Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sander Selnes Toresen

Concurrent Bkd-tree

An Insert-Focused Multidimensional Data Structure with Eventual Consistency

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

NTNU
Norwegian University of
Science and Technology

Sander Selnes Toresen

Concurrent Bkd-tree

An Insert-Focused Multidimensional Data Structure with Eventual Consistency

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Sander Selnes Toresen

Concurrent Bkd-tree

An Insert-Focused Multidimensional Data Structure with Eventual Consistency

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU

ABSTRACT

Modern CPUs are released with more and more computational units. To take advantage of the increasing number of cores on modern CPUs, programs should be created to support concurrent and multithreaded workloads. With this starting point, a multidimensional structure called a Balanced Kd-tree(Bkd-tree) have been taken as a study subject. This thesis covers the creation and benchmarking of a serial Bkd-tree, a multidimensional index. The findings is further used as a general blueprint for modernizing the data structure and improving the solution by utilizing multithreading and concurrency.
Multiple tests and configurations of the Concurrent Bkd-tree have been profiled and benchmarked. The overall results indicate that synchronization overhead should be avoided were possible and that multithreaded performance is largely dependent on reducing communication between threads. The Concurrent Bkd-tree is tested in both a laptop and server environment and the results conclude the importance of utilizing configurations which benefit the hardware of the given machine. Findings indicate the importance of tuning the workload based on the system and that utilizing all threads may slow down performance in data intensive application due to multiple threads sharing the cache. The configuration should also reflect the the expected workload, either creating large structures to support read performance or smaller structures to increase Inserts per Second(IPS).

The final result is a Concurrent Bkd-tree which demonstrates an average thread efficiency of 34.8% during inserts in data intensive workloads with 4228 Inserts per Second(IPS), in a close to optimal configuration.

SAMMENDRAG

Moderne CPUer utgis med fler og fler beregningsenheter. For å utnytte det økende antallet kjerner på moderne CPUer, bør programmer utvikles for å håndtere samkjørende og flertrådede arbeidsbelastninger. Med dette utgangspunktet har en flerdimensjonal datastruktur kalt Balansert Kd-tre(Bkd-tre) blitt utvalgt som et studieobjekt. Fokuset på denne masteren omhandler utviklingen og ytelsestesting av et serielt Bkd-tre, en flerdimensjonalt indeks. Funnene blir deretter brukt som en generell mal for å modernisere datastrukturen og tilpasse løsningen til å kunne utnytte flere tråder og samkjøring.
Flere tester og konfigurasjoner av det Samkjørende Bkd-treet har blitt profilert og ytelsestestet. De overordnede resultatene indikerer at tidsbruk på kommunikasjon burde reduseres hvor det er mulig og at flertrådet ytelse i stor grad avhenger av å redusere kommunikasjonen mellom tråder. Det Samkjørende Bkd-treet er testet både på en bærbar datamaskin og på en større server. Resultatene konkluderer med viktigheten av å utnytte konfigurasjoner som drar nytte av maskinvaren til den gitte maskinen. Funnene indikerer viktigheten av å tilpasse arbeidsmengden basert på systemet og at bruk av alle tråder kan redusere ytelsen i dataintensive applikasjoner, da flere tråder må dele hurtigbufferen. Konfigurasjonen bør også gjenspeile den forventede arbeidsmengden, enten ved å opprette store strukturer for å støtte leseytelse eller mindre strukturer for å øke antall innsettinger per sekund(IPS).

Sluttresultatet er et Samkjørende Bkd-tre som oppnår en gjennomsnittlig trådeffektivitet på 34,8% under innsettinger i datatunge arbeidsmengder med 4228 innsettinger per sekund(IPS), i en nær optimal konfigurasjon.

PREFACE

This thesis was written for the Department of Computer Science (IDI) at NTNU, with supervision from Svein Erik Bratsberg. The theory and groundwork of the thesis are primarily based on research conducted during a specialization project for the course IT3915 in the fall 2022, with certain modifications.

I want to thank my supervisor Svein Erik Bratsberg for his expertise and guidance. I am also grateful to my friends for our companionship and memorable moments. Thanks to my family for their support, comfort and for never being longer than a phone call away. Thanks to my girlfriend Elisabeth for her love and encouragement. Her support and encouragement made the writing of this thesis way more pleasant and I look forward to our future together.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

INTRODUCTION

Computer hardware manufacturers have gotten closer and closer to a power wall determining how much power can be sustained by a single chip without overheating. This has led to a larger focus on computational units with more chips rather than a single powerful chip. To fully take advantage of modern CPUs, modern programs should therefore be able to take advantage of multiple cores by creating multithreaded and concurrent applications. This paper will cover the research of how multithreading can be used to increase performance of the Bkd-tree data structure. The rest of this section will present the Bkd-tree and give a brief overview of the Bkd-tree's structure. The rest of the paper is structured as: Background, Serial implementation, Serial results, Serial discussion, Concurrent implementation, Concurrent results, Concurrent discussion and Conclusion. The serial sections from chapter 3-5 covers the implementation, results and discussion related to a serial Bkd-tree solution. The knowledge and findings from this section is used to suggest a concurrent Bkd-tree architecture, which will be further built upon in the concurrent section in chapter 6-8. This section uses the findings from the serial Bkd-tree to create a concurrent Bkd-tree closely based on the suggested Bkd-tree architecture presented at the end of the serial section.

1.1 Problem definition

The Bkd-tree was first introduced in a paper in 2003 and was created as a solution to handle large amounts of inserts while sustaining great space utilization. The experiments from the authors shows great insert performance compared to other multidimensional indexes such as HB-trees and Kdb-trees. As the Bkd-tree structure consists of multiple balanced kd-trees, a query needs to be performed on all the trees in the structure. This results in a larger computational overhead when performing deletes or window queries as multiple trees needs to be traversed. Results from range queries shows that despite having similar or less I/O operations compared to the Kdb-tree, the Bkd-tree's querying is still slower due to the computational overhead[1]. This inspired the research area of this paper which will look into possibilities to overcome this overhead with multithreaded processing. This thesis will use the Bkd-tree structure as a starting point to first look into possible solutions and limitations of the structure, using them to attempt to improve performance by creating a concurrent Bkd-tree solution. The focus of

this master thesis is therefore to attempt to create a concurrent Bkd-tree which benefits from increasing the number of computational units.

1.2 Brief overview of the Bkd-tree

A Bkd-tree is a data structure for storing multidimensional data in a fast manner. The Bkd-tree is similar to the structure of LSM-trees in that it is a structure consisting of multiple structures. Like the LSM-tree, the Bkd-tree is dynamic and the size of each of its structures increase in size lower down in the tree. The Bkd-tree is made to have fast inserts and few updates to the structure per insert. This is achieved by inserting data to memory until full and then flush the data to disk in a balanced tree structure. Data is always added in the smallest available tree which then get merged into larger structures as more data is inserted. As data is always inserted in the smallest available tree, changes to the largest structures containing the oldest data will happen the least frequently. The dynamic basic structure can be seen in figure 1.2.1. All trees on disk will consist of a number of nodes which is a power of the memory structure size. When T_0^M becomes full in figure 1.2.1, All the trees to the left of the first empty tree will be merged into a larger tree. So T_0^M, T_0 and T_1 will be merged together to the new tree T_2.[1]

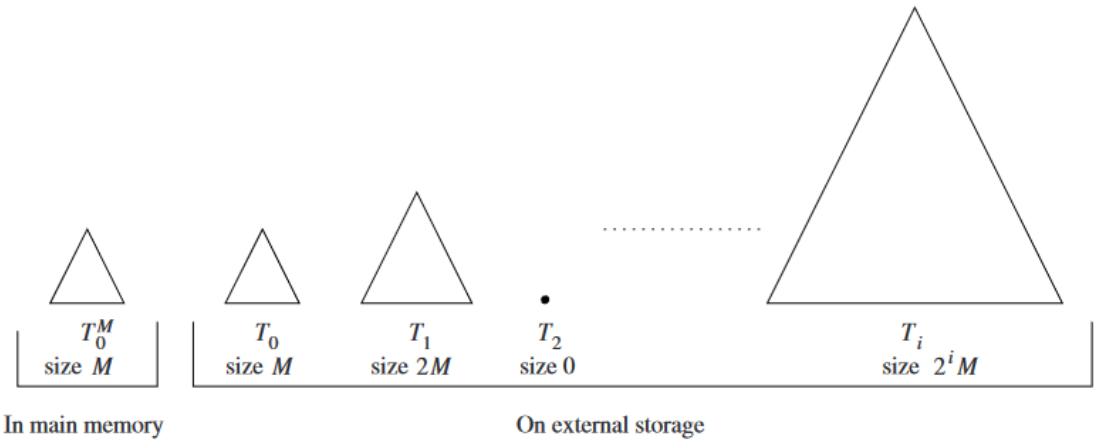


Figure 1.2.1: The forest of trees that makes up the data structure. In this instance, T_2 is empty

[1]

To merge the trees together to a new tree the authors present two kinds of bulkloading, binary and grid loading. The binary version includes a top down approach where the tree is built and filled with nodes in the same section. The grid bulkloading splits the tasks. It works by keeping track of the distribution of nodes in a matrix which is then used to generate a tree structure. As a final step the data is inserted after the tree structure is built. In this thesis, the focus will be on the binary bulkloading strategy. The Bkd-tree structure is primarily built to store multi dimensional data and access it efficiently. The primary query for accessing data in multi dimensional structures is using a window query. A window query works by asking for all points within a given area. This means that for each dimension, the structure should return all nodes within the given coordinates. An

example of a window query on a two dimensional data set can be seen in figure 1.2.2. Here all nodes where $3 < x < 9$ and $2 < y < 8$ are selected. The result of the window query is therefore all the nodes within the selected area. So the query would return the values of A, D and F from the data set.

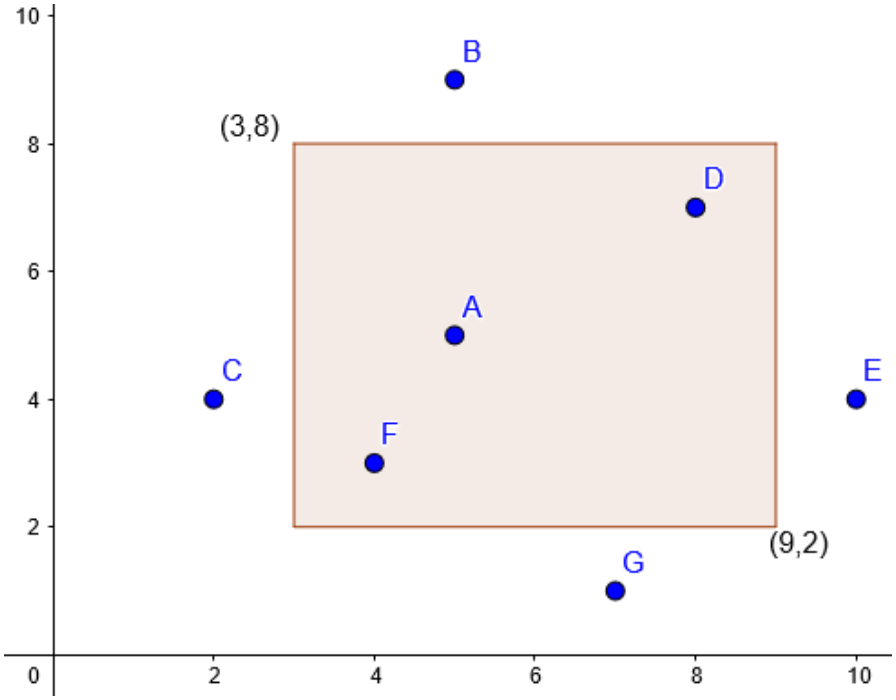


Figure 1.2.2: 2D window query example.

CHAPTER

TWO

RELATED WORK AND BACKGROUND

Related work

The focus of this thesis, is to take a multidimensional data structure and look into solutions for achieving speedup. The primary focus will be on achieving speedup with the use of extra computing power, by sharing the workload between multiple computing units. This section will therefore cover papers presenting data structures which has been improved by multiple compute units and also structures similar to the Bkd-tree.

2.1 RCU-HTM

The RCU-HTM is a generic synchronization technique for highly efficient concurrent search trees. The paper covers a synchronization technique which takes advantage of two synchronization mechanisms, Read-Copy-Update(RCU) and Hardware Transactional Memory(HTM). The solution is generic as it can be applied to any structure which can take advantage of the RCU technique. RCU allows for asynchronous reads meaning all concurrent readers have no overhead and each thread perform as a serial solution. The downside to RCU is that it can only have one computational unit writing to the structure at once. This tanks performance in workloads with any significant amount of updates. HTM allows for optimistic concurrent execution, meaning threads can write and read data and if a collision between threads occur, the colliding operations are aborted and restarted. Together RCU-HTM allows for a RCU solution which can have multiple updaters with a HTM implementation which restarts and eventually serialize colliding operations. The paper further compares the RCU-HTM implementations against multiple state of the art binary search trees that use different kinds of synchronization mechanisms[2].

2.2 Log-Structured Merge-Tree

The Log-structured Merge-tree(LSM-tree) is a tree which was created as a solution for applications with high amounts of updates. The B-tree effectively doubled the I/O cost of inserts due to real time restructuring of the tree and keeping the tree

balanced. The LSM-tree set out to be an alternative and works by adding new indexes in a memory resident structure and merging it together with larger trees on disk. This is created in such a way that the larger the trees become and the older data they store, the fewer updates is made to the structure. This is done in the same way as seen in figure **??**. The LSM-tree achieve this by always putting new data nodes in the memory structure and merging the data nodes into the smallest possible structures. The LSM-tree is a key value store with a one dimensional index[3].

2.3 Plush

Plush is a write-optimized persistent log-structured hash-table. Plush is similar to the Bkd-tree in that it is a log structured data structure. Plush therefore uses the same technique as the LSM-tree where it has a memory buffer which gets flushed to a lower level storage unit and dynamically merged into bigger collections of data lower down in the tree. Plush is created to be efficient on systems using Persistent memory. When Plush deletes a record, it adds the deleted record to a list and removes it when it shows up at a later migration[4].

Background

2.4 Process, multithreading and concurrency

When a program is executed the underlying operating system will spawn a process which runs the program code. The process is run by a main thread, which in turn may spawn new threads. A multithreaded program is therefore a program which runs on multiple threads. Multithreaded programs can increase performance as the workload can be split among multiple threads[5]. This may increase performance in computational heavy workloads, but also result in new problems regarding data sharing between threads and asserting data correctness.

2.5 Parallelism

Moore's law states that due to shrinking size of transistors, every two years the number of transistors which can fit on a chip will double and hardware will become exponentially more powerful. This law was true until the size and amount of transistors no longer was the determining factor, but how much power a single chip can sustain. To overcome this, chip manufacturers focus shifted to create processors with multiple cores and chips. Therefore, to fully take advantage of modern hardware, highly efficient software should utilize multiple cores and threads and support parallelism or concurrency.

2.5.1 Parallelism, concurrency and distributed computing

Parallel, concurrent and distributed computing are all ways to take serial code and achieve speedup by sharing the workload with multiple kinds of computational

units.

In parallel computing, a program solve multiple tasks cooperating closely with other threads to solve a problem. An example of a parallelizable task is matrix multiplication as multiple threads can each solve a small fraction of the problem while working in parallel. Parallel computing is highly efficient at large scale calculations, but is less generalizable and is not suitable for complex workloads with branching and nodes with different tasks.

In concurrent computing a program may have multiple different tasks running at a given instant. There may be multiple threads working within one program, all having different responsibilities. A suitable program to make concurrent could be a queue where you both have producers and consumers. Concurrent computing is suitable for complex workloads with workers being able to work independently from one-another. Concurrent code is run on an CPU, which have fewer threads available compared to parallel solutions running on an GPU.

Distributed computing is a program which communicates with other programs. This can for example be done over a network and can be beneficially to distribute a workload over multiple machines. Hardware for distributed computing can be cheaper compared to one powerful concurrent solution. Instead of having an expensive machine with a CPU with multiple powerful cores, a distributed program can run on multiple cheaper machines, each having a few powerful cores. Distributed computing do however often have a larger communication overhead as it could be expensive to communicate with other machines over a network compared to communication between cores in a single CPU.

In this thesis, the goal is to lay the groundwork for solutions which can be used to achieve a speedup of a Bkd-tree. As a starting point, concurrent computing will be looked into as a base as it is most fitting for the problem. Concurrent programming gives the flexibility needed for designing a Concurrent Bkd-tree which can have multiple worker threads which all can work independently and share memory. Distributed computing could also be a suitable solution which would be worth looking into. However, when assuming the final solution will require more communication between units than raw computing power, concurrent computing where deemed the best fit.

2.6 Amdahl's law, Gustafson's law and Thread pooling

Amdahl's law states that the maximum speedup of a concurrent or parallel system is limited to the serial overhead of the solution. The formula for Amdahl's law can be seen in equation 2.1. Consider a parallel solution where 10% of the code is serial and the final 90% is parralelizable. Using Amdahl's equation shows that the greatest speedup which can be achieved is 10x the speed of the serial solution. This is a result of when the number of processing units reaches infinity, the solution performance is still limited to the serial section of the code[6, p. 58]. Amdahl's law assumes that the workload is a set size. Meaning that it shows the maximum speedup of a given program without considering the workload. However, when the amount of computational resources increases, the problem size can also increase to take advantage of the new resources. This is better encapsulated in Gustafson's

law, which has a more optimistic look on the scaling of high-computational computing. Gustafson's law states that with the increase in computational units, the parallel or concurrent workload tend to increase at a similar rate as programmers increase the problem size to fully take advantage of the processing power. While the parallelizable section tend to increase as the program size grows, the serial section do not tend to increase significantly[7]. Using Amdahl's law, each computational unit added will have a exponentially downwards speedup effect on the runtime. Using principles from Gustafson's law, the workload can be scaled at a similar rate as the computational units are added. By doing this, the performance increase from added computational units can be closer to linearly scalable.

$$Speedup_{total} = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

Scaling up the workload size could work to get more performance out of each added core. Each computational unit can be assigned a significant amount of work instead of only being assigned a small fraction, this increases the usefulness of each individual thread. The critique from Amdahl's law is still valid as the serial section of a parallel or concurrent solution can be a bottleneck when the workload is small compared to the computational power of the system. When a concurrent program starts, a single thread usually starts the program, assigns work and spawns new threads to perform the work. This is the primary critique. No matter the number of cores of the system, a main thread still needs to partition data and spawn the new threads. In some cases, using multiple compute units may be slower than a serial solution. This happens when the work performed by the spawned threads uses less time than the time it took for the thread to spawn. For this reason, it is important that spawned threads perform a significant amount of work to overcome this overhead. One solution to assert that each thread performs more work than the cost of the overhead is to use thread pooling. Thread pooling is the technique of having multiple threads on standby which all can be assigned work. With this technique, the overhead of spawning the thread is minimized by spawning a thread once and then having it wait to be assigned more work instead of exiting[8, pp. 274-288]. Thread pooling is useful when the process do not need to share resources. In systems where a process do not share resources, having threads idle and on standby makes them fast to initiate while also not taking away resources from other processes. The Concurrent Bkd-tree implementation will utilize multiple threads to increase performance. Thread pooling could therefore be a useful strategy to overcome the bottleneck related to spawning new threads. With the combination of thread pooling and the high workload which will be handled by the Concurrent Bkd-tree, the serial section of the program will be a small amount of the total runtime. The program will therefore likely scale closer to linearly as predicted in Gustafson's law rather than exponentially downwards as predicted by Amdahl's law.

2.6.1 Correctness in concurrent data structures

In concurrent data structures there might be several threads working on the same data or the same structure. If threads are not working together, the data structure and the data can become inconsistent. To achieve correctness in concurrent data

structures, synchronization mechanisms are needed for the threads to be able to work on the same structure concurrently[2]. Synchronization mechanisms offers correctness by letting the threads work on the data structure as if it was a serial solution. Some of the more common synchronization mechanisms used to achieve correctness are locks and atomic operations.

2.6.2 Locks

There are different kinds of locks such as spin locks, semaphores and barriers. Different kinds have different locking conditions, but the general concepts still apply. Locks achieve correctness by locking resources asserting that other threads do not reach them while a set amount of threads are working on the resource or that other threads reach the resource before intended.
The most basic lock is a spin lock. Spin locks work by locking a thread in a loop where it "spins" until the resource is freed and the thread is able to obtain the lock. A lock could also be made non-blocking, allowing the thread to work on other problems and periodically checking if the resource has been released.
Semaphores work by having a given number of threads which can reach a resource and locks new threads from accessing the resource when the threshold is met. Semaphores could help reduce strain on a shared resource by asserting only a specified number of threads can access it at once.
Barriers are a lock variant which works opposite to the semaphore. Barriers are used to accomplish synchronization between threads. The barrier will lock all threads and only release them when the specified number of threads has entered the barrier. Barriers are therefore useful to achieve synchronization between threads asserting no thread continue on their work before the other threads are finished with their work. Barriers are therefore useful in cases where all threads needs to finish before continuing, but could affect performance as the fastest threads needs to wait for the slowest[9].

2.6.3 Atomic operations

Atomic operations are instructions which are guaranteed to run in full without interference from other threads in a multithreaded environment. Atomic operations work by making a small change which is done without interference from other threads. An atomic instruction is uninterruptible, as the whole instruction is performed before other threads can interfere. Atomic functions are not possible to divide into smaller parts, meaning that the whole instruction will be performed or it will not be performed before another thread is scheduled. Atomic functions can be implemented on software and hardware level. The most well known atomic operations are compare and swap(CAS). Compare and swap works by comparing a value with a shared resource, and if the values are the same, the shared resource is updated. CAS can therefore be used to assert that a shared resource have not been altered by another thread and is safe to update. CAS instructions are more expensive when compared to other machine instructions. However CAS instructions are still faster when compared to the overhead involved in using a lock to protect a shared variable[9].

2.6.4 Read-Copy-Update

As mentioned in section 2.1, Read-Copy-Update(RCU) is an synchronization mechanism which allows for asynchronous reads of data structures, but the generic implementation is limited to a single updater. RCU support autonomous reads by having the updater always work on a copy of the data and atomically updating the structure therefore avoiding collisions. This is achieved by the updater first reading or traversing down to the area which will be changed, then copying the parent node of all the affected nodes, performing the update and atomically inserting the updated memory address as a parent node. With this strategy, a reader will either read from the old replaced nodes, which can be deleted after all readers are done reading from the outdated structure or if accessing the node after the atomic update, will read the updated value. As the RCU only support one updater at a time, it should be combined with other synchronization mechanisms for any structure with a significant amount of inserts.

CHAPTER

THREE

SERIAL IMPLEMENTATION

This section will cover some technical details of the Bkd-tree implementation. To lay the ground work for a Concurrent Bkd-tree, a serial solution of the Bkd-tree were implemented in C++. This sections will cover how the tree was implemented, technical details and how the solution differs from the implementation presented by the original authors. The structure will later be benchmarked to spot bottlenecks and high computational areas which then can be optimized and improved with multithreading.

3.1 Programming language

For implementing the Bkd-tree C++ was chosen as the programming language. C++ extends C with functionality and offers a larger set of standard library functions. C++ is known for being efficient and gives the programmer access to low level memory allocation giving the programmer more control[10].

3.2 Solution structure

The project is structured to increase readability and to abstract away code. The project is split into 3 sections. The main components of the implementation was the Bkd-tree, Kdb-tree and the configuration file. The Bkd-tree contains everything related to the Bkd-tree's memory buffer implementation. The Bkd-tree's memory buffer structure mostly handles the unstructured Memory and Disk arrays and makes calls to the Kdb-trees which is used to store the trees lower down in the structure. The Kdb-trees are the Balanced kd-trees which makes up the bulk storage of the structure. Finally, the configuration file is used to configure global variables used by the implementation such as the structure size of the memory storage and leaf node sizes. The Bkd-tree itself is implemented to be configurable and able to support multiple dimensions. As the primary focus is on studying improvements gained by concurrency, the tree was implemented as a proof of concept memory structure. This was done based on the assumption that making a disk based storage system would not give any additional insight on the subject of concurrency or multithreading compared to a proof of concept memory implementation.

3.3 Bkd-tree

The Bkd-tree itself is the primary structure which manages all structures, including creating new Kdb-trees when needed. Both the memory and disk buffer is located in memory. When the Memory buffer fills up, it is flushed to the disk buffer if it is not full. If the disk buffer is full, the arrays will be merged together into the first Kdb-tree. This is the general structure of all operations, both updates and reads. Reads work by either searching up a single value or performing a window query. The search start with the most recent added data in the memory and disk arrays and gradually move downward into the Kdb-trees. In the case of a window query, all the structures needs to be scanned. When inserting a new value into the structure, the most common operation is to write the value into the memory array. However if the memory array is full, the inserter will first need to bulkload all previous structures into the first empty structure. This can be as simple as turning the memory array into a disk array and allocating a new memory array. In the worse case, the inserter will need to merge all full structures located to the left of the first free tree, into the new tree. Such as turning memory array(T_0^M), disk array(T_0) and T_1 into the new tree T_2 as would be necessary in figure 1.2.1.

3.4 Kdb-tree

The Kdb-tree is a bare bones Kdb-tree implementation with no auto-balancing. This is a design decision made as the structure only changes during deletions. Instead of restructuring the tree during deletion, the idea is that the tree will be replaced with a completely balanced tree during its next restructure. The difference between the normal Kdb-tree and the Kdb-tree presented in the Bkd-tree paper, is that it is not self balancing and each leaf node in the Bkd-tree version stores multiple values. The Kdb-tree from the Bkd-tree stores enough values in its leaf node to completely fill a disk block. With this design, it is possible to achieve perfect space utilization on the disk based structures. In the implemented solution, the Kdb-leaf size can be configured to an arbitrary value as the solution currently is not disk based. However, the Kdb-leaf value should be compatible with the memory array size in such a way that one memory array can fill a set amount of kdb-leafs without empty space to keep perfect space utilization. The Kdb-tree implementation differs from the one used in the Bkd-tree paper as pointers between leaf nodes has been added to support faster fetching of data when bulkloading trees. As mentioned, the Kdb-leaf nodes should be able to perfectly fill a disk block, so to make sure a single pointer did not come in the way of that, the pointer is stored by the branch which points at the leaf node. The pointers can also be reached by the root node of the tree. All these changes allows for fast access of all the leaf nodes. This is beneficially when bulkloading as during bulkloading, all nodes should be fetched from the tree. The structure of the implemented Kdb-tree can be seen in figure 3.4.1.

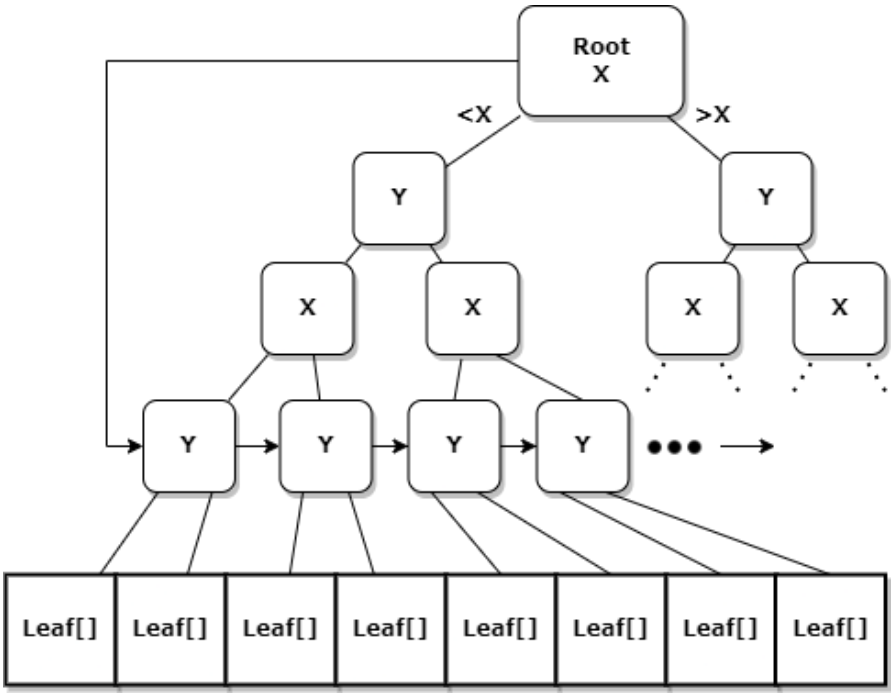


Figure 3.4.1: The structure of a 2 dimensional Kdb-tree, each traversal change the dimension value that split the tree.

3.5 Binary bulkloading

The binary bulkloading strategy is a solution for taking a list of inputs and building a balanced tree based on the data rather than inserting the values one by one. Bulkloading is often used as a mean of setting up a data structure and refilling it with values that should be stored in the data structure. In the case of log structured merge trees such as the Bkd-tree, the bulkloading algorithms are constantly used to merge trees into larger structures. The values are first fetched form all the structures that will be merged together. Both the memory array and disk array are added to a file together with all nodes from the trees which also should be merged together. The values from the trees are fetched using the Kdb-trees linked leaf node list mentioned in the previous section. To create a balanced Kdb-tree, all the values needs to be sorted on each dimension, so the values are stored once for each dimension and then sorted on each of them. The trees first left and right branches are then created by splitting the values on the first dimension and splitting the other nodes such that the values sorted on the other dimension ends up with their matching nodes. The Kdb-tree is then created recursively in a similar manner with the branches being split on the middle value of the revolving dimensions which is iterated between as seen in the traversal of the tree in figure 3.4.1. This happens recursively until the number of values is less or equal to the Kdb-leaf size. The bottom of the recursion is then reached and the remaining values are inserted into a Kdb-leaf node and the linked leaf list is updated to later allow for fast access to all nodes of the tree.

CHAPTER

FOUR

SERIAL RESULTS

4.1 Experimental platform

The Bkd-tree was implemented in C++ and the solution have been tested and benchmarked using Chrono, a C++ library for tracking time[11]. In the experiments, it is used to time sections of the Bkd-tree implementation. Chrono support accuracy down to 100s of a nanosecond so in some of the fastest operations, the time is measured to 0 nanoseconds. The solution were run on a System with a Intel(R) Core(TM) i5-6600 3.30GHz processor, 16 Gb of 1600MHz RAM and a Windows 10 Home operating system. The code was run and compiled on g++ (GCC) 10.2.0 with the $-g$ flag for debug support. Each presented data point is the result of the average from 10 individual runs. Each value used in the experiment were a generated pseudo random integer. Each data node had a key as a multidimensional index consisting of integers and each node stored an unique integer as its data. Experiments were conducted with the tree configurations listed in table 4.1.1. And the experiments were performed with different amount of inserts to see how tree load affected performance.

	Value	Description
Array size	32768	Size of memory and disk array in number of nodes.
Kdb-leaf size	128	Number of nodes in each Kdb-leaf nodes.
Dimensions	2	Number of dimensions of multidimensional index.

Table 4.1.1: Configuration used in serial experiments.

4.2 Deletions

As mentioned in the previous section, deletions were implemented as suggested in the Bkd-tree paper and works by iterating over all structures, finding the specified node and removing it. The tests in this section test the worst case scenario deletions by deleting the 100 first nodes inserted in the tree. This is the worst case scenario as it means the nodes will be stored in the bottom of the last structure and that all other structures will be traversed before locating the node. The 100 values are deleted and timed one by one.

4.2.1 Deleting single tree

The purpose of this test is to see how increasing tree sizes affect the deletion time of a node. The results of the experiment can be seen in figure 4.2.1. The trees are first filled by inserting the nodes into the tree normally until a size of $T_n + 1$ is reached. The final insert fills the previous structures and the tree is then created by bulkloading. As the trees are completely balanced with $32768 \times Treesize$, data nodes, all leaf nodes reside on the same level in a given tree. In T_2, accessing the leaf nodes therefore requires 9 traversals to reach the bottom branch which points to the data. With the set configuration the number of traversals can be expressed as $traversals(x) = \log_2\left(\frac{32768x}{128}\right)$ where x is a tree size to the power of two. The graph shows the deletion time increasing with the tree sizes. This is most likely caused by the increasing numbers of traversals as the trees grows and might be a result of increasing data sizes which may cause reduced performance either because of virtual memory written to disk or reduced cache hits due to an increased data load.

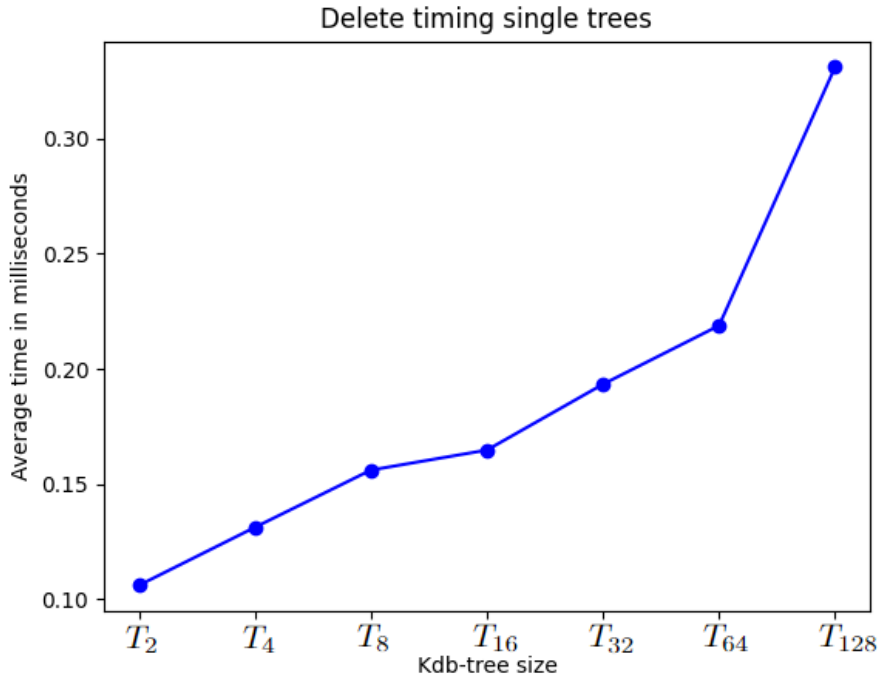


Figure 4.2.1: Delete instruction single tree time in milliseconds.

4.2.2 Deleting multiple trees

To see how searching through multiple structures affected the delete performance, this tests performs the same test as when deleting a single tree. The only difference is the tree size which is now set to $T_n - 1$. This tree size makes it so all previous trees will be filled. So for $T_8 - 1$ the structures T_4,T_2, T_0 and T_0^M are filled while T_8 is empty. The hypothesis before performing the experiment were to see the graph have a growth similar to the results in figure 4.2.1. As seen in figure 4.2.2 the results instead seem to be more sporadic. The slowest run, though by a fraction is made by the smallest structure. What this figure really shows is the weight of the overhead added by traversing the unsorted memory and disk arrays. Consider the

deletion traversal of structure $T_8 - 1$. As mentioned, it consists of the structures T_4,T_2, T_0 and T_0^M, which all needs to be traversed. The cost of traversal of each tree then becomes $traversal(4)+traversal(2) = 19$. Combining that with the iteration over the 2 leaf nodes we get 19 traversals and 256 leaf nodes. This is only 0.4% of the total traversals compared with the disk and memory array structures which combined has 65536 nodes. As both T_0 and T_0^M are full in every trees in the test, this becomes the most expensive part of the operation and makes the tree traversals insignificant. This also adds into the reason for why the smallest trees have the worst performance. When deleting a node from the trees larger than $T_2 - 1$, first all nodes in the unsorted arrays are traversed, then traversing down every tree until the node is found. When the node is found the node can be deleted by copying the last leaf into the removed nodes place and decreasing the leaf size. As the leaf is only 128 nodes, the values needed for the replacement might be higher up in the memory hierarchy resulting in a faster operation. This may not be the same case as $T_2 - 1$. This is because $T_2 - 1$ needs to perform the delete on the disk array which seem to be a more expensive operation compared to deleting nodes from trees. This is likely a result of the disk array size of 32768 nodes which might result in slower performance due to working on a larger structure.

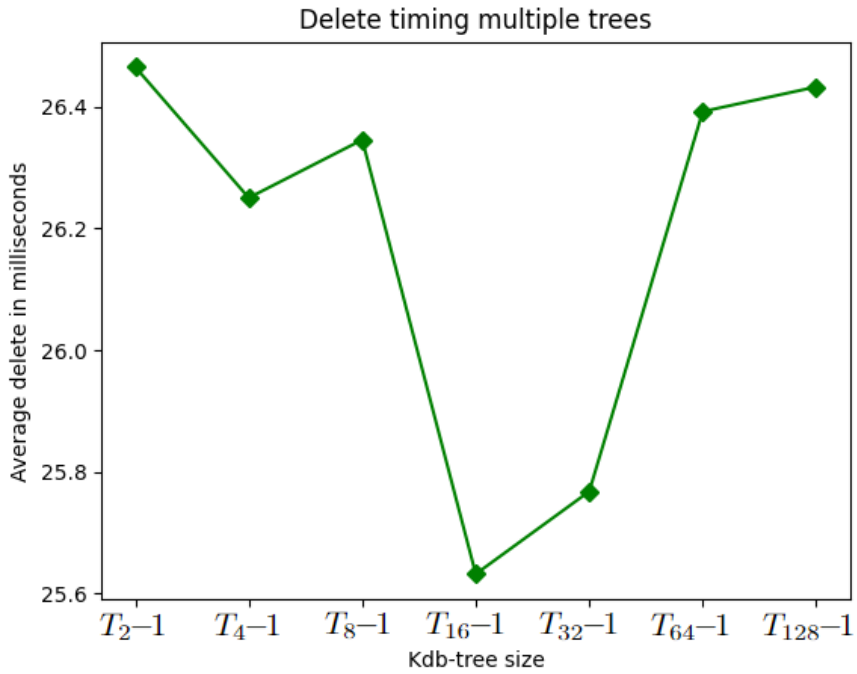


Figure 4.2.2: Delete instruction multiple tree. Each test is -1 the tree size.

4.3 Inserts

The insert implementation works by placing nodes in the memory buffer and further bulkloading them into larger and larger structures. This section covers the insert performance with an emphasis on the performance hit experienced when the bulkloading operation is necessary. The results are based on inserting a T_{256} tree worth of data so T_{256+1} is inserted to cause the final bulkloading. This results in the insertion of almost 8.4 million data points (256×32768). The data is again

pseudo randomly generated integers with the timing result being an average of 10 runs.

4.3.1 Bulkloading trees

Figure 4.3.1 shows the bulkloading performance of all the primary bulkloading operations. For example, bulkloading T_8, the structures T_4,T_2, T_0 and T_0^M are combined together to create T_8. This happens when the previous mentioned structures are full and the tree must be bulkloaded into a larger structure. So the measured time on the graph, is the time it takes one insert to combine the previous occurring structures into one and then inserting its own data into the new memory array. Figure 4.3.1 shows how the time spent bulkloading, starts slow but grows exponentially as the amount of data managed per bulkload doubles per newly introduced tree size. It should be noted that the pictured results are the case where all previous data is combined into one large new structure. There are however other cases such as between the step of T_8 and T_{16}, there are multiple smaller restructures while T_2, T_4 gets filled up, so smaller bulkloads happens every time T_0 and T_0^M gets filled.

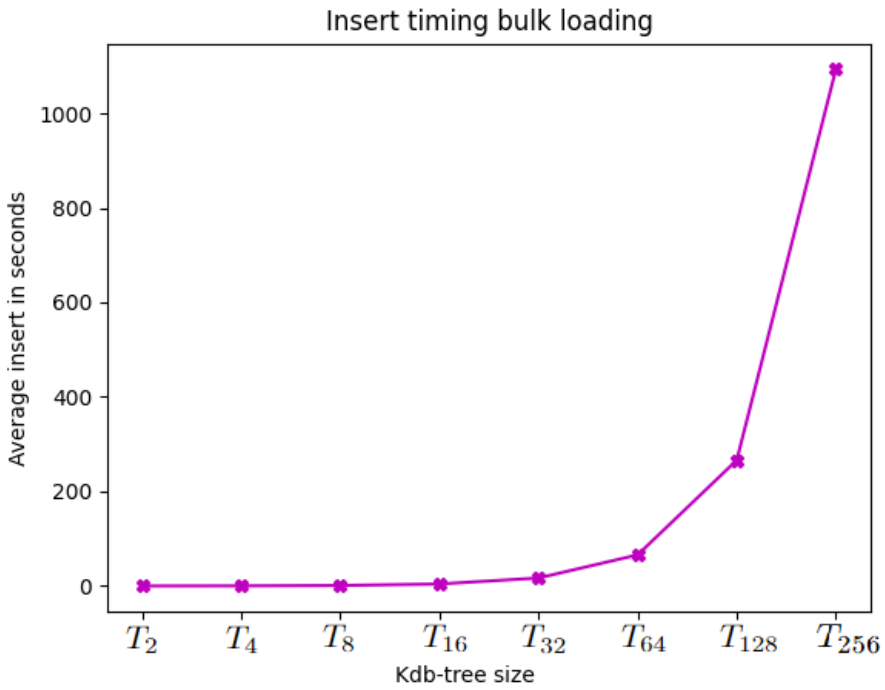


Figure 4.3.1: Timing of the leaf node which causes bulkloading.

4.3.2 Average insert performance

Due to the design of log structured merge trees, most inserts are cheap. The insert that are the slowest are the cause of needing to restructure the tree. In 99% of cases, the inserts are close to instant, as seen in table 4.3.1. This is because the most common scenario is to just write the inserted node into the memory buffer. The final 1% is where things get more problematic. The worst insert completed in over 18 minutes. This is a result of the tree growing large and all previous tree

Percentage	**Nanoseconds**	**Times slower than 1%**
1%	20	1x
10%	40	2x
20%	50	2.5x
50%	60	3x
80%	70	3.5x
90%	80	4x
99%	100	5x
99.9%	1850	92.5x
99.99%	5070	253.5x
99.999%	100423710	5021185.5x
99.9999%	4576106470	228805323.5x
100%	1093112409220	54655620461x

Table 4.3.1: Percentage worst inserts.

structures gets combined into one. As the test was performed on two dimensions, the data will first need to be fetched into a central unit, create a copy of all nodes and then sort each on their own dimension. Even though the time for each insert is close to instant, the average insert performance is 194362 nanoseconds. This means that the top slowest 1% is so slow that the average time is also in the 99.99%. It should however be noted that the average without the final bulkload is on 64052 nanoseconds, which is a decrease of 67%. So though the final, bulkload is very slow, there would not be another as large operation until the inserted data doubles what it currently is.

CHAPTER

FIVE

SERIAL DISCUSSION

This section will discuss the serial results and together with the background serve as tools to discuss which parts of the implementation could be worth further exploring with concurrent computing. At the end of the chapter, the findings will be combined to create a Concurrent Bkd-tree architecture which further will be built upon in the final sections.

5.1 Inserts

The insertion test proved that insertions can be quite expensive and is definitely an area which should receive more attention when attempting to create a more optimized solution. The insert show that bulkloading is very expensive and should be sped up either with the help of concurrent computing, using another bulkloading algorithm or improve the current one.

5.1.1 Bulkloading algorithm

In the Bkd-tree paper two bulkloading algorithms were presented. In the demo implemented in this paper the binary bulkloading algorithm were used. The binary bulkloading algorithm were simpler as it requires no additional structures and can be implemented as a straight forward recursion function. The grid bulkloading presented were an algorithm that splits the data into grids based on where the data will be partitioned in the structure. The distribution of counts of points are stored in a matrix which further can be read and used to make the tree structure before inserting the data. The problem with the grid bulkloading strategy is that it requires an additional data structure to partition the data as it requires a matrix which might vary in size depending on the size of the data. The matrix structure also need to be modified if it should support more than 2 dimensions. The implementation presented in the paper present a grid bulkloading strategy for a 2 dimensional tree. To use grid bulkloading on a structure with more dimensions a matrix supporting multiple dimensions would need to be created or imported. However if the results of the Bkd-tree bulkloading experiments applies to structures with more than 2 dimensions it could still be beneficial due to the grid bulkloading algorithms faster bulkloading speed.
So the strength of the binary bulkloading strategy is it versatility. As it works

recursively and do not rely on other structures, the same algorithm can be applied to any data set with any number of dimensions. However this strength might not overcome the fact that it is slower than its grid loading counterpart. In most use cases it will be beneficial to have a specialized solution made for one specific problem instead of a solution which is one size fits all.
The binary bulk algorithm also seem to have been designed primarily with two dimensions in mind. What makes it seem like it is designed for a two dimensional structure is an oversight in the original algorithm. The algorithm suggests starting by sorting the values on all dimensions. However the first split in the tree will only split the data on one dimension meaning that the rest of the dimensions could rather be sorted after the first recursion. By not sorting the data right before its needed could effectively cut the number of nodes needing to be sorted down to $log_2(N)$ nodes. To prove this, consider the following thought experiment.
An D dimensional data set will be bulkloaded. First all the N nodes are sorted, costing $D \times N \times log_2(N)$. The data is split based on the first dimension and the array sorted on the first dimension is halved. As the final $D-1$ dimensions are not sorted on the first dimension, they require to be iterated over to check if their data or coordinate belongs in the left or right branch. In the worst case scenario, the whole of the sorted dimension will need to be iterated through to determine where each of the nodes belongs causing an additional $D \times N$ of overhead. So for sorting the data first, extra overhead is added as iteration slows down the algorithm giving the split an unnecessary overhead of $O(N) = (D-1) \times N$. An improved solution would be to only sort when it is needed. Then the unnecessary overhead would be avoided and would only need to sort the first array for the split. It should be noted that the next left and right split would still need to sort and split their data so the overhead from sorting is unavoidable, but iteration over the data can be evaded. However, after the data has gone down D recursive calls, the data would either need to be sorted again on the first dimension or iterate over the sorted data. For this reason it would be beneficial to still send the data already sorted down in the recursion as it is cheaper to iterate over the sorted data which would be $O(N) = N$ instead of sorting it in $O(N) = N \times log_2(N)$.

5.2 Deletion

In the results section the main focus has been on insertion and deletion. The reason for going in dept on the deletion functionality instead of range queries is that they both work quite similar. Both algorithms go down in the tree searching for a node in a given range. The primary difference is that the range query will often have to go down multiple branches while the deletion only searches for one node. The deletion were therefore chosen as the timings and results would be easier to verify because of the way the tree is traversed and not the way the random data were structured on a particular run. For this reason, general similarities can be drawn between the result of the deletion and how a range query test would have performed.

5.2.1 Flaws with the current solution

In the suggested solution, each structure are searched until the deleted node is found. This means that in the case of false positives, where a non existent node is attempted deleted, all the trees needs to be scanned through to assert the node does not exist. This becomes quite expensive as the structure grows and more and more trees needs to be iterated through. A possible solution to this used by other log structured merge trees, is to use bloomfilters. Bloomfilter could help reduce the cost of iterating through trees by asserting some trees definitely do not store the given index. However, this solution is better suited for single indexes as their use case are more aimed at single updates and fetches, while Bkd-trees use case aims more towards range queries. From the results it were shown that when deleting a node, the most significant bottleneck were the traversal of the unsorted memory and disk arrays. This bottleneck would also be the same for the range queries and other operations which requires to scan the tree for data. To keep these operations fast, the memory and disk array could be replaced with a more sophisticated structure with faster lookup time.

5.3 Tombstone strategy

One strategy used by Plush is to use tombstones instead of deleting nodes[4]. Tombstones work by instead of iterating thorough structures and deleting nodes, the delete order is placed in a tombstone list. Then when a restructure happens, each node is compared against the tombstone values and if a match is found, the node will be deleted. This could be a good alternative to needing to traverse all nodes and would make restructuring less expensive. It would however require more monitoring of the structures to assert correctness when considering operation order. If not considered, it could happen that a delete instruction is sent for a node not in the system, then the node gets added and will be deleted the next restructure. This could be resolved by having extensive checks that the data and coordinates match, but this would not resolve the issue if the nodes were completely alike. A better solution could be to have a timestamp stored with each tombstone entry and have a timestamp of every data node. This could then be used to compare and assert if a node was inserted before or after the deletion operation were called on the given node.

5.4 General structure strategy

To determine how the Concurrent Bkd-tree could be implemented and improved, it is important to roughly define the imagined use case for the final structure. By defining the use case, the structure can be further improved based on the kind of operation and workload the tree will need to handle. For example, if the tree should support multiple range queries, but other operations were less important, this changes the scope of the project as the range query performance should be the determining factor when it comes to what the structure should be optimized for. There is also a possibility that recycling structures is a viable strategy and question of how to achieve a good memory structure performance in a Concurrent

Bkd-tree.

5.4.1 Local or global structure

As stated in the introduction, the reason for implementing a concurrent data structure is to expand on the serial Bkd-tree solution to create a solution that can better utilize the multiprocessing and multithreading capabilities of modern processors. The scope of this task is narrowed down to being a concurrent system instead of a distributed system scattered over a network of multiple machines. There is however still a question if the final structure should have the use case of being part of a larger system on a single machine or be made to support transactions sent from other machines.
If the system is made to support local processes, the system resources would need to be shared with other potential resource intensive applications. This would affect what size the memory buffer could sustain before taking a performance hit from the operating system writing memory to disk as virtual memory. This could drastically increase the cost of reading data from the structure. Another problem is not knowing the amount of computational resources used by other processes. This is a problem as if the system supports for example 8 threads, but the threads is overworked by other processes. This could result in multiple concurrent nodes being scheduled to the same processing unit and needs to share its power while getting the overhead caused by concurrency such as communication and synchronization mechanisms. If other local systems calls the structure directly, the are also some fair concern related to potential stalling. Consider a thread from another program inserting a value into the Bkd-tree structure. The thread might be unlucky and become responsible for a bulkloading sequence. This would result in massive stalling and the calling program could become unresponsive for a vast amount of time, depending on the size of the bulkload. This could be resolved by having the local nodes offload the insert to another thread dedicated to answer Bkd-tree calls. However this would require to have available computational resources on standby or at the very least not fully utilizing the system resources to have available resources in reserve.
The other possibility is to implement the solution as a standalone storage system supporting external processes while having its own resources. This could be beneficial as it would be easier to monitor and best use system resources knowing that all resources are dedicated to the Bkd-tree structure. This would make the system work as a singular unit which receives data and queries over a network. This opens for some interesting possibilities as the load of the tree could be better monitored. Operations that are scheduled on the tree could be used to adapt which tasks are prioritized depending on the current workload and what kind of operations are called. If similar queries becomes common or performed often, it could be possible to cache the result of queries, particularly in the case of range queries and normal data fetching. Another possibility would be to avoid bulkloadings of a certain size during peek hours and rather schedule the expensive bulkloaings for when less operations are called on the structure. Knowing that all resources is free to the system could also improve management of system resources. By knowing the resources is available for the structure at any given time, all resources could be used without fear of putting a to big of a strain on the system lowering

performance due to high workloads from the combined toll of multiple processes.

5.4.2 Reusing tree structures

In the current implementation, new trees are created by bulkloading. When the bulkloading is complete, the trees previously storing the bulkloaded data will get deleted. As trees will have the same or close to the same structure each creation, it could be a viable strategy to reuse the structures instead of deleting them. This could work by storing deleted trees in a list and fetching them when they should be reused. In a typical scenario, the trees should have the same size as its previous counterpart as both trees would be sized to the power of 2 of the memory buffer. This means that when a T_4 tree is created from T_0^M, T_0 and T_2, the T_2 could be kept in a list and later reused. When the tree should be reused, the bulkloading algorithm would not need to create a new structure, but could rather update the data and replace the leaf blocks with updated values. This could be viable especially for the smallest trees. The smallest trees are the ones that are created and deleted the most. For example before one T_{16} tree is created, 4 T_2 trees will be created and deleted. If it proves to be faster to reuse old structures rather than create new trees, it could be beneficial to recycle the trees considering how many times they are used.
The benefit would be the largest for the smaller trees. As mentioned this is because the smaller the trees, the more the structure is created and deleted. The other reason is due to how larger structures might slightly mutate over time due to deletions. Deletions might change the shape of the larger trees given enough removals. This might cause the structure to require more modification if more than a leaf node worth of data nodes are removed. Another reason for not storing the larger trees is that their less used than the smaller structures as the bigger the structure, the rarer it gets updated and changed. It might not be beneficial to use storage space to store an outdated structure which will be rarely used. For this reason it could be beneficial to store smaller structures, but only structures up to a given size may be beneficial to store and reuse.

5.4.3 Concurrent search trees performance

This section will go over and cover some of the results found in the RCU-HTM paper[2]. The results will be taken into consideration on how different synchronization mechanisms can affect performance. The RCU-HTM benchmarks multiple state of the art search trees using different synchronization mechanisms and these results will be generalized and used as a basis for a discussion on when different structures perform their best. The results this section is based on can be seen in figure 5.4.1 which is a snippet of the results from the RCU-HTM paper. The performance of the tests show how different state of the art binary search trees compare to the RCU-HTM solution presented in the paper. The discussion in the paper were more centered around how each implementation compared against the RCU-HTM, but in this section the general trend of the results will be the main focus. The results are based on testing different amounts of updates and look-ups on 3 different key ranges 200 keys, $20K$ keys and $20M$ keys. The 100% look-ups results are usually of the faster variant as most trees manage to avoid using to

many synchronization techniques on data during look-ups as there will not be any collisions. However the lock based solutions still requires checking if the variables are locked before reading adding a small overhead compared to some of the other structures. The interesting thing about the result for the purpose of a potential improved Bkd-tree is how the structures perform on different workloads during mixed workloads.
The trend of the data is that the more data nodes there is, the better the solutions perform. There are however some outliers such as the RCU implementation as the performance is overrun by the fact that the basic RCU implementation only allow for one thread performing updates, therefore not being able to fully take advantage of the other cores. For the rest of the implementations, the trend shows how increased workload and structure size increases performance. On the different implementations this is largely because of the same reason. There is more data, which mean the threads are generally more spread out and work on different memory locations and values, which result in less collisions and generally less needs for synchronization between threads. This lets threads work on their own memory area without being disrupted. In the case of lock based synchronization mechanisms, this means that fewer threads are locked up due to the threads working on separate memory locations. This phenomenon can especially be seen in the base HTM implementation where the performance skyrockets when working on $20M$ keys. This again is a result of HTM beeing an optimistic strategy which assumes no writes to memory collide and rather aborts instructions if two threads collide. As the threads work on such a large memory area, there are few collisions and the tree can have asynchronous like performance.
Where this becomes important for the Bkd-tree implementation, is in the memory buffer. If the concurrent structure were to use a simple array such as in the implementation and the one presented in the paper, inserts would end up being close to serial performance. The serial performance would be the result of multiple threads working on the same memory area. Multiple threads would try to fetch the current size of the array to know where it should insert a new data node, and this collision would lead to high overhead as multiple insert attempts would need to be rerun due to aborting or would be serialized if a lock were to lock the entirety of the memory structure. To overcome this obstacle and still have good memory performance, threads should optimally be made to work on their own memory area to avoid as many collisions as possible.

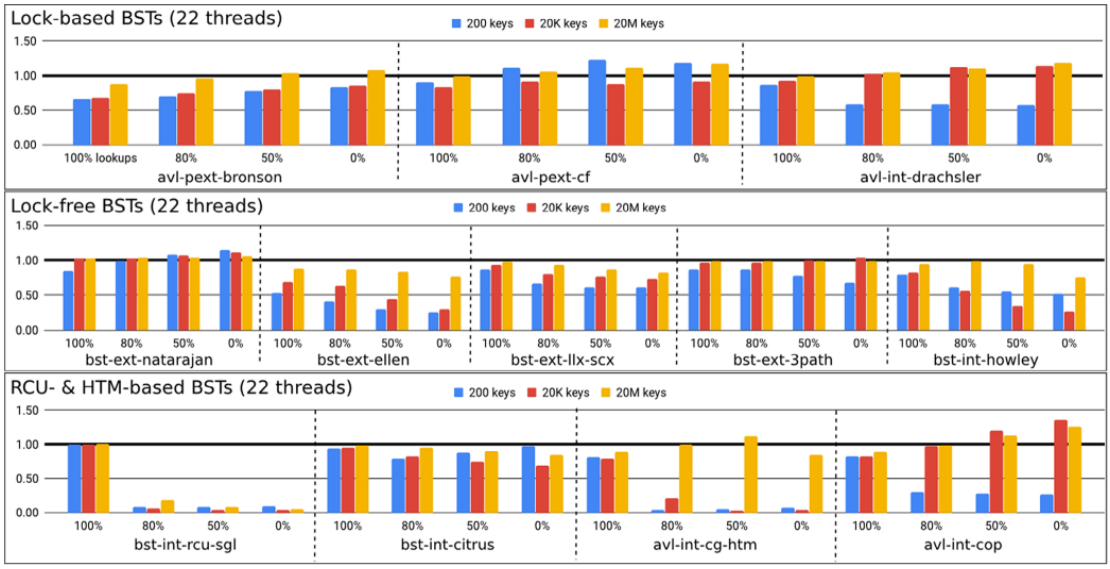


Figure 5.4.1: This is a snippet of results from the RCU-HTM paper. The results of each structure is normalized on the performance of the tree's RCU-HTM variant. [2]

5.5 Designing a concurrent solution

This section will take the results and observations from the serial results and serial discussion into consideration to suggest possible design and solution principles and how a Concurrent Bkd-tree best could be implemented. The goal of this section is to lay the ground work for designing a concurrent solution which can be the starting point for implementing a Concurrent Bkd-tree solution. The section will first cover some of the smaller hacks which can be used to boost performance. Then the most important challenge will be covered which is the memory buffer structure. The reason for the importance of the memory buffer, is that it is the area with the most density considering both threads working together and the most density of the data making it one of the most important sections for synchronization mechanisms and potential overhead.

5.5.1 Deletion strategy

The deletion strategy should use a tombstone strategy. This would greatly increase individual deletions while slightly increasing the cost of bulkloading as each value in the bulkloading file would need to be compared against the entries in the tombstone list. This could however be beneficial to a concurrent solution. The computational load of a single deletion instruction and gathering more of the highly computational tasks in one section allows for reducing toil on a single thread performing a operation. This is good as the highly computational task becomes a good subject for achieving speedup with concurrency. The tombstone list or structure is also a good area to take advantage of bloomfilters to speedup look ups.

5.5.2 Reusing tree structures

Small trees at the start of the structure are currently created and deleted multiple times. Instead the first structures could be stored in storage instead of being deleted. The structures could then be reused by updating their meta data, split values and leaf nodes. What tree sizes that are beneficial to store would vary based on the configuration of the given Bkd-tree. For example if the memory buffer is very large in a configuration there might be less benefit in storing larger trees as they would be less used and take up more storage space. On a configuration with a smaller memory buffer it could be more beneficial to store trees to a larger power of 2 as they would store less data as their size is dependent on the memory buffer size. To reduce the cost of storing used trees, their leaf nodes could be deleted as they are not needed. This would reduce the amount of storage needed by a given tree as it is the largest part of the structure. However, this might reduce the benefit as then new leaf blocks would need to be allocated instead of reusing the old ones.
One issue which could be problematic is when a tree is safe to overwrite. This would depend on the design of the whole tree and which synchronization mechanisms are used. For example when using an RCU like strategy, some threads might be reading old data that are no longer part of the updated tree. To overcome this a mechanism for keeping track of if a structure is safe to remove or overwrite should be used.

5.5.3 Bulkloading scheduling

From the testing, bulkloading proved to be the most expensive operation. Deletions, lockups and range search will in the worst case require to iterate through each tree structure. This might be expensive as each traversed tree would add a cost of $log_2(\frac{N}{LeafSize})$ where N is the number of nodes in a given tree. bulkloading data on the other hand requires fetching all data from potentially multiple structures, sorting them and iterating through the data in multiple occasions to send data sorted on different dimensions down the correct recursive call.
This makes the bulkloading algorithm a prime target to speed up with optimizations and more computational power with the use of multithreading. As the bulkloading algorithm have potential to take up so much of computational power, it might be beneficial to schedule larger bulkloads when the strain and workload on the structure is low. This could be done by implementing a scheduler overlooking the system. The scheduler could use live numbers such as number of operations performed by the system to determine if system resources should be allocated to larger bulkloading operations. A scheduler only reacting to live data could be problematic as there might be an unexpected increase in data inserts or other operations. It could therefore be beneficial to look at other factors such as history of previous operations over time. Then if it emerges a pattern of when there are peek hours. In those peek hours the scheduler could be less aggressive with bulkloading and rather save them for when the operation intensity is lower.

5.5.4 Memory structure

As mentioned the memory buffer structure where all nodes first get imported, is the most important structure to achieve good insert performance and utilize the threads the best way possible. When looking at inserts, all values will first be written to the memory buffer and without a proper synchronization mechanism adapted for the structure's purpose, the structure will not gain much speedup with the help of multithreading. An example of an improper solution would be a global lock over the whole memory buffer. This would mean that only one thread could work on the structure at once and effectively achieving serial performance as all other inserting threads would be locked and do no work. As the locked node might be in the middle of changing data, the structure would not be safe to read either meaning that readers would also be locked. Generally the more threads would need to work on the same memory location, the more overhead would be caused by requiring synchronization mechanisms.

5.5.4.1 Concurrency and thread cooperation

For this reason, it could be beneficial for each thread to insert data into its own memory location before further merging it into a global memory buffer shared with all threads. This would work as a intermediate merging step resulting in most inserts requiring little write synchronization as they would be written to a memory location only accessed by the inserting node. When the memory location is filled up, it could be turned into a shared resource by atomically placing the pointer to the structure into the global memory structure. The inserting node could then generate a new private structure. With this implementation the amount of synchronization and communication between nodes could be greatly reduced as the most common scenario would be to insert data in a private memory location, resulting in less synchronisation overhead. This implementation would work similar to RCU as each thread would work on their own private memory and then atomically update and share the resource with the other nodes. However, this solution would not have the problem of only being able to have one updater as this is overcome by having multiple private memory locations where nodes work. When the memory buffer eventually gets filled, the node which updated the final pointer could perform the first bulkload by taking the structure and making it into a disk array.

5.5.4.2 Lookup operation challenges

This solution would prioritize insert performance over lookup performance as some choices would need to be made to support lookup of all values. It might be problematic for a lookup thread to read values from a private area as it would have little guarantee that the area is not being changed by the inserting node. Depending on the use case of the structure and the necessity of updated data, one solution could be to have the solution be an eventually consistent structure where data is not considered fully inserted before it is placed in the global memory buffer. Another solution could be to have the lookup node first read the number

of nodes in the private structure and then only read over the confirmed written nodes. However if a lookup node are first going over the private memory areas and the the global memory buffer, it could happen that it read the same data twice. This could happen if the private data pointer is placed into the memory buffer during the lookup operation. A fix for this could be that each private memory structure could have an id and the lookup could assert if the structure had been read. This would however add more overhead to the lookup operation, so going with an eventually consistency approach could be easier and cheaper.

5.5.4.3 Deletion operation challenges

With the suggested architecture of the memory buffer, some questions regarding the deletion of nodes needs to be considered. The tombstone method for deleting nodes has been chosen as a good candidate for the deletion operation. The tombstone strategy takes the responsibility of the delete itself from the thread calling the operation over to the thread which ends up performing bulkloading. As this is the case, if a data node is attempted deleted while it is in the private memory location of a given thread, it becomes that threads responsibility to read the tombstone log and remove the data node. The thread should therefore assert that none of its inserted nodes is located in the tombstone log. The thread could do this by checking its values once the private memory buffer is full and if a node is deleted, the thread could fit a new node into the array. This makes it so nodes can be deleted before inserted into the global memory buffer resulting in less empty spaces in the tree.

5.5.4.4 Incoming operation queue

So far the general structure of how a Concurrent Bkd-tree could be implemented has been presented. However, there still is the question of how the structure should handle incoming operation calls and how operations should be split amongst threads. As the use case which will be looked into are operations received over a network from other machines, other computers would need a reliable and responsive API that can handle multiple operations. To achieve this, the incoming data management should optimally be scalable with more threads as to not be a bottleneck throttling the rest of the application. To create this the API should be able to use multiple threads which could answer calls and increase and decrease system resources to managing API calls depending on the load. If possible the threads answering the request should be the ones to perform the operation in the structure. This would make it so the thread would be able to answer the request as soon as the result is ready. The alternative would achieve the added overhead of returning the data to another communication thread which may be busy with another request. This could slow down the system due to the computational overhead. One solution could be to have a communication thread which schedules work depending on which threads are available, then the thread performing the computation or lookup could reply with the result of the request rather than the primary communication thread.

5.5.5 Thread manager

As mentioned through out this section, there are multiple areas of the solution where multiple threads can be beneficial to speed up performance. This includes everything from creating and bulkloading the structure, to receiving and answering API calls or operation requests. These parts of the implementation all might have a need for spawning new threads as workload increases. However a system for knowing how many threads a given section can spawn needs to be created. Each sections needs to know what resources are available and how many of those resources can be used. If this is not monitored the system resources could spawn more threads than the system supports causing crashes or throttle performance. To avoid this, there should be a central thread manager which can keep track of system resources and distribute the system resources to different parts of the application. For example, when there are a high operation workload the thread manager could dedicate more threads to handling these request. A given unit of the application could have a count of how many threads is given to the unit and the unit could check the count before spawning more threads. This way the solution could have a central resource manager which could be used to prioritize tasks based on the available resources.

5.5.6 Suggested design of Concurrent Bkd-tree

Based on the previous section, a general design for a Concurrent Bkd-tree has been created. The design can be used as a starting point for implementing a Concurrent Bkd-tree with focus on effective use of synchronization mechanisms with little overhead and sharing computational resources depending on the workload. This section will present the design seen in figure 5.5.1.

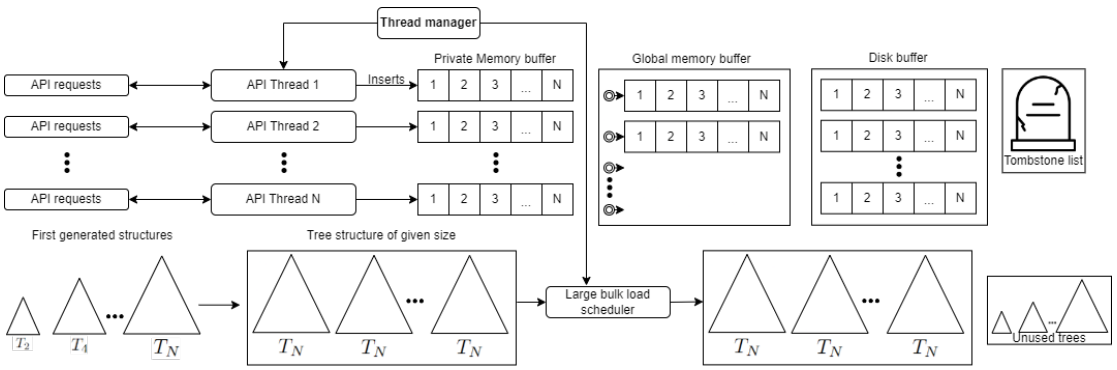


Figure 5.5.1: Suggested Concurrent Bkd-tree design

The API requests can be sent by other machines over a network and their request will be handled by a thread. The thread is responsible for communication with the API and for performing the operation specified by the caller. If a request for inserting is received, the thread will insert the data node in its own private memory buffer. If the buffer gets filled, the thread must first iterate over its values and assert none of them are located in the tombstone list. If a values is located in the tombstone list it gets deleted from the private memory buffer, which then has room for more nodes. When the private buffer eventually gets

filled, the pointer gets atomically inserted into the first available pointer in the global memory buffer. The thread creates a new private memory buffer as the old one now is part of the global memory buffer.
When the memory buffer eventually gets filled, it will be written to T_0 which is the first tree and is the disk buffer. This operation should be performed by the thread which filled the memory buffer. If the disk buffer is filled, a bulkload would be necessary. The calling thread then first look in the unused tree storage to look for a tree which can be reused of the correct size. If the tree is found, it is used, otherwise a new tree is created. The data in the old tree gets overwritten if the structure is reused, otherwise the tree is created based on the data.
Trees which got bulkloaded and are no longer in use should be placed in the unused trees list. If the new tree is of a set large size, it should be inserted into the tree storage list. The reason for this implementation is to avoid the calling nodes to be locked up in too large operations. So the trees will only be bulkloaded up to a certain size before being placed in storage. The trees in storage can be further bulkloaded by the thread manager scheduling larger bulkloads when there are system resources available. The larger trees gets merged together in the background when there is available resources.
When a delete operation is performed, the deleted value gets added into the tombstone list. This requires that each bulkload asserts that each data node to be bulkloaded is not located in the tombstone list. To be safe, other operations such as look ups an window queries should also assert that the return values do not occur in the tombstone list. It might be necessary to have writers and bulkloaders to work on a copy of the data to assert that the data read by look ups always is safe to read.

CHAPTER

SIX

CONCURRENT IMPLEMENTATION

In this section the design and implementation of a Concurrent Bkd-tree will be presented. The Concurrent Bkd-tree were designed based on findings and the suggested architecture in figure **??**. The concurrent implementation reuses code from the serial section and modifies it to suit a concurrent implementation. This chapter provides an overview of the modifications made to the different parts of the serial implementation as well as presenting the new components added to better suit a concurrent solution.

6.1 Programming language

The programming language used for the Concurrent Bkd-tree is C++. To implement concurrent C++ code, POSIX threads, also known as pthreads were used. Pthreads is a C library for creating and managing threads in multithreaded programs[12]. It allows the main thread to spawn new pthreads for concurrent execution within a single process. The pthread library also provides synchronization mechanisms such as different kinds of locks which will be utilized to keep the Concurrent Bkd-tree threadsafe. The C++ atomics library is used to make atomic updates and is used as an alternative to locks in critical sections throughout the application.

6.2 Solution structure

The project structure have been kept relatively similar to the serial implementation. Structures have been modified slightly to better suit the concurrent solution and some new building blocks have been added to support the new concurrency criteria. In the Concurrent Bkd-tree solution, the main sections are the Bkd-tree, Kdb-tree, MockAPI and the Scheduler. To support faster deletions, a simple tombstone has also been implemented. The structures are run and used by thread functions which work together asynchronously. The structure can still be configured through a configuration file which have been updated to fine tune the new structures and the communication between them. The Kdb-tree still works the same as in the serial version. The code of the Kdb-tree has been refactored and

each tree now include meta data to give the threads more control over the structures in the concurrent system. The Bkd-tree also works with the same general principles as presented in the serial implementation. However, the concurrent structure is more complex and therefore uses more underlying structures to both support synchronization and thread efficiency. The MockAPI is the API request block from figure **??**. Its primary function is to imitate an inserter API and is used by inserter threads to fetch data. The Scheduler is responsible for managing threads and to free old structures. The main thread runs the Scheduler function which then spawn other threads such as as inserter, reader and the large bulkloader threads. Together the mentioned sections make up the Concurrent Bkd-tree.

6.3 Kdb-tree

In the Concurrent Bkd-tree implementation, the Kdb-tree has been close to unchanged. The primary parts of the Kdb-tree structure is the same as in the serial solution. To keep track of which variation a Kdb-tree has, more meta data fields have been added. The fields include level and id. Level is used to Easily find out the approximate size of a given tree. When generating a Kdb-tree, the tree will be marked with a level which corresponds with the size and bulkload stage it were created at. This data point is primarily used by the thread handling larger bulkloads to match up trees of similar sizes. The id field is used for identification and each tree should have a unique id. This is used throughout the solution when a specific tree needs to be found.

6.4 Bkd-tree

The actual Bkd-tree structure of the Concurrent Bkd-tree is responsible for storing and supplying the data of the tree while protecting the resources with synchronisation mechanisms to keep the structure threadsafe. The Bkd-tree has four primary structures for storing and managing data. After being added to the tree by the inserter threads, the data gets stored in the global memory structure. The global memory structure is still an array of data points, but the way data gets added to the array has changed. The global memory structure is a multiple of the thread buffer size. This allows inserter threads to insert data in different sections of the array without colliding. To keep the structure threadsafe, threads will load the current global memory size and attempt to update it with a CAS operation. This lets the thread update the size and then copy its added nodes into the structure. Same as the serial version, the global memory gets pushed to the disk array when full. To avoid bottlenecks related to both array performance and synchronization mechanism overhead, the Concurrent Bkd-tree is designed as an eventually consistent structure. To elaborate further on this design choice, it means that the data points stored from the private thread buffers, global memory and global disk, will not be visible to readers. This means that the result of queries will be less accurate, but the overhead from synchronization mechanisms is lowered for inserters and readers.
Readers are however guaranteed to access all the tree structures generated at the point their query is called. This is implemented through a readable trees hash-

table. The readable trees structure is atomically updated with RCU to guarantee no need for blocking of reader threads. The readable trees structure works by being a collection of all the trees in the system which gets updated whenever a new tree is added or trees gets merged into larger trees. The readable tree gets updated by creating a copy of the old structure, updating it by deleting old and adding new trees, then replacing the original. As readers might still be reading deleted trees, the data will be reclaimed at a later time by the Scheduler.
When the global memory and disk is full, the inserter thread will bulkload them into the first tree structure. The tree will be inserted into the small tree storage which keep a small preconfigured number of trees. The small tree structure is an array of tree pointers. When an inserter thread performs the small bulkload, the thread will take all previous trees and combine them into a larger tree. This is the same strategy as seen in figure 1.2.1. The bulkloading inserter thread will use a small tree structure lock to assert that its the only thread inserting trees into the small tree structure. Another solution would be to use more fine grained synchronization mechanisms, this were not prioritized, but would allow multiple small bulkloaders. If multiple threads were waiting to bulkload, it would mean that the bulkload makes up most of the computational time of the inserter threads. This is undesirable as it would mean that inserter threads use more time restructuring data than working on the insertions. The small tree structure should therefore only perform bulkloads up to a level which will not throttle insertions.
When the new tree has been inserted into the small tree structure, it also needs to be inserted into the readable trees structure so reader threads can access the data. The readable tree structure then gets locked from writers to assert that there is only one updater of the structure. To update the structure in a read safe manner, a RCU update is performed. This works by taking a copy of the Readable tree structure, deleting maps which have been merged into the new tree and inserting the new tree. The readable map is then given an epoch value to mark which iteration the structure is on. When the updates are performed, the readable tree structure replaces the old pointer in the Bkd-tree and sends the old structure to the Scheduler.
As mentioned the small tree structure should be of such a size that bulkloads can be performed without throttling inserts. The structure therefore gets filled swiftly. When trees gets bulkloaded to a size the structure should not room, the tree will instead be inserted into a medium tree structure. The medium tree structure is a simple lock protected list which stores the largest trees bulkloaded by the insertion threads. The medium tree list is protected by a lock as it can be accessed by both an inserter thread inserting trees and by the large bulkload thread. The inserters and large bulkload thread therefore has a producer consumer relationship, where inserters produce trees which are further merged together by the larger bulkload thread. The trees pushed to the medium tree list are also added to the readable trees structure in the same manner as trees in the small trees list.
The final structure containing trees is the large trees structure. The large trees structure consist of trees merged together by the large bulkloader thread. It contains both trees merged together from trees from the medium tree list and the large tree structure. The structure itself is a list containing Kdb-tree pointers and is not locked as in the current solution its only accessed by the bulkloading thread.

As a proof of concept the Bkd-tree also has a tombstone structure for storing deleted nodes. This strategy could be an improvement over the implementation presented by the original Bkd-tree authors and the one presented in section 3.4. This outlines the primary components which together form the Bkd-tree structure of the concurrent implementation. The data flow of the structure will be presented in more detail in section 6.8.

6.5 MockAPI

The Concurrent Bkd-tree is implemented as a standalone structure which answer queries through an API. To simulate both inserts and window queries, a MockAPI structure were created. Depending on the test, the MockAPI may either generate random uniform data or data points with normal distribution. In order to make the data pseudo random, a thread local number generator were used. This avoids similar generated values which may occur in some timing-based random generators when using multiple threads.
The time it takes to access data nodes from the MockAPI is not representative of the time it would take to answer a request over a network, but serves as a tool to test the application. The MockAPI can be configured to add a delay to answering queries. This can be used to simulate network overhead. The MockAPI is a module which will be called directly by inserter and reader threads and offer functionality through function calls without its own Mock thread supplying data. This in turn lowers communication overhead, but may be less accurate to a real systems application flow as no communication is needed when fetching new data nodes or window queries.

6.6 Scheduler

The Scheduler structure oversees the entire system and is responsible for handling and communicating with threads. In the current implementation, the Scheduler is responsible for starting up threads, assigning tasks, reclaiming memory and shutting down the application. The Scheduler is a structure used by the Scheduler thread to communicate with other threads. It creates structures for the inserter and reader threads and can shut down the application by atomically updating an integer to set a given threads running status. After starting up the threads and the program, the Scheduler structure gives the Scheduler thread the needed tools to reclaim memory. The memory being reclaimed, is the memory of deleted Kdb-trees. As mentioned, the Concurrent Bkd-tree supplies reader threads with data through a RCU updated readable trees structure. At any given point, a reader thread may be reading the old version of the readable trees structure, which makes them unsafe to delete even after the readable tree in the Bkd-tree structure has been updated. When a bulkloading thread has updated the readable tree structure, the old readable tree together with the list of deleted Kdb-trees gets placed in a struct and the pointer gets atomically inserted into an array in the Scheduler structure. To check if a given structure is safe to delete, an epoch value is checked to assert threadsafety. The epoch value is a numerical value which is incremented each time the readable trees structure gets updated. The

epoch is stored in the readable tree structure, and when a reader thread access the readable tree, it stores the epoch value in that given tree. The Scheduler structure can then be used to compare a deleted structure's epoch against all readers to assert the thread is safe to delete. For example, if the smallest epoch stored by any read thread is 32, then no readable trees will be deleted with the epoch larger than 31. The current implementation the Scheduler is only responsible for memory reclamation during runtime. Since this task is small and dependent on waiting for reader threads reading updated memory, the process is set to yield while waiting. The thread will yield until the global epoch value has increased for a given amount, then attempt to delete old structures. The Scheduler is still an important component in an expanded solution where it would have responsibilities as dynamically allocating resources depending on the current system load.

6.7 Tombstone

The deletion strategy of the Concurrent Bkd-tree is a tombstone strategy. The tombstone consist of two sections. Each deleted node will be used to update the bloomfilter such that the deleted value can be searched for in the bloomfilter before searching the next structure. The tombstone itself is a hash table storing the data of the deleted data node. The tombstone is currently only a proof of concept solution and will not be the main focus. To be efficient, the tombstone should also be used with an eventual consistent strategy. This would mean the only thread checking and filtering out deleted nodes would be the large bulkloader thread. Not being eventually consistent would require checking each data node in every reader query against the bloomfilter which would significantly impact performance. The current tombstone implementation is made threadsafe using a read write lock. This means that readers may be blocked every time the bloomfilter or hash table is updated by a deleted node which would also affect performance greatly.

6.8 Thread functions

This section will cover the threads and the data flow. The threads uses the different data structures to together perform the tasks necessary for running the Concurrent Bkd-tree. The design of the structures lets the threads work together while attempting to minimize overhead related to synchronization mechanisms.

6.8.1 Scheduler

The Scheduler thread is primarily focused on tasks related to program startup, garbage collection through memory reclamation and shutting down the program. In the current solution the Scheduler thread starts by creating the Scheduler component, which again creates the other components such as Bkd-tree and the Mock-API. Based on the configured amount of threads specified in the configuration file, the Scheduler threads spawn the other worker threads. When program execution starts, the Scheduler thread will yield until the readable trees epoch reaches a set amount. At that point, its assumed that there is old data ready to be reclaimed. Garbage collection works by deleting all old Kdb-trees supplied in a list by the

thread which performed the bulkload. Another possibility would be to iterate over the old readable tree structure and deleting the trees marked as deleted. However, this adds overhead as each tree would be iterated through and each deleted tree is most likely not from the epoch currently being deleted. If these trees were deleted it could result in deleting structures still accessed by readers. It is important that the old trees gets deleted regularly as to not fill the array storing the deleted structures. This would cause bulkloaders to have to wait and cause the program to slow down while bulkloading threads waits for available slots. The Scheduler thread is also responsible for shutting down the program. To shut down, the Scheduler first signals all threads to stop operation. This is done by updating each threads' running integer which holds them in an infinity loop. This causes the loop to exit and the thread cleans up memory used during operation and exits. Before exiting, the thread again updates its running integer, signaling to the Scheduler that its no longer running. When all other threads has seized operation and signaled so to the Scheduler, the Scheduler can safely delete memory without crashing the program. Each structure can then be safely deleted before the Scheduler exits as well.

6.8.2 Large bulkloader

The large bulkloader thread is responsible for performing large bulkloads, allowing the inserter threads to have focus on inserting data. Trees which are inserted into the medium tree structure by the inserter thread gets combined into larger trees which are stored in the large trees structure. These trees will again be combined into larger trees by the bulkloader thread. In the current implementation, the medium tree structure only requires locking while inserter threads insert large trees and when the large bulkloader fetches trees from the structure. In an implementation with multiple bulkloader threads, the large bulkloader structure would also need to be protected. However, in the current solution there is only one large bulkloader accessing the large tree structure, therefore requiring no synchronization mechanism protecting the structure. The bulkloads work by selecting a set amount of trees from a given structure, then combining them into a larger tree. Each generated tree gets assigned a bulkloading level to indicate the approximate size of any given tree. This can again be used to match up trees of similar sizes when new bulkloads shall be performed. In the current solution the bulkloading level is iterated through in a loop were a bulkload of each level is attempted. If there are enough trees of a given size to meet the criteria set by the configuration, a bulkload will be performed. The current large bulkloading strategy do not consider the age of any given tree when performing a bulkload. This could be implemented, but it is assumed it would add no benefits. The original Bkd-tree is log structured where the oldest data nodes are stored in the largest structure. However as window queries require to look through all structures, its assumed that the age of data nodes is not of importance. If the primary focus was fetching singular data points, the age of a given data node may be more crucial to performance, assuming users usually access new data nodes. The bulkloading data structure is currently the only structure responsible for removing deleted nodes. Before creating a new tree, the bulkloader will need to iterate through all data nodes to look for deleted nodes, assuming deletions are enabled.

6.8.3 Inserter

As discussed in section 5.5.4, threads should optimally work on their own memory locations to avoid the need for synchronization mechanisms when possible. To avoid synchronization mechamism overhead, each inserter thread works on filling its own private thread buffer. When the thread buffer is filled, the thread atomically updates the global memory data node count. When the count is updated, the thread has reserved this memory space, and can safely insert its data by copying its thread buffer into global memory. This is a similar strategy as visualized and suggested in figure **??**. The data flow of the inserting node before generating trees can be seen in figure 6.8.1. Each individual inserter thread works on its private thread buffer. The threads insert data fetched from the MockAPI. When the global memory gets filled, it becomes the global disk if it is free and a new global memory is allocated. If both structures are full, the thread will store them both locally, updating the global memory to a new allocated structure and removing the global disk pointer. The thread can then safely work on performing a small bulkload while the other inserter threads can continue inserting data into the new global memory. The inserter thread could also be modified to skip global memory and global disk and instead go straight to creating tree structures. This modification lowers the communication between inserter threads. Each thread would then create a Kdb-tree structure based on their thread buffer, then insert it into readable trees and either the small or medium tree structure depending on the configuration.

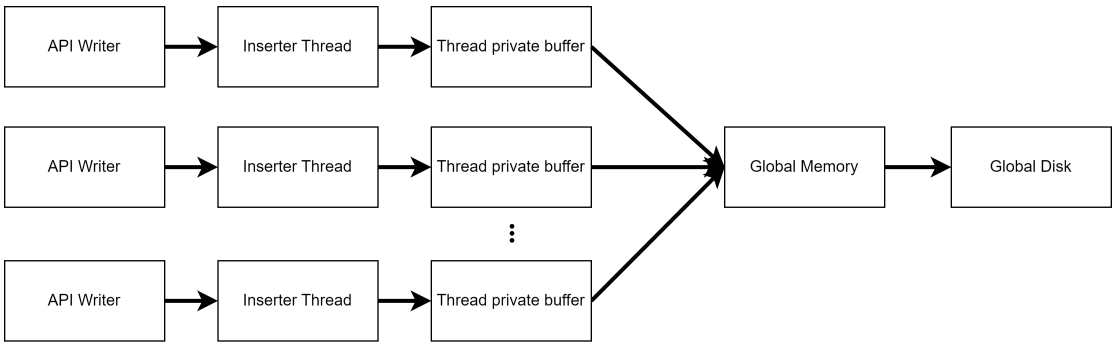


Figure 6.8.1: The data flow of the memory structure T_0^M and disk structure T_0 of the Concurrent Bkd-tree.

6.8.4 Reader

The reader threads are responsible for accessing data from the structure to simulate a typical workload. The reader threads works by calling the MockAPI, requesting a window size and then performing the query. The MockAPI will supply a window query specifying the range of data nodes in each dimension which should be extracted. The reader thread will then fetch the readable tree from the Bkd-tree and update its own epoch value. This will signal to Scheduler which old structures are safe to delete. The reader thread can then safely iterate through all the trees in the structure and perform the window query on each tree. In the current solution the reader thread adds the data to a list. The data nodes are only fetched for testing and benchmarking purposes, so the fetched data in the list

gets deleted before a new query is called. The reader threads will keep performing such queries until the Scheduler thread shuts them down. The reader threads are primarily created for testing and benchmarking, so singular data node queries have not been prioritized. The reader threads will fetch all data nodes matching their query from their current version of the readable tree structure. Kdb-tree structures which are added after the query has fetched the readable tree structure will not be part of the query. To assert data nodes fetched in the query are not deleted, all nodes would need to be checked against the Tombstone. This would be a viable solution, but the downside is the impact this would have on performance. The cost would come from iteration over all fetched nodes and synchronization mechanism calls needed to access the shared structure.

6.9 Data flow

The overall data flow of the structure can be seen illustrated in figure 6.9.1. It highlights how all tree structures in the trees insert their entries into the readable trees structure. As the small tree structure is protected as multiple writer thread may attempt to access it at once. Its protected by a small bulkloading lock asserting only one writer thread performs small bulkloads at a given time. The medium tree list creates a consumer producer relationship where writer threads will insert their largest trees and the bulkloading thread will take the trees and combining them into larger trees. The larger tree list do not have a lock as its only accessed by the single large bulkloader thread. It does however having the added computational overhead of checking for deleted nodes. The tree size will grow going from left to right in the structure. Small trees only go up to a set size as to not throttle inserts. Medium trees will only contain trees with a certain size. They will be merged into large trees and stored in the large tree list. The large tree list can have trees in a range of different sizes depending on the current bulkloading situation. All the tree storage structures will need to update the readable trees structures. This guarantees that all generated trees can be accessed by the readers of the structure. Reader threads can answer queries supplied from the reader API. The reader threads grab the most updated readable tree structure currently stored in the Bkd-tree and iterates through all trees performing a read query on every tree.

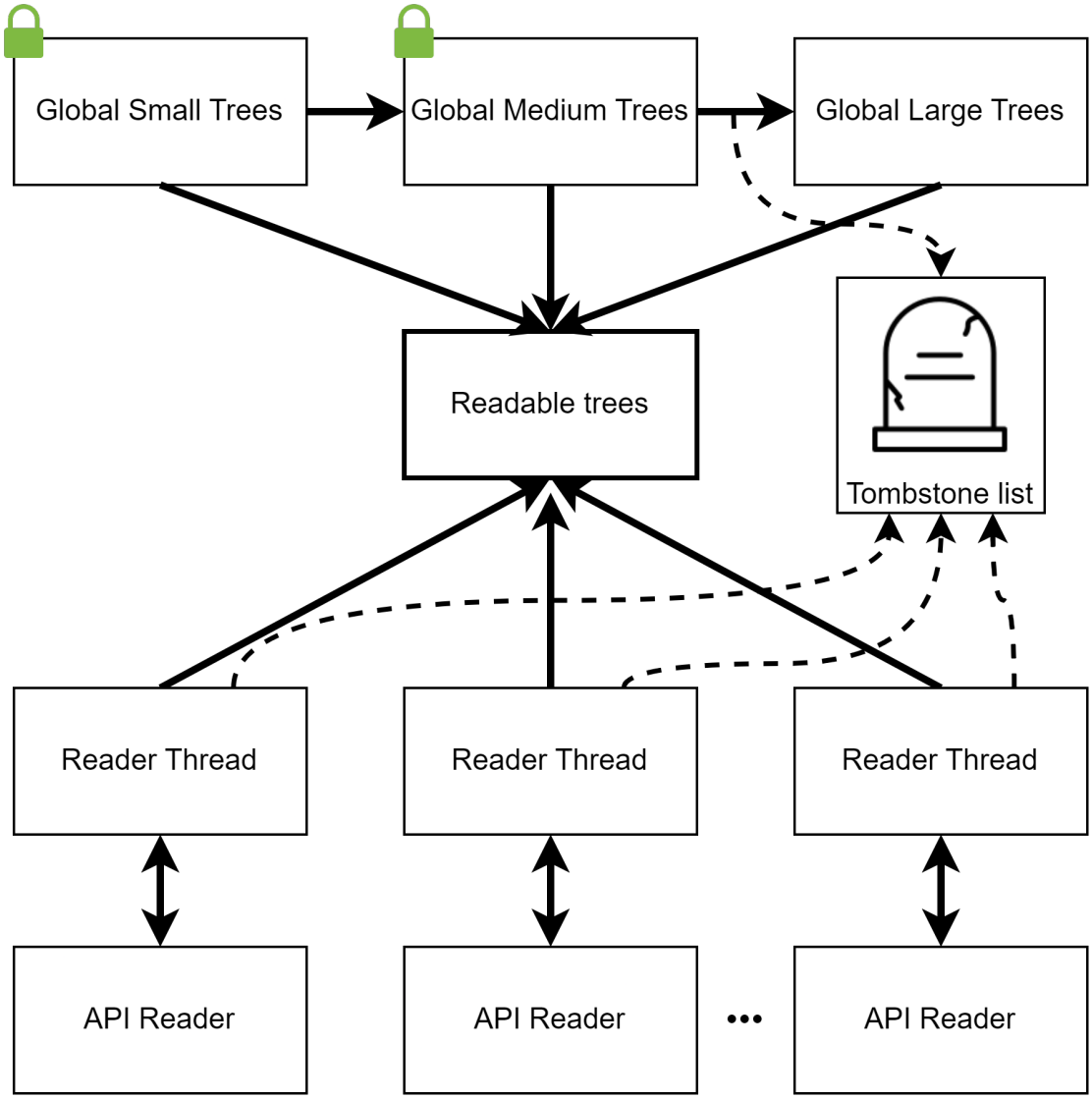


Figure 6.9.1: The data flow of the Kdb-trees in the Concurrent Bkd-tree

CHAPTER

SEVEN

CONCURRENT RESULTS

This chapter will cover the results and benchmark of the Concurrent Bkd-tree. Each data point in the graph figures is based on the average result of 10 independent runs. Each test may have a different configuration, with changes outlined for each test. A common aspect of each test is the use of a two-dimensional float array used as keys. Each data node stores a pseudo random string of 16 characters and each key is generated as a pseudo random uniform data float value ranging from 0 to 10000. In order to measure the performance of different parts of the solution, the code may have been altered to tests different sections of the implementation. The specific code used for each test can be accessed in the project's Github repository found in Appendix A.

7.1 Experimental platform

Similarly to the serial test, the Concurrent Bkd-tree was benchmarked with Chrono[11]. Performance tests were ran on a laptop with an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz processor with 4 cores and support for 8 threads. The system had 8GB of 2400 MHz RAM. The Fedora Linux 36 (Workstation Edition) operating system were utilized for the tests. The code were compiled with the -O2 flag for optimization and -lpthreads for pthread support. The compiler used were g++ (GCC) 12.2.1 20221121. To see how the Concurrent Bkd-tree scales on a server environment, some tests were also tested on a server leveraging multiple CPUs. The server used for testing had 16 Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz processors. Each physical processors has 8 cores, for a total of 128 available cores adding up to a possible 256 threads using hyper-threading. The server had 131.9 GB of RAM and the tests ran on a Ubuntu 22.04.2 LTS operating system.

7.2 Insertions in Concurrent Bkd-tree

This section covers results related to inserting data nodes into the Concurrent Bkd-tree. Two primary solutions have been benchmarked, one utilizing global memory and global disk and another without. In the first solution, inserter threads insert their nodes into a global structure which will later be bulkloaded into a tree. The latter works by having a larger thread buffer, which the inserter thread bulkloads

into a tree when full. Each solution works by fetching pseudo random uniform data from the MockAPI generated between 0 and 10000. Common for each test configuration is the Kdb-leaf size of 128 data nodes and two-dimensional structure. The insert tests vary between inserting 64 and 128 trees of a set size. In each test the main thread is responsible for scheduling the inserter threads. After starting the threads, the scheduler thread yields until all trees are created, at which point an inserter thread shuts the program off as the test is completed.

7.2.1 Insertions without global structures

This section will focus on tests benchmarking the time it takes to create trees without global structures such as global memory and global disk. In this implementation, each thread fills its private thread buffer and bulkloads it into a Kdb-tree of a given size. The trees are then inserted into a copy of the global readable tree structure which then replaces the global one and completes the RCU. The tests are performed with different tree sizes and switches between 64 and 128 trees per test. Smaller tests are benchmarked by measuring the time used in milliseconds(ms) and larger tests are measured in seconds(s). The timings in these tests are the time between when the first inserter thread starts running the insertion thread function and until all the 64 or 128 trees have been created. As inserter threads are spawned one after another, this means that when the first inserter thread starts, not all of the other inserter threads have been initialized. This may result in a slight favour for tests with few threads.

7.2.1.1 Inserting 64 trees of size 16384

The graph in figure 7.2.1 show the performance of inserting 64 trees with 16384 data nodes into the Concurrent Bkd-tree with increasing number of writer threads. The results show a massive improvement from going from a single inserter thread to utilizing 2 inserter threads. This improvement comes from having 2 threads which can work on inserting trees with data nodes with little communication overhead. Adding more inserter threads past the two first is still beneficial, but the performance gained is less compared to the initial increase from two threads instead of one. The reason for this is likely the small tree size. Since the tree size is only 16384 data nodes, each thread may have a smaller workload meaning each individual thread have less work to contribute. With the given workload, a single inserter thread completed the task in 136ms and 7 inserter threads completed it in 45ms. This is a speedup of 66.9%. However, it should be noted that on smaller tasks the overhead of spawning threads may be greater than the performance gained.

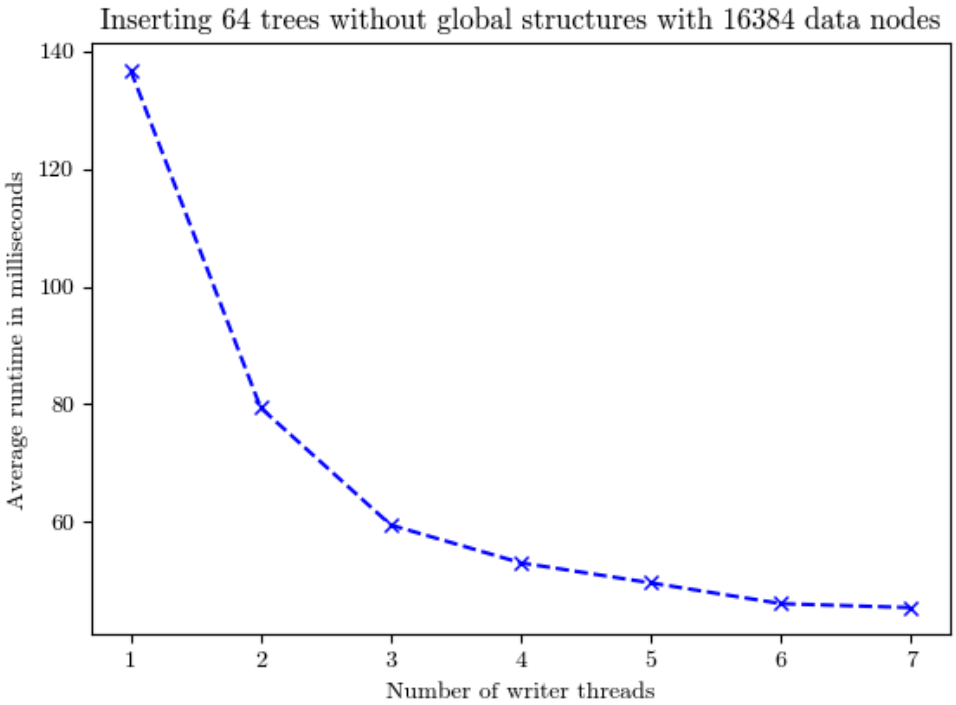


Figure 7.2.1: Inserting 64 trees of size 16384 without global structures

7.2.1.2 Inserting 64 trees of size 65536

As noted in the previous section, there is a large performance gain by having two inserter threads instead of one. In figure 7.2.2, the tree size is set to 65536, 4x larger than in the previous test. Due to the larger workload, the runtime of the program has increased, but the trajectory of the graph has improved. There is still a larger gap between one and two inserter threads, but the improvement of each added thread makes a more significant difference than in the previous test. This is a result of having more work, so the improvement from going from 1 inserter thread to 7 inserter threads has increased. Even though trajectory of the graph in figure 7.2.2 is similar to the trajectory seen in figure 7.2.1, the runtime has decreased from 576ms to 192ms. This is a speedup of 197.2%. These results point to the fact that the benefit of adding more threads is dependent on the work size.

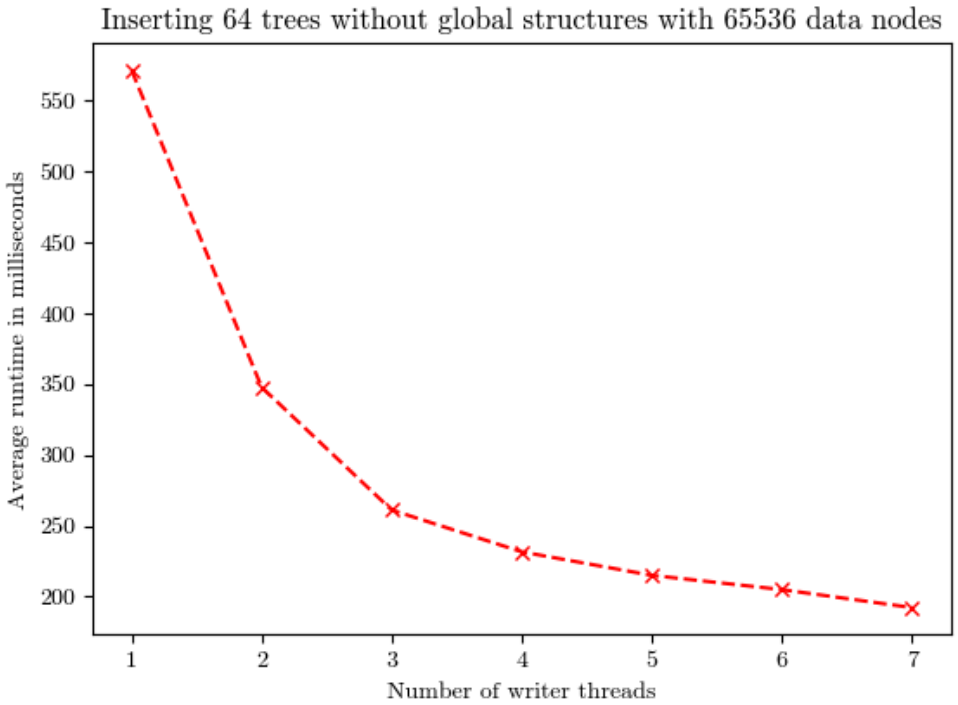


Figure 7.2.2: Inserting 64 trees of size 65536 without global structures

7.2.1.3 Inserting 64 trees of size 262144

When increasing the tree size to 262144 data nodes, the workload is again increased which increase the benefit of using more threads. In figure 7.2.3, there is a larger performance gain than seen in the previous figures. This is a result of scaling the workload 8x higher than in figure 7.2.1. As each individual thread has more computationally heavy work, the runtime increases, but the benefit of each added inserter thread is greater. A single inserter thread spend 26.31s on creating the 64 trees. While 7 inserter threads only spend 8.53s. This is equivalent to a speedup of approximately 207.9%, which again shows the increasing benefit of adding more threads as the workload grows.

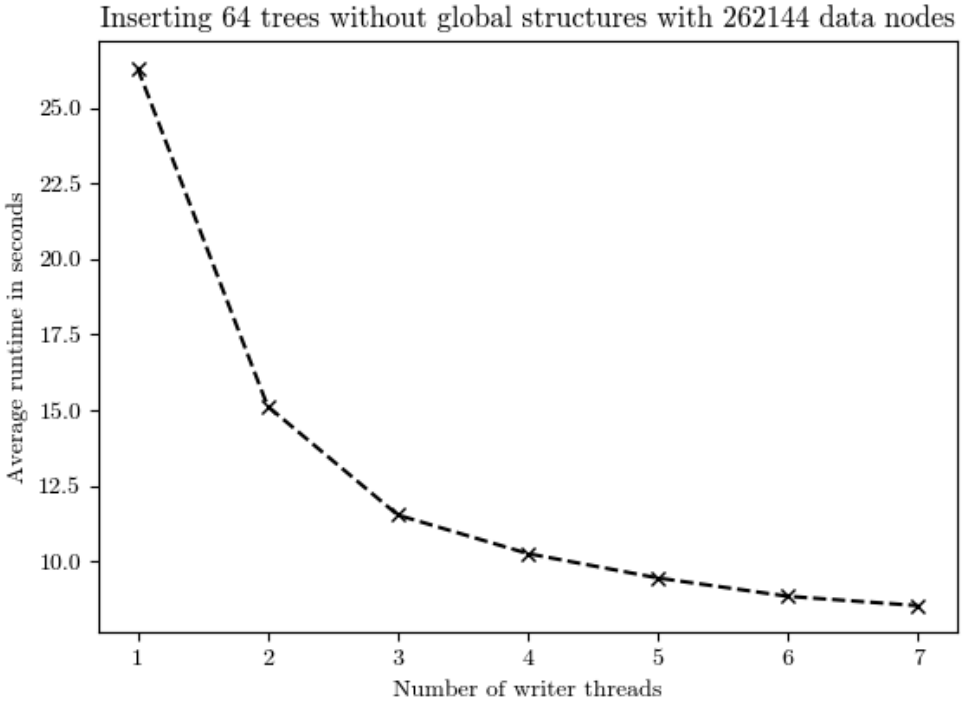


Figure 7.2.3: Inserting 64 trees of size 262144 without global structures

7.2.1.4 Inserting 128 trees of size 16384

So far, the workload has been increased by increasing the Kdb-tree size. Another way to increase the workload for each inserter thread, is to increase the amount of trees which needs to be created. In the previous tests, the workload has been increased by increasing the tree sizes. The next sections will focus on increasing the number of trees created to see how it affects scaling. In figure 7.2.4, 128 trees each with 16384 data nodes are created. The performance gained is similar to figure 7.2.1, but due to a larger workload, the performance gain by each inserter thread is larger as the program runs for longer meaning more inserter threads are beneficial. A single inserter thread spend 273ms creating the 128 trees, while 7 inserter threads only spend 85ms. This is equivalent to a speedup of approximately 221.7%. Even though the workload is only doubled, this improvement is over 3x larger than seen in the equivalent test with 64 trees in figure 7.2.1. This may point to the solution scaling better when increasing the number of trees compared to increasing the tree size.

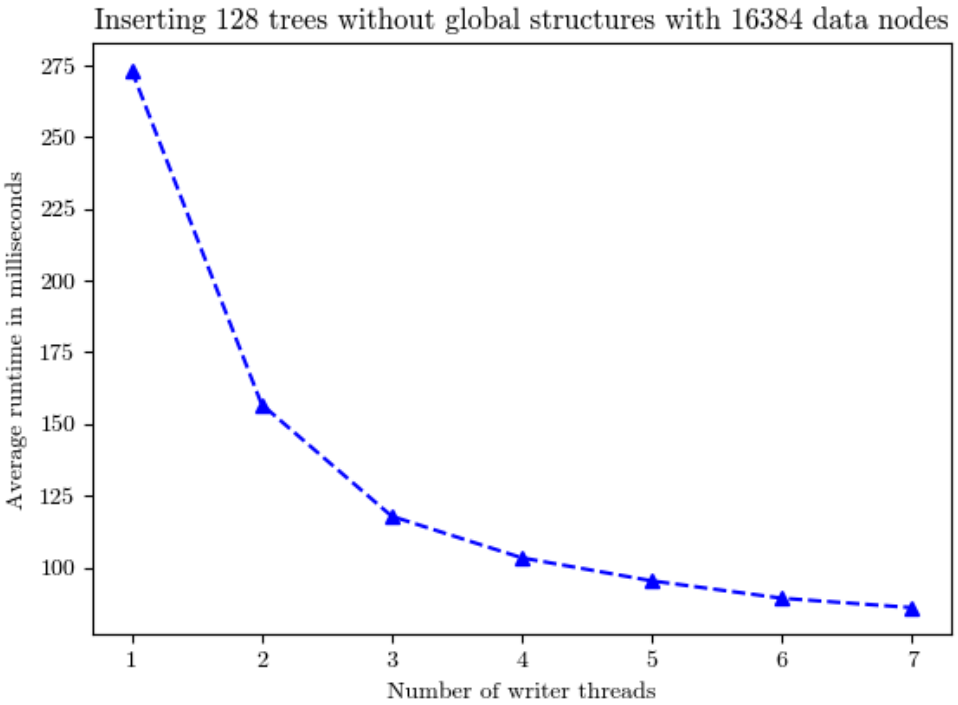


Figure 7.2.4: Inserting 128 trees of size 16384 without global structures

7.2.1.5 Inserting 128 trees of size 65536

When inserting 128 trees of size 65536 data nodes, the graph trajectory in figure 7.2.5 is similar to previous runs. Having just one extra inserter is greatly beneficial, but each inserter thread added after, gradually make less of an impact for a set workload. For this workload, a single inserter thread completes the task in 595ms, while 7 inserter threads uses 193ms. This is equivalent to a speedup of 308.8%, which continues the trend of greatly increasing performance as both the workload and number of threads scale together.

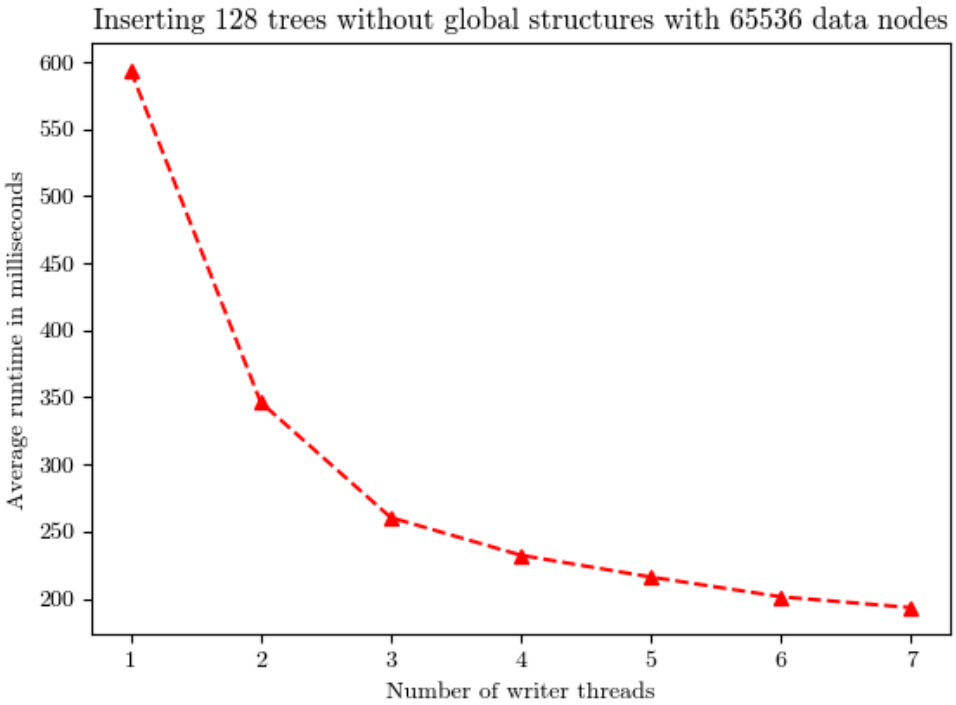


Figure 7.2.5: Inserting 128 trees of size 65536 without global structures

7.2.1.6 Inserting 128 trees of size 262144

Creating 128 trees each with 262144 data nodes, scales similarly to the other large workload inserts. As performance is gained by increasing the amount of inserter threads, scaling the problem size larger seem to always be beneficial to multiple inserter threads. Figure 7.2.6 continues this trend. It does however prove that for a given workload, the Concurrent Bkd-tree scales in accordance to Amdahl's law. However, as the communication overhead stay relatively the same when increasing the workload, the application can still benefit from more threads by increasing the workload to scale in accordance to Gustafson's law. This is again seen in this test were a single inserter thread spends 52.53s on the given tasks while 7 inserter threads only spend 16.27s. This resulted in a speedup of 222.1%. This is a smaller increase than seen in other tests, which may insinuate that when the individual workload grows too large, the benefit of concurrency decreases. This may be a result of creating larger tree structures which causes more cache misses as each individual thread sorts and store more data nodes during bulkloading. This could explain why 1 inserter thread is less affected as it has more of the cache to itself causing fewer cache misses.

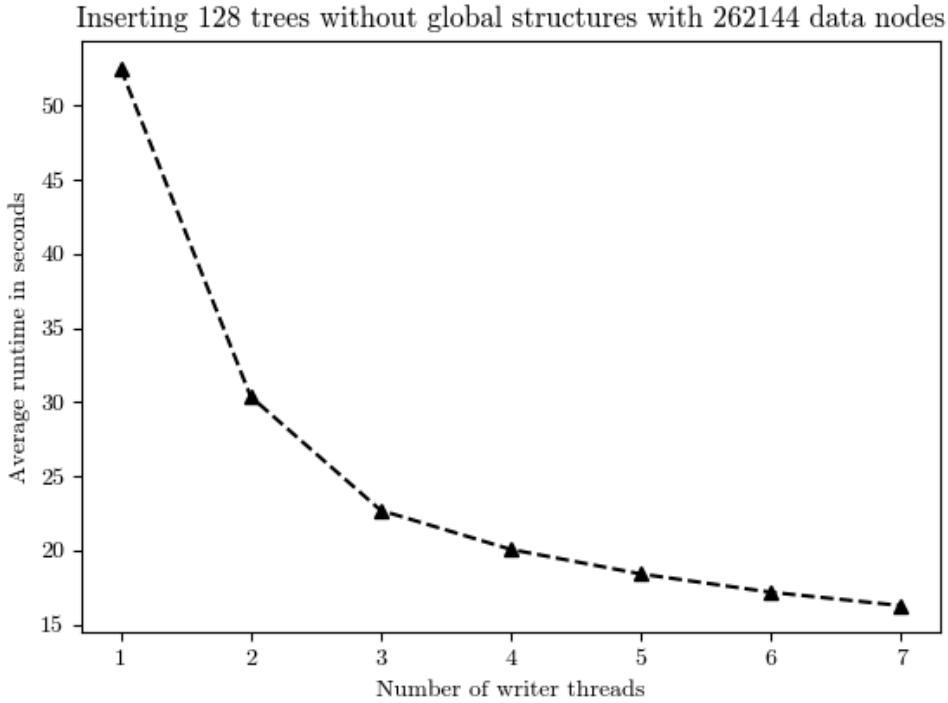


Figure 7.2.6: Inserting 128 trees of size 262144 without global structures

7.2.1.7 Inserts per Second

This chapter has so far covered the performance of inserts without global structures by comparing runtime of different configurations. To get a better understanding of the results, the approximate Inserts per Second(IPS) have been calculated. The table of all the IPS for every run performed in the laptop environment can be seen in table 7.2.1. The result is an approximate calculated with the formula $IPS = (treeSize * numberOfTrees)/timeSpent$. From the results it is clear that for the laptop environment, increasing the number of inserter threads is always beneficial. The results do also show that there is a trade off between increasing tree size and IPS. The sweet spot for achieving the most inserts is 64 trees each with 65536 data nodes.

Table 7.2.1: Inserts per Second (IPS) for Different Configurations

Number of Trees and Tree Size	Inserter Threads						
	1	2	3	4	5	6	7
64x16384	1191.71	2073.73	2742.15	3072.68	3287.72	3545.89	3590.03
64x65536	1134.38	2350.13	3122.89	3498.57	3786.91	3985.81	4228.15
64x262144	997.68	1736.97	2273.83	2567.69	2791.03	2970.01	3083.26
128x16384	598.40	1041.36	1387.68	1577.63	1710.33	1833.83	1896.46
128x65536	1093.44	1894.83	2524.44	2827.86	3050.37	3246.88	3381.19
128x262144	497.47	858.06	1147.46	1295.32	1411.75	1510.23	1598.69

7.2.1.8 Inserting 128 trees of varying sizes on server

To further test the scalability of the Concurrent Bkd-tree, the structure were tested in a server environment. To test how different workloads affect performance, trees with 16384, 65536 and 262144 data nodes were tested. Each test monitored the time it took to generate 128 trees of the given size. The results from the tests can be seen in figure 7.2.7. The results show that increasing the workload by having each thread creating larger trees, increase the performance gain. Comparing the graphs, creating 262144 data node Kdb-trees, yields a much larger performance gain than smaller trees. This is likely due to the powerful server CPUs require a higher workload to fully utilize the computational power. An interesting observation from the server tests is how adding more threads than required for a given workload, ends up decreasing performance. In all three tests, the computational time spent decreases and reach its lowest point when using 16 threads. After that the computational time increases. The reason may be that creating 128 trees is too small of a workload for more than 16 threads and that more trees should be created. However, if this were the case, creating larger trees should also help to increase the workload. The tree sizes 16384, 65536 and 262144 experience a decrease in their performance of -216.2%, -315.1% and -171.4% between 16 and 128 threads. As the largest tree size comparatively looses less performance, the tree size might be a small contributing factor, but likely not the main bottleneck. Another likely factor for the reduced performance is the server's hardware infrastructure. The server has 16 physical CPUs. Considering each physical processor has 8 cores for a total of 16 threads in the server environment, the decrease in performance may be a result of communication between physical CPUs being expensive. Even though communication between threads is limited in the Concurrent Bkd-tree solution, threads still need to share data by updating the readable tree structure. This may be more expensive in an environment were threads need to communicate across CPUs. This is not an issue when utilizing 16 threads or less. Then the task can be performed on a single CPU, when going over that threshold, more communication cross CPUs are required, which may significantly decrease performance.

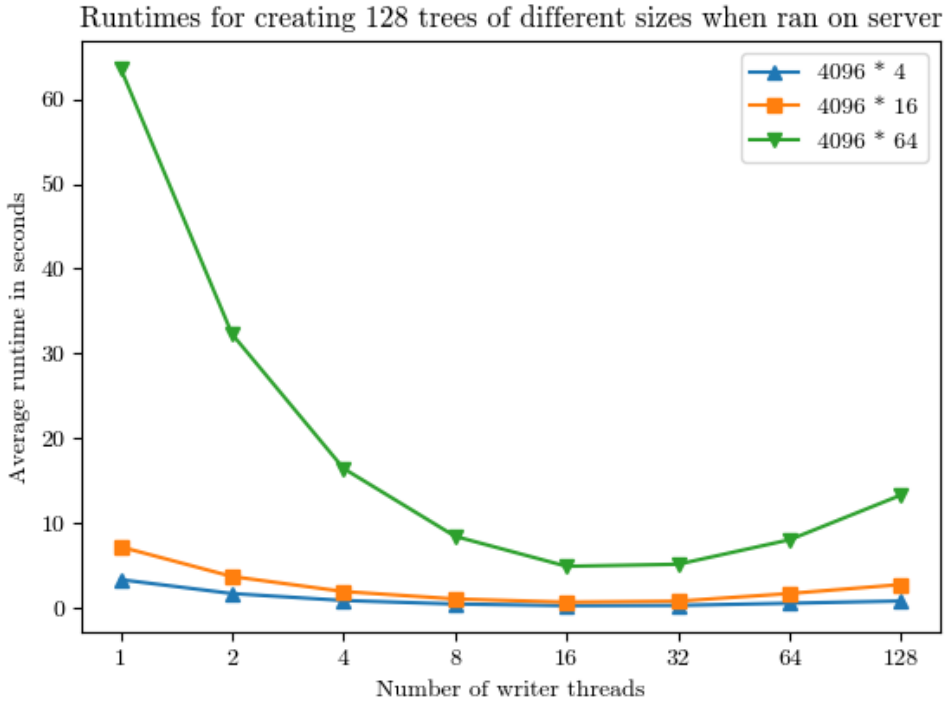


Figure 7.2.7: Inserting 128 trees of varying sizes on a server

7.2.2 Inserting trees with global structures

This section will cover the performance of the Concurrent Bkd-tree using global structures. The original Bkd-tree structure used an array of data nodes as a memory structure, when the array got filled up, it would be flushed to disk. To optimize this approach for a multithreaded environment, the Concurrent Bkd-tree were created with a global memory structure shared between inserter threads. Figure 7.2.8 shows how the performance of creating trees with the global structure. The global memory is set to store 64 thread buffers, each with 4096 data nodes which then creates a Kdb-tree of 524288 data nodes. As seen from the trajectory of the graph, it is beneficial to use two inserters instead of one, each inserter thread added after that, gradually slows down the performance. This make it seem like the synchronization overhead related to keeping the global structures threadsafe slows the program down. This would also explain why the performance slightly decrease when adding more threads, as more inserter threads access shared resources. In an attempt to increase the individual work of each thread, the global structures were set to store 16 thread buffers and the thread buffer size were set to 16384. This change in the configuration still kept the same workload of creating trees of 524288 data nodes, but each thread would in theory spend more time inserting data into its own thread buffer and less time inserting data into the global structure. The results of this modification can be seen in figure 7.2.9. From the result, it is clear that the solution is bottlenecked by the global structure which slows down the application. The primary bottleneck is the fact that only one thread is configured to bulkload at once. In theory this should let all threads work on the global structures while a single thread bulkloads data. This

approach may work if inserting data nodes were more computational expensive than bulkloading. In the current solution, it manifests on inserter threads waiting on global resources. To avoid this communication overhead, global structures should rather be avoided where possible and only use the necessary amount of communication between threads.

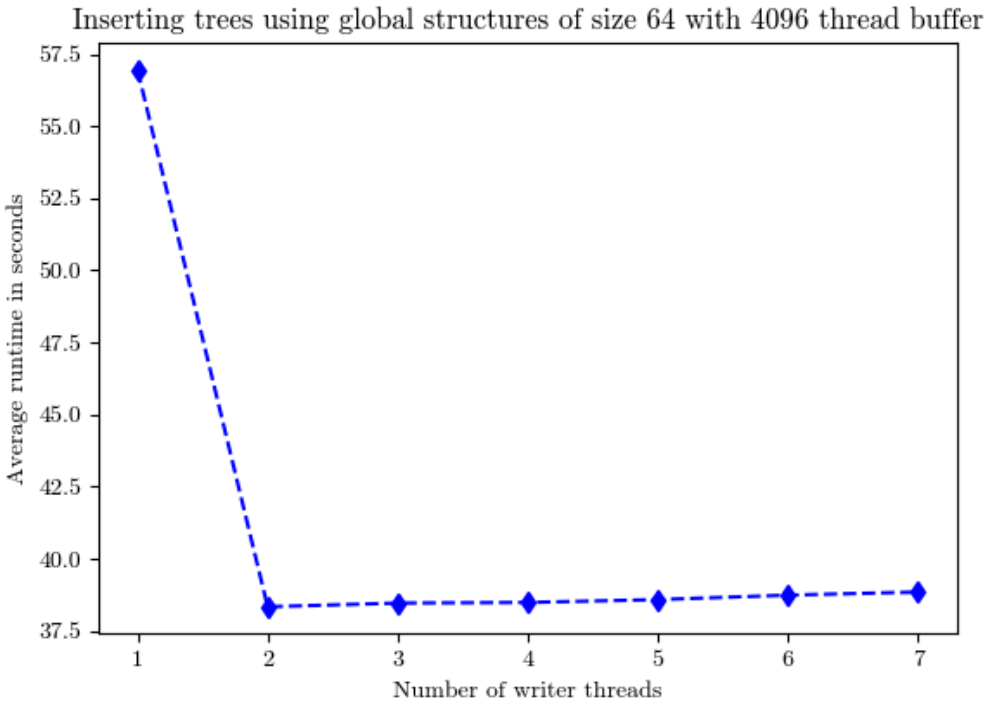


Figure 7.2.8: Inserting 64 trees using global structures of size 64 with a thread buffer size of 4096

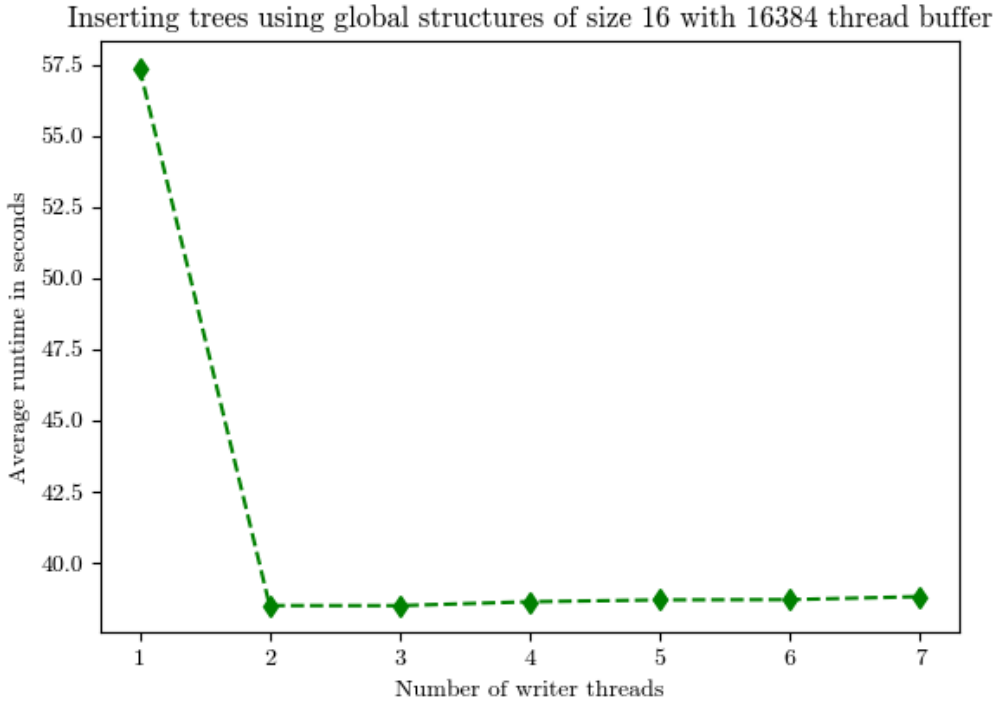


Figure 7.2.9: Inserting 64 trees using global structures of size 16 with a thread buffer size of 16384

7.3 Fetching data from Concurrent Bkd-tree

This section will cover the benchmarking of reader threads fetching data nodes from the Concurrent Bkd-tree. These tests will benchmark performing window queries on the Concurrent Bkd-tree configured with different tree sizes. A large data pool were used for the tests, so the tests were performed on the server environment to supply the program with enough RAM. The Concurrent Bkd-tree were first filled with 16777216 data nodes, in different configurations. After the tree was filled with a given amount of Kdb-trees of a set size, a reader thread was spawned and given a window query of a set range. Data nodes were pseudo randomly generated in the key range 0 - 10000 of each dimension with a uniform distribution. To fetch approximately 10%, 25% and 100% of data nodes, window queries of the size (0-3162), (0-5000) and (0-10000) were selected. New data were generated for each test. To test how distribution of the data affected the query results, the 16777216 data nodes were distributed between 1,2,4,8, ... and up to 16384 Kdb-trees.

7.3.1 Average window query time

Figure 7.3.1 shows the time it takes to perform a window query and fetch different percentages of the data. The x-axis show the number of trees the 16777216 data nodes were distributed across and the y-axis show the query time in milliseconds(ms). The results show a slight improvement when accessing data from a few larger trees rather then multiple small ones. The trend of the data seem to skew

towards that window query time is slightly improved as data nodes are distributed across fewer larger trees. The performance gain seem to be similar between query ranges, though accessing a larger percentage of data takes more time. There is however only a 11-18ms improvement of having larger trees in the tests compared to multiple smaller trees. This may suggest that frequent bulkloading may not be necessary due to the small improvement gained. When normalizing each data point between 0 and 1 as seen in figure 7.3.2, there is a clear correlation between the results. The mean Pearson correlation coefficient between the data sets is 0.93, which points to a strong correlation between the results. This indicates that the overhead of fetching higher percentages of the data, primarily comes from fetching larger sections of the tree structure and that the overhead is consistent across the query sizes.

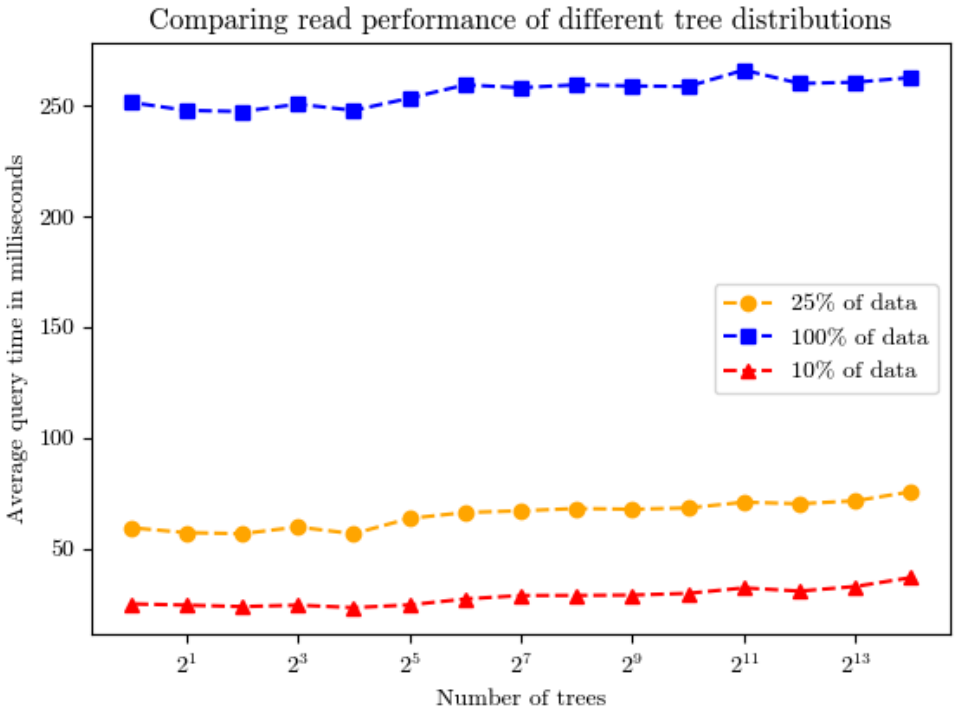


Figure 7.3.1: Window query times when fetching 10%, 25% and 100% of data.

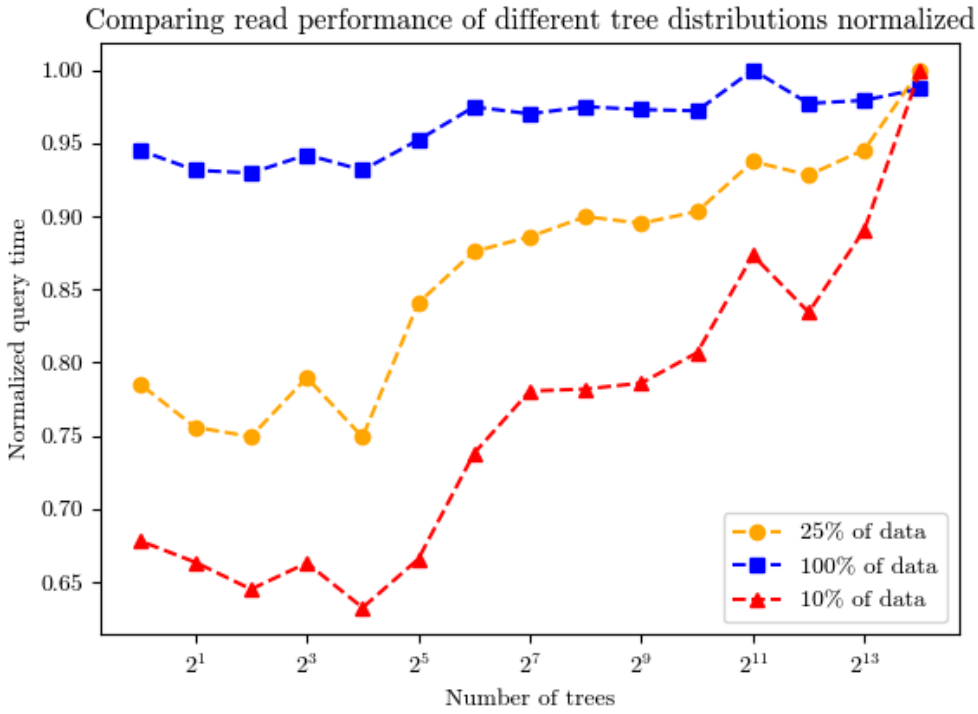


Figure 7.3.2: Window query times when fetching 10%, 25% and 100% of data, with the results normalized.

7.4 Thread performance

This section will take a closer look at the thread performance through different tests. To profile the thread performance, Intel® VTune™ Profiler(Vtune)[13] were used. Vtune can be used to benchmark many performance metrics and look at many aspects of a solution. This thesis looks at the performance benefit of multithreading. Therefore, the focus area has been put on performance related to threading. To gain the most insight, insertion tests with and without global structures have been analyzed to look at differences related to communication overhead. Due to limitations related to system access, GNU gprof(gprof)[14] were used to perform a flat profile of code executed on the server. Gprof version 2.37-37.fc36 were utilized.

7.4.1 Insertion without global structures

To get a deeper understanding of how the execution time is used, Vtune were used to benchmark the application. Figure 7.4.3 shows the top down view of the execution time given in percentages for each function call. The figures display an overview of resource usage, but not all functions have been included in the figure. Such as the recursive function calls of 'KdbCreateBranch´ and the underlying functions of 'fetchRandom'. The full analyses of each benchmark can be found in the test branches of the project's GitHub repository found in Appendix A. The program executed in this profile, is the same program as tested in figure 7.2.1. However, this section only profiles the program using 1 inserter thread and

another run utilizing 7 inserter threads. Figure 7.4.1 shows the execution time when running with 1 insertion thread. As seen in the figure, the effective runtime is 49.8% while the total spin time is 50.2%. The spin time comes from the scheduler, which only responsibility is to spawn the thread in the test. The same can be seen in figure 7.4.2, were the spin time is 12.8% due to utilizing 7 inserter threads instead of 1. Normalizing the results gives a better indication of how the results compare with each other. After normalizing the effective time percentage to 100%, the two runs becomes more comparable. The normalized results shows that a single inserter thread spend approximately 50.8% fetching data nodes and 48.7% bulkloading. The bulkload time is distributed between 35.1% sorting data nodes and 13.4% building Kdb-trees. The 7 thread inserters spend 60% fetching data nodes and 48.7% bulkloading. Of that time, 34.9% is spent sorting and only 4.8% of the total run time is spent building tree structures. This shows that the 7 inserter threads spend approximately 9.5% more time on creating and fetching values compared to the single thread on the same workload. This may be caused by an initial cost of the ‘fetchRandom‘ function. To keep the data random and avoid shared resources, each thread needs to initialize its own random generator which may be an expensive cost which is reflected in the results. This cost is extra apparent due to the short runtime of the given test as seen in figure 7.2.1 to be between 136-45ms.

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	49.8%	50.2%
_start	0.0%	50.2%
__clone3	49.8%	0.0%
start_thread	49.8%	0.0%
_threadInserterTree	49.8%	0.0%
MockApi::fetchRandom	25.3%	0.0%
func@0x9ea30	0.4%	0.0%
BkdTree::_smallBulkloadT	24.2%	0.0%
std::sort<DataNode*, da	17.5%	0.0%
KdbCreateTree	6.7%	0.0%
KdbCreateBranch	5.6%	0.0%

Figure 7.4.1: Performance of 1 inserter thread

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	87.2%	12.8%
_start	0.0%	12.8%
__clone3	87.2%	0.0%
start_thread	87.2%	0.0%
_threadInserterTree	87.2%	0.0%
MockApi::fetchRandom	52.6%	0.0%
BkdTree::_smallBulkloadT	34.6%	0.0%
std::sort<DataNode*, da	30.5%	0.0%
KdbCreateTree	4.2%	0.0%
KdbCreateBranch	3.7%	0.0%
__memcpy_avx_unal	0.4%	0.0%

Figure 7.4.2: Performance 7 inserter threads

Figure 7.4.3: Thread performance of smaller insert workload

Figure 7.4.6 shows the profile of creating 128 trees of size 262144, the same workload as seen in figure 7.2.6. Running the program with both 1 and 7 inserter threads yield similar results as seen in the previous section. After normalizing the results to not include the spin time, 1 inserter thread spend approximately 41.4% fetching data nodes and 57.7% on bulkloading. Utilizing 7 inserter threads gives similar results with 43.5% spent fetching data nodes and 55.9% spent bulkloading as seen after normalizing the results. An interesting observation is how a single inserter thread spend 39% sorting and only 18.4% building Kdb-trees. Seven inserters spend 35.6% sorting and 20% building trees. This may be caused by added expense of creating and managing large structures when multiple threads share the same cache and memory, making data handling slightly more expensive. In this test, time spent fetching values is significantly reduced. A single inserter thread spend 9.4% less time and 7 inserter threads spend 16.75% less time on the fetching random values, compared to the smaller test in figure 7.4.3. The

gap between percentage spent generating data has also drastically reduced with 1 inserter thread using 41.4% and 7 inserter threads using 43.5% of the runtime generating data nodes. This seem to support the previous assumption of the startup cost of the 'fetchRandom' function being expensive and the percentage of the runtime is therefore reduced as the workload grows. Comparing the time spent building trees with the larger workload, shows that a higher percentage is spent creating Kdb-trees in both runs in figure 7.4.6. A single inserter thread spends approximately 18.7% creating Kdb-trees. This is an increase of 5.3% compared to the results from the smaller workload. Seven inserter threads have a even larger increase, now spending 20%, an increase in 15.2%. This is a result of increasing the workload by creating more and larger trees. Another observation is the lack of the 'updateReadTrees' function in the results. The 'updateReadTrees´ function is responsible for updating the readable trees structure and is assumed to be the most expensive communication call in the current solution. This points to the synchronization mechanisms causing less overhead than initially expected. This means that a significant amount of runtime is spend computing and that the inserter threads likely have a smaller communication overhead than expected.

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	50.1%	49.9%
_start	0.1%	49.9%
__clone3	50.0%	0.0%
start_thread	50.0%	0.0%
_threadInserterTree	50.0%	0.0%
MockApi::fetchRandom	20.8%	0.0%
func@0x9ea30	0.0%	0.0%
BkdTree::_smallBulkloadT	28.8%	0.0%
std::sort<DataNode*, da	19.6%	0.0%
KdbCreateTree	9.3%	0.0%
KdbCreateBranch	8.6%	0.0%

Figure 7.4.4: Performance of 1 inserter thread

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	87.2%	12.8%
_start	0.1%	12.8%
__clone3	87.2%	0.0%
start_thread	87.2%	0.0%
_threadInserterTree	87.2%	0.0%
MockApi::fetchRandom	38.0%	0.0%
func@0x9ea30	0.1%	0.0%
BkdTree::_smallBulkloadT	48.8%	0.0%
std::sort<DataNode*, da	31.1%	0.0%
KdbCreateTree	17.5%	0.0%
KdbCreateBranch	16.2%	0.0%

Figure 7.4.5: Performance 7 inserter threads

Figure 7.4.6: Thread performance of larger insert workload

7.4.2 Insertion with global structures

From the previous sections, it is clear that insertion with global structures greatly increase runtime and the added synchronization mechanisms increase overhead. To take a closer look on why and how performance takes a hit, insertions with global structures have been profiled. The configuration used for the test is the same as used in figure 7.2.9 with a thread buffer size of 16384 data nodes, a global structure size of 16 and creating 64 trees. Figure 7.4.8 shows the total runtime of the program to be 64.7%. In the profiles without global structures, the total runtime is set to 87.2%. The yielding scheduler thread accounts for the remaining 12.8% of the runtime. Figure 7.4.8 only has 64.7% instead of 87.2% because about 20% of the total runtime is lost due to synchronization overhead. Most of the time is spent waiting for the bulkloading lock. The time spent fetching and generating data nodes is also reduced from 38% as seen in figure 7.4.5, to only 25.9% in figure 7.4.8. This points to global structures causing synchronization overhead which greatly reduce the insert speed of the solution compared to the insertion strategy without global structures.

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	51.7%	48.3%
_start	1.7%	48.3%
__clone3	50.0%	0.0%
start_thread	50.0%	0.0%
_threadInserter	50.0%	0.0%
MockApi::fetchRandom	19.6%	0.0%
func@0x9ea30	0.0%	0.0%
BkdTree::_bulkloadTree	30.1%	0.0%
std::sort<DataNode*, da	19.2%	0.0%
KdbCreateTree	10.6%	0.0%
__memcpy_avx_unalig	0.2%	0.0%

Figure 7.4.7: Performance of 1 inserter thread

Function Stack ▾	CPU Time: Total	
	Effective Time	Spin Time
Total	64.7%	35.3%
_start	1.5%	35.3%
__clone3	63.2%	0.0%
start_thread	63.2%	0.0%
_threadInserter	63.2%	0.0%
MockApi::fetchRandom	25.9%	0.0%
func@0x9ea34	0.0%	0.0%
func@0x9ea24	0.0%	0.0%
BkdTree::_bulkloadTree	36.7%	0.0%
std::sort<DataNode*, da	22.8%	0.0%
KdbCreateTree	13.4%	0.0%

Figure 7.4.8: Performance 7 inserter threads

Figure 7.4.9: Thread performance of inserts with global structures

7.4.3 Active thread time

To give a better look into the global structure overhead, this section will look at the thread runtime of different solutions. Figure 7.4.10 shows the performance of running the program with a single inserter thread. The main thread is only responsible for spawning the inserter thread and then yields. The single inserter thread is still affected by synchronization overhead, but it runs constantly. The same can not be said for figure 7.4.11. Figure 7.4.11 shows the thread performance of running 7 inserter threads with global structures. The white area marked during each threads' run, is time spent locked and waiting for resources to unlock. Most of this time is spent waiting for the bulkload lock to unlock. It is still faster than using a single inserter thread, but a lot of computational power is lost waiting for locks. Figure 7.4.12 shows the performance of an inserter test run without using global structures, the run has some pitfalls, likely related to other tasks being prioritized by the system, but no significant amount of computational time seem to be wasted.

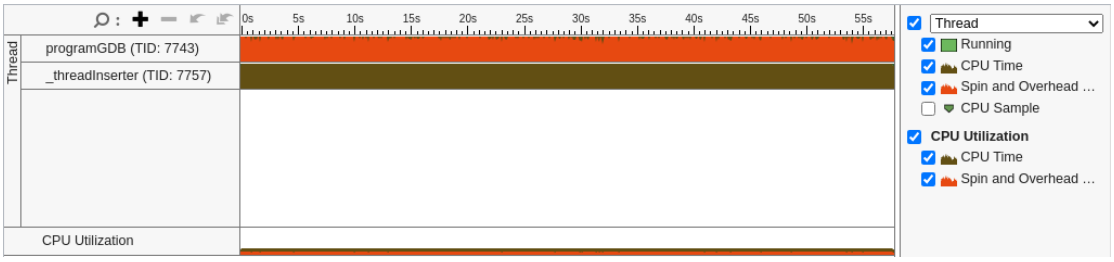


Figure 7.4.10: Performance of 1 inserter test with global structures

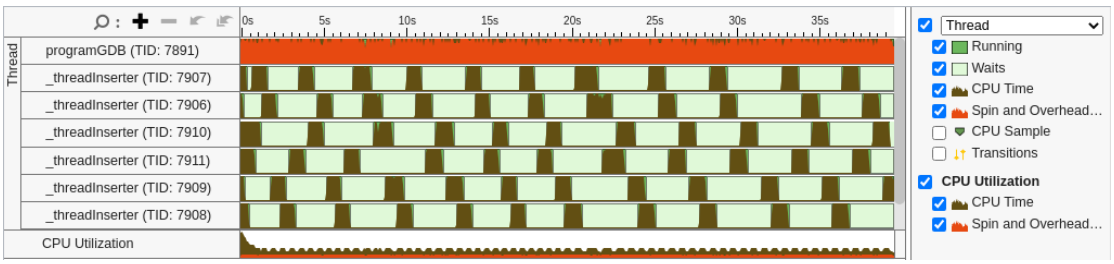


Figure 7.4.11: Performance of 7 inserter test with global structures

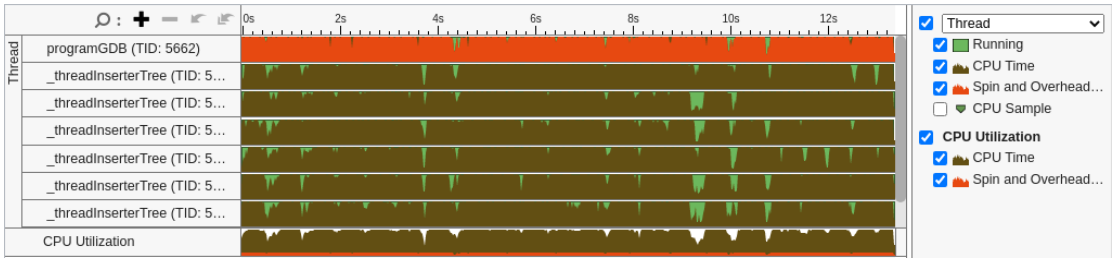
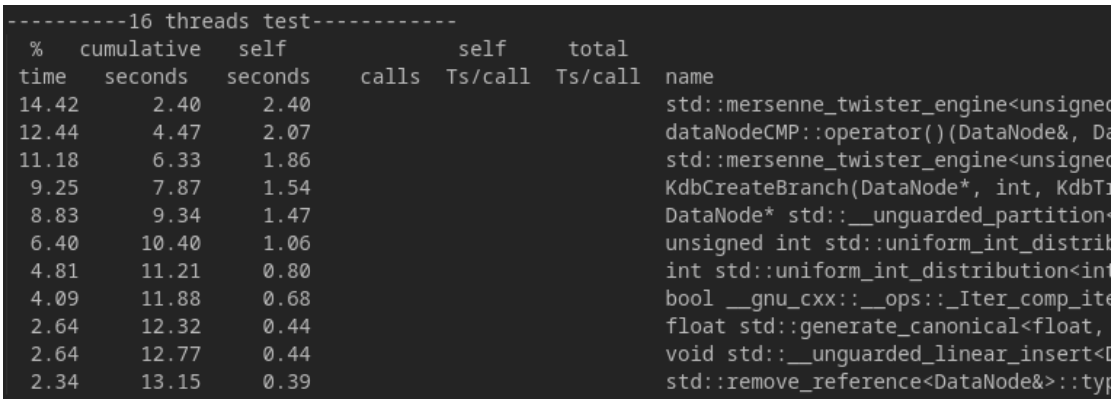


Figure 7.4.12: Performance of 7 inserter test without global structures

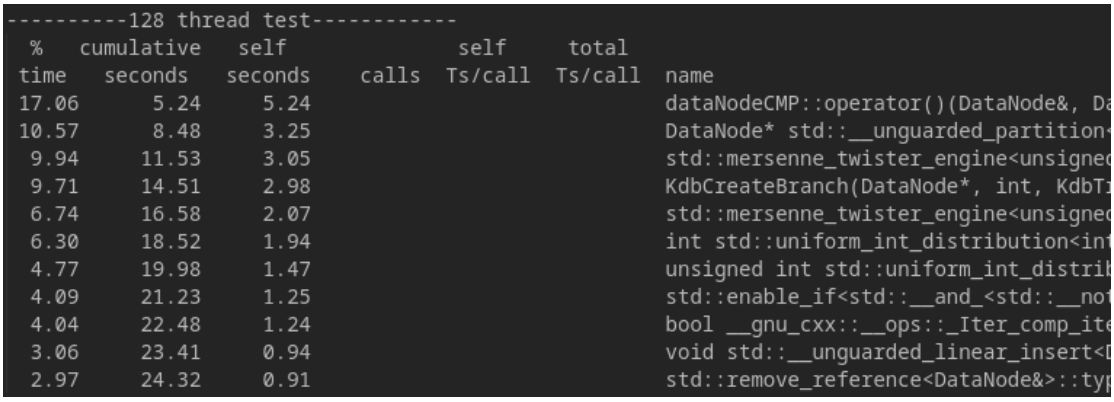
7.4.4 Server insert performance

Figure 7.2.7 clearly show that increasing the thread count is initially beneficial for thread performance under large workloads. As the number of inserter thread outgrows the size of the task, adding more threads reduce performance instead of increase it. To further investigate how more threads slows down the application, the code were benchmarked with gprof. A flat profile were created from running a executable compiled with the -pg and -pthreads flags which were used for testing and multithreading support. Gprof were not created for benchmarking multithreaded programs, therefore it can only generate a flat profile. The flat profile shows the total time spent in each function, but no information related to how time is spent by each thread. To get a overview of the largest time spenders, the top 10 most expensive function calls were included from a 16 and 128 inserter threads run. The complete profile can be found in the project's Github repository linked in Appendix A. The largest workload from figure 7.2.7 were analyzed, with a configuration with 128 trees each of a size of 262144 data nodes. The results of the two runs can be seen in figure 7.4.13 and figure 7.4.14. The 16 inserter thread run shows that much of the computational time is spent on functions related to generating pseudo random uniform values for the data nodes, sorting data nodes and creating the tree structure. Functions related to ‘mersenne_twister_engine‘ and ‘uniform_int_distribution‘ is typically related to the threadsafe random generator. Functions like ‘dataNodeCmp::operator()‘ and ‘std::__unguarded_partition‘ is related to sorting as its calling the comparing function used to sort the data nodes and the partitioning function used to partition values in the sorting functions. When increasing the inserter thread amount to 128, the time usage changes. Time spent in the sorting sections increases significantly. The percentage spent on generating data nodes comparatively decreases. This points to increasing the thread count beyond a certain point increases the sorting cost and therefore more time is spent sorting and less is spent on creating data nodes. Having to many threads at once may cause reduced performance as threads needs to share the cache with more threads. This may cause more cache misses, increasing the overall cost of fetching and comparing data as more values need to be read from memory instead of the cache.



```
----------16 threads test------------
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 14.42      2.40     2.40                             std::mersenne_twister_engine<unsigned
 12.44      4.47     2.07                             dataNodeCMP::operator()(DataNode&, Da
 11.18      6.33     1.86                             std::mersenne_twister_engine<unsigned
  9.25      7.87     1.54                             KdbCreateBranch(DataNode*, int, KdbTr
  8.83      9.34     1.47                             DataNode* std::__unguarded_partition<
  6.40     10.40     1.06                             unsigned int std::uniform_int_distrib
  4.81     11.21     0.80                             int std::uniform_int_distribution<int
  4.09     11.88     0.68                             bool __gnu_cxx::__ops::_Iter_comp_ite
  2.64     12.32     0.44                             float std::generate_canonical<float,
  2.64     12.77     0.44                             void std::__unguarded_linear_insert<D
  2.34     13.15     0.39                             std::remove_reference<DataNode&>::typ
```

Figure 7.4.13: Top 10 most expensive function calls utilizing 16 inserter threads in a server environment



```
----------128 thread test------------
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 17.06      5.24     5.24                             dataNodeCMP::operator()(DataNode&, Da
 10.57      8.48     3.25                             DataNode* std::__unguarded_partition<
  9.94     11.53     3.05                             std::mersenne_twister_engine<unsigned
  9.71     14.51     2.98                             KdbCreateBranch(DataNode*, int, KdbTr
  6.74     16.58     2.07                             std::mersenne_twister_engine<unsigned
  6.30     18.52     1.94                             int std::uniform_int_distribution<int
  4.77     19.98     1.47                             unsigned int std::uniform_int_distrib
  4.09     21.23     1.25                             std::enable_if<std::__and_<std::__not
  4.04     22.48     1.24                             bool __gnu_cxx::__ops::_Iter_comp_ite
  3.06     23.41     0.94                             void std::__unguarded_linear_insert<D
  2.97     24.32     0.91                             std::remove_reference<DataNode&>::typ
```

Figure 7.4.14: Top 10 most expensive function calls utilizing 128 inserter threads in a server environment

CHAPTER

EIGHT

CONCURRENT DISCUSSION

This chapter cover the discussion of the Concurrent Bkd-tree. The Concurrent Bkd-tree is a proof of concept data structure prioritizing inserter performance. Gained knowledge from concurrent results, insights and other findings will be used to further discuss and refine the Concurrent Bkd-tree.

8.1 Global memory and disk

Using the global memory and disk greatly increase the cost of insertion due to the added communication overhead. It should therefore not be used and each inserter thread should rather work on its own structure. With the eventually consistency approach of the concurrent Bkd-tree, it is unnecessary to store data nodes in a global structure. Data nodes are not available for readers before being placed in trees which then need to be inserted into the readable trees structure. As readers currently are not implemented to access the the global memory and disk structures, using them only add overhead and serve little purpose. Based on the findings in figure 4.2.2, it is also clear that iterating over many data nodes is expensive and should be avoided. With these points in mind, each inserter thread should rather work on its own tree instead of inserting values to global structures. This way communication between threads are reduced which lower overhead and speedup inserts.

8.2 Inserter performance

This section covers the performance of inserts in the Concurrent Bkd-tree. The performance of the Concurrent Bkd-tree is not directly comparable to a fully utilized solution, but serve as a proof of concept of how a data structure can be modified to better take advantage of modern hardware. As seen in the concurrent results, a lot of the CPU time is spend generating data nodes. In a real life scenario the Concurrent Bkd-tree would likely perform differently. In the current solution, a significant part of the computational time is spent generating pseudo random data nodes. In a complete solution, the data would likely be received over a network. This may be less computational expensive, but it could also take longer time due to added latency and communication overhead.

The performance of the Concurrent Bkd-tree would also be better if data nodes were generated before execution, but this would also require an approach were each inserter had its own data pool to avoid the need of syncronization. It would also require close to double the amount of memory for each test as each data node would be stored twice, once in the MockAPI and once in the Concurrent Bkd-tree structure. One of the strengths of the Concurrent Bkd-tree is the use of the eventual consistency approach. Eventual consistency allows writer threads to create and manage their own thread buffers without the additional overhead of synchronization. This helps the insert performance as each writer can fill and bulkload its buffer into Kdb-tree without being slowed down by readers. This gives the writer thread the freedom to only focus on managing its own data which is beneficial to insert performance. This approach is great for applications were readers only require an estimate or a snapshot of the data and not an exact snapshot of all current data nodes. This approach may not be suitable for all workloads, especially not workloads which needs precise and timely results.

The Concurrent Bkd-tree were tested with different workloads to see how performance scaled with different configurations. One of the more surprising findings were how well the solution scales when increasing the number of inserted Kdb-trees. Comparing figure 7.2.1 and figure 7.2.4 showed that doubling the amount of trees for the same configuration caused a speedup of more than three times as large when comparing the 7 inserter threads results of both figures. The startup cost related to creating the local thread random generator may be partially responsible, but it is still surprising to see the performance increase. The initial hypothesis were that more trees would slow down the performance due to the cost related to updating the readable tree structure. This cost may grow larger over time as the readable tree structure grows, but in all the inserter tests, increasing the workload by creating more trees were beneficial. Another important factor for the results may be the overhead of using a single thread. As the single inserter thread still updates the structures in a threadsafe manner, it is slower than a thread implemented to work without interference from other writers. From the Inserts per Second(IPS) test it is clear that the performance is not only related to runtime, but there is also a question of overhead related to creating larger structures. The larger structures may increase the speedup compared to a single node, but more time is spent creating large structures than multiple small ones. For this reason a trade off needs to be made between Kdb-tree size and IPS.

A reason for the low synchronization overhead seem to be the low cost of the ‘updateReadTrees‘ function. The function did not show up in the Vtune profiles due to its low overhead. This overhead may grow over time as the readable trees structure grows and becomes more expensive to update, which is why the structure should be replaced with a data structure with more fine grained synchronization.

Even though increasing the workload by creating more trees is beneficial, increasing the tree size affect the performance in a different manner. This can be seen in the tests of inserting 128 trees of size 16384, 65536 and 262144. In these tests the performance increases 221.7%, 308.8% and 221.1% comparatively from 1 to 7 inserter threads. These results may vary on different computers, but it may indicate that there is a sweet spot tree size which is most beneficial for a given workload. The smallest tree test may have a smaller performance increase as smaller trees are less computationally expensive, causing them to be more man-

ageable for a single thread. Trees with 65536 data nodes may be a good middle ground were the workload is to high for a single thread, but not to high to overwork the CPU. The largest tree size may be to large for the laptop to handle. The workload may be to large causing less of the values used in bulkloadings to fit in the cache which causes more expensive calls fetching of data from memory. This would explain why fewer threads are not affected in the same way, as they have fewer threads to share the cache with. This is also supported by the server performance profile which showed an increase in sorting related functions when going from 16 to 128 inserter threads.

When running the same tests on the server environment compared to the laptop, the results clearly show that the workload needs to scale in accordance to the capacity of the hardware. Running tests with 16384 and 65536 data nodes on the server gained significantly less performance compared to the larger 262144 data node test. This is a result of the more powerful server CPUs requires a larger workload to fully utilize their capability. The result from figure 7.2.7 also showed that even though a server may have multiple threads, they may not be suitable for data intensive applications as multiple threads may slow down the execution time due to poor cache utilization.

In an optimal scenario, each thread added to an application would increase the performance in accordance to the number of threads. This would mean that 2 threads ran twice as fast as a serial solution, 3 threads would run three times as fast and so on. In the Concurrent Bkd-tree, the highest improvement seen in any test is 308.8% which is equivalent to around 3.1 times as fast as the serial solution. This were achieved with 7 inserter threads with a workload of 128 trees each with 65536 data nodes. This is equivalent of a thread efficiency average of 34.8% per added thread. Out of this the highest increase is seen when going from 1 to 2 inserter threads. This increase is equivalent to a thread efficiency of 41.6%. Each thread added after that point gradually decreases in usefulness with the difference between 6 and 7 inserter threads only being 4.02% speedup. From these results, it is clear that for a set workload, the usefulness of each individual thread will eventually be insignificant. For a set workload the application will scale in accordance to Amdahl's law, meaning the performance will never increase past the initial startup overhead. This is why it's important to scale the workload together with the number of threads to fully utilize the computational power of a given system. Using this approach the application will scale in accordance to Gustafson's law and each added thread will be able to make a more significant impact on the performance.

8.3 Reader performance and bulkloading

Reader performance is currently decent. It is clear that readers are less affected by the number of trees than first assumed. Therefore, frequent bulkloadings are not as beneficial. Instead larger and fewer bulkloadings should be performed. From the reader results, it is also clear that reader performance scales at a similar rate with different tree sizes. There is a small overhead to reading data from multiple smaller trees, this overhead may be manageable, but it depends on what a typical workload is. If readers are prioritized, larger trees should be created

as this will over time improve window query performance. This would require inserter threads to spend more time sorting data nodes to create larger structures which would negatively affect the IPS. If a typical workload for the system were to perform 90% window queries and 10% write or insert queries, reader performance would be more important and bulkloading may be a higher priority. In a system were 90% of queries are inserts and only 10% are window queries, it may be more important to maintain high insertion performance and therefore creating smaller trees may be sufficient. This is something which could be preconfigured in a complete solution, or may be a task for the Scheduler thread to dynamically adapt bulkloadings and Kdb-tree sizes based on the current workload.

8.4 Scheduler's role

In the current solution the Scheduler's role is primarily to start threads and reallocate memory to the application. As these responsibilities are relatively small, the Scheduler thread is under utilized in the current implementation. Given a complete program there may be more tasks such as load balancing and adapting system resources to the current workload. In that case, a dedicated Scheduler thread may be more necessary than in the current implementation.
In the current implementation, readers slows down the Scheduler thread from reallocating data. The Scheduler cannot delete old structures before all readers have exited. As writer threads currently create a new readable tree structure for every new tree, readable trees structures become outdated quick. Each structure is pushed to the Scheduler which is made responsible for clearing the old structures. The Scheduler then has a lot of structures to reclaim, while at the same time the structures are held up by readers which access them. This could be improved by creating a fine grained synchronization readable tree structure. This would mean that readers would only lock a small part of the readable tree structure instead of an entire copy. This could let the Scheduler clear data faster as there would be smaller structures needed to be reallocated and fewer readers per section.

8.5 Readable trees

In the current solution the readable trees are updated each time a new tree is created. The structure is both updated by inserter threads and potential bulkloader threads. The readable trees structure is currently being updated with a RCU approach. For the thread performing the update this entails that the global structure first needs to be copied, the new Kdb-tree can be inserted and if the new tree is built with values from old trees, the old trees needs to be removed from the structure. The updated readable tree structure copy can then be pushed globally were it can be accessed by all threads. This may become expensive over time, especially when synchronization with RCU only allow for one thread to modify the structure at once. For this reason, it might be reasonable to use a different, more fine grained structure for the RCU update.

8.6 Syncronization

The current readable tree structure uses coarse grained synchronization with a single lock protecting the structure. To make a RCU update in the current solution, a inserter or bulkloader thread must lock the entire structure to keep the structure thread safe. The structure itself is not locked as readers can still access it, but only a single updater thread can access it at once. This is to avoid race conditions which could cause trees to be lost due to multiple simultaneous updaters. A more fine grained synchronization structure could be created by having more locks. This could help performance as each inserter would only need to lock a small part of the structure. In turn this would let multiple inserters work at different sections of the solution at once, further reducing the communication overhead. Having fine grained locking combined with an RCU approach, would also make it possible to create a RCU-HTM readable tree. Fine grained locking could let one inserter lock a single branch of a readable tree structure and let other writers work on different branches.
Comparing how inserter threads scales when using global structures and not using them, it is clear that communication should be kept to a minimum. Performance gained from multithreading comes from having multiple compute units which each can perform individual tasks. The more communication between the compute units, the more time is spent on synchronization and waiting on other computational units. Adding structures with more locks would increase communication as each lock adds some synchronization overhead. However, as the alternative is to have a singular lock which would lock all but one updater, the performance gained from having multiple updaters, being able to update the structure at once, would likely overcome the overhead.

8.7 Tombstone list

The tombstone list have not been fully tested and remain as a proof of concept deletion approach in the Concurrent Bkd-tree implementation. The current solution is not sustainable for a large scale workload as it currently is a shared global resource with a coarse grained synchronization mechanism. With the current synchronization approach, the tombstone strategy would add a significant overhead if it were used as a global shared resource. In this scenario the bulkloader thread could iterate over all its data nodes and assert none were deleted. This would be a possible approach, but to ensure no deleted values were accessed in window queries, each read thread would have to check each data node against the tombstone bloomfilter. This would cause a significant overhead as the structure would both be accessed by readers and updaters which both would block each other and cause delay. Another approach would be to only have the bulkloader thread access the tombstone structure and having the lone responsibility of managing and deleting nodes. This would make the deletions eventually consistent as deleted nodes would be deleted by the bulkloader after the structure gets merged. This still leaves some problems due to the current primitive solution of the tombstone list. The list currently has no meta data about a given node's age. This means that a deletion request could be sent before the actual node were inserted and cause the new node to be deleted by an old delete request. Currently there is not

implemented any update of the bloomfilter. This would be required to periodically update the filter after deleted nodes have been removed to decrease the amount of false positives. The tombstone list has potential to be a good addition to the Concurrent Bkd-tree structure, but may be to primitive in its current state and require further work to be a suitable deletion strategy.

8.8 Futher work

The Concurrent Bkd-tree is a proof of concept multithreaded data structure, which have multiple sections which needs more improvements and future work. As mentioned further work should go into improving the readable tree structure. Using a more light weight solution would both allow for more fine grained synchronization mechanisms and make inserter threads spend less time on the RCU update and allow the Scheduler thread to spend less time on the reallocation of the old structures. In its current form the tombstone strategy needs improvement. Its a proof of concept which may be usable with an eventually deleted approach, but should be improved. The focus of this paper has been on creating a proof of concept structure and improving it with concurrency. The current solution show potential on the concurrency front, but the code should still be improved by other means. The program could be made faster with a grid bulkloading approach as suggested in the original Bkd-tree paper. The solution should also be improved and adapted to a specific use case, by tailoring it to a set number of dimensions and create a tailored solution for a more predefined workload.

CHAPTER
NINE

CONCLUSION

This thesis has covered the implementation and benchmark of a Bkd-tree structure. The findings have further been used as starting point for implementing a Concurrent Bkd-tree. The final implementation is a concurrent solution with the highest average thread utilization of approximately 34.8% when using 7 inserter threads, compared to a single thread. The same workload achieves 4228 IPS. From the results it is clear that synchronization overhead should be kept to a minimum to increase performance when utilizing multiple threads. To fully utilize the CPUs of a given system, solutions should be tuned to best utilize the hardware. Even if lots of computational power is available, it may not be beneficial to utilize every thread, especially in data intensive workloads. The impact of multiple threads sharing a cache may negatively impact performance as each thread fetches more values from memory. When it comes to which workload should be given to a system, it is important to adapt the program to the requirements set by the particular workload. If the workload primarily requires good read performance, then larger trees should be created to lower window query cost. If the application is insertion heavy, smaller trees best suited for the given system, should be utilized to keep a high IPS. The focus on this thesis has been on improving a data structure with concurrency and multithreading. The Concurrent Bkd-tree has achieved this, but it should be noted that the solution would still benefit from other means. The structure is still a proof of concept and requires further work to be utilized and reach its fullest potential. The Concurrent Bkd-tree shows that performance can be gained by utilizing multiple threads in a data intensive application.

REFERENCES

[1] Octavian Procopiuc et al. "Bkd-Tree: A Dynamic Scalable kd-Tree". In: *Advances in Spatial and Temporal Databases*. Ed. by Thanasis Hadzilacos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 46–65. ISBN: 978-3-540-45072-6.

[2] Dimitrios Siakavaras et al. "RCU-HTM: A generic synchronization technique for highly efficient concurrent search trees". In: *Concurrency and Computation: Practice and Experience* 33.10 (2021), e6174. DOI: https://doi.org/10.1002/cpe.6174. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6174. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6174.

[3] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. ISSN: 1432-0525. DOI: 10.1007/s002360050048. URL: https://doi.org/10.1007/s002360050048.

[4] Lukas Vogel et al. "Plush: A Write-Optimized Persistent Log-Structured Hash-Table". In: *Proc. VLDB Endow.* 15.11 (Sept. 2022), pp. 2895–2907. ISSN: 2150-8097. DOI: 10.14778/3551793.3551839. URL: https://doi.org/10.14778/3551793.3551839.

[5] Kuo-Chunf Tai Richard H. Carver. *MODERN MULTITHREADING Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. JOHN WILEY SONS, INC., 2005.

[6] Randal E. Bryant and O'Hallaron David. *Computer Systems: A Programmer's Perspective*. 3rd ed. Boston, MA: Pearson Education, 2016, p. 58. ISBN: 978-1-488-67207-1.

[7] John L Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.509.6892.

[8] Anthony Williams. *C++ Concurrency in Action*. Shelter Island, NY: Manning Publications, 2012. ISBN: 978-1-935182-57-4.

[9] Tom Dickens. "Chapter 12 - Software Synchronization". In: *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013, pp. 385–445. ISBN: 978-0-12-416018-7. DOI: https://doi.org/10.1016/B978-0-12-416018-7.00012-2. URL: https://www.sciencedirect.com/science/article/pii/B9780124160187000122.

[10] Bjarne Stroustrup. *Bjarne Stroustrup - The Essence of C++*. 2014. URL: https://www.youtube.com/watch?v=86xWVb4XIyE (visited on 11/21/2022).

[11] *Chrono*. URL: https://cplusplus.com/reference/chrono/.

[12] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997, p. 381. ISBN: 978-0201633924.

[13] *Intel® VTune™ Profiler*. Software. Intel Corporation, 2023. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

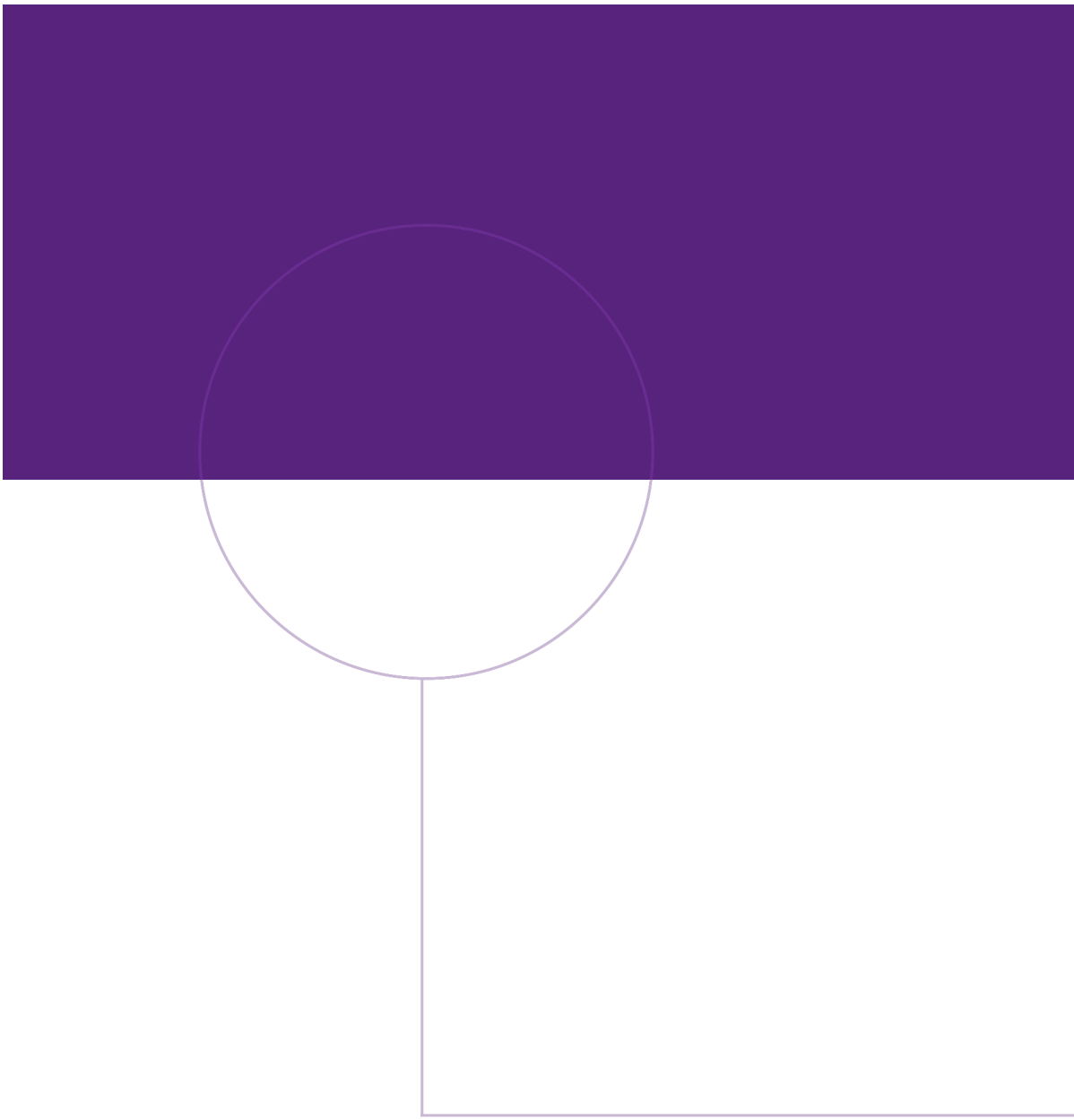
[14] GNU Project. *GNU gprof*. [Computer software]. Version 2.37-37.fc36. 2023. URL: https://sourceware.org/binutils/docs/gprof/.

APPENDIX A - GITHUB REPOSITORY

All code, tests and scripts to generate figures in this paper can be found in the Github repository linked below.

Github repository links

- `https://github.com/sandertoresen/Concurrent-Bkd`
- `https://github.com/sandertoresen/BKD-SERIAL-PREPARATORY-PROJECT`



NTNU

Norwegian University of
Science and Technology