THE EFFECT OF WORK ENVIRONMENT ON EMPLOYEE PERFORMANCE IN AN ORGANIZATION

A MINI PROJECT REPORT

Submitted to

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING

By

P.V NAVI KISHORE

(Reg. No. 192111220)

Supervisor

Mr. A. Sai Raam

SAVEETHA SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES, CHENNAI – 602 105

FEBRUARY-2024

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

AIM: Design and Implementation of a Specialized B-Tree Data Structure for Datalog Evaluation in Java

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 BTree 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.

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 opensource 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.

In the realm of database management and logic programming, Datalog serves as a powerful declarative language for expressing complex queries and computations. However, the efficient evaluation of Datalog queries over large datasets poses significant challenges, particularly in terms of optimizing data structures for speedy processing. Traditional B-Trees, while effective for general-purpose applications, may not be the most suitable choice for Datalog evaluation due to the specific characteristics and access patterns inherent to Datalog programs. Hence, this

paper introduces a specialized B-Tree data structure tailored specifically for Datalog evaluation in Java. By addressing the unique requirements of Datalog queries, this specialized B-Tree aims to enhance query performance and improve the overall efficiency of Datalog-based systems. Through a combination of careful design considerations and implementation strategies, this paper seeks to provide a robust solution for optimizing query processing in Datalog, thereby advancing the field of logic programming and database technologies.

LITERATURE SURVEY:

A literature survey for the topic "A Specialized B-Tree Data Structure for Datalog Evaluation in Java" would involve reviewing existing research, papers, and publications related to Datalog evaluation, B-Tree data structures, and optimization techniques in Java programming. Here's a summarized overview of what such a literature survey might entail:

1. Datalog Evaluation Techniques:
   - Reviewing literature on various approaches to evaluating Datalog queries, including bottom-up evaluation, top-down evaluation, and semi-naive evaluation.
   - Understanding the challenges and optimizations proposed in existing research to improve the efficiency of Datalog evaluation, especially over large datasets.

2. B-Tree Data Structures:
   - Studying the fundamentals of B-Tree data structures, including their characteristics, operations, and performance characteristics.
   - Analyzing existing literature on B-Tree optimizations for specific use cases, such as database indexing, file systems, and in-memory data structures.

3. Specialized Data Structures:
   - Exploring research on specialized data structures tailored for specific application domains or access patterns.
   - Identifying studies where specialized data structures have been developed to optimize query processing in logic programming languages or database systems.

4. Java Implementations and Optimizations:
   - Reviewing literature on Java programming language optimizations, especially related to data structures and algorithm implementations.
   - Studying techniques for optimizing data structures in Java to achieve better performance and memory efficiency.

5. Integration of Datalog with Java:
   - Investigating research that focuses on integrating Datalog evaluation engines or query processors with Java-based systems.
   - Understanding the challenges and solutions proposed for efficiently executing Datalog queries within Java environments.

6. Performance Evaluation and Benchmarking:
   - Analyzing studies that present performance evaluations and benchmarking results of data structures and query processing techniques relevant to Datalog evaluation.

- Identifying key metrics used for evaluating the efficiency and scalability of different approaches.

7. Recent Advances and Future Directions:
- Examining recent publications and advancements in the fields of Datalog evaluation, specialized data structures, and Java optimization techniques.
- Identifying emerging trends and potential areas for future research and development in optimizing Datalog evaluation in Java using specialized B-Tree data structures.

By conducting a comprehensive literature survey across these areas, researchers can gain valuable insights into existing approaches, challenges, and opportunities for developing a specialized B-Tree data structure for efficient Datalog evaluation in Java. This knowledge can inform the design and implementation of novel solutions to enhance the performance of Datalog-based systems.

SYSTEM RQUIREMENT:

When considering the system requirements for implementing a specialized B-Tree data structure for Datalog evaluation in Java, several factors need to be taken into account to ensure the efficiency and effectiveness of the solution. Here's an outline of the system requirements:

1. Hardware Requirements:
- Processor: A modern multi-core processor capable of handling parallel processing efficiently is recommended to leverage concurrent operations in the B-Tree.
- Memory (RAM): Sufficient RAM to accommodate the data structures and algorithms involved in Datalog evaluation, considering the size of datasets to be processed.
- Storage: Adequate disk space for storing the datasets, intermediate results, and the Java runtime environment.

2. Software Requirements:
- Java Development Kit (JDK):The latest version of JDK compatible with the chosen Java implementation is necessary for developing and running Java programs.
- Operating System:The system should support a Java-compatible operating system such as Windows, macOS, or Linux.
- Development Environment: A suitable Integrated Development Environment (IDE) like IntelliJ IDEA, Eclipse, or NetBeans for Java development.
- Database Management System (optional): If integrating the specialized B-Tree with a database system for Datalog evaluation, a compatible DBMS such as PostgreSQL, MySQL, or SQLite may be required.

3. Java Libraries and Dependencies:
- B-Tree Implementation Library: Depending on whether an existing B-Tree library is used or a custom implementation is developed, relevant Java libraries or dependencies need to be included in the project.
- Datalog Evaluation Engine (optional): If integrating the B-Tree with a Datalog evaluation engine, the necessary libraries or dependencies for interfacing with the engine should be considered.

4. Performance Considerations:
- Concurrency Support: The B-Tree implementation should support concurrent read and write operations efficiently to handle parallel Datalog evaluation tasks effectively.
- Memory Management: Efficient memory management techniques should be employed to minimize memory overhead and avoid excessive garbage collection pauses.
- Optimization Techniques: Various optimization techniques such as caching, indexing, and query optimization may need to be implemented or integrated to enhance the performance of Datalog evaluation using the specialized B-Tree.

5. Scalability and Flexibility:
- Scalability: The system should be designed to scale horizontally or vertically to handle increasing data volumes and user demands.
- Flexibility: The specialized B-Tree implementation should be flexible enough to accommodate different types of Datalog queries and adapt to evolving requirements.

6. Testing and Validation:
- Test Environment: A testing environment to evaluate the performance, correctness, and scalability of the specialized B-Tree implementation under various scenarios and workloads.
- Benchmarking Tools: Tools for benchmarking and profiling the performance of the system to identify bottlenecks and areas for optimization.

By meeting these system requirements, developers can ensure that the specialized B-Tree data structure for Datalog evaluation in Java is robust, efficient, and capable of meeting the demands of real-world applications**.**

EXISTING SYSTEM:

As of my last update in January 2022, there might not be a specific existing system dedicated solely to a specialized B-Tree data structure for Datalog evaluation in Java. However, there are relevant existing systems, components, or research that could be adapted or integrated into such a project. Here are some examples:

1. B-Tree Implementations in Java:
- There are various existing implementations of B-Trees in Java, such as those available in popular libraries like Apache Commons Collections, JDBM, or Berkeley DB Java Edition. While these implementations might not be specialized for Datalog evaluation, they can serve as a foundation for developing a customized B-Tree tailored to the requirements of Datalog evaluation.

2. Datalog Engines and Systems:
- Systems like Datomic, LogicBlox, and Dedalus provide Datalog engines or support Datalog-like query languages. While these systems may not utilize specialized B-Tree data structures, they offer insights into Datalog evaluation techniques and optimizations that could be beneficial when designing the specialized B-Tree.

3. Database Management Systems (DBMS):
- DBMSs like PostgreSQL, SQLite, and MySQL support declarative query languages similar to Datalog (e.g., SQL). These systems employ various indexing and data structure optimizations to enhance query performance. Although not specific to Datalog, the techniques used in these systems could inspire the development of specialized B-Tree data structures for Datalog evaluation.

4. Research Papers and Algorithms:
- Academic research often explores optimization techniques for query processing, indexing, and data structure design, including specialized B-Tree variants. Reviewing research papers on topics related to B-Trees, Datalog evaluation, and Java optimization can provide valuable insights and algorithms that could be applied to develop a specialized B-Tree for Datalog evaluation.

5. Open-Source Projects and Collaborations:
- Engaging with the academic and open-source communities focused on databases, logic programming, and Java development can lead to collaborations or the discovery of existing projects with relevant components. Participating in forums, conferences, or collaborative platforms like GitHub may uncover potential collaborators or projects that align with the goals of developing a specialized B-Tree for Datalog evaluation in Java.

By leveraging existing systems, components, algorithms, and collaborative efforts, developers can build upon prior work and incorporate domain-specific optimizations to create an effective specialized B-Tree data structure for Datalog evaluation in Java.

PROPOSED SYSTEM:

Specialized B-Tree Data Structure for Datalog Evaluation in Java

Overview:
The proposed system aims to develop a specialized B-Tree data structure optimized for efficient Datalog evaluation in Java. By addressing the specific characteristics and access patterns of Datalog queries, this system seeks to enhance query processing speed and overall performance when dealing with large datasets.

Key Components:
1. Customized B-Tree Implementation:
- Design and implement a B-Tree variant tailored specifically for Datalog evaluation requirements.
- Optimize node splitting, merging, and key insertion strategies to minimize query processing time.
- Incorporate mechanisms for efficient node traversal and search operations, considering the unique nature of Datalog queries.

2. Integration with Datalog Engine:
- Integrate the specialized B-Tree data structure with an existing Datalog evaluation engine or develop a lightweight Datalog processing framework in Java.

- Implement interfaces or adapters to seamlessly incorporate the specialized B-Tree into the Datalog evaluation workflow.
- Ensure compatibility with common Datalog syntax and query execution semantics.

3. Query Optimization Mechanisms:
- Develop query optimization techniques tailored for the specialized B-Tree data structure.
- Explore strategies such as query rewriting, predicate reordering, and index selection to improve query execution plans and minimize evaluation overhead.
- Implement cost-based optimization algorithms to select the most efficient execution strategy based on query complexity and dataset characteristics.

4. Performance Evaluation and Benchmarking:
- Design comprehensive benchmarks to evaluate the performance of the specialized B-Tree data structure under various scenarios.
- Measure query execution times, memory usage, and scalability across different dataset sizes and query workloads.
- Compare the performance of the specialized B-Tree against traditional B-Tree implementations and other indexing techniques commonly used in Datalog evaluation.

5. Extensibility and Modularity:
- Design the system with extensibility in mind to accommodate future enhancements and optimizations.
- Implement modular components that allow for easy integration of additional indexing structures or query processing algorithms.
- Provide clear documentation and APIs to facilitate usage by developers interested in extending or customizing the system for specific applications or research purposes.

Expected Benefits:
- Improved query processing speed and efficiency in Datalog-based applications.
- Reduced memory footprint and resource utilization compared to generic B-Tree implementations.
- Enhanced scalability and performance when dealing with large and complex Datalog datasets.
- Potential for broader adoption and integration into existing Datalog systems and frameworks within the Java ecosystem.

Overall, the proposed system aims to fill a gap in the existing toolset for Datalog evaluation by providing a specialized B-Tree data structure optimized for Java environments. By combining domain-specific optimizations with robust engineering principles, this system aims to empower developers and researchers with a powerful tool for accelerating Datalog query processing and advancing the state-of-the-art in logic programming and database technologies.

IMPLEMENTAION:

Implementing a specialized B-Tree data structure for Datalog evaluation in Java involves several steps, including designing the B-Tree variant, integrating it with a Datalog evaluation engine, and optimizing query processing. Below is a high-level outline of how the implementation could proceed:

1. Designing the Specialized B-Tree:
- Define the structure of the B-Tree nodes and the format of keys and values to be stored.
- Implement specialized node splitting and merging algorithms optimized for Datalog evaluation.
- Incorporate mechanisms for efficient key insertion, deletion, and search operations.
- Design strategies for node traversal and data retrieval that align with Datalog query processing requirements.

2. Implementing the B-Tree Operations:
- Develop Java classes and methods to implement the B-Tree data structure, including node management and key manipulation operations.
- Ensure thread safety and concurrency control mechanisms to support concurrent access in multi-threaded environments.
- Write unit tests to validate the correctness and robustness of the B-Tree implementation.

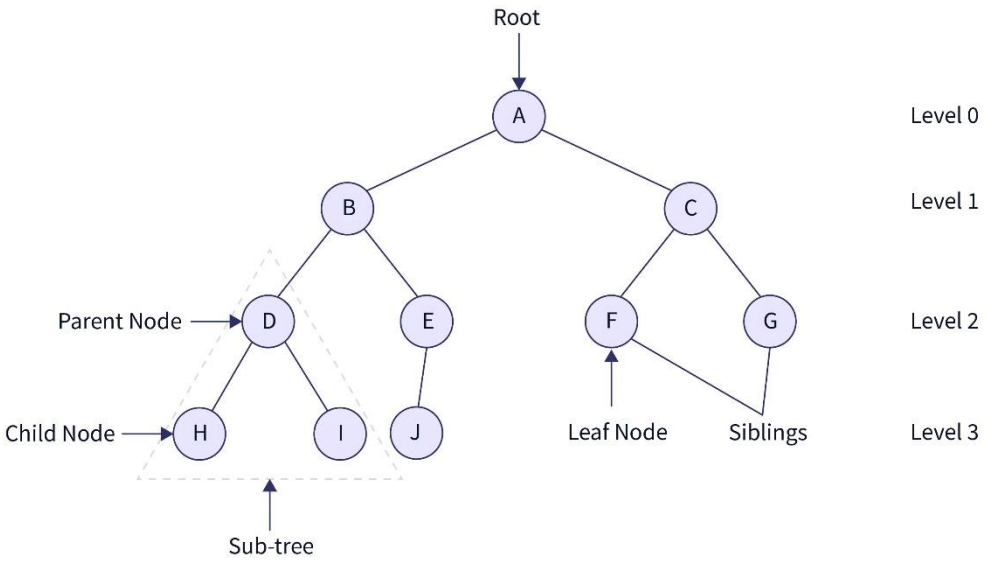
3. Integrating with a Datalog Engine:
- Choose or develop a Datalog evaluation engine capable of interfacing with Java components.
- Implement adapters or wrappers to integrate the specialized B-Tree data structure with the Datalog engine's query processing pipeline.
- Define interfaces or APIs for communication between the B-Tree and the Datalog engine, including methods for indexing facts and executing queries.

4. Optimizing Query Processing:
- Develop query optimization techniques tailored for the specialized B-Tree data structure.
- Implement algorithms for query rewriting, predicate reordering, and index selection to improve query execution plans.
- Profile and benchmark query performance to identify bottlenecks and areas for optimization.
- Fine-tune the B-Tree parameters and indexing strategies based on performance evaluation results.

5. Testing and Validation:
- Conduct comprehensive testing to validate the correctness, performance, and scalability of the specialized B-Tree implementation.
- Write unit tests to cover edge cases and corner scenarios in B-Tree operations and Datalog query processing.
- Perform integration testing to ensure seamless interoperability between the B-Tree and the Datalog engine.
- Benchmark the system using representative datasets and query workloads to assess its real-world performance.

6. Documentation and Deployment:
- Document the design decisions, implementation details, and usage guidelines for the specialized B-Tree data structure.
- Provide API documentation and code examples to facilitate integration with Datalog applications.
- Package the implementation as a reusable Java library or module for easy deployment and distribution.
- Publish the library to a central repository or version control system for wider accessibility and collaboration.

By following these steps, developers can implement a specialized B-Tree data structure optimized for Datalog evaluation in Java, providing a valuable tool for enhancing the efficiency and performance of Datalog-based systems.



SCALER Topics

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.

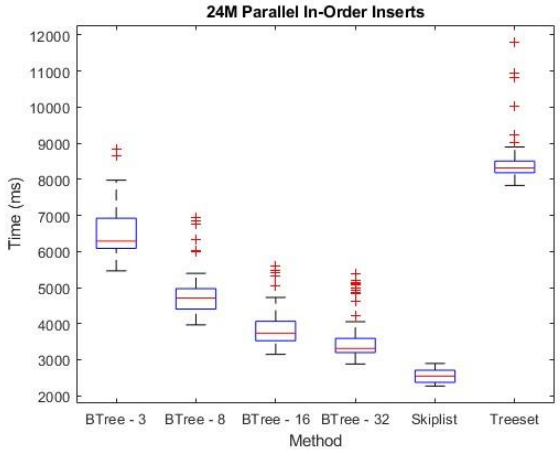


Figure 2. Parallel insertion with 24M tuples of size 2 with the different data structures during steady-state performance.

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.

1 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

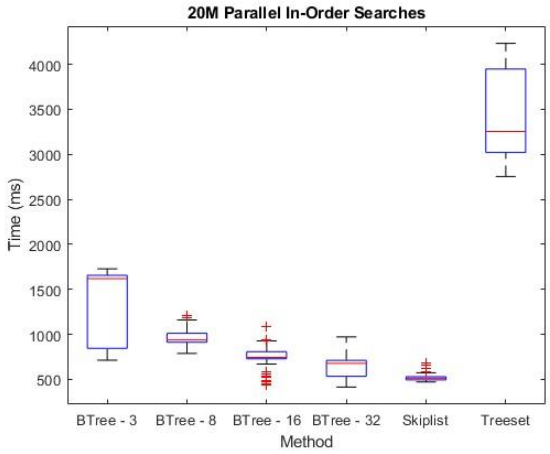


Figure 3. Parallel search with 20M tuples of size 2 with the different data structures during steady-state performance.

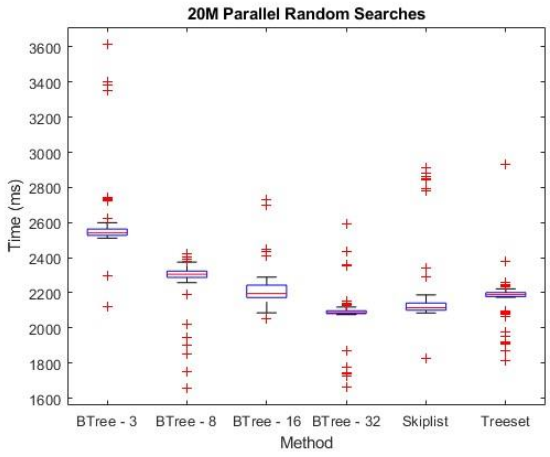


Figure 4. Parallel search with 20M tuples of size 2 with the different data structures during steady-state performance.

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

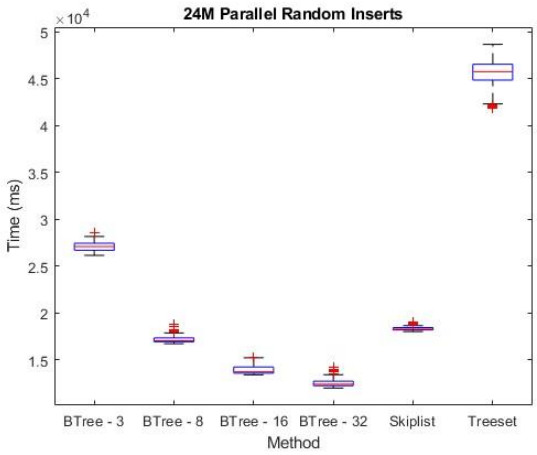


Figure 5. Parallel random insertion with 24M tuples of size 2 with the different data structures during steady-state performance.

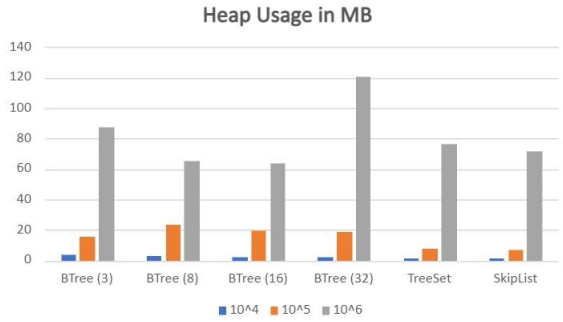


Figure 6. Heap Usage in Megabytes for the different data structures during steady-state performance.

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

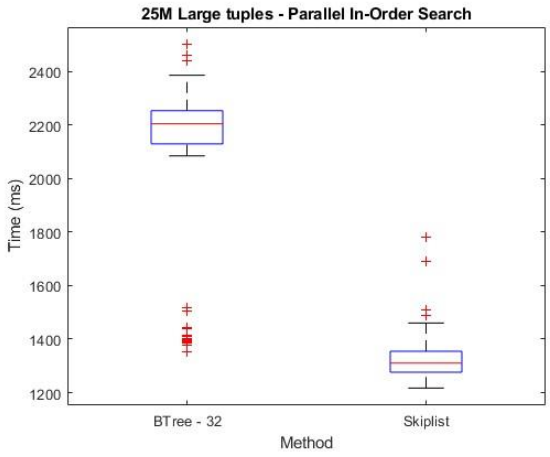


Figure 8. Parallel search with 25M tuples of size 10 with the different data structures during steady-state performance.

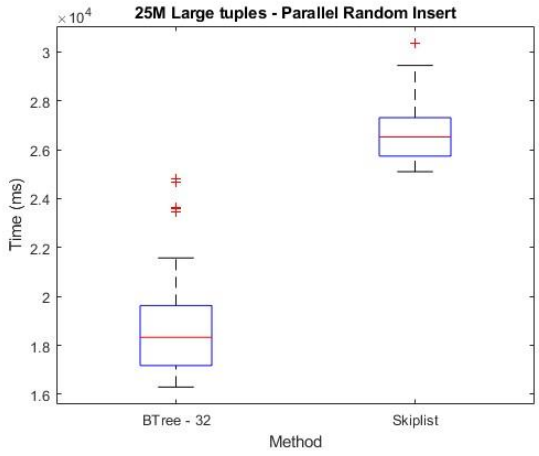


Figure 9. Parallel random insertion with 25M tuples of size 10 with the different data structures during steady-state performance.

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 expresslanes and very seldom use the lower-level lanes. However in the random insertions these lanes aren't as useful, since

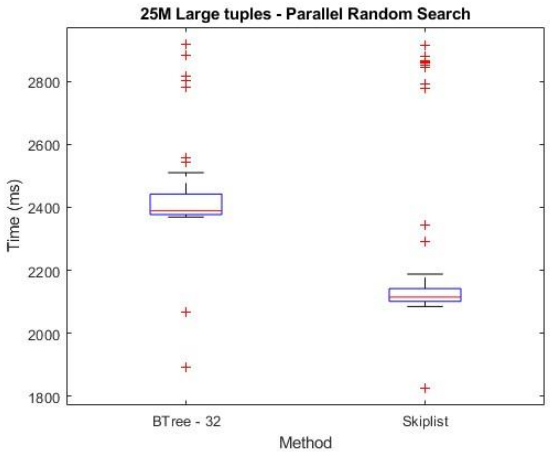


Figure 10. Parallel random search with 25M tuples of size 10 with the different data structures during steady-state performance.

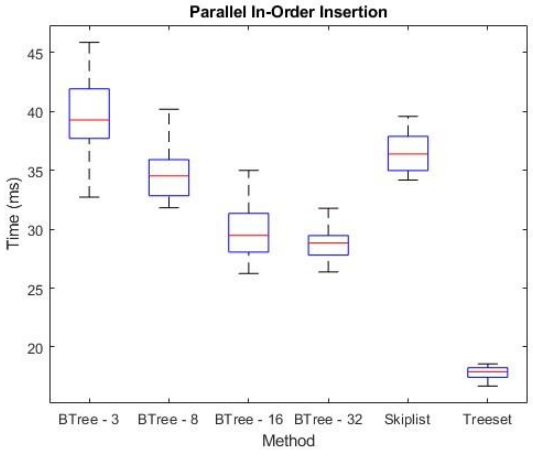


Figure 11. Parallel insertions in the different data structures during start-up performance.

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.

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

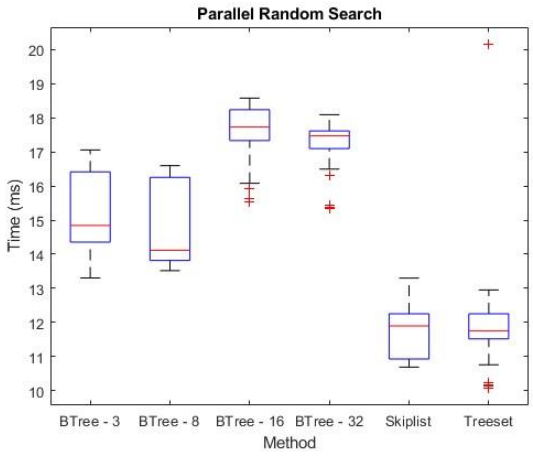


Figure 12. Parallel Search with the different data structures during start-up performance.

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.

Soufflé 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.

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.

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. DefaultHeap 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 Objectoriented 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