ran on six dedicated threads for operations on the data structures, and benchmarks were not ran for varying number of threads, the results from the B-Tree implemented in Soufflé showed that a higher number of threads resulted in a

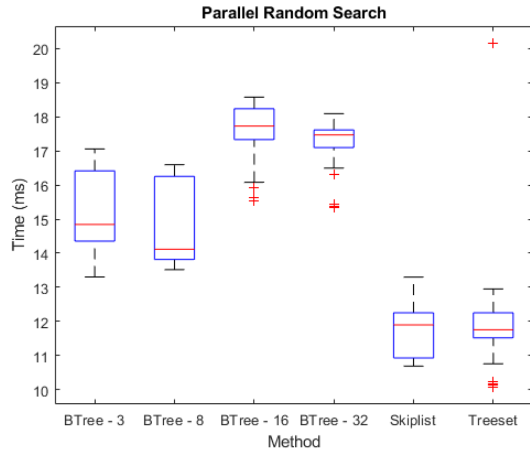

Figure 12. Parallel Search with the different data structures during start-up performance.

larger number of insertions per second for the B-Tree. This might also hold true for the Java implementation. To give this a fair evaluation the benchmarks would have to be reran on a machine with a CPU with several more hardware cores. The authors were limited to a consumer PC with a CPU with eight dedicated hardware threads, and decided to leave two of these threads dedicated to various system effects such as garbage collection.

The order of the B-Tree significantly changed the performance of the data structure, showing increasingly better results when using an higher order in most benchmarks, however with some diminishing returns. E.g. the performance difference between the mean execution time for random parallel insertion between order 3 and 8 were 65% while changing the order from 16 to 32 only showed a performance increase of 9%, see figure 5. This suggest that there is an optimal order that remains to be found for a given element size. This would probably have to be determined by experimentation by using different sizes of elements at different orders of the B-Tree.

The memory usage of the B-Tree was surprisingly large after one million insertions compared to the TreeSet and the SkipList. Especially since the memory usage at the lower orders were significantly lower. At one million insertions the B-Tree at order 32 used significantly more heap space. See figure 6. Why the memory of the large B-Tree is so large is probably due the pre-allocated pointers in the nodes for both keys and children that are yet to be assigned. For example a child with only a few keys inserted at an high order would have most of its allocated data unused, this is especially apparent in the benchmark that was ran since the elements were relatively small, being only an array of length 2 of long-values.

The B-Tree has not yet been profiled for either execution or memory optimizations, and bottlenecks for both could probably be found.

Soufllé compared their implementation of the B-Tree against different data structures, namely a synchronized hashset, and other available B-Trees. It might be that a C++ implementation of the concurrent SkipList would give great results in their benchmarks as well.

7 Threats to Validity

All benchmarks were run on the Windows Subsystem for Linux 2 on a personal computer running several other applications in the background. The authors made an effort to minimize the number of other applications running during the benchmarks, however this couldn't be completely controlled on the machine where several hidden processes might be running during the execution of the benchmarks.

8 Related work

Soufflé [1] is an Datalog engine that created a B-Tree and Brie implementation. These turned out successful and gave life to papers about improved versions of them [7, 8]. In [8] they developed a specialized B-Tree with features like an optimistic locking protocol and a hint system that re-use the results from previous tree traversals. Another interesting data structure is the Brie. In [7] they developed a specialized Trie, the Brie. It's a specialized data structure for storing high amounts of dense data. They showed a improvement of up to 59 x (B-Tree) respectively 15 x (Brie) higher performance than state-of-the-art industrial standard data structures, but also when integrated with a Datalog engine they would have an overall system performance increase of 3 x higher when using the B-Tree, respectively 4 x higher for the Brie and with a compression ratio of 3.6 x when running a point-to analysis.

This shows the great potential of the B-Tree and Brie and why it is interesting to develop them in Java and see if that could improve the performance further.

9 Conclusion

The B-Tree shows potential for being a suitable replacement as a data structure for Datalog evaluation. Outperforming the TreeSet in all steady-state benchmarks, and performing better than the Concurrent SkipList Set in randomized parallel insertion and ties with the SkipList in several benchmarks only being outperformed in the in-order tests as well as random searches at larger tuple sizes. As of right now the B-Tree has to be profiled to find eventual performance bottlenecks, and should not be a replacement to the SkipList as is. The B-Tree should also go through further benchmarks on a computer equipped with a CPU with several more cores to examine the eventual performance increase of insertions that was seen in the C++ implementation of the B-Tree. It might

be the case as it was in C++ that the other data structures have a performance cap that is reached at a lesser amount of threads compared to the B-Tree.

Acknowledgments

The authors would like to acknowledge our supervisor Alexandru Dura, who is also developing MetaDL which utilizes Datalog to bring static program analysis to the masses. The authors would also like to acknowledge Professor Görel Hedin for a great introduction to the wonderful world of compilers.

References

[1] 2020. Soufflé : A Datalog Synthesis Tool for Static Analysis. (2020). https://souffle-lang.github.io/
[2] 2020. Specalized B-Tree source code. (2020). https://github.com/souffle-lang/souffle/blob/master/src/include/souffle/datastructure/BTree.h
[3] 2020. Universal Permissive License (UPL). (2020). https://github.com/souffle-lang/souffle/blob/master/LICENSE
[4] 2021. Default Heap Size, Oracle Documentation. (2021). https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#default_heap_size
[5] Georges Andy, Buytaert Dries, and Eeckhout Lieven. 2007. Statistically Rigorous Java Performance Evaluation. *In OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications* (2007), 57–76. https://dl.acm.org/doi/10.1145/1297027.1297033
[6] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *SOAP 2019 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. Association for Computing Machinery (ACM), United States, 38–43. https://doi.org/10.1145/3315568.3329970 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2019 ; Conference date: 22-06-2019 Through 26-06-2019.
[7] Jordan Herbert, Suboti´c Pavel, Zhao David, and Scholz Bernhard. 2019. Brie: A Specialized Trie for Concurrent Datalog. *in PMAM'19: Programming Models and Applications for Multicores and Manycores* (2019), 10. https://doi.org/10.1145/3303084.3309490
[8] Jordan Herbert, Suboti´c Pavel, Zhao David, and Scholz Bernhard. 2019. A Specialized B-tree for Concurrent Datalog Evaluation. *in PPoPP'19: Symposium on Principles and Practise of Parallel Programming* (2019), 13. https://doi.org/10.1145/3293883.3295719
[9] Christoph Lameter. 2005. Effective Synchronization on Linux/NUMA Systems. *Gelato Conference* (2005). https://mirrors.edge.kernel.org/pub/linux/kernel/people/christoph/gelato/gelato2005-paper.pdf
[10] Blackburn M. Stephen, McKinley S. Kathryn, Garner Robin, Hoffman Chris, Khan M. Asjad, Bentzur Rotem, Diwan Amer, Fienberg Daniel, Frampton Daniel, Guyer Z. Samuel, Hirzel Martin, Hosking Anthony, Jump Maria, Lee Han, Moss B. Eliot J., Phanasalkar Aashish, Stefanovik Darko, VanDrunen Thomas, von Dincklage Daniel, and Wiedermann Ben. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* (2008). https://doi.org/10.1145/1378704.1378723