Otto-von-Guericke-University Magdeburg

Faculty of Computer Science

Department of Databases and Software Engineering

Parallelizing the Elf - A Task Parallel Approach

Bachelor Thesis

Author:

Paul Blockhaus

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dr.-Ing. David Broneske

Otto-von-Guericke-University Magdeburg
Department of Databases and Software Engineering

Dr.-Ing. Martin Schäler

Karlsruhe Institute of Technology
Department of Databases and Information Systems

Magdeburg, November 29, 2019

Acknowledgements

First and foremost I wish to thank my advisor, Dr.-Ing. David Broneske for sparking my interest in database systems, introducing me to the Elf and supporting me throughout the work on this thesis.

I would also like to thank Prof. Dr. rer. nat. habil. Gunter Saake for the opportunity to write this thesis in the DBSE research group. Also a big thank you to Dr.-Ing. Martin Schäler for his valuable feedback.

Finally I want to thank my family and my cat for their support and encouragement throughout my study.

Abstract

Analytical database queries become more and more compute intensive while at the same time the computing power of the CPUs stagnates. This leads to new approaches to accelerate complex selection predicates, which are common in analytical queries. One state of the art approach is the Elf, a highly optimized tree structure which utilizes prefix sharing to optimize multi-predicate selection queries to aim for bare metal speed. However, this approach leaves many capabilities that modern hardware offers aside. One of the most popular capabilities that results in huge speed-ups is the utilization of multiple cores, offered by modern CPUs. At this point our work comes into play, to bring multi threading to the Elf.

In the scope of this thesis we introduce parallel queries to the Elf to accelerate multi-predicate selections even further. Furthermore we will introduce concurrent insertions to the Elf to bring it on a level that satisfies the requirements for state-of-the-art database systems. To evaluate the results of our work, we compare our parallel query approach to the original Elf. Therefore we utilize the state-of-the-art TPC-H benchmark to obtain meaningful results for real-world workloads. Our findings indicate a speed-up of factor 4 to 11 compared to the serial Elf version. For our concurrent insertion algorithms, we evaluate their performance in a synthetic benchmark under different write workloads. Our algorithms are able to achieve a speed-up of up to factor 2.6 to 5.8 depending on the workload.

Contents

List of Figures

List of Code Listings

List of Algorithms

1 Introduction

In the last decades, the demand for database systems grew explosively. At the same time, alongside with online transaction processing (OLTP), which facilitates large amounts of short concurrent read-write workloads, long running complex mostly read workloads for analysis (OLAP) became more important [Plattner, 2009]. Also on the hardware site, used technologies evolved immensely. Disk-based systems, which were bottlenecked by the access to rotating disks developed to main-memory systems and the bottleneck of latency and bandwidth to second level storage was replaced by the bandwidth and latency bottleneck of main-memory. Research is now focused on main-memory structures to minimize this bottleneck [Rao & Ross, 2000], [Boncz, Kersten, & Manegold, 2008].

Nowadays there exist large data warehouses, holding thousands of terabytes of data. They need to scan this data within the smallest amount of time possible to answer complex analytical queries. Those analytics often contain queries with multiple selection predicates. For these kind of queries, multidimensional data structures [Gaede & Günther, 1998] are used to accelerate the queries. State-of-the-art multi-dimensional main-memory index structures [Sprenger, Schäfer, & Leser, 2019], [Zäschke, Zimmerli, & Norrie, 2014] even go a step further and optimize the cache efficiency and use extended capabilities of the CPU to parallelize the queries any further.

However, most state-of-the-art multi-attribute data structures do not utilize the fact that the combination of multiple selection predicates drastically reduces the selectivity of the data. This is where the Elf comes into play. The Elf is "a tree-based index structure for multi-column selection predicate queries featuring prefix-redundancy elimination and a memory layout tailored to exploit modern in-memory technology" [Broneske, Köppen, G., & Schäler, 2017]. By exploiting multiple selection predicates at once, the Elf is able to outperform other main-memory index structures [Sprenger et al., 2019] and even parallel scans [Willhalm et al., 2009] provided the selectivity of the query is low enough. However, this comparison is not particularly fair, since unlike parallel scans, the Elf is only a serial data structure and as thus does not fully utilize all capabilities of modern CPU architectures to reach bare metal speed. Consequently, we want to examine the parallelization capabilities of the Elf. Therefore we focus on using multiple CPU cores concurrently to accelerate Elf queries and insertions.

Goals

The goal of this thesis is to evaluate the impact of multithreaded algorithms on the performance of the Elf and thus prove our hypothesis that task parallelism is able to speed-up the Elf approach even more. Therefore we contribute the following key contributions in the frame of this thesis:

1. We present the design of three different query algorithm using two different parallelization strategies.
2. We propose concurrent insert algorithms adapting three state-of-the-art approaches.
3. We compare the performance of our algorithms to the serial Elf using the TPC-H benchmark queries to get robust results which represent current real world workloads.
4. We compare our concurrent insertion implementation against the serial approach in a microbenchmark to show the speed-up we are able to achieve.
5. We give an outlook to some promising research tasks to enhance the performance of the Elf even further.

Thesis Outline

The structure of this thesis is as follows. We start by introducing the necessary background needed to understand this thesis in Chapter 2, starting with parallel computing architectures with a focus on multithreaded structures and various concurrency schemes. This is followed by an introduction into the concepts of the Elf. At the end we introduce various non-blocking data structures we utilize in the next chapters. In Chapter 3, we will discuss related approaches for parallel tree traversal and different insertion algorithms. Our own contribution starts in Chapter 4, where we define our parallel query algorithms as well as the related merge algorithms. his is followed by Chapter 5, in which we introduce our concurrent insertion algorithms. Subsequentially, in Chapter 6 we evaluate our contribution and discuss the evaluation results. Finally, in Chapter 7 we summarize the thesis and give an outlook to possible future work.

2 Background

The goal of this thesis is to parallelize queries and modifications for the multi-dimensional main-memory index structure *Elf*. Therefore in this chapter we introduce parallelization techniques for modern x86 multi-core CPUs with particular regard to task-based parallelism. Furthermore, we introduce the structure of the multi-dimensional main-memory index structure called *Elf*, as well as some further data structures.

2.1 Parallel Computer Architectures

In current parallel computer architecture, there are various different parallelization capabilities, which can be classified by the Flynn taxonomy [Flynn, 2011]. The Flynn taxonomy comprises of four quadrants:

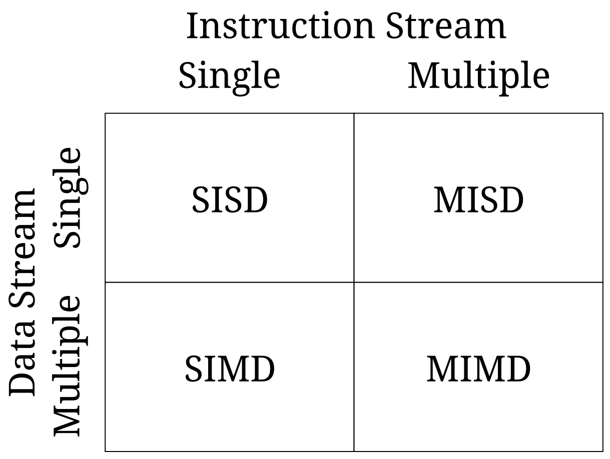


Figure 2.1: Flynn's Taxonomy

Single Instruction Single Data (SISD) describes the traditional uni-processor architecture. Instructions are executed sequentially on single data.

Single Instruction Multiple Data (SIMD) is the class of vector processing machines, which execute the same instruction on multiple data in parallel.

Multiple Instructions Single Data (MISD) is an exotic redundant architecture, where multiple processors execute instructions on the same data.

Multiple Instructions Multiple Data (MIMD) is often referred to as Multiprocessing architecture and can execute various instructions on various data in parallel.

In the context of this thesis we focus on MIMD parallelism especially Multithreading since it is widely available and suites best for our demands.

2.2 Multithreading

Multithreading is a concurrency model for the MIMD architecture. It is available on multiprocessor as well as uniprocessor machines even though it is not a MIMD architecture in this case. To introduce concurrency, a process is split into smaller lightweight subprocesses, so called threads.

Each thread runs independently from the other, which allows the execution of multiple threads at the same time by multiple CPU cores, which is called Simultaneous Multithreading. For communication between threads, either a memory region is needed, where all threads can write to and read from, or messaging channels for regulated communication must exist. The former is called *Shared Memory* the other is the *Message Passing Interface*. *Shared Memory* has the advantage of being very lightweight compared to a *Message Passing Interface* Protocol with relatively large communication overhead. For this reason we decided to use shared memory, even though this approach must be used carefully, otherwise data races can occur.

Since every thread can access shared memory at any time, multiple threads can parallelly change, or access the same memory location. This can lead to a so called data race; data races can occur at parallel changes on shared data, as for instance on Listing 2.1, which is an example for two parallel banking transactions.

```
 1  thread T1;
 2  thread T2;
 3  var Account = 100;
 4  start T1;
 5  //execution block of T1
 6  var tmp_t1 = Account; //100
 7  preempt T1;
 8  start T2;
 9  //execution block of T2
10  var tmp_t2 = Account; //100
11  Account = tmp_t2 + 50; //add 50
12  T2 preempt;
13  //execution block of T1
14  Account = tmp_t1 + 70; //add 70
15  /* Account = 170, one transaction got lost! This data race occurred,  because both ↩
        variables were read and written back independently.*/
```

Listing 2.1: Example Data Race During Concurrent Modification of a Bank Account

In this example, each of the two threads `T1` and `T2` performs a modification on the same `Account`. Therefore, each thread reads the current value of `Account`-variable into its own memory (q. v. Lines 6 and 10) and gets preempted before it can write the result back. In the next step the actual data race occurs, which is in this case a lost update. `T2` writes its result as first back into the `Account` variable in Line 11. Then `T2` gets preempted and `T1` writes its data back in Line 14, ignoring that the value of `Account` has just changed, which leads to the data race.

2.2.1 Blocking and Non-Blocking Algorithm

To avoid data races, locks are usually used to protect the critical section. The critical section should not be executed in parallel for all threads except the one that is currently executing in this section. In order to avoid concurrent access on critical sections, several different locking techniques exist:

Semaphore are resource counters, which allow only a specified amount of concurrent executions of the critical section[Dijkstra, 1962]. When the limit is reached, all threads that want to execute the critical section code must wait until a contingent for them is available.

Mutexes are essentially binary semaphore, where only one thread is allowed to execute the critical section at once.

RW-Mutexes are mutexes, that can be locked in two ways [Fraser, 2004]. One way is to lock the mutex in read-only mode, this lock does not acquire exclusive access to the protected resources and can be acquired multiple times concurrently. The write lock only allows read operations to be safe, write operations can be executed by a read lock. A read lock blocks any access to the protected resources and thus prohibits concurrent writes or reads while modification. With this properties a RW-mutex is able to allow multiple readers safe concurrent access to the protected resources but also make it possible to protect the resource from concurrent writes.

Spinlocks are condition variables on which all threads loop until the thread in the critical section finishes. This locking is done in user space and can be faster sometimes but has the drawback that the waiting threads are actively using resources while busy waiting. This can lead to delaying of the thread currently executing the critical section, because the scheduler can not preempt the spinning threads early.

In most cases locking is achieved with mutexes; there are different techniques how mutexes can be implemented. One is hardware lock support or atomic operations such as test-and-set or Compare-And-Swap (CAS). As a substitute when no hardware support is available, algorithms such as Dekker's algorithm come into play. However, these algorithms require a strict ordering of instructions and memory access, which is not granted by modern CPUs which perform out-of-order execution.

To overcome the disadvantages of locking, there exist non-blocking algorithms with different guarantees regarding the progress of an algorithm. M. Herlihy et al. classified three guarantees, that non-blocking algorithms can give [M. Herlihy, Luchangco, & Moir, 2003]:

Obstruction-Free: An algorithm is obstruction-free if at any point, a single thread is executed in isolation (i.e., with all obstructing threads suspended) and will complete its operation in a bounded number of steps.

Lock-Free: An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress (for some reasonable definition of progress).

Wait-Free: An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. This property is critical for real-time systems and is always nice to have as long as the performance cost is not too high. It guarantees lock-freedom and starvation freedom.

Obstruction freedom is the smallest guarantee that a non-blocking algorithm can give, it only guarantees termination. The next better guarantees come from lock-freedom which gives the additional guarantee of system throughput. The third and strongest guarantee is the wait-freedom, which guarantees lock-freedom and starvation freedom, which means the system is guaranteed to make progress. To implement non-blocking algorithms mostly atomic instructions are used.

2.2.2 Atomic Instructions

Atomic instructions are lock-free instructions that are free of data races. Since it is complicated to achieve atomicity, the set of atomic instructions is quite small. Most of the instructions are simple arithmetic, fetch and store, as well as binary instructions. The most important instruction in this work in particular is the atomic CAS instruction. It allows to compare a value with a desired value and swaps the value with a third when both are identical; all this is done in one step without locks needed.

```
pointer L;
do {
  var A = *L;
  var B = new var V
} while (!cas(L, A ,B)); //Meanwhile L could have changed to B and back to A again
```

Listing 2.2: Speculative CAS Execution Example

Unfortunately most complex algorithms such as database transactions or even a simple doubly linked list require reads and writes at multiple locations. Those algorithms can not be simply adapted to the use of atomic instructions, since most architectures have no support for multi-word CAS instructions to modify multiple memory locations atomically. Usually the workaround is to read a memory location multiple times and execute a CAS speculatively like we show in Listing 2.2 adapted from [Dechev, Pirkelbauer, & Stroustrup, 2010]. The disadvantage of this workaround is, that it is prone to the ABA Problem.

The ABA Problem

The ABA in the ABA Problem is no abbreviation, ABA stands for the variable changes from A to B and back to A. The problem occurs when multiple threads access the same data interleaved; Dechev et al. give a definition as follows in Listing 2.3:

```
1  val A;
2  val B;
3  pointer *L = A;
4  thread T1;
5  thread T2;
6  start T1; \\execution block of T1
7  read *L; \\read A from L
8  T1 preempt;
9  \\execution block of T2
10 read L; \\read A from L
11 *L =  B;
12 *L =A;
13 T2 preempt;
14 \\execution block of T1
15 read *L; //*L is A again and T1 does not  see the changes made by T2
```

Listing 2.3: The ABA Problem

The former listing defined the sequence of operations which result in the ABA Problem. Two threads `T1` and `T2` both read from the shared memory location `L`. The first reading thread is `T1` in Line 7. Afterwards, the thread is preempted and `T2` modifies the value of L from `A` to `B` in Line 11. Since the processing quantum of `T2` is not exceeded yet it also modifies the value of `L` back to `A` in Line 12. Now, `T2` is preempted and `T1` reads `L` again in Line 15 to ensure that no other thread accessed `L` in the meantime. Since `L` was changed from `A` to `B` and back and `T1` is not able to recognise this changes made by `T2` and assumes that no other thread accessed `L` in the meantime.

While this behaviour might not be a problem for integral data types, it can have immense effects on the semantic of an algorithm if the value was a reference or a pointer. Even though the program usually continues without problems, its state can be inconsistent due to modifications that were missed by `T1`. To overcome this issue, there are several solutions for this problem partially listed in "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs" by Dechev et al.:

Deferred Reclamation through garbage collection e.g. with reference counting [Detlefs, Martin, Moir, & Steele, 2001], Hazard Pointers [Michael, 2004] or Read Copy Update (RCU) [Mckenney et al., 2001] is one solution for the ABA Problem. This technique tracks the memory uses and defers the reuse of memory locations until the ABA Problem can not occur.

Tagged Pointers first proposed in "IBM System/370 Extended Architecture — Principles of Operation" are another circumvention technique, which couples a pointer with a tag, e.g., by using a part of the address as a mutation counter. This effectively prevents the ABA Problem by always mutating the A to an A', provided the mutation counter is chosen at least large enough to prevent an overflow. If this is not the case, this technique is ineffective because tags would also underlie the ABA Problem. Even though a 64 bit tag would overflow after 10 years permanently permuting [1].

Extra level of indirection Indirection implicates that all values are stored in shared memory indirectly through pointers. Each write of a given value to a shared location needs to allocate a new reference on the heap and finally safely delete the pointer value removed from the shared location.

[1]http://ithare.com/cas-reactor-for-non-blocking-multithreaded-primitives/

Descriptors proposed by Dechev et al. are objects which describe the current operation and the current state of the operation, which has to be executed. All threads are helping to finish these operations and when finished, the next descriptor is inserted in a way that no ABA Problem occurs.[Dechev et al., 2010]

In the scope of this thesis, we presume, that no memory is reused in our algorithms. Furthermore the ABA Problem does not harm the correctness of our evaluation results, since we do only insert and query integral data and pointers whose memory reuse is prohibited.

Fortunately there exists another more optimistic approach, which does not need complex ABA Problem prevention techniques. And is thus is potentially able to achieve comparable performance without CAS instructions, instead the approach relies on transactional memory.

2.2.3 Transactional Memory

Transactional memory works in transactions similar to transactions in database systems [Saake, Sattler, & Heuer, 2011],[Maurice Herlihy & Moss, 1993] as they either complete successfully or not at all. A transaction is a sequence of memory modifications, which are executed on a best effort basis. If another concurrent modification interferes with the transaction, all modifications made by the transaction are rolled back. Furthermore transactions are also serializable, which means transactions are executed serially and can never run interleaved. This model can be extended to support overlapping as well as nested transactions but is rarely used because of the large overhead they introduce. Thus, they are insignificant in practice and rarely implemented. Transactional memory can be implemented in hardware - so called *Hardware Transactional Memory* (HTM) - as well as in software - *Software Transactional Memory* (STM).

Hardware Transactional Memory

In HTM, the processor has a special instruction set extension for memory transactions and thus is able to perform transactions with hardware support. This can speed up memory transactions drastically against software transactions but has currently some limitations. One possible implementation for transactional memory on hardware is to add a transactional first level cache of the processor and thus split the cache into a regular non-transactional, and a full-associative transactional cache with additional logic to support the commit and abort of transactions [Maurice Herlihy & Moss, 1993]. This implementation of transactional memory needs only minimal changes to the existing cache coherency protocols and is thus one of the easiest to implement. Furthermore it has no drawbacks.

At the beginning of a transaction, the processor starts using the transactional cache and blocks the non-transactional cache to save all the writes to memory locations but no changes are written to memory. When the transaction aborts, the cache is invalidated and the transaction is discarded. Otherwise, when the transaction commits, the changes in the transactional cache are written to the memory in parallel in a single cache cycle since the cache is fully associative. As a consequence, the size of the transactions is limited by the size of the cache, a single scheduling quantum and other architectural limits.

Currently the only implemented instruction set extension for transactional memory on the x86 architecture comes from Intel and is called TSX-NI. TSX-NI delivers two functionalities, one is *Hardware Lock Elision* (HLE), the other is the *Restricted Transactional Memory* extension (RTM), which implements the four following instructions with the following functionalities [Intel, 2019a]:

`XABORT` forces an RTM abort. Following an RTM abort, the logical processor resumes the execution of the fallback path of the outermost `XABORT` instruction.

`XBEGIN` The `XBEGIN` instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the `XBEGIN` instruction causes the logical processor to transition into transactional execution. The `XBEGIN` instruction that transitions the logical processor into transactional execution is referred to as the outermost `XBEGIN` instruction. The instruction also specifies a relative offset to the fallback path following a transactional abort. On an RTM abort, the logical processor discards all memory updates performed during the RTM execution and restores the state corresponding to the outermost `XBEGIN` instruction. The fallback path of the outermost `XBEGIN` instruction following an abort is used.

`XEND` The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this `XEND` instruction, the number of `XBEGIN` instructions is the same as number of `XEND` instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all memory updates performed during the RTM execution. The logical processor will resume execution at the fallback path from the outermost `XBEGIN` instruction.

`XTEST` The `XTEST` instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region it sets the corresponding flag.

Software Transactional Memory

Software transactional memory follows the same concept as HTM but does not require any form of architectural support for memory transactions. There exist many different implementations [Spear, Dalessandro, Marathe, & Scott, 2009] [Maurice Herlihy, 1993] [Lev & Maessen, 2008] [Shavit & Touitou, 1997] [Felber, Fetzer, & Riegel, 2008].

The first and most simple one is known as Herlihy's method. A transaction with this method consists of a k-word CAS, which commits the transaction previously cached in regular memory. The nature of CAS guarantees system progress, so that the transaction eventually succeeds.

For this theses we use a library called libitm, which supports HTM as well as STM with various different algorithms as fallback. The STM implementation that comes closest to the one used per default by the libitm implementation is the one by Felber et al. The solution used by libitm is based on a lazy multi-locking scheme. This means, that the locations which are to be modified by a transaction are locked only in the last step, when the transaction is not aborted and commits its changes. Before this locking takes place, all modifications are added to a transaction log which is only flushed when no conflicting transaction accesses data that should be modified in the meantime.

2.2.4 Linearisability

To reason about correctness about non-blocking algorithm, one of the most used criteria is the linearisability formulated by M. P. Herlihy and Wing. Linearisability states [M. P. Herlihy & Wing, 1990]:

1. Processes act as if they were interleaved at the granularity of complete operations.
2. The operations seen as apparently sequential interleaving respects the real-time precedence ordering of operations.

This allows intuitive reasoning about the behaviour of the algorithm, making it suitable for all purpose use without paying attention to complicated special cases.

2.2.5 Asynchronous Programming

Until now we described rather low level architectural details. For our work we also used some techniques form another programming model, the asynchronous programming model. This model is more abstract than Multithreading and bases on the idea that portions of code can be run asynchronously in so called tasks. Later, after the execution of the task finished, results can be retrieved, but tasks that depend on other tasks can also wait on the calculation results.

To reach this kind of cooperation, special objects exist which hold the calculated data and can be used to wait for results [Friedman & Wise, 1978]. These objects are called promises and futures.

Promise is an object, which represents the assurance of the called object, that can pass the result to the caller via the associated future.

Future is an object, which represents a future value, which will become available when the associated promise is redeemed and finishes. Until the result value is not ready, it is possible to block and wait for the result.

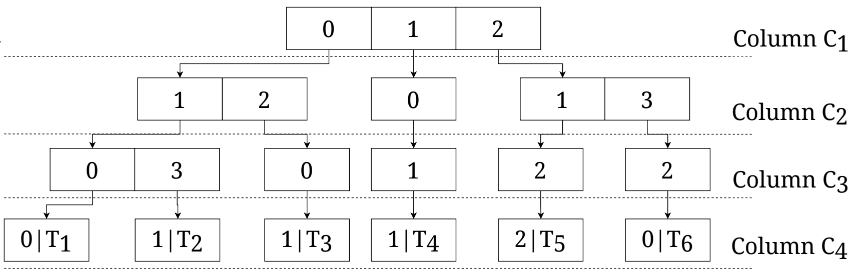
With these two synchronisation primitives, it is possible to hide locking effectively behind design patterns as well as reaching data race freedom, since the objects itself act as a message passing protocol.

2.3 The Elf

The *Elf* is a prefix-sharing multi-dimensional main-memory data structure invented by Broneske et al.

C_1	C_2	C_3	C_4	TID
0	1	0	0	T_1
0	1	3	1	T_2
0	2	0	1	T_3
1	0	1	1	T_4
2	1	2	2	T_5
2	3	2	0	T_6

(a) Example Table



(b) Conceptual Elf

Figure 2.2: Example Table Data with Corresponding Conceptual Elf

In Figure 2.2, we show an example of the *Elf*s structure. Each level of the *Elf* corresponds to one Column and parent nodes act as shared prefix from the previous dimensions. For example the left most path represents the tuple T_1 with the column data $\langle 0, 1, 0, 1 \rangle$. In the last dimension besides the last value of the column stands the *TID*. The *Elf* comes in two variants, each optimized for a either read or write workloads, but the overall conceptual structure remains.

2.3.1 Write-Optimized Elf

The write-optimized *Elf* consists of as much level as number of columns indexed, whereas each *DimensionList* is a sorted array with key-value pairs. The key is a 32-bit value of the column, the value is a pointer to the next dimension. The last dimension only consists of the last value and *TID*s; this structure leads to prefix sharing and drastically reduces the storage consumption. Furthermore the dimension lists can be searched in logarithmic time since the lists are sorted and have random access. On the other hand, prefix sharing leads unlucky distribution and sparse subtrees that degenerate to linked lists, which have a rather large traversal overhead compared to the *DimensionList*s. This problem is addressed together with further query performance optimizations in the read-optimized *Elf*.

2.3.2 Read-Optimized Elf

The read-optimized *Elf* features three major optimizations *Linearisation*, *Perfect Hash Map* for first dimension and *MonoLists* to improve the query performance at the cost of more complex modification.

MonoLists

As one can observe in Figure 2.2, the Elf gets parse and degrades to a linked list with growing dimension. To optimize the storage consumption and traversal for this case, Broneske et al. introduced a technique called *MonoLists*, which compresses the linked-list subtree to an array with the remaining dimensions and *TID* data, as we show in Figure 2.3.

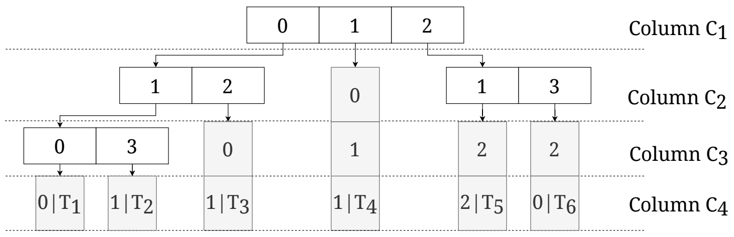
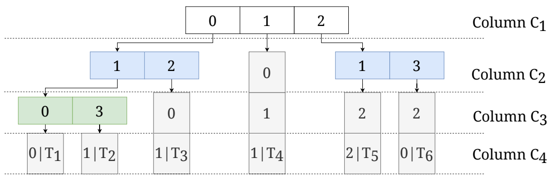


Figure 2.3: Conceptual Elf Example with *MonoLists*

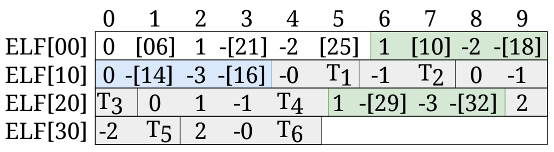
To signal, that the pointer to the next dimension points to a *MonoList*, the most significant bit (MSB) is set on the pointer value.

Linearisation

As we show in the example linearisation in Figure 2.4, the *Elf* is linearised into an array and the pointers to the next dimension are replaced with array offsets. Additionally, the end of a dimension list must be marked, since there is no longer a separate memory region per *DimensionList*. This is achieved by setting the MSB on the last value of the *DimensionList*. This linearisation results in a better cache performance and less page faults due to the absence of pointers and one large continuous memory region in which all data lie.



(a) Conceptual Elf



	0	1	2	3	4	5	6	7	8	9
ELF[00]	0	[06]	1	-[21]	-2	[25]	1	[10]	-2	-[18]
ELF[10]	0	-[14]	-3	-[16]	-0	T_1	-1	T_2	0	-1
ELF[20]	T_3	0	1	-1	T_4	1	-[29]	-3	-[32]	2
ELF[30]	-2	T_5	2	-0	T_6					

(b) Linearised Elf Layout

Figure 2.4: Linearisation of the Example Elf

First Dimension Hash Map

Since all values of the first dimension are listed together with their pointers to the next dimension, it is rather costly to perform a search for lower and upper boundaries on this list. To overcome this issue, a perfect hash map in form of a dense array is used as we depict in Figure 2.5.

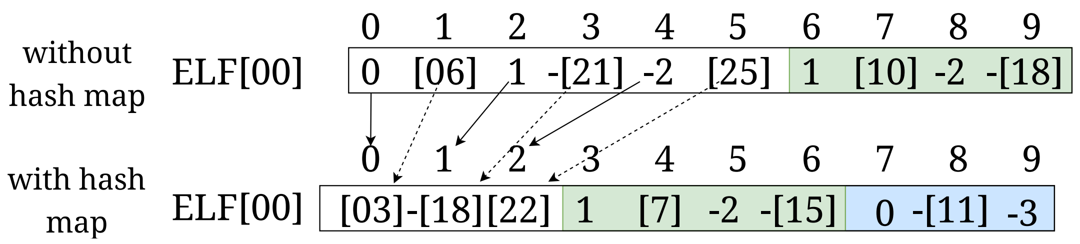


Figure 2.5: First Dimension Hash Map Transformation

Now every index of the array corresponds to the value to which it holds the pointer. Additionally some space is freed, since the value must no longer be stored explicitly. On the other hand this technique can lead to big holes in the first dimension array, which costs much memory. However when assumed the values in the first dimension are dense this is not the case.

2.4 Non-Blocking Data Structures

2.4.1 Skip Lists

A Skip List is a probabilistic linked data structure invented by Pugh et al., which has average case logarithmic search and insertion time complexity [Pugh, 1990].

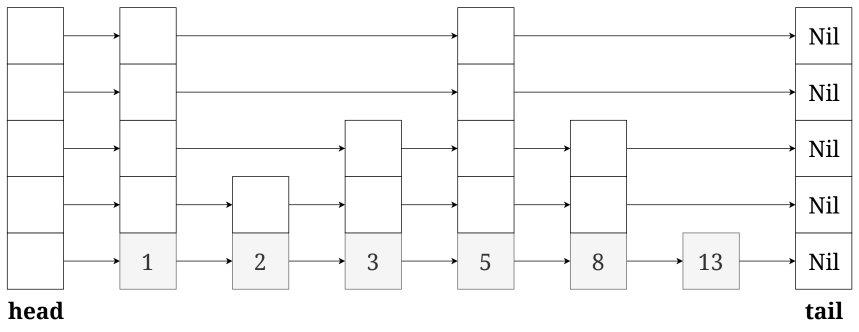


Figure 2.6: Example Skip List

As one can see in Figure 2.6, the structure of a skip list is a sorted linked list of all elements on the bottom lane and all other lanes act as fast lanes to skip some of the intermediate nodes.

Traversal

The traversal of a skip list is shaped very simply. The traversal begins at the head sentinel node of the skip list with the entry on the highest dimension and traverses the dimension until the next value is larger than the value to find. In the next step, the next lower fast lane is used; this repeats until the level is zero and the corresponding value is either found or the next value is higher than the searched one.

Insertion

For an insertion into a skip list, a traversal is used to determine the insert position, while recording the preceding nodes on every level. Then a insert level of the value is chosen randomly with a probability of the geometric distribution, where typical probabilities are $p = \frac{1}{2}$ or $p = \frac{1}{4}$. From this level on to level 0 all previous nodes are changed to point to the new inserted node and all next elements of the preceding nodes are now the predecessors of the newly inserted node.

Lock-Free Skip List

The lock-free skip list is a modification of the original skip list by Fraser et al. to become a non-blocking concurrent data structure. Therefore he utilized a technique called *pointer marking* [Harris, 2001] to describe the current state of a list element.

Pointer marking bases on the assumption, that pointers are aligned to the native architectural bandwidth. Therefore some bits which are unused due to the alignment can be used for signalling the state of the list nodes. In combination with atomic pointers this allows logic operation announcement before the actual operation is executed, which effectively prevents concurrent operations to interfere with others. In case a list pointer is marked, the operation is aborted and restarted until it succeeds.

HTM Skip List

The hardware transactional skip list also utilizes *pointer marking* to guarantee a consistent state between the transactions. Furthermore, in the actual insertion of new nodes a transaction is used to insert the node on all levels, assumed the predecessors haven't changed their logical state. In this case the transaction is aborted and the complete insertion is retried.

2.4.2 Non-Blocking Vector

While a resizeable array, often called vector of continuous memory is one of the simplest, yet cache friendliest and most perfect fit data structures, it becomes significantly more complex to handle in concurrent Programming. The idea of a resizeable array is, that once it becomes full, a new larger array is allocated, the contents from the old array copied over to the new one and finally the old array is deallocated and replaced by the new.

This process is simple when executed serially, but because the process may need a long time and at this time no access is allowed to the resizeable array. This is complicated to achieve in a parallel environment without blocking access to the data structure. Nevertheless there are some lock-free versions of resizeable arrays which could also improve performance over a locking implementation [Dechev, Pirkelbauer, & Stroustrup, 2006], [Feldman, Valera-Leon, & Dechev, 2016] as well as the `concurrent_vector` [2]. The

[2]https://software.intel.com/en-us/node/506203

`concurrent_vector` is the non-blocking vector implementation in Intel's Thread Building Blocks library, a library which aims to provide common concurrent data structures and algorithms.

3 Related Work

In this chapter, we briefly introduce some works which are related to the context of this thesis and from which we adapted some techniques for our algorithms.

Hardware-sensitive Index Structures

State of the art multidimensional index structures are clearly all focussing on the use-case as a main-memory index structure. Sprenger et al. proposed an index structure, which utilizes SIMD parallelism to accelerate its node scans and parallelize the result gathering phase with multithreading. Kim et al. proposed an index structure called FAST, which is architecture sensitive supporting parallelized CPU and GPU traversal. Similar approaches to make index structures hardware sensitive do also exist for one dimensional data structures such as the B-Tree [Levandoski, Lomet, & Sengupta, 2013]. Most of this work utilizes multiprocessing for concurrent queries and only SIMD for parallel traversal. Since vector parallelism is out of this scope, we do not adapt most of the proposed optimizations of this works.

Parallel Tree Traversal

Proposed strategies from which we can adapt some techniques are at first a general approach for distributed tree traversal. The distributed tree traversal by Ferguson and Korf , is until now one of the most used algorithm in this category. When we come to some fine-tuning decisions. There was some recently published work aiming to parallelize conventional multi-dimensional index structures such as the kd-Tree, R^*-Tree and the VA-File using horizontal parallelization [Sprenger, Schäfer, & Leser, 2018]. Since the concept of horizontal parallelism has shown as a very effective technique for task parallelism, also all our query algorithms are designed in a horizontal parallel fashion. Furthermore, for our node-parallel traversal we adapt the distributed tree traversal with a vertical parallel traversal.

Related Concurrent Insertion Approaches

Regarding concurrent insertions, there exist several works, which develop advanced locking schemes. One major work in this area is a RW-locking scheme used for parallel insertion into a B-Tree [Bayer & Schkolnick, 1977] similar to the one we use in our work to reduce contention during insertions. We will utilize this approach in our locking insertion algorithm, however since the Elf is dissimilar with a B-Tree in many points, we need do some major adaptions to the scheme. Also for lock-free concurrent insertions, there exists some similar work utilizing CAS-instructions for concurrent insert in B-trees

[Braginsky & Petrank, 2012]. Together with another approach by Chatterjee, Walulya, and Tsigas, which propose a linearisable nearest neighbour query algorithm for a lock-free kd-Tree, we use the tagging flags and a similar algorithm utilizing CAS-instructions to insert new data.

4 Query Parallelization

In the previous chapter we introduced the multi-dimensional main-memory index structure *Elf*. The Elf, even though a state-of-the-art highly optimized data structure, currently only supports serial queries and modification. This leads to the fact that a lot of optimization potential is unused. For this reason, we introduce in this chapter some approaches to parallelize the queries for the Elf without modifying its structure as well as concurrent modifications on the write-optimized Elf including adaption of the original Insert Elf version in the next chapter.

The Elf approach features several important properties and optimizations that render it the perfect structure for serial query execution. However, these properties lead to an imbalanced work when traversing the tree, which makes it challenging to find the perfect workload partitioning for concurrent Threads. In the following, we shortly outline three possible parallelization strategies for the Elf. Queries on Elf are executed in a depth first manner, whereas the first dimension does not have to be traversed sequentially to find the start and end of the traversal range. It can just be accessed directly by the value as offset in the array. This structure leads to a simple and intuitive first parallelization approach. The first dimension is just partitioned into P approximately equal parts, where P is the number of processor cores of the system. Then each thread traverses its subtree range independently.

Now, that the actual traversal has finished, the results can either be returned directly or merged into a single array. In the first case, the results are a list of *TID* arrays, which contain the *TID*s in depth first search order. In the other case the data need to be merged into one result array. We decided to use the latter approach, not only to remain fully compatible with the original Elf implementation, but also preserve the comparability between our approach and the original implementation as well as other approaches by attaining the same output.

As previously mentioned, the next step is merging of the results into one array. Therefore, the result storage size is determined and the corresponding write positions for every thread are calculated by prefix summing. Finally those results are written in parallel to the result storage.

One property of the *Elf* is, that it is not balanced due to prefix sharing and postfix linearisation in form of *MonoList*s. This could potentially lead to a performance loss due to waiting of the finished threads for one thread that has more to traverse. Therefore we changed the subtree traversal range to a single subtree per thread at a time in a round-robin fashion. This should make the traversal more insensitive to strong imbalance in the *Elf*. However this approach can still suffer from a non-dense first dimension.

In the case that sub trees have a big variation in their fan-out, most threads could have finished their execution since the last values in the first dimension have no subtree, but the last thread could still run a long time if its subtree is large. For such a case, an approach similar to distributed tree search [Ferguson & Korf, 1988] could help preventing the problem. Our third traversal is based on per node traversal and work sharing between the threads. And while this can be seen as the best version, this algorithm is far more complex and needs potentially costly inter-thread communication. Since all algorithm we introduce consist of two stages, the actual traversal and the merging phase, there are also multiple merging strategies too. They range from simple serial merge to a full parallel one.

In the following we will discuss in detail every of the three traversal algorithms we introduce starting with the naive range-parallel traversal, following the subtree-parallel traversal and at last the node-parallel traversal. Furthermore, we introduce strategies which we use for our result merging.

4.1 Range-Parallel Traversal

To achieve maximum parallelizability, an algorithm should divide its problems into smaller pieces, which can be independently solved. This means that if there is no inter-thread communication and no merging step of the sub-results, we should reach the best speed up possible. For parallel tree traversal it generally is not possible to achieve both because either there must be some communication of the threads to avoid result merging, or there is an extra merging phase. Furthermore for better maintainability of code especially in parallel programming, it is good to build on the single threaded version for the parallel algorithm. Resulting from this consideration, our reference implementation is to split the single threaded traversal into equal parts and then use the single threaded traversal independently in parallel for the sub-range traversal. We show an example traversal in Figure 4.1.

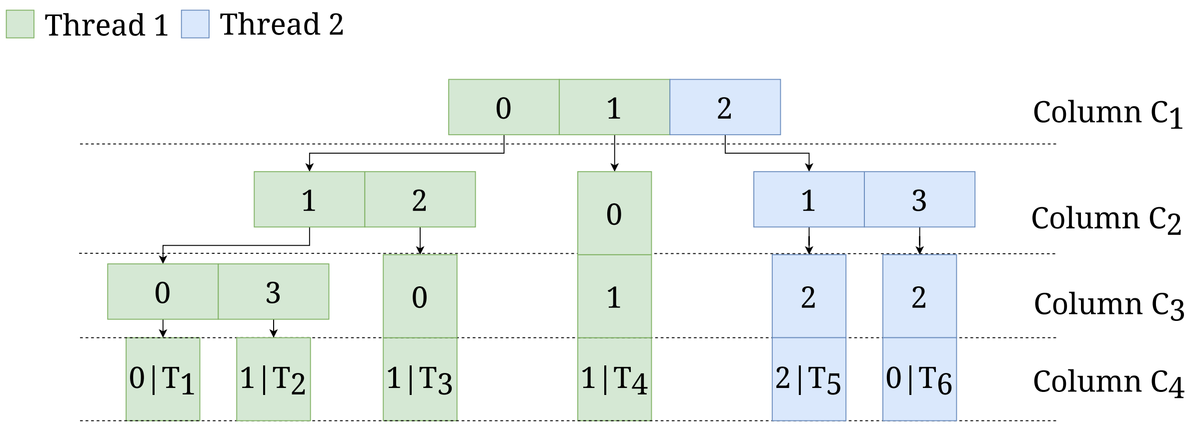


Figure 4.1: Example First Dimension Partition

In this example, the Elf is traversed by two threads, partitioning the first dimension as equally es possible between them. Afterwards each thread traverses its own subtree. This split traversal technique is, on the one hand very efficient since the sub-problems can be independently solved, on the other hand this approach needs a merging phase of the sub-results after the parallel traversal has finished. Nevertheless with this approach we have reused already optimized existing code. Furthermore this approach is very common to parallelize problems efficiently and fits perfectly as a reference implementation which we try to improve with the following three approaches. In fact, the *Range-Parallel Traversal* is a hybrid of the single threaded depth first traversal and a breadth first traversal. Whereas multiple threads traverse each of their ranges in a depth first search while the first dimension is traversed in breadth by the partitioning into sub-ranges.

4.1.1 Implementation

The simplest yet smallest overhead version is the range-parallel traversal. Since the first dimension of the read-optimized Elf consists of a perfect Hash Map designed as an array with indexes as keys, we can simply partition the search range of the first dimension as we show in Figure 4.1.

Algorithm 1: Range Traversal

```
   Input: Int[] &lower, Int[] & upper, Int numThreads
   Result: Vector<TID> result
 1 Function searchMCSP(lower, upper, numThreads) {
 2     Future<Vector<TID>> threads[numThreads];
 3     Int stepSize = (upper[0] - lower[0] + numThreads - 1) / numThreads;
 4     Atomic<Int> resultSize = 0;
 5
 6     for Int i = 0 to numThreads do
 7         threads[i] = async searchMCSPThread(lower, upper, i, stepSize, resultSize);
 8     end
 9
10     Vector<TID> intermediateResults[numThreads];
11     foreach thread ∈ threads do
12         intermeds += thread.get();
13     end
14
15     return merge(intermediateResults, resultSize);
16 }
```

To achieve this, we partition the search space from the lower bound to upper bound in P approximately equally sized parts as in Line 3.

Therefore we utilize the following sequence which represents the approximate equal partitioning of the search range into n parts:

$$(a_i) = start + \left\lceil \frac{upper - lower - i}{P} \right\rceil |i \in \mathbb{N}_0$$

Afterwards in Line 7, we asynchronously start P threads with Algorithm 2, passing each one its query bounds for the first dimension. The main thread will then wait for all started threads to return Line 12, and afterwards merge the results as described in Section 4.4 and return in Line 15.

Algorithm 2: Range Traversal Thread

```
Input: Int[] &lower, Int[] & upper, Int start, Int numElements, Atomic<Int>
       &resultSize
Result: Vector<TID> result
 1 Function searchMCSPThread(lower, upper, start, numElements, resultSize) {
 2     Vector<TID> result;
 3
 4     for Int offset = start to start + numElements do
 5         Int pointer = ELF[offset];
 6
 7         if pointer == NOT_FOUND then
 8             continue;
 9         else if noMonoList(pointer) then
10             SearchDimList(lower, upper, pointer, 1, result);
11         else
12             result += SearchML(lower, upper, unsetMSB(pointer), 1, result);
13         end
14     end
15
16     resultSize += result.size();
17     return result;
18 }
```

The threads will traverse their part of the first dimension starting from `start` for `numElements` pointers in Line 4. Then on Lines 7, 9 and 11 each pointer is checked whether it points to a *DimensionList* , *MonoList* or nothing at all because the first dimension is not dense. Afterwards, for each of the values, the corresponding existing traversal functions by Broneske et al. is called - in case of a dimension list pointer in Line 10 and for *MonoList*s in Line 12 - and the results of this functions is added to the result vector. When the traversal of the queried range is done, the result size is atomically added to the overall result size and the result vector is returned.

The major advantage despite its simplicity is the fact, that the total ordering of the result data is kept. This can be important for further query processing.

4.2 Subtree-Parallel Traversal

A major flaw of the first implementation is that the Elf is no balanced tree since it uses *MonoLists*, which compresses suffix dimensions of the inserted data. This leads to unequal traversal times for a range traversal since this unbalance can become even more dramatic between two thread ranges. To address this flaw in our second algorithm we do not traverse subtree ranges, but only single subtrees. We show an example traversal in Figure 4.2.

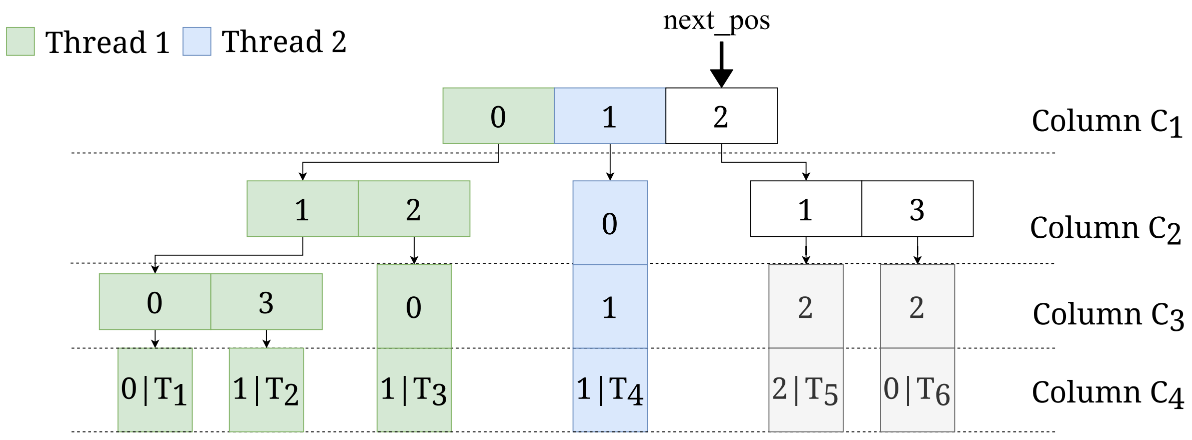


Figure 4.2: First Dimension Traversal

This traversal does not partition the first dimension beforehand, but instead each of the two threads has its start position. When a thread finishes traversing its current subtree, it just continues traversal at the position of the `next_pos` cursor which points to the next untraversed subtree. This way the traversal workload is equally distributed between all threads.
However, this leads to unsorted results, because threads do no longer traverse the Elf in order. Furthermore, this approach needs some kind of coordination between the threads, which subtree to traverse next, when a thread has finished its previous traversal. To ensure, that the cursor is properly changed for every thread, this cursor is an atomic, which guarantees that as little overhead as possible is generated for synchronisation.

4.2.1 Implementation

This algorithm works similar to the one in Section 4.1, but in this version every thread repeatedly traverses single subtrees as we show in Figure 4.2.

Algorithm 3: Subtree-Parallel Traversal

```
   Input: Int[][] lower, Int[] upper, Int numThreads
   Result: Vector<TID> result
1  Function ParallelSearchMCSP(lower, upper, numThreads) {
2      Future<Vector<TID>> threads[numThreads];
3      Atomic<Int> resultSize = 0;
4      Atomic<Int> nextIdx = lower[0];
5
6      for Int i = 0 to numThreads do
7          threads[i] = async searchMCSPThread(lower, upper, nextIdx, resultSize);
8      end
9
10     Vector<TID> intermediateResults[numThreads];
11     foreach thread ∈ threads do
12         intermediateResults += thread.get();
13     end
14
15     return merge(intermeds, resultSize);
16 }
```

To achieve interleaved sub-tree traversal, a global atomic variable in Line 4 gives the next traversal position in the first dimension and is always incremented by the thread which traverses the position pointed to by the atomic. As before we at first discuss the algorithm responsible for the thread initialization, work coordination and result merging in Algorithm 3 and afterwards the algorithm of the threads in Algorithm 4. The traversal of the first dimension is this time partitioned by the threads itself by communication of the atomic. Algorithm 3 just starts the threads in Line 7 passing each the query bounds as well as the atomic as traversal position cursor. Afterwards it waits for all threads to finish their work in Line 12 and returns the merged results in Line 15.

The worker threads start obtaining their traversal position by obtaining the next position from the atomic variable by atomically post-incrementing the atomic in Line 2, which returns the value of the position before the increment. Then, in Line 7, the actual traversal starts, a loop traverses the first dimension until the upper limit is reached in Line 3, the next position is always determined by a post-increment of the atomics's next position cursor in Line 15. The remaining subtree traversal is equal to the traversal in Algorithm 2. The algorithm just calls the matching traversal function for the subtree.

Algorithm 4: Subtree Traversal Thread

```
Input: Int[] &lower, Int[] & upper, Atomic<Int> &nextPos, Atomic<Int>
       &resultSize
Result: Vector<TID> result
Function searchMCSPThread(lower, upper, nextPos, resultSize) {
    position = nextPos++;
    while position ≤ upper[0] do
        pointer = ELF[position];

        if pointer == NOT_FOUND then
            position = nextPos++;
            continue;
        else if noMonoList(pointer) then
            SearchDimList(lower, upper, poiunter, 1, result);
        else
            result += SearchML(lower, upper, unsetMB(offset), 1, result);
        end

        position = nextPos++;
    end

    resultSize += result.size();
    return result;
}
```

However it is important to note that this kind of traversal is *not* order preserving so that the results must be sorted if this is required in further query execution. Furthermore, the jumps to new possibly not adjacent subtrees lead to page faults and cache misses, even though it reduces the effects of imbalance on the traversal time.

4.3 Node-Parallel Traversal

To reduce the time penalty introduced by waiting for threads to finish the traversal of imbalanced subtrees and furthermore remove the merge phase, our third proposed algorithm works differently from the previous two. Instead of splitting the problem in breadth on the first dimension we now have one main thread traversing the complete tree and starting helper threads similar to the distributed tree traversal by Ferguson and Korf. The helper threads then traverse their subtree *DimensionList* which will no longer be traversed by the parent thread as part of his depth-first traversal. This procedure is done recursively until P threads are busy. We show an illustration of this procedure in Figure 4.3.

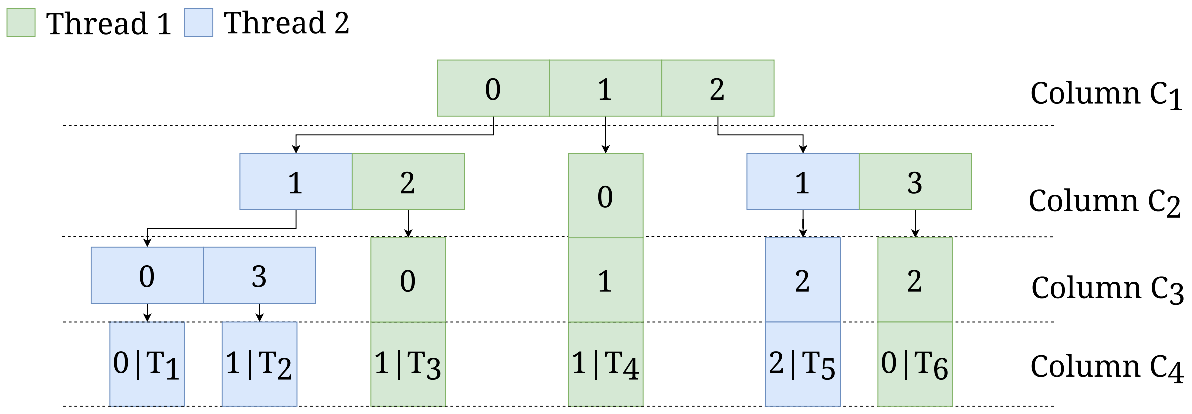


Figure 4.3: Example Node-Parallel Traversal

In the node-parallel traversal, one main thread traverses the Elf starting from the first dimension down to the next. If the next dimension is no *MonoList* and the number of running threads is smaller than a upper bound, new threads starting to traverse the *DimensionList*. Therefore, each of the started threads gets one subtree of the *DimensionList* to traverse. This way, the workload is not only balanced between the subtrees, but also between the different dimensions.

Once a thread finishes, it checks whether its potentially started child threads finished their traversal as well. If this is the case, the results from the recursive threads are merged into the result vector of the thread. Afterwards, the thread decrements a thread working counter and returns its results to its parent thread. The counter is used to limit the number of concurrent running threads to optimally utilize the CPU. If one thread finishes, the thread counter is decremented, which is a signal to the next thread checking the thread counter, that it can start more threads. This check is only done for each traversal of a dimension list. Since the traversal of a *MonoList* is only a sequential read of a vector and copying the *MonoList*'s data to the result array. This will not lead to any performance benefits when parallelized due to the fact that sub-results must be merged back from the parent thread anyway. The thread counter itself must once more be an atomic data type to guarantee that the variable is always synchronized between all threads. Otherwise more threads than advisable could be running and lead to an overload of the CPU reducing the overall performance.

4.3.1 Implementation

The implementation of the third algorithm we introduce is even finer grained than the one we presented before. Instead of traversing complete subtrees from the first dimension, threads are started during depth first traversal of a subtree until P threads are running. When a thread finishes its traversal, it decrements the thread counter and finishes its corresponding future. The next traversing thread *MonoList* will then start a new thread traversing the next untraversed subtree which is not a *MonoList* in the *DimensionList* currently traversed by the creating thread. When the thread finishes backtracking to the point where it started threads they are incrementally merged into the results. We show the corresponding code to this algorithm in Algorithm 6.

The actual traversal interface for the node level is rather simple. As for the two previous algorithms, all necessary variables are initialized in Lines 2 - 5. Afterwards the first dimension of the Elf is traversed in the range given by the query in Line 7. If a subtree exists for the value, it is either a MonoList, or a *DimensionList*. In the case of a MonoList, the corresponding traversal function is called in Line 13, which serially traverses the MonoList. In the case of a *DimensionList* , the node-parallel traversal function is called in Line 12 and the subtree traversal starts. If no subtree to the search values exists, or the traversal of the subtree is finished, the traversal continues from the next position until the search range is fully traversed. Finally, the traversal function waits for running child threads until they finish in Line 17, merge their result into the overall result in Line 18 and returns in Line 21.

Algorithm 5: Node-Parallel Traversal

```
   Input: Int[] lower, Int[] upper, Int numThreads
   Result: Vector<TID> result
 1 Function searchMCSP(lower, upper, numThreads) {
 2     Atomic<Int> runningThreads = 0;
 3     List<Future<Vector<Int>>> children;
 4     Vector<Int> result;
 5     ThreadId parentId = thisThread::getId();
 6
 7     for Int offset = lower[0] to upper[0] do
 8         Int pointer = ELF[offset];
 9
10         if pointer == NOT_FOUND then
11             continue;
12         else if noMonoList(pointer) then
13             SearchDimListParallel(lower, upper, pointer, 1, runningThreads, children,
                 parentId, result);
14         result += SearchML(lower, upper, unsetMSB(pointer), 1, result);
15     end
16
17     foreach child ∈ children do
18         result += child.get();
19     end
20
21     return result;
22 }
```

The *DimensionList* parallel traversal starts traversing one entry after another in Line 3 until the complete *DimensionList* is traversed. For each entry in the list, its value is checked in Line 4, whether it lies in the query range. If this is not the case, the position is incremented to its next value in Line 17 and checked, whether the current value is lower than the upper query bound in Line 18. If the upper bound is reached, all threads are queried whether they have finished their execution and the sub-results from all finished threads are merged to the complete result and returned in Line 20.

Otherwise the next value of the dimension list is traversed. If the value contains a pointer to a *DimensionList*, the number of runningThreads is incremented atomically in Line 7 and afterwards tested whether the maximum number of threads is reached. If the limit was reached, `runningThreads` is decremented in Line 8 and the function calls itself recursively with the new *DimensionList* in Line 9. If the number of running threads was smaller or equal to the number of `runningThreads`, the node parallel traversal function is started in a new thread and the thread is added to the child thread list in Line 11.

If the subtree pointer points to a *MonoList*, the *MonoList* is traversed and its results are merged in Line 14. After all values in the *DimensionList* are traversed, all remaining child threads in the child thread list are queried whether their results are ready to retrieve in Line 26. If the thread result is ready, the result is merged to the result of the parent thread in Line 27 and the thread removed from the child thread list in Line 28. Before the function terminates, it checks whether it was started as a new thread or was recursively called by checking if the own thread ID is equal to the `parentId` in Line 31. If this is not the case, the function was started in a new thread and decrements the `runningThreads` counter before the function returns its result in Line 34.

Algorithm 6: *DimensionList* Parallel Traversal

```
Input: Int dimension, Int listStart, Vector<Int> &result, Int[] lower, Int[] upper,
       Atomic<Int> &runningThreads, List<Future<Vector<TID>>> &children,
       Int parentId
Result: Vector<TID> result
 1 Function SearchDimListParallel(lower, upper, listStart, dimension,
    runningThreads, children, parentId, result) {
 2     Int toCompare = ELF::getValue(listStart);
 3     while notEndOfList(toCompare) do
 4         if isIn(lower[dimension], upper[dimension], toCompare) then
 5             pointer = ELF[++listStart];
 6             if noMonoList(pointer) then
 7                 if ++runningThreads > numThreads then
 8                     −−runningThreads;
 9                     SearchDimListParallel(dimension + 1, pointer, result, query,
                        runningThreads, children, parentId);
10                 else
11                     children += async SearchDimListParallel(lower, upper, poniter,
                        dimension + 1, runningThreads, new
                        List<Future<Vector<TID>>>, parentId, new Vector<TID>));
12                 end
13             else
14                 result += SearchML(lower, upper, unsetMSB(pointer), dimension + 1,
                    result);
15             end
16         end
17         position += 2;
18         if upper[dimension] < toCompare then
               /* merge thread results and remove futures from list as in
                  Lines 25 and following                                  */
19              ...
20             return result;
21         end
22         toCompare = ELF::getValue(position);
23     end
24
25     foreach elem ∈ children do
26         if elem.ready() then
27             result += elem.get();
28             children -= elem;
29         end
30     end
31     if thisThread::getId() ≠ parentId then
32         −−runningThreads;
33     end
34     return result;
35 }
```

This approach has the advantage, that it uses a more fine grained parallel approach to maximise the parallel work. The work is distributed as evenly as possible, leading to almost no waiting situations. Furthermore the merging is not done in an extra step, but during traversal and in parallel, eliminating an extra merge step with additional overhead. However it introduces some inter-thread communication, which can be costly.

4.4 Result Merging Strategies

For the result merging step of our algorithm we need to merge all P result arrays to one array. For this task exist three following options.

The simplest option is the Serial Merge in Section 4.4.1, which inserts the results serially into the final result array. While this approach may be very simple it does not fully utilize the available memory bandwidth and effectively turns out as bottleneck for the algorithm since merging is easily parallelizable. Therefore we implement a Hybrid Merge in Section 4.4.2, which merges the results in parallel.

To parallelize the merge step, the start positions of the partial results in the final result vector must be known. Otherwise data may be overwritten or the result array will not be dense. To determine the results we use a prefix summation [Hillis & Steele, 1986]. This operation adds subsequentially the result sizes of the partial results, leading to a list of partial sums. These partial sums are the start positions of the result data partition in the final results (cf. Serial Merge in Section 4.4.1).

The prefix sum operation itself can be implemented work-efficient in parallel [Ladner & Fischer, 1980], but since P and thus our array to prefix sum is expected to be rather small for our workloads, we do not expect it to payout [Nguyen, 2007]. Nevertheless we include the Parallel Merge variant for completeness reasons in Section 4.4.3.

The trade-off between the two versions previously introduced is a partially parallel merge, which computes the prefix sum serially and afterwards starts P threads to copy the intermediate results. This should lead to a big performance benefit, since multiple threads are writing in parallel to disjunct memory regions, which usually leads to a higher bandwidth and transfer rate.

In the following we will provide and explain algorithms of all three approaches.

4.4.1 Serial Merge

As previously mentioned, the serial merge is the simplest merge version, which just reserves a Vector of *TID*s preallocated with the number of elements after the merge in Line 2. Afterwards it just iterates over the array of sub-results and adds the values to the result array in Line 5. When this step has succeeded the merge is finished and the results are returned in Line 7.

Algorithm 7: Serial Merge

```
   Input: Vector<TID>[] intermediateResults, Int resultSize
   Result: Vector<TID> result
 1 Function merge(intermediateResults, resultSize) {
 2     Vector<TID>result(resultSize);
 3
 4     foreach elem ∈ intermediateResults do
 5         result += elem;
 6     end
 7     return result;
 8 }
```

4.4.2 Hybrid Merge

For a parallel merging phase, either the result data structure must have a thread safe way to randomly insert data, or the result size as well as the insert positions must be known in advance to have no overlapping writes at merge time. Since, to the best of our knowledge, no good performing concurrent data structure with random position insert exists, we determine the insert position in advance through prefix summation.

Algorithm 8: Hybrid Merge

```
    Input: Vector<TID> intermediateResults, Int resultSize
    Result: Vector<TID> result
 1  Function merge(intermediateResults, resultSize) {
 2      Vector<TID>result(resultSize);
 3      Int prefixes[intermediateResults.size()];
 4      prefixes[0] = 0;
 5
 6      for Int i = 1 to intermediateResults.size() do
 7          prefixes[i] = prefixes[i-1];
 8      end
 9
10      for  Int i = 0 to intermediateResults.size() pardo
11          for Int idx = 0 to intermediateResults[i].size() do
12              result[prefixes[i] + idx] = intermediateResults[i][idx];
13          end
14      end
15      return result;
16  }
```

The prefix summation is done by traversing the input data in Line 6, which have to be merged and add their sizes successively, writing every partial sum in an array in Line 7. The insert positions for the data now correspond to the prefix sums in the array. In the next step, P threads are started, passing each one the prefix sum of the corresponding merge data as well as the result vector in Line 10. Each thread now inserts its data, starting from its prefix sum as insert position in the result vector in Line 12. Afterwards the merge is finished and the result is returned in Line 15.

4.4.3 Parallel Merge

The full parallel merge variant merges the data as in the partially parallel Algorithm 8, but instead of calculating the prefix sum serially, it is also parallelized; in our example on Lines 4-13 with Horn's Algorithm [Horn, 2005].

Algorithm 9: Parallel Merge

Input: Vector<TID> intermediateResults, Int resultSize
Result: Vector<TID> result

```
1  Function merge(intermediateResults, resultSize) {
2      Vector<TID>result(resultSize);
3      Int prefixes[intermediateResults.size()];
4
5      for Int i = 1 to $\log_2(intermediateResults.size())$ do
6          for Int k = 0 to intermediateResults.size() pardo
7              if $k \geq 2^i$ then
8                  prefixes[k] = prefixes[k-$2^i$] + prefixes[k];
9              else
10                 prefixes[k] = prefixes[k];
11             end
12         end
13     end
14
15     for Int i = 0 to intermediateResults.size() pardo
16         for Int idx = 0 to intermediateResults[i].size() do
17             result[prefixes[i] + idx] = intermediateResults[i][idx];
18         end
19     end
20
21     return result;
22 }
```

To calculate the prefix sum in parallel, the algorithms iterates binary logarithmic in times depending on the size of the array to calculate the full prefix sum as shown in Line 5. In each of the iterations, the algorithm calculates in parallel the sum of two elements which are 2^i elements interleaved as in Line 10. This means, in the first iteration all neighbouring elements are summarized, in the second every fourth element and so on. By this scheme subsequentially all elements are summed up with all preceding elements. Afterwards we just merge the results as previously in Algorithm 8.

4.5 Linearisability

We introduced our parallel query algorithm for the read-optimized Elf, which does not allow other operations than querying. This effectively reduces the proof of linearisability to the question whether any operation modifies the Elf during traversal. Since this is not the case for any of our introduced algorithms, we do not need to consider any ordering of operations, since all operations occur always on the same data. Therefore, our query algorithms for the read-only Elf are linearisable.

This is different on the write-optimized Elf, since this data structure allows insertions. Thus we need to consider two operations and it turns out, that on the write-optimized Elf none of our introduced algorithms is linearisable.

5 Insert Parallelization

In the previous section, we described three different parallel search algorithms for the read-optimized Elf, which can easily be adapted for the write-optimized Elf. In this section we introduce three different algorithms to parallelize the insertion into the write-optimized Elf. Therefore we will first introduce a parallelized blocking version of the algorithm as reference implementation, followed by two non-blocking versions featuring state-of-the-art techniques to overcome the disadvantages of the blocking algorithm.

5.1 Blocking Parallel Insertion

Our blocking concurrent insert is based on the original insertion algorithm of the Elf with some minor changes regarding protection from data races. In the original version of the Elf by [Broneske et al., 2017], a node of the Elf is represented as an array with the respective values of the dimension, respectively the *TID*s of the rows having the values of the path. To insert values into the Elf, the Elf is traversed until a value is not found in the `Nodes` array. When this is the case, the missing value is inserted and a subtree with the values to insert from there is created.

In a parallel environment, it may occur that two values must be inserted in the same array. This insertion can lead to a data race because the data are inserted at the same position, or the array must be resized. Both are actions that are inherently not atomic and thus can lead to inconsistencies. To prevent those situations, we associate each `Node` array with a mutex, which protects the array from concurrent writes which would lead to a data race. Even though a simple mutex would be enough to prevent data races, this approach is not very efficient since read operations are also blocking the entire array even though they have no influence on correctness of the Elf. Furthermore, our critical section is rather large and the locks are potentially hold for a rather long time period, since the complete non-existing subtree is generated and inserted. This can lead to the situation that multiple threads wait for a long time until the lock is released and they can continue.

For both problems we provide the following solutions for our locking insertion algorithm: To eliminate exclusive locking on read access, we use RW-mutexes instead of normal mutexes. Shared mutexes can be locked in two ways. One way is exclusive access to the protected resource, which blocks every further locking of the mutex and thus makes writes to the resource free of data races. The other way is RW-mutex, which prevents acquiring locks for reads and allows concurrent reads while blocking modifications at the same time. This kind of mutex can greatly reduce the amount of exclusive locking and thus increase parallelizability and throughput.

For our second problem, we use a finer grained locking scheme. Instead of locking a `Node` only once and insert a complete subtree, we just insert single nodes during traversal. This ensures smaller locking periods and thus ensures more throughput. In the following, we introduce the structural changes we made on the node structure as well as the complete insertion algorithm together with our fine grained locking scheme and RW-mutexes.

5.1.1 Node Structure

The serial Insert Elf nodes by [Broneske et al., 2017] use for every node, inner- as well as leaf node the same structure; an array of Integral data types which can be interpreted as regular *DimensionList* or Leaf Node, which only contains a list of *TID*s, depending on the node type. To improve the maintainability and guarantee consistent concurrency, we define the abstract parent class *LKElfNode* for inner nodes as well as leaf nodes and specialize both types for their specific needs. We replaced the integer array of the inner node (*LKLeafNode*) with a Vector of key-pointer pairs. This guarantees a correct ordering on concurrent insertion of multiple key-pointer pairs while giving the flexibility of inserting keys and pointers separated from each other in different but concurrent steps. Furthermore, this restructuring improves maintainability by structuring keys and pointers without introducing an overhead. This is due to the definition of a Pair, which is just a container for in this case key and pointer data with no own space requirements. In addition to the change of the entry representation, we store the previously mentioned, corresponding mutex for entries in the node.

To make the node functionalities more independent from its implementation and simplify the implementation of the insertion and traversal algorithms, we identify a small but sufficient set of functionalities required to insert and traverse inner nodes. We add this set of functionalities in form of member functions to the node, from which some ensure, that all access to the node's data are data race free, and others allow manual locking control without modification. Our set of functionalities includes:

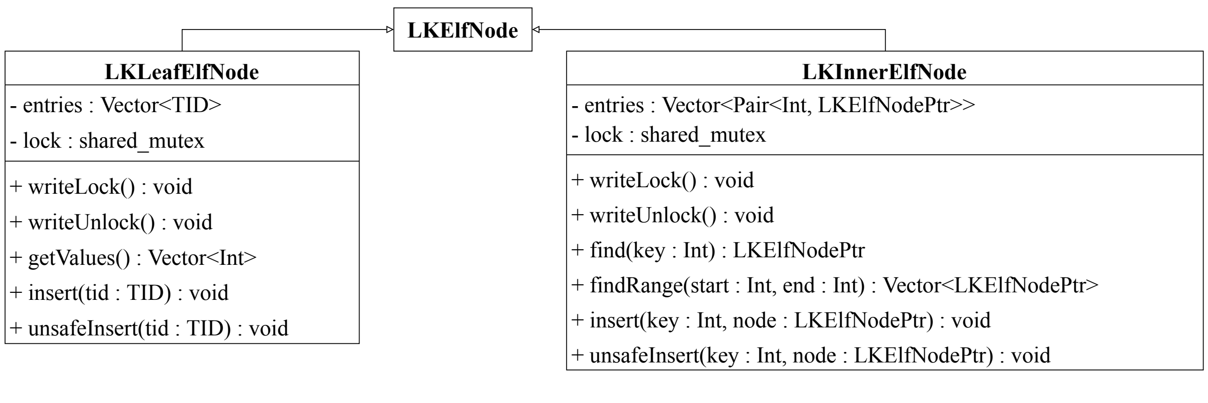

LKElfNode
LKLeafElfNode
- entries : Vector<TID>
- lock : shared_mutex
+ writeLock() : void
+ writeUnlock() : void
+ getValues() : Vector<Int>
+ insert(tid : TID) : void
+ unsafeInsert(tid : TID) : void
LKInnerElfNode
- entries : Vector<Pair<Int, LKElfNodePtr>>
- lock : shared_mutex
+ writeLock() : void
+ writeUnlock() : void
+ find(key : Int) : LKElfNodePtr
+ findRange(start : Int, end : Int) : Vector<LKElfNodePtr>
+ insert(key : Int, node : LKElfNodePtr) : void
+ unsafeInsert(key : Int, node : LKElfNodePtr) : void

Figure 5.1: Locking Elf Nodes

find(key : Int) This function fulfils the need to query single values for exact match queries. Therefore, it takes a *key* and returns a pointer to the corresponding *LKElfNode* if the key exists. Otherwise it returns a *nil*. Since the operation accesses the `entries` vector only reading, this operation needs to lock the data only with a non-acquiring lock to ensure the correctness of the query results.

rangeFind(start : Int, end : Int) is the pendant for our first find function for range queries. This function does not only return a single *LKElfNode*, but instead a Vector of *LKElfNode*s whose corresponding keys lie in the search range. Since this operation is also read only, a non-acquiring lock is sufficient.

insert(key : Int, node : LKElfNodePtr) This operation allows safe insertion of a key-pointer pair into the node. Therefore, move operations may be necessary to insert the new data in the sorted Vector. Since this is a write operation, the entries are locked for exclusive access and afterwards released.

unsafeInsert(key : Int, node : LKElfNodePtr) The pendant to the insert operation, but potentially unsafe, since no locking takes place and synchronisation must be handled from outside by the following **writeLock()** and **writeUnlock()** functions. However this allows more control over the locking granularity.

writeLock() This function obtains the acquiring lock of the node to make exclusive modifications from outside of the node possible without using the provided functionality while protecting the entries from concurrent modification.

writeUnlock() This function releases a previously acquired lock via *writeLock* on the node's data.

For the *LKLeafElfNode* the find functions are not needed, because leaf nodes hold only *TID*s which match with the columns that build the path from the Elf root to the leaf. Thus also key-pointer pairs are not needed, only a vector of *TID*s with a getter for the vector, which obtains a lock when copying the vector. Furthermore the insert functions are adapted to append new *TID*s to the end of the vector and if needed to lock the node during this operation. Now that we introduced the Elf node structure and its functions, we can introduce the locking insertion algorithm.

5.1.2 Implementation

Our insert algorithm takes the *TID* to insert together with its dimension values. To locate the insert position, the traversal starts from the root node in Line 2, querying each subsequent node on the path in Line 7 until the last dimension is reached in Line 6, or no corresponding value in the dimension is found in Line 11. During each call of the `find` function, the nodes traversed by the function are locked one by one. When no value is found in the dimension, we construct a new *LKInnerElfNode* in Line 12 manually write locking its internal structures in Line 13 to protect the new empty node from concurrent modifications from other nodes. Afterwards we try to insert the newly constructed node pointer with its corresponding key into the Elf in Line 14. If this step is not successful, in the meantime another concurrent insertion could have inserted a node with the same key. In this case, the newly created node is unlocked and destructed in Line 18.

If this step succeeded, which is the case when no other concurrent insert operation successfully inserted a node with the same key in the meantime, then an insertion flag is set in Line 16. This signals that a new inner node was inserted and must be unlocked later. The traversal now continues in Line 25 with subsequent insertions in the newly generated subtree by constructing new nodes in Line 26, inserting them in the locked node inserted previously in Line 28, unblock the previous node in Line 29 and change the node to traverse to the new node in Line 30.

Algorithm 10: Locking Parallel Insertion

```
Input: Int[] toInsert, TID tid
Function insert(toInsert, tid) {
    LKElfNodePtr currentNode = root;
    Bool newInserted = false;
    Int dim = 0;

    while dim < NUM_DIM do
        // locks nodes traversed with non-acquiring lock during serach
        LKElfNodePtr ptr = currentNode->find(toInsert[dim]);
        if $ptr \neq nil$ then
            currentNode = ptr;
            ++dim;
        else
            LKElfNodePtr newNode = new LKInnerElfNode;
            newNode->writeLock();
            if currentNode->insert(toInsert[dim++], newNode) then
                currentNode = newNode;
                newInserted = true;
                break;
            else
                newNode->writeUnlock();
                delete newNode;
            end
        end
    end

    while dim < NUM_DIM - 1 do
        LKElfNodePtr newNode = new LKInnerElfNode;
        newNode->writeLock();
        currentNode->unsafeInsert(toInsert[dim++], newNode);
        currentNode->writeUnlock();
        currentNode = newNode;
    end

    if newInserted then
        LKElfNodePtr newNode = new LKLeafElfNode;
        newNode->writeLock();
        currentNode->unsafeInsert(toInsert[dim++], newNode);
        currentNode->writeUnlock();
        currentNode = newNode;
        currentNode->unsafeInsert(tid);
        currentNode->writeUnlock();
    else
        currentNode->insert(tid);
    end
}
```

This scheme guarantees, that concurrent traversals or insertions in the same node are blocked and wait until the node contains data. Furthermore this reduces the lock times since only single nodes are inserted and not a full subtree. This also enables concurrent operations to work with partially inserted subtrees. When all inner nodes up to the last dimension are inserted or traversed, we check the insertion flag in Line 33 whether we inserted new nodes. If this is the case, we also need to construct and insert a new leaf node in Line 36. Then insert the *TID* and unlock the leaf node. Otherwise, if we have not inserted any inner nodes, we can just insert the new *TID*, using the safe insert function of the leaf in Line 42.

5.2 Non-Blocking Parallel Insertion

In our blocking reference implementation we have not greatly altered the structure of an Elf node. This leads to the limitation that concurrent insertion per node is limited to one at a time, because the only way to guarantee correctness of the node's content is to lock the complete array. To overcome this limitation, the data structure holding the node's data must be changed.

Inner Node Adaptions

Alternative data structures suited for concurrent insertions are either a concurrent array implementation as previously introduced, or completely different data structures such as trees, linked lists or hash maps. However, hash maps are not well suited for range searches since the data are unordered and concurrent arrays are too slow for in-order insertion in an sorted array. This is, to the best of our knowledge, due to the fact that random position insertion in a concurrent vector has much larger overhead compared to serial insertion in all existing implementations [Feldman et al., 2016], [Dechev et al., 2006].

The remaining variants are linked lists and trees, which are especially well suited for our use case because they are linked data structures, which have generally very good random position insertion complexity. A tree has logarithmic traversal complexity in case of balanced search trees while having no space overhead against linked lists, making it superior to the list. The disadvantage of trees is, that they need rebalancing which introduces some overhead at insertion or traversal time. As a trade-off between the simplicity of linked lists and the performance of trees we choose a skip list as data structure with the actual column data as key and a pointer to the next node as value for our *DimensionList* data. This data structure is furthermore well suited as concurrent data structure while being well studied for parallelization [Fraser, 2004], [Kardaras, Siakavaras, Nikas, Goumas, & Koziris, 2018], [Sprenger, Zeuch, & Leser, 2017]. As random distribution probability for the geometric distribution of fast lanes, we decided to use a probability of $p = 0.5$, since with this probability provides the best balance of fast lanes on average [Pugh, 1990]. Furthermore, we choose a fixed size array to store the fast lane data for a simpler node structure and expected better performance [Sprenger et al., 2017]. As size of the fast lane array we choose twelve pointers. At this size we are able to balance up to $2^1 2$ entries.

Unfortunately no lock-free, or transactional range search operations for our chosen skip lists exist, to the best of our knowledge. However, they are rather straight forward to implement and thus we will introduce our modifications of the provided search algorithm to support range queries in the following sections.

Leaf Node Adaptions

While the skip list suits well as a structure for the inner nodes, due to their ordering and simple random insertion, this criteria are not important for leaf nodes. The *TID* data in a leaf node are usually unsorted, data can be always appended and need no ordering during insertion. This leads us to the decision to use the *concurrent_vector* as data structure for the leaf nodes. This continuous data structure, gives us optimal performance when fetching *TID*s. Since usually all *TID*s in a node are fetched, an array is the best performing data structure for those kind of tasks.

Flags for State Indication

For a non-blocking implementation there is furthermore the need to adapt the insertion algorithm. We adapt the reference implementation by introducing a flag on every pointer to an Elf node. This flag signals whether a node is either inserted but uninitialized or ready for traversal. The flag only needs one bit memory for signalling and thus can easily be stored in the least significant bit on architectures aligning their pointer to at least a 16-bit boundary, which is practically the case on any modern architecture. With this modifications we are able to introduce the modifications of the non-blocking skip lists, following the structures of the transactional and the lock-free Insert Elf nodes and finally the implementation of the insertion algorithm.

5.2.1 Range Find for Transactional Skip List

For our implementation of a range search (cf. Algorithm 11) for the transactional skip list by Kardaras et al., we adapted the search algorithm for single value search. The idea is to search in the skip list to the value which is the closest one to the lower bound possible. Afterwards we traverse and collect all values from this point until the upper bound or the end of the skip list is reached.

Algorithm 11: Transactional Skip List Range Search

```
   Input: Int lowerBound, Int upperBound
   Output: Vector<ElfNodePtr> result
 1 Function find(lowerBound, upperBound) {
 2     Vector<ElfNodePtr> result;
 3     SkipListNode curr, pred;
 4     pred = head;
 5
 6     for Int h = MAX_LEVEL to 0 do
 7         curr = pred.next[h];
 8         while lowerBound > curr.key do
 9             pred = curr;
10             curr = pred.next[h];
11         end
12     end
13
14     while curr.key ≤ upperBound do
15         while curr.state == INITIAL do
16             ;
17         end
18         if curr.state ≠ DELETED then
19             result += curr.value;
20         end
21     end
22     return result;
23 }
```

To determine the start position of the queried range in the list, we use the part of the `find` implementation to traverse to the point where the first value in the search range lies. Thus we start at the head of the list in Line 4 and traverse down to the last dimension from Line 6 to 12. When the last dimension is reached, the current node is the first node within the search range or a node past the `upperBound` in case no value within the search range was found.

Now all values in the linked list that are smaller than `upperBound` are traversed in Line 14. For each of the values its state is tested. If the state is *INITIAL*, the insertion of the value is not completed yet and the traversal waits for completion in Line 15. Afterwards, if the value is not marked as *DELETED*, the value is added to the results in Line 19. When all values in the search range are traversed, the function returns the result values in Line 22.

5.2.2 Range Find for Lock-Free Skip List

The lock-free skip list must be adapted in a similar manner as the transactional skip list. Fortunately, for the initial search for the beginning of the search ranges a `list_search` function from the reference implementation by Fraser already exists. Therefore, the range query algorithm is slightly simpler, even though in this algorithm we need to deal with atomics and a different syntax for manipulation of flags. In this algorithm we utilize the flag manipulation functions introduced by Fraser et al. Furthermore, flags are only used to signal deletion, since new values are always fully initialized when inserted.

The Algorithm 12 itself is despite the adaptions mentioned earlier the same for the lock-free skip list, i.e., the same as in Algorithm 11.

Algorithm 12: Lock-Free Skip List Range Search

```
Input: Int lowerBound, Int upperBound
Output: Vector<ElfNodePtr> result
 1 Function find(lowerBound, upperBound) {
 2     Vector<ElfNodePtr> result;
 3     SkipListNode *curr = list_search(lowerBound).second[0];
 4
 5     while curr->key ≤ upperBound do
 6         if is_marked(curr) then
 7             curr = unmark(curr)->next[0].load();
 8             continue;
 9         end
10         result += curr.value;
11         curr = curr->next[0].load();
12     end
13
14     return result;
15 }
```

The algorithm starts by initializing the cursor with the lowest value of the skip list, which is enclosed in the search range. If no such value exists, the lowest element, that is higher than `upperBound` is returned in Line 3. In the next step, all values in the skip list that match the search range from the start element onwards are subsequentially traversed in Line 5. If the node currently in traversal is marked as deleted, the flag is removed from the tagged pointer to obtain the real pointer and the pointer to the next element is fetched atomically in Line 7. In this way, the marked pointer is ignored and the traversal is continued. If an element is not marked as deleted, its value is added to the results in Line 10 and the next element is loaded in Line 11. When all elements in the search range are traversed, the traversal ends and returns the result Vector in Line 14.

5.2.3 Transactional Node Structure

The node structure for the transactional Elf, which we show in Figure 5.2, is, despite the same class hierarchy and some common node functions, highly different from the structure of the first introduced blocking Elf nodes. In the *TMInnerElfNode*, we replaced the vector of pairs by a transactional key-pointer skip list which does not need a mutex. Hence, the mutex is omitted from the *TMElfNode*s. This leads to the fact, that no unsafe and manual locking functions are included. In exchange we add new functions which manipulate or prompt the state of the node. There are two possible states, `INITIAL` and `INSERT` which can be mapped to a single indicating flag. To represent this flag, we utilize the least significant bit (LSB) from the pointer to the skip list. This is possible, since on modern architectures all data types are aligned to their respective size boundary for performance reasons.

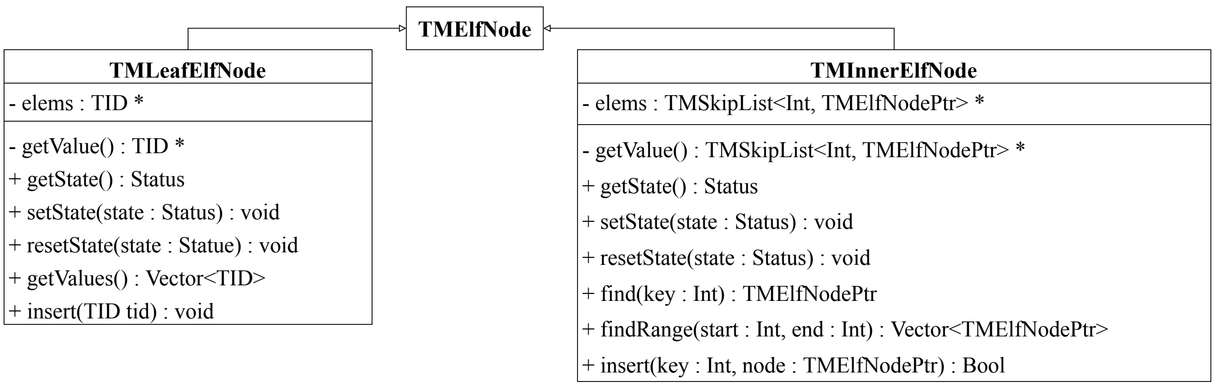

TMElfNode
TMLeafElfNode
- elems : TID *
- getValue() : TID *
+ getState() : Status
+ setState(state : Status) : void
+ resetState(state : Statue) : void
+ getValues() : Vector<TID>
+ insert(TID tid) : void
TMInnerElfNode
- elems : TMSkipList<Int, TMElfNodePtr> *
- getValue() : TMSkipList<Int, TMElfNodePtr> *
+ getState() : Status
+ setState(state : Status) : void
+ resetState(state : Status) : void
+ find(key : Int) : TMElfNodePtr
+ findRange(start : Int, end : Int) : Vector<TMElfNodePtr>
+ insert(key : Int, node : TMElfNodePtr) : Bool

Figure 5.2: Transactional Elf Nodes

For the *TMLeafNode* we replaced the vector with an own resizeable array, which we represent as a plain memory area of type *TID*. Allocations and freeing of plain memory is always possible as memory transactions, thus we need no special data structure and just include the resizing of the array in the insert function. In the following we introduce the previously mentioned and further member functions:

getValue() This function removes the state flag from the pointer to restore the original location of the skip list. This operation is used in all find and insert operations to guarantee modifications to occur in correct memory locations.

getState() This operation reads the LSB from the pointer to the skip list and interprets its state.

setState(state : Status) sets the LSB to the state passed as parameter.

resetState(state : Status) resets the state passed as parameter.

find(key : Int) As in the *LKInnerElfNode* this function is needed for *exact match queries*, but this time does not use a linear search on the vector. It calls the traversal function of the transactional skip list and returns either the corresponding *TMElfNodePtr* to the key or a *nil* when the key does not exist in the *TMInnerElfNode*.

findRange(start : Int, end : Int) This function acts as a wrapper to the skip lists `findRange` call. First it recovers the pointer to the skip list and then returns the skip lists range search results.

insert(key : Int, node : TMElfNodePtr) This function recovers the skip list pointer and forwards the insertion parameters to the skip list's insert function.

5.2.4 Lock-Free Node Structure

As previously mentioned, we use the lock-free skip list by Fraser to store the dimension data for *LFInnerElfNode*s 5.3. Furthermore, we utilize the `concurrent_vector` to store the *TID* data in the *LFLeafElfNode* 5.3. With this modification we are able to implement the node's member functions in a lock-free way. Since the algorithm for insertion in the lock-free Elf also makes use of node states, this functionality is implemented in the same way as in the transactional Elf.

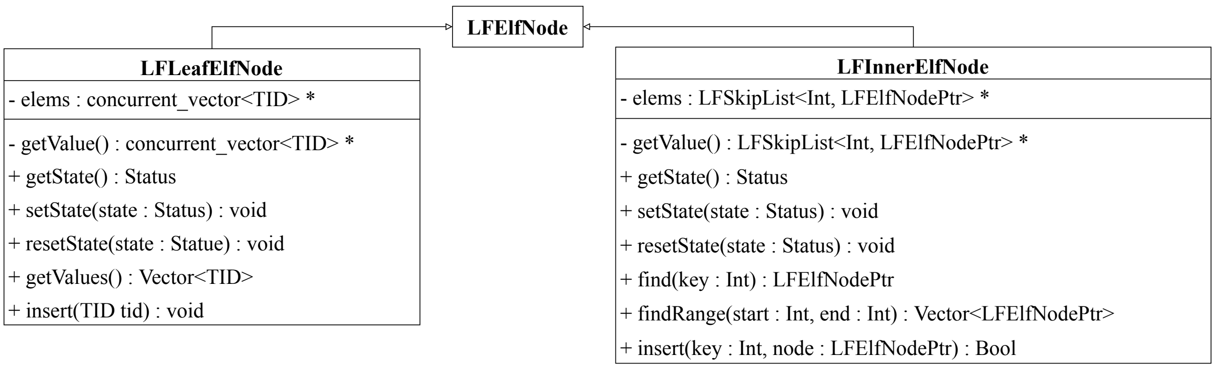

LFElfNode
LFLeafElfNode
- elems : concurrent_vector<TID> *
- getValue() : concurrent_vector<TID> *
+ getState() : Status
+ setState(state : Status) : void
+ resetState(state : Statue) : void
+ getValues() : Vector<TID>
+ insert(TID tid) : void
LFInnerElfNode
- elems : LFSkipList<Int, LFElfNodePtr> *
- getValue() : LFSkipList<Int, LFElfNodePtr> *
+ getState() : Status
+ setState(state : Status) : void
+ resetState(state : Status) : void
+ find(key : Int) : LFElfNodePtr
+ findRange(start : Int, end : Int) : Vector<LFElfNodePtr>
+ insert(key : Int, node : LFElfNodePtr) : Bool

Figure 5.3: Lock-Free Elf Nodes

getValue() This function removes the state flag from the pointer to restore the original location of the skip list. This operation is used in all find and insert operations to guarantee modifications to occur in correct memory locations.

getState() This operation reads the LSB from the pointer to the skip list and interprets its state.

setState(state : Status) sets the LSB of the `elems` pointer to the state passed as parameter.

resetState(state : Status) resets the `entry`-pointers LSB according to the state passed as parameter.

find(key : Int) As in the previous two node implementations, this function is provided for the exact match query algorithm.

findRange(start : Int, end : Int) This function acts as a wrapper to the skip list `findRange` call. First it recovers the pointer to the skip list and then returns the skip lists range search results as a vector of *LFElfNodePtr*.

insert(key : Int, node : LFElfNodePtr) This functions recovers the skip list pointer and forwards the insertion parameters to the skip lists lock-free insert function.

5.2.5 Implementation

Similar to the locking insertion algorithm, we start our non-blocking parallel insertion algorithm, Algorithm 13, by traversing the Elf from the root node in Line 2 until the tree is traversed to the last dimensional node in Line 3.

Algorithm 13: Non-Blocking Parallel Insertion

```
Input: Int[] toInsert, TID tid
 1 Function insert(toInsert, tid) {
 2     ElfNodePtr currentNode = root;
 3     for Int dim = 0 to NUM_DIM - 1 do
 4         RETRY:
 5         ElfNodePtr pos = currentNode->find(toInsert[dim]);
 6         if $pos \neq nil$ then
 7             currentNode = pos;
 8         else
 9             ElfNodePtr newNode = new InnerElfNode;
10             newNode->setState(INITIAL);
11             Bool success = currentNode->insert(toInsert[dim], newNode);
12             if $\neg$ success then
13                 delete newNode;
14                 goto RETRY;
15             end
16             currentNode->resetState(INITIAL);
17             currentNode = newNode;
18         end
19     end
20     while true do
21         ElfNodePtr pos = currentNode->find(toInsert[NUM_DIM - 1]);
22         if $pos \neq nil$ then
23             pos->insert(TID);
24             pos->resetState(INITIAL);
25             return;
26         else
27             ElfNodePtr newNode = new LeafElfNode;
28             newNode->setState(INITIAL);
29             Bool success = currentNode->insert(toInsert[NUM_DIM - 1], newNode);
30             if $\neg$ success then
31                 delete newNode;
32                 continue;
33             end
34             currentNode->resetState(INITIAL);
35             newNode->insert(tid);
36             newNode->resetState(INITIAL);
37             return;
38         end
39     end
40 }
```

It is possible that during the traversal of inner nodes, the subtree in which data should be inserted does not exist (cf. Line 6). In this case, a new inner node for the corresponding key is created in Line 9 and its state is marked as `INITIAL` in Line 10. Then we insert the node on a best effort basis in Line 11. If now a node with the corresponding key exists, a concurrent operation inserted the subtree node in the meantime. If this is the case, the newly generated node is destroyed in Line 13 and the traversal of the dimension is redone in Line 14. Otherwise, the state of the node in which was inserted is reset to default in Line 16.

Since the `currentNode` could have already been inserted in a previous iteration, the node could now be in an `INITIAL` state, signalling concurrent queries to wait until the initialisation is finished. In the case that the node was not newly created, these instructions have no influence. After the modification, traversal continues and missing nodes are inserted as described beforehand. Then the traversal finishes and the child node from the current node, being the last dimensional node is queried for the leaf node in Line 21. If the leaf node exists, the *TID* is inserted into the node in Line 23 and the state of the leaf node is reset in Line 24 due to the concurrent nature of the algorithm, which allows the insertion of a new uninitialized node in the meantime. When the node status is reset, the traversal is finished and the function returns in Line 25.

In the case, that no leaf node with the searched key exists, a new leaf node is instantiated in Line 27 and its state is set accordingly in Line 28. Now we attempt to insert the newly created node with its corresponding key in Line 29. If this fails, in the meantime another node with the same key has already been inserted. In this case the newly generated node is deleted in Line 31 and the last dimension traversal is redone in Line 32. Otherwise, the insertion was successful and the state of the inner node is reset in Line 36. Afterwards the *TID* is inserted into the leaf node in Line 35 and the leaf nodes status is reset in Line 36. This finishes the insertion and the algorithm returns in Line 37.

5.3 Linearisability

For the proof of linearisability we need to show, that all possible operation histories are linearisable. This essentially gives us three possible cases:

Query operations only In this case it is easily observable, that this is equivalent to the previously proven read-optimized Elf case, thus concurrent queries are linearisable.

Insert operations only For this case it is a little more complex to show its linearisability. To do so, we set the linearisation point of a successful insert to the insertion of the *TID* in the leaf node. This is linearisable, since nodes are marked as `INITIAL` as long as the insertion is about to be done. This linearises all other insert operations on the same new subtree behind the first insert operation. For insert operations on different subtrees, operations are simply ordered by termination order, which is linearisable, since until the successful return no other operation made changes on the same subtree as well as different subtrees. For failed insertion the linearisation point is the successful search of the *TID* in the leaf node, since the leaf node structure holding the *TID*s is itself linearisable, it is guaranteed, that the complete operation is linearisable.

Concurrent insertion and queries In this case queries and insertions can be intermixed arbitrarily, but the linearisability criterium remains true anyway. Unfortunately this is not the case for our implementations. A simple counter sample is the sequence of operations in Figure 5.4, which run concurrently.

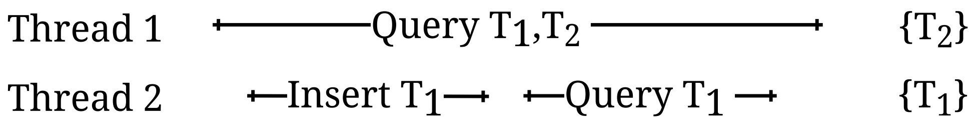


Figure 5.4: Non-Linearisable History

At first, thread 1 queries for tuple T_1 and T_2 as range query, during this query an insert operation in thread 2 is started, which inserts T_1. When the insertion has finished, the thread queries for the previously inserted value. With this constellation of operations it is possible, that the data are inserted into an already traversed subtree by the first query. Now, that the insertion finished and the query results do not contain the newly inserted tuple, the insert operation must be ordered after the query operation. When this is the case our second query operation can still find the newly inserted data and finishes next. Since the first query must be ordered before the insert operation but in the meantime another query already returned the inserted data, this leads to inconsistencies in the observed results and thus breaks the linearisability.

Even though our introduced algorithms are not linearisable, there exist some approaches to linearise range queries on the nodes data structures as well as overall linearisation approaches which can possibly be ported to the Elf [Chatterjee, 2017], [Brown & Avni, 2012]. However, due to the limited time and scope of this thesis, we leave this adaptions for future work.

6 Evaluation

In this chapter we will evaluate our parallel query, as well as concurrent insertion and merge algorithms from an empirical perspective. For the empirical aspect, we conduct the following two kinds of experiments.

Selected Queries

The first one is our micro-benchmark, with which we will determine the scaling behaviour of the different query and merging algorithms with growing concurrency.

For our second aspect we use the TPC-H benchmark queries [Council, 2014] to analyse how our algorithms behave in a real world scenario. In this benchmark, we are interested in the speed-up of query times with growing concurrency, as well as the break-even point of query times against the implementation of [Broneske et al., 2017]. With the TPC-H data we are also benchmarking our concurrent insertion algorithms. For these, we examine the consistency and scalability of the read and write times with growing concurrency under different work-loads.

For our evaluation, all algorithms are written in C++, as the original Elf implementation. Furthermore, all algorithm are tuned to an equal extend to ensure a fair comparison. For our experiments we used Version 8.3 of the GCC with the following compiler flags `-O3 -march=native -ffast-math -fgnu-tm`. These build flags arrange that all available optimizations for the target architecture, such as target specific instruction sets, for example AVX are enabled and used by the compiler. It also activates support for transactional memory and takes over the necessary linking for the hybrid transactional memory library.

Test Machine

Our tests run on a Ubuntu 18.04.2 with Linux kernel version 4.15.0-51. The test machine is a dual Intel® Xeon® Gold 6130 processor with 2.1 GHz base clock, 16 cores and 22.5MB L3 cache each. The available memory of the system was 768GB main DDR4-2666 ECC memory. We left hyper-threading and Turbo Boost functionalities turned on and did not set any scheduling or NUMA settings. While this may lead to some irritating latency peaks in our measurement results at the first sight, we come more closer to a real world performance profile.

For our insertion benchmark we used the non-standard heap manager `tcmalloc`, which features non-blocking allocations. This is needed in order to benchmark the actual performance of the non-blocking insertion algorithm and to not falsify the non-blocking property by blocking allocation algorithms.

In the following sections, we will evaluate the performance of our presented algorithms, starting with our selected merging strategies, following the query evaluation and leading to the concurrent insertion.

6.1 Merging Strategies

To determine the best merging strategy, we decided to use a microbenchmark with P arrays of random data, which have to be merged. The array sizes vary between 1,000 and 1 million values per array. To have as less derivation as possible, each benchmark was repeated 1,000 times and the average of all results was taken as the final value. In the following Figure 6.1, we show the merging times for the serial, hybrid and parallel merge dependent from the size of the arrays.

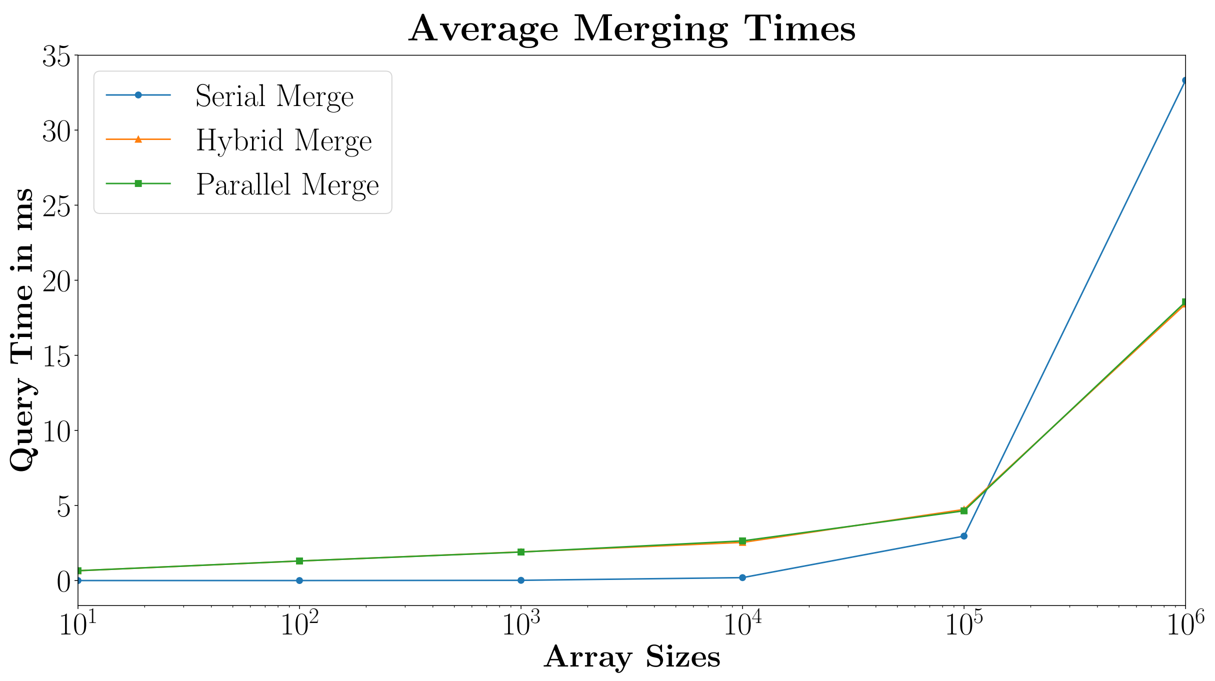


Figure 6.1: Average Merging Times

The serial merge algorithm scales linear and is due to its simplicity and absence of thread initialization overhead the fastest algorithm for small merge sizes. The hybrid and parallel merge algorithms perform significantly worse for small merge sizes, since for this case the thread initiation overhead is bigger than the parallelized work. This changes for result sizes larger than 100,000 elements. From thereon, the improved throughput of parallel merge algorithm starts to payout against its initialization overhead. However, the full parallel algorithm does not perform better than its hybrid counterpart. This is probably due to the small prefix sum size of 64 in our case. For more parts to merge, this behaviour will change and the parallel merge algorithm will become the fastest.

To reduce the variability of the remaining evaluation, we decided to refer to the hybrid merge strategy as our benchmark choice, since this strategy is a good trade off between a full parallel merge algorithm, which is for the rather small amount of processors used in this evaluation too slow. Furthermore, the serial algorithm, which performs excellent for result sizes below 100,000 elements, drops performance drastically with higher result sizes. By choosing the hybrid merge algorithm, we hope to get a better speed up for queries on large tables with a with a lower selectivity. By doing so, we aim to close the gap between a table scan and the Elf for low selectivity a little bit.

6.2 Query Approaches

To evaluate our parallel query algorithms, we start by comparing the original Elf implementation by [Broneske et al., 2017] as well as the parallel algorithms against each other, using a microbenchmark. Afterwards, we evaluate the real world performance of the range- and subtree-parallel query algorithms.

6.2.1 Microbenchmark

The microbenchmark consists of range queries of the TPC-H `Lineitem` table data with a scaling factor of 100. We measure the average query time of 1,000 queries, while using different numbers of threads for the queries. In Figure 6.2, we show the average query times depending on the number of threads for all three approaches.

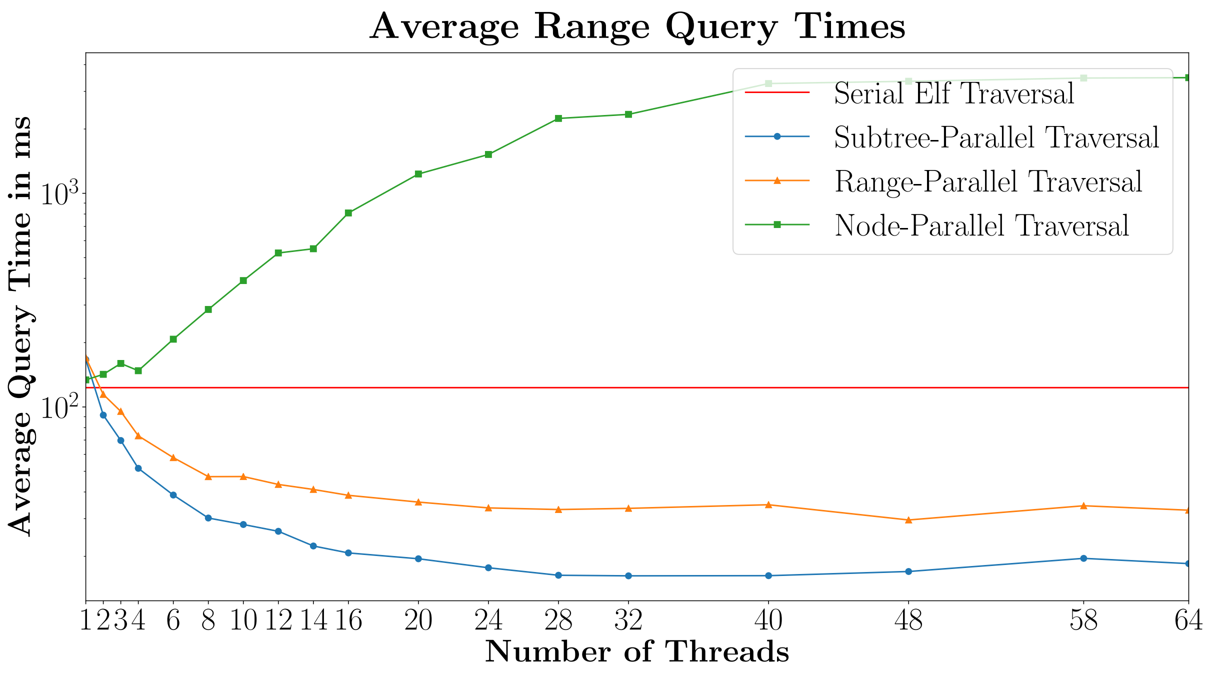


Figure 6.2: Average Range Query Time for 1,000 Repetitions With Scaling Factor 100

As we show in Figure 6.2, our node parallel algorithms does not perform well and is not able to accelerate the traversal at all. This behaviour comes from the big communication and synchronisation overhead introduced by the fine grained parallelism on the level of single dimension lists. However, with a very large number of threads, the query time seems to stabilize on a high level. The subtree as well as the range parallel traversal behave differently. With only one thread, both have a significant overhead of around 73% compared to the original Elf traversal algorithm. This latency is introduced by the thread setup and spawning overhead and since the overall query times are rather low, this overhead as a significant influence on the query times. However, the range- and subtree-parallel traversal benefit greatly from more physical core and the query times shrink drastically. The range-parallel algorithm is able to speed-up range queries by factor 3.7 and the subtree-parallel algorithm even by factor 6.6. Furthermore, they both reach their lowest query times with the number of threads matching the number of physical cores. With more threads than physical cores, the performance becomes worse, probably due to the high dependence of the query time from the bandwidth of the system, added interconnect traffic for NUMA and increased latency due to hyper-threading. Likewise it is remarkable that the single subtree traversal is by a factor of 1.7 faster than the range-parallel traversal, despite the additional communication overhead. This speed-up can be explained by the fact that for range queries the data are probably not equally distributed. Thus the Elf is imbalanced and some search ranges contain less values while others are rather dense. For the range-parallel traversal this means, that some threads need significantly longer than other threads. The subtree-parallel traversal does not have this drawback, since a thread does only traverse one subtree at a time. Thus, the traversal of more, or less dense ranges are equally distributed between all threads. This results in a drastic reduction of query times due to less busy waiting.

Due to the bad performance of our node parallel traversal compared to the other two traversal algorithms, we decided to only compare subtree- and range-parallel traversal against each other in the TPC-H benchmark.

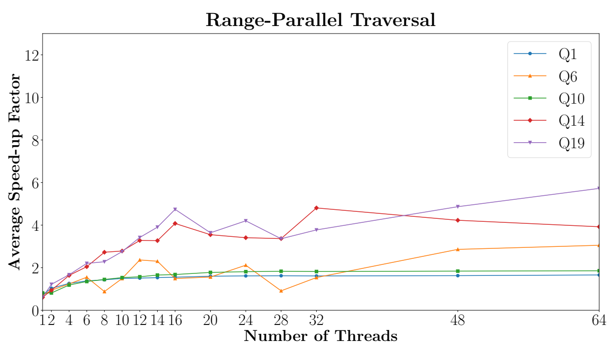
6.2.2 TPC-H Benchmark

This benchmark reflects the performance of our algorithm under common real world workloads. To simulate such workloads, we use the TPC-H benchmark queries on the `Lineitem` table with scaling factor 100. Our used queries are Q1, Q6, Q10, Q14 and Q19, whereas Q19 is only partially executed on the `Lineitem` table, thus we only benchmark this part. To obtain meaningful results, we use the arithmetic mean as our average query time and repeat each query 1,000 times.

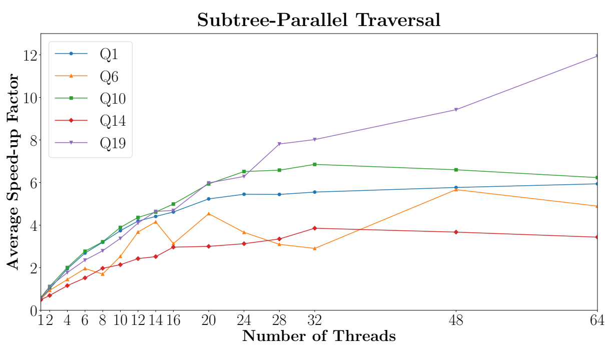
In our first Figure 6.3, we show the latency speed-up of the subtree- and range-parallel algorithms against the average serial query time dependent on the number of threads used for parallel traversal. The speed-up is determined as $S_n = \frac{T_{serial}}{T_{parallel}}$ where T_{serial} is the average serial query time and $T_{parallel}$ is the average query time for the parallel algorithm with n threads.

Range-Parallel Traversal

Both algorithm show a significant speed-up compared to the serial query algorithm. On Figure 6.3a, we show the scaling behaviour of the range-parallel query algorithm. Its single threaded performance is only around 70% of the serial algorithm, because it introduces some overhead to setup thread spawning and actually starting the thread. The range-parallel algorithm is able to outperform the original Elf clearly in every query from



(a) Average Range-Parallel Speed-Ups



(b) Average Subtree-Parallel Speed-Ups

Figure 6.3: TPC-H Queries Speed-Up for Subtree- and Range-Parallel Elf Compared to Serial Query Times

the break-even point of 4 threads per query. Afterwards, the speed-up factor is highly dependent from the workload of the query. For Q1, the algorithm is not able to accelerate the query times much more than factor 1.6, since this query has a very high selectivity of 92-97% [Council, 2014], but we expected this behaviour, since the Elf is designed to work particularly well for queries with low selectivity such as Q19. Q19 scales very well and reached a speed up of 5.7, and were not even able to determine an asymptotic behaviour, which hints to an upper limit for the speed-up.

However, for Q19 and the queries Q6 and Q14, there are some sweet spots at which the speed-up reaches a local maximum. These extrema come from load distribution and the associated frequency scaling of the processor cores[1]. This frequency scaling automatically reduces the clock significantly on more than four, eight and twelve cores used in our scenario, since our code is optimized with AVX2 instructions, which result in a different clock than vector-free code. With respect to the NUMA load balancing behaviour for adjacent ranges on same pages, this clock reductions match exactly to the performance drops at eight cores, 16 cores and more than 24 cores overall. Only Q6 shows some inconsistencies with this strategy, which can be traced back to the query type, which leads to a slightly different scheduling behaviour. Furthermore, Q14 shows a fall off in performance, when more threads than physical cores are used.

Subtree-Parallel Traversal

The behaviour of the subtree-parallel traversal algorithms, which we depict in Figure 6.3b, shows a bigger overall speed up of up to factor 11.9 for Q19 and behaves similar to the range-parallel traversal. Especially queries with low selectivity, such as Q1 profit from this algorithm. The lowest reached speed-up is factor 3.4 for Q14, which is still more than for most of the range-parallel traversal. Furthermore, the impact of the frequency scaling by the CPU is less dramatic. This can be deduced to the different NUMA balancing, since the memory ranges accessed are not adjacent any more which allows better distribution of the work between the NUMA nodes.

The major difference between the range-parallel traversal and the subtree-parallel traversal is the highly different behaviour under hyper-threading. While the range-parallel traversal

[1]https://en.wikichip.org/w/index.php?title=intel/xeon_gold/6130&oldid=91076

profited from more virtual cores, except Q14, this is most of the time different for the subtree-parallel traversal. The speed-up degrades with the utilization of virtual cores for Q6, Q10 and Q14. We can only observe a significant speed-up for Q19, while Q1 stagnates as in the range-parallel traversal.

Average Query Times

In this section, we directly compare our proposed algorithms against the current serial implementation for each of the `Lineitem` queries. Therefore we measure the query times of 100 queries on the `Lineitem` table with scaling factor 100.To obtain robust average times we use the mean of all query times and give the maximum outliers in both directions. We are comparing the serial algorithm to a singlethreaded version of our algorithms as well as our parallel algorithms with maximum number of threads available, since this gives the overall best speed-up. We show our findings in Figure 6.4.

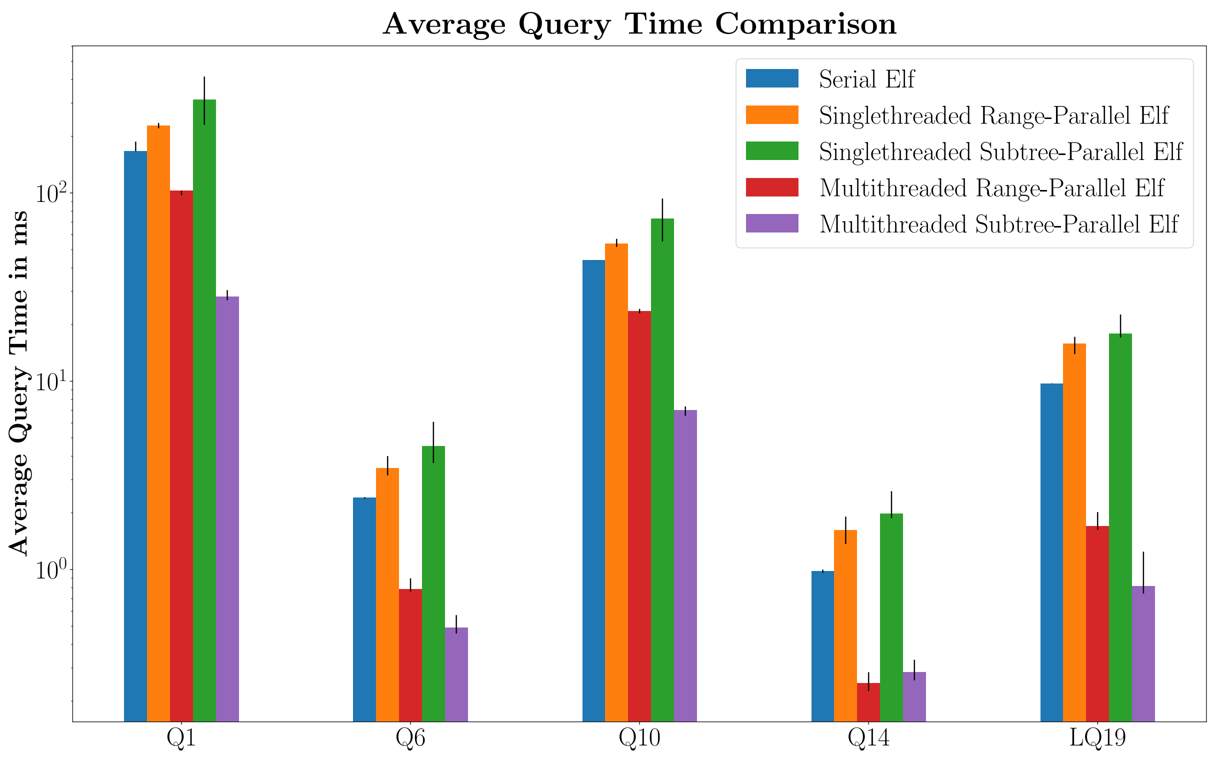


Figure 6.4: Average Query Times of all Traversals for the `Lineitem` Table With Scaling Factor 100

The overhead we introduce in our parallel algorithm compared to the serial algorithm is rather small. For the range-parallel traversal, we have an overhead of 20-40% for a single threaded traversal and for the subtree-parallel traversal we can observe a performance penalty of 50-60%. Despite the overhead introduced for single threaded parallel traversal, both algorithms scale well. The multithreaded range-parallel traversal with 64 threads is able to outperform the serial traversal by factor 1.6-5.7 and for the subtree-parallel traversal with 64 threads by factor 3.4-11.9.

6.3 Concurrent insertion

In this section, we evaluate the performance of our adapted insertion along with their proposed algorithms. Therefore, we changed the standard memory allocation algorithms of the `libstdc++` with the `tcmalloc`[2] implementation. These give us contention-free allocations, which make it possible to have a real lock-free implementation of our algorithms and thus meaningful evaluation results. To benchmark the performance of our implementation, we decided to do a mixed workload analysis. This means that we measure the insertion times as well as the exact match query times under different load situations from high query loads to high write workloads. As benchmark data we used once more the TPC-H `Lineitem` table with shuffled tuple data and repeated the benchmark 1 million times for robust results. For the desired workload, we started the amount of threads requested with one global atomic variable synchronising the total number of repetitions between all running threads. Each thread generates the read, respectively write workload utilizing a Bernoulli distribution with the wanted percentage for a write as probability. This random booleans give in each repetition of the experiment whether a read or insertion shall be executed. This random distribution over all threads guarantees that no wait situation occurs where all readers finished their work and writers were blocked until all reads finished.

We show the results of our evaluation with two box plots per workload, one for the read performance and one for write. The box plots have the mean as robust average and the boxes represent the quartiles. The whiskers show the maximum, respectively minimum time of all repetitions.

In the following we discuss the behaviour of the three parallelized algorithms under two representative workloads - one read-heavy with only 10% writes and one write-heavy with 75% writes workload - and afterwards compare them to each other. For further workloads we included our full results in the Appendices A, B and C

6.3.1 Locking Parallel Insertion

The first algorithm we examine is the locking parallel insertion algorithm with 10% write workload in Figure 6.5.

As we show in Figure 6.5a, the single thread performance of our locking read algorithm is even a bit faster than the serial read time. This is due to an iterative version of the traversal against the default recursive implementation of the serial Elf. With growing number of threads, the read times for the concurrent Elf degrades slightly with a decrease in performance of factor 2.2 with 64 threads, but the outliers do not change much. Only the quartile boxes shrink, which is a signal that the mutex overhead stabilizes.

The insertion time of the locking parallel Elf degrades similar to the read performance. While the average insertion time with one core is rarely slower than the serial insertion, the performance degrades quickly and shows an asymptotic behaviour with an extreme at 8 threads. Afterwards the read times converge, resulting in a performance loss of

[2]https://goog-perftools.sourceforge.net/doc/tcmalloc.html

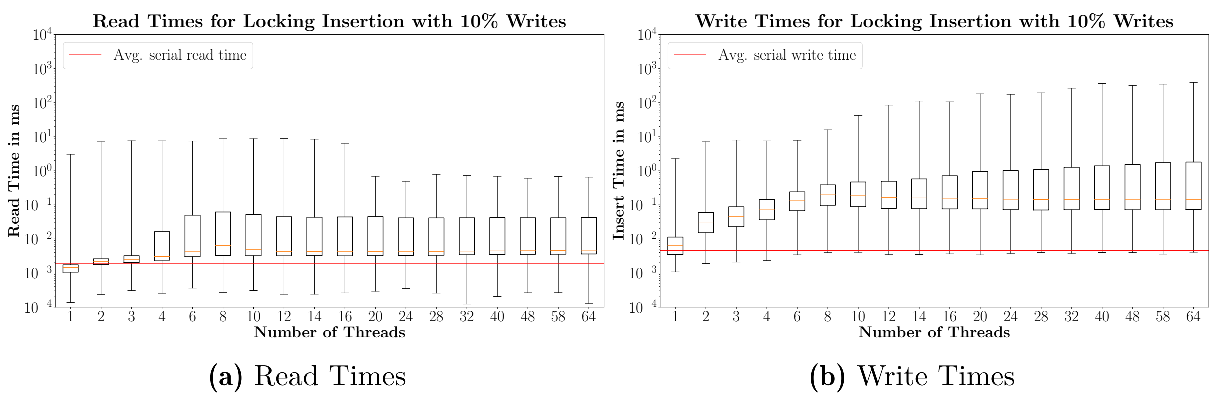


Figure 6.5: Locking Parallel Insertion Times With 10% Writes

in the magnitude of 31 times the insertion time of the serial insertion at 64 threads on average.

The overall behaviour does not changes much with a higher write, than read workload, which we show in Figure 6.6. For the read workloads, the performance curve is very similar, but the local maximum can be located at 3 threads instead of 8 with 10% writes. After this extreme, the query times slowly start converging back to the serial read times.

At 75% writes, for read times, which we illustrate in Figure 6.6a we observed the same behaviour. As before, insertion times grow quickly until they reach their maximum at 3 threads and then start to converge back to the average serial insertion time. However the algorithm behaves the same at 75% as before on 10%, just by the order of magnitude higher as the serial insertion time as well. This means that for higher write workloads, the writes become slower and thus the divergence is not as big as before.

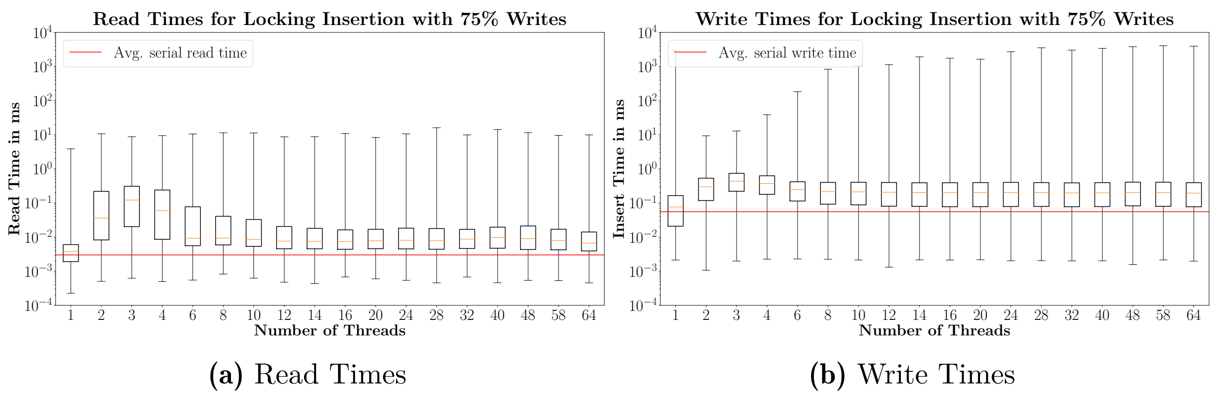


Figure 6.6: Locking Parallel Insertion Times With 75% Writes

This behaviour can not be explained by any hardware characteristics, but by the design of the rw-locking mutex we used. This kind of mutex generally seems to have a higher processing overhead because it allows two different locking states. Therefore a lot additional logic is needed, which can make a RW-mutex slower than an exclusive mutex[3]. This leads to the fact that the critical section must be reasonably large, such that the

[3]https://askldjd.com/2011/03/06/performance-comparison-on-reader-writer-locks/

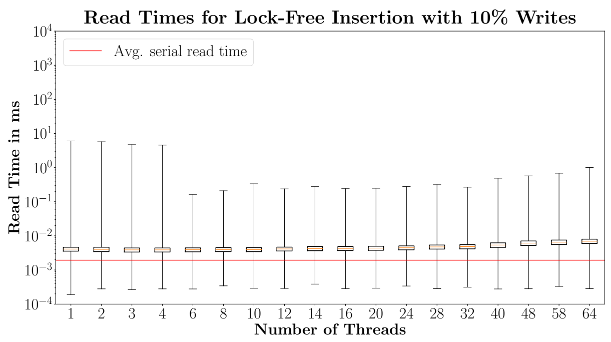
more costly mutex pays off and there must be more contention than under normal locks that the additional effort pays off. This is the reason why the performance degrades that quickly and begins to a amortize only at reasonably high concurrency, which leads to more contention. While our read critical section is rather small, the write section is much larger and additionally for high read loads, there are naturally more read locks. Since these read locks also block write accesses, this leads to a rather poor write performance for the 10% writes benchmark. In contrast, the read performance degradations is small due to the non-exclusive read access.

For high write workloads, this behaviour changes, since writes require exclusive access, which prevents concurrent reads. This is visible by the generally higher read times and the big performance hit with a small number of threads. However, these higher contention times due to rather large critical write sections, leads to a faster amortization of the mutex costs and the performance reaches its worst point at 3 threads, instead of 8 before on high read workloads.

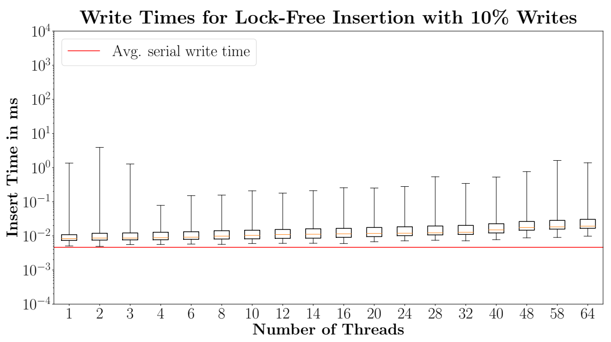
6.3.2 Lock-Free Parallel Insertion

Our second version is the lock-free version of the insert algorithm, which shows a slightly different behaviour than the locking version. In Figure 6.7, we show the read, and write times of the insertion algorithm benchmark with 10% writes. The first notable fact we can see is that the read, as well as the write times only show a slight degradation of their performance with the number of threads larger than the number of cores.

Furthermore it is notable that with Figure 6.7a we generally reach very reliable query times with at least 50% of the read operation within $\pm 1\mu s$. For the insert operation, we show the same behaviour in Figure 6.7b, where 50% of the insert operations has an insertion time window within $\pm 20\mu s$ from average with a tendency to the lower times. Even though, there are some outliers, they are generally lower than the one we observed for the locking implementation and also rather unbiased for the read operations. For writes, the outliers towards lower times are somewhat nearer to the median than towards larger insertion times, but still not as large as for the locking insertion.



(a) Read Times

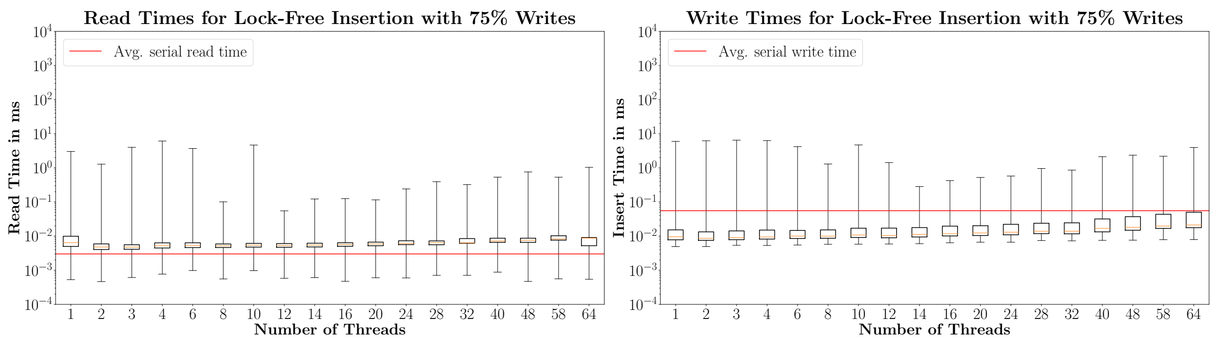


(b) Write Times

Figure 6.7: Lock-Free Parallel Insertion Times 10% Writes

In Figure 6.7 we showed the steady behaviour of the read as well as the insert times. We also observe the same behaviour in our write heavy benchmark with 75% writes, whose results we show in Figure 6.8. As expected, the read performance under high write workload is marginally lower than under low write workload. However the performance kept nearly identical with respect to the serial read time, as we depict in Figure 6.8a. The higher read times can thus be explained by the overall bigger Elf to traverse, since there are more data in the Elf on 75% writes than for 10% writes.

Another interesting observation from Figure 6.8b in comparison to Figure 6.7b is that the serial write performance degraded with the amount of inserted data. This is not the case for our lock-free parallel insertion. Its insertion time kept nearly constant under both workloads, which provides a speed-up of up to factor 5.8 and even more for workloads with more than 75% writes.



(a) Read Times **(b)** Write Times

Figure 6.8: Lock-Free Parallel Insertion Times With 75% Writes

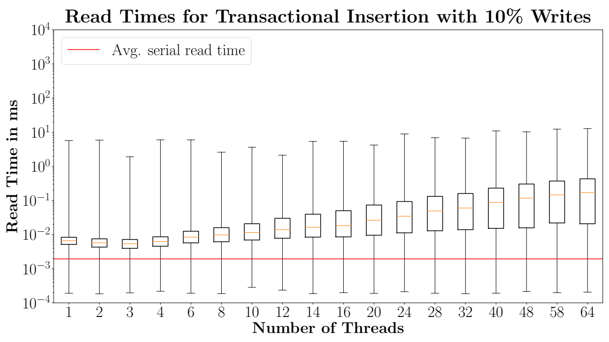
The steady behaviour we showed is typical for lock-free algorithms since no waiting situation occurs. That the performance degrades at all with more threads than cores is the consequence of having only one RMW-unit per core [Intel, 2019b]. This unit is needed for atomic instructions and thus generates a latency bottleneck when the instructions can not be pipelined. On the other hand, this responsiveness comes with the price of added overhead due to a less efficient caching behaviour. The caching is worse since serial access to continuous memory is the most cache efficient access strategy, but since we replaced the vectors by skip lists we trade-off some cache performance for more steady query times and parallelism. However under workloads with higher write than read percentage, the lock-free algorithm on average provides a speed-up of at least factor 2.6 for insertions compared to the serial version.

6.3.3 Transactional Parallel Insertion

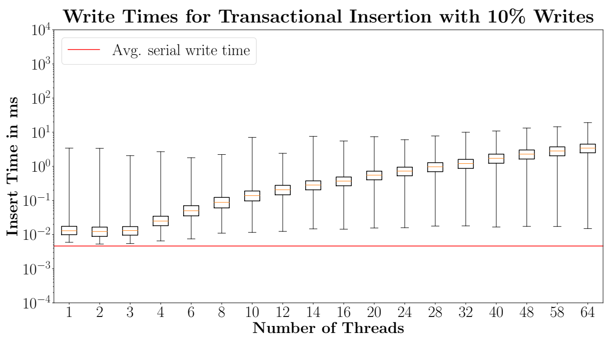
The third variant we introduced is the transactional parallel insertion. As for the previous two algorithms, in the following we will show our benchmark results for the read heavy insertion with a write ratio of 10% and afterwards a write heavy benchmark with 75% insertion.

For the 10% writes benchmark, whose visualisation of query and write times is shown in Figure 6.9, we can see a similar behaviour between the read and insertion times. The insertion times, we show in 6.9a, are approximately 3.4 times higher in the single threaded traversal, than the default query algorithm. With growing concurrency, this behaviour does not change and eventually the query time as well as the quartiles of the query times grow significantly with the number of threads concurrently accessing the Elf. However, with low concurrency, the query times shrink marginally before growing and the outliers keep steady with growing concurrency.

In Figure 6.9b we show the insertion times, where we can observe a similar behaviour. Similar to the read times, the write times increase with growing concurrency, but the outliers remain in the same order of magnitude. Only the average and quartiles are increasing. Furthermore, the quartiles are significantly smaller than for the read times. The reason behind is probably that the writes are largely independent from the reads and vice versa, since the reads traverse the Elf without any transactions. However under high concurrency during insertions, the number of transaction conflicts increases, which increases the transaction latency and thus the overall insertion time.



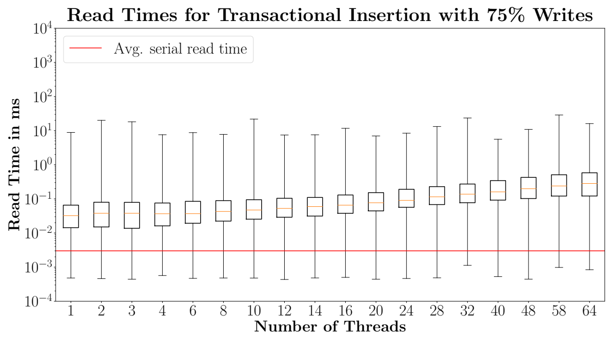
(a) Read Times



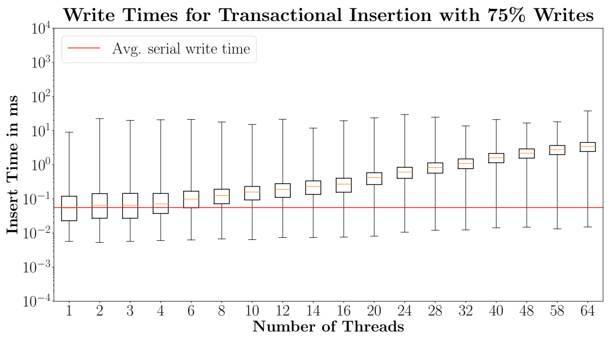
(b) Write Times

Figure 6.9: Transactional Parallel Insertion Times With 10% Writes

This assumption is supported, when we look at the write intensive benchmark, whose results are shown in Figure 6.10. The higher write load reduced the quartile derivation for read workloads shown in 6.10a because there are overall less reads that can conflict with each other. Furthermore, the write time quartiles we depict in 6.10b are larger than for the 10% write workload. Even though they slowly converge to the average with growing number of concurrent operations. This convergence happens, because the average insertion time increases with more concurrency and thus, the conflicting operations do no longer have that much influence on the overall insertion time.



(a) Read Times



(b) Write Times

Figure 6.10: Transactional Parallel Insertion Times With 75% Writes

The overall performance of the transactional insertion approach is worse than expected. The algorithm does not particularly scale well and is slower than the serial algorithm under medium to high concurrency. However, in this approach, the query time window was smaller than the one for locking insertions and completely independent from the concurrency level. This makes the algorithm suited for real time requirements where reliable query times are needed. Furthermore, the performance probably can be improved by allowing more hardware transaction retries and maybe even migrating from a transactional library to intrinsics to manage the transactions. Also a larger amount of smaller transactions could be considered, since this mostly leads to less dramatic transaction conflicts.

6.3.4 Elf Storage Consumption

In the previous sections, we discussed the behaviour of the different insertion algorithms proposed. Another important factor beside latency is the storage consumption of the Elf. Since the Elf is a data structure utilizing prefix sharing it features a light compression, which should not be destroyed by our adaptions. In the following, we determine the storage consumption of our adapted Elf structures and compare them to the read optimized Elf as well as the conceptual Elf and the raw data.

For our evaluation we compare the sizes of the data structures with 1 million randomly chosen tuples inserted form the `Lineitem` table with scaling factor one. Since we choose our tuples uniform at random, this does not change the overall distribution of the data in the `Lineitem` table. We determined the storage consumption empirically with the tool `heaptrack`[4], which replaces the standard allocation mechanisms and logs all allocations of the program started with `heaptrack`. Since that way also memory which is not used by the Elf is tracked, we filtered the allocations for its origin in the corresponding build and insert algorithms. We show our results in Figure 6.11.

[4]https://github.com/KDE/heaptrack

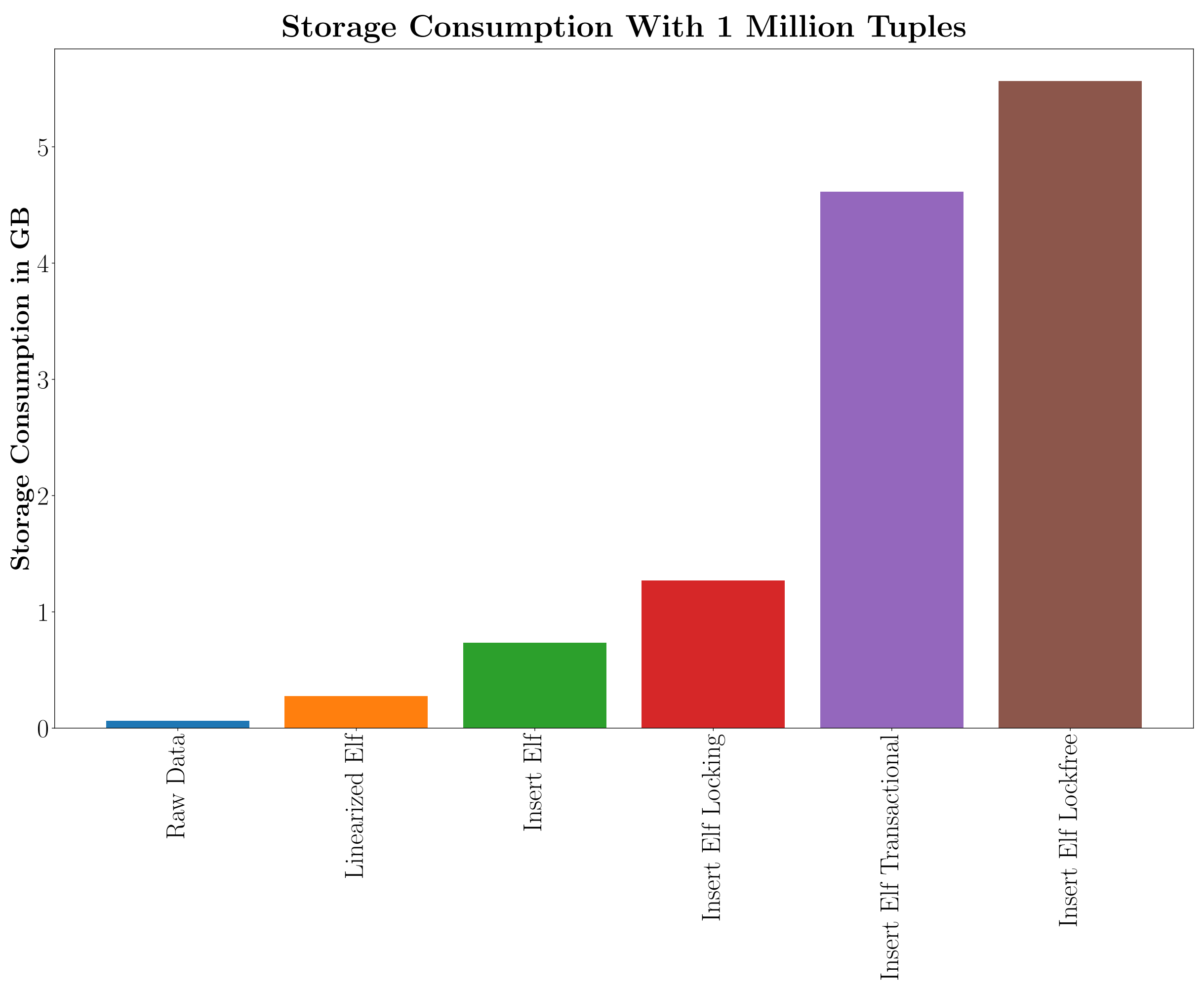


Figure 6.11: Storage Consumption of the Different Elf Structures for 1 Million Tuples of the `Lineitem` Table

The first result of this experiment is, that the Linearized Elf still has the lowest storage consumption of all Elf variants, but does not show any compression. While this observation at the first sight seems to contradict with the observations of Broneske et al., this is not the case. Our dataset only consists of 1 million tuples, while the `Lineitem` dataset used in the work of Broneske et al. contained more than 600 million tuples. At this size, there are many more prefixes than in a random subset of just 1 million tuples and thus more prefix redundancy elimination. The Linearized Elf just does not have enough data inserted to eliminate enough prefix redundancies to show a compression effect.

Locking Elf

As we can see, all our adaptions in the structure of the Elf led to a drastic increase in the storage consumption. The locking Insert Elf implementation has the smallest increase in size of our three proposed structures with 58% larger than the normal insert Elf implementation. This is due to the added overhead of a RW-mutex and the additional vector, which holds the pointers instead of mixing values and pointer in one vector for

DimensionLists. For each *DimensionLists*, this doubles the node initial size compared to the serial Insert Elf nodes.

Transactional Elf

The transactional Elf has a far bigger overhead, which can be explained by the use of skip lists instead of vectors. Since skip lists are a data structure with linked nodes, its storage overhead is far bigger, since every value in a *DimensionList* does not only need a pointer to the next *DimensionList*, but also a pointer to the next value in the list. Additionally, each skip list node contains an array of fast lanes, which take up additional memory of eleven more pointers. However, against the lock-free Elf, the leaf nodes of the transactional Elf has lightweight leaf nodes, which consist just of a continuous memory region for the *TID*s and the size of the memory region. This leads to an overall space overhead of around factor 6.3 compared to the serial Insert Elf.

Lock-Free Elf

The largest space overhead of all three adapted versions has the lock-free Elf, its is approximately by factor 7.6 larger than the space utilized by the serial Insert Elf. This is in large parts the responsibility of the skip list as for the transactional Elf. Furthermore there are several other factors, which lead to an even higher space utilization than the transactional approach. One factor is the `concurrent_vector` used for the leaf nodes, which is designed as an array of vectors to ensure non-blocking insertions. However, this design leads to a higher space consumption and even more memory fragmentation. Furthermore, the atomic pointers to the skip lists within the inner lock-free elf nodes are aligned through the architecture specific destructive cache inference size, which is in our case 64 byte. This ensures that concurrent access to the same atomic does not lead to cache conflicts due to false sharing

The biggest downside of our approaches is the replacement of the enormous space efficient vectors against the skip lists. This also leads to fragmentation and therefore more page and cache misses. A trade-off which effects becomes worse with growing size of the Elf and reduces the performance drastically. Unfortunately we are not able to empirically evaluate the impact of memory fragmentation on the performance in the limited time scope of this thesis. However, the step away from continuous data structures was needed to reach optimal concurrency, since to the best of our knowledge no efficient non-blocking data structures with continuous storage and fast random insertion times exists.

7 Conclusion

In the previous chapters we presented our parallelization approaches and empirically evaluated them. In this chapter we will wrap-up and summarise the contributions of this thesis as well as the performance of our suggested approaches. Afterwards we outline some research questions that arise in the context of this thesis.

7.1 Concept & Implementation

In this work, we brought task parallelism to the Elf in two aspects. At first, we proposed three thread parallel approaches to parallelize the query algorithms. The range- and subtree-parallel traversal as a rather coarse-grained approach on subtree- and the node-parallel traversal as a fine grained task parallel approach on node level. For the subtree- and range-parallel traversal, we also adapted different result-merging variants to further optimize the query times and find the best merging technique for our approaches.

Furthermore, we adapted the insertion algorithm for concurrent insertions. Therefore we choose three different approaches. The first one with minimal adaptions to the Elf node structure to enable data-race-free concurrency with a fine grained locking scheme. The second one with a major adaption to the Elf, by replacing the data array of the nodes with a lock-free skip list to reach non-blocking concurrency for better scalability at high contention with CAS-instructions. At last a transactional approach with similar adaptions to the Elf structure as for the lock-free algorithm but instead of using CAS-instructions we rely on a hybrid version using both hardware and software memory transactions. Since the skip lists we utilized had no range query, we also implemented a range query algorithm based on their exact match algorithm.

Subsequentially, we also empirically evaluated our proposed query and insertion algorithms to determine their performance as well as potential drawbacks in our approaches.

7.2 Evaluation

For our evaluation, we started with a microbenchmark of the merge algorithm, with the result that the serial merge algorithm outreaches our parallel variants by several orders of magnitude until the input sizes become high enough. However, since we await a large amount of data to merge, we decided to use the merge algorithm with a parallel merge strategy but only serial prefix summing. We did this, since the number of inputs to merge is not as big, to expect a parallel prefix sum algorithm to pay-off.

Parallel Queries

With the hybrid merge algorithm, we compared the node parallel, as well as the range- and subtree-parallel algorithms in a benchmark of range queries. The results showed that the performance of our node parallel query algorithm did not show the expected behaviour. Instead of minimizing the query times with a more fine grained parallelism, the query times grew with more threads used in the traversal. Our explanation for this behaviour is that the communication overhead introduced by the node traversal does not pay-off for the small *DimensionList* sizes found in the Elf. However our other proposed algorithm behaved as expected and was able to speed up the range queries compared to the serial counterpart.

To understand the behaviour of the two well behaving algorithms under real world workloads we evaluated their speed-up utilizing the TPC-H benchmark queries on the `Lineitem` table with scaling factor 100. The result of this benchmark is that both algorithms scale well with the growing number of threads used in the query. Furthermore, the subtree-parallel query is able to outperform the range query for every workload by more than factor 2 and the serial algorithm for factor 3.4 up to 11.9 depending on the query characteristics. Even though, the performance of both algorithms is not only dependent on the number of threads working in parallel, but also the frequency of the cores, which is responsible for the sweet spots we found for both algorithm.

Concurrent Insertion

In the second part of our evaluation, we tested the behaviour of our insertion algorithms regarding performance and space utilization. Therefore we measured the read, as well as insertion times under different read-write workloads and different concurrency levels. We observed that all three algorithms were not able to outperform the serial algorithm regarding single read performance. Furthermore only our lock-free algorithm was able to outperform the serial write performance under high write workloads. The locking algorithm performed better under high write workloads and concurrency than high read workloads. The transactional insertion is not able to keep up with the performance of the other approaches probably due to the poor utilization of hardware transactions by our used library. Anyway, the transactional approach showed a similar consistent behaviour as the lock-free variant. Overall the lock-free and the locking algorithms were able to speed-up the insertions for the locking algorithm by at least factor 2 and for the lock-free by at least factor 2.6 compared to a serial execution. However, the benchmarks cannot be seen as a completely general analysis of the algorithms behaviour, since for time reasons we are only able to determine the performance for a uniform random distribution of the `Lineitem` table with scaling factor 100, but not any other distribution. Yet it is likely for the algorithms to show a completely different behaviour for different distributions such as the insertion of a sorted sequence. Furthermore we have no data how the algorithms behave for insertions into a pre-build Elf with a larger number of tuples already inserted. Furthermore, we only collected data for single execution times, which only gives us the possibility to reason about the behaviour of single queries but not the overall speed-up achievable by running multiple operations concurrently.

To complete our evaluation we also measured the space requirements of our adapted Elf structures. The results proved that the most lightweight adaptions were made on the locking Elf with only 58% higher than the serial Insert Elf. The replacement of a vector with the skip list for the lock-free and transactional algorithms lead to a significantly higher memory consumption by up to factor 7.6 compared to the original Insert Elf implementation. Furthermore, it is likely that larger Elfs with skip lists are prone to high memory fragmentation and thus reduced performance due to page and cache misses.

7.3 Future Work

Since the build time of the read-optimized Elf takes a major amount of time, a parallelisation of the build and linearisation of the read-optimized Elf could bring huge speed-up. Since we have only a limited scope of time for this theses, we left this parallelization part out of our consideration. Not at last, since the build time amortizes over the time. However a parallelisation could be beneficial anyway, especially for large tables or dynamic data.

Parallel Queries

The parallel node level traversal approach utilizes a depth first traversal, which could be a reason for the bad performance. With a restructuring of the Elf nodes by adding a length field at the start of each node replacing the MSB markers, a breadth first traversal (BFS) will be possible. Changing the node parallel traversal to a BFS could greatly improve its performance.

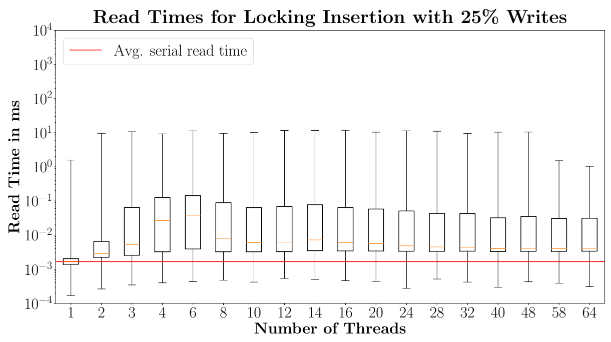
Furthermore, the subtree- and range-parallel traversal does not show asymptotic behaviour for some queries, which is a sign that these algorithms can be further parallelized on GPU and even adapted for heterogeneous computing to improve the performance even more.

Concurrent Insertion

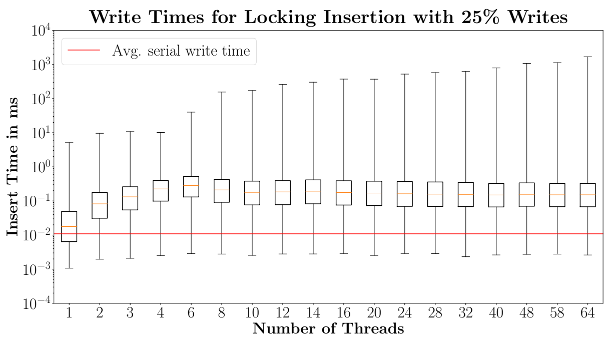
In our work we introduced concurrent insertions, which we proved to be not linearisable. This is a big disadvantage, since this criteria is crucial for the use in most database systems. However, there exist some approaches to make range queries linearisable that may be adaptable to our approach [Chatterjee, 2017], [Chatterjee et al., 2018], [Brown & Avni, 2012]. This would bring the concurrent Elf a step further to be used in modern database systems.

Appendices

A Locking Insert Benchmarks

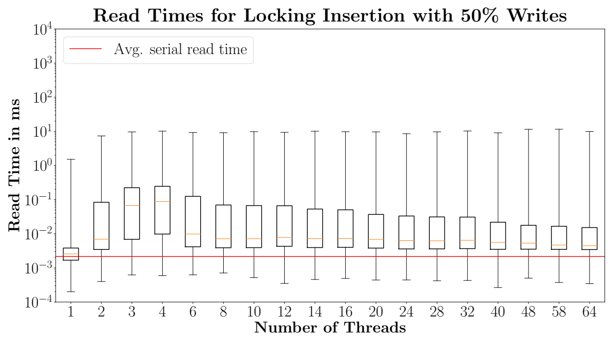


(a) Read Times

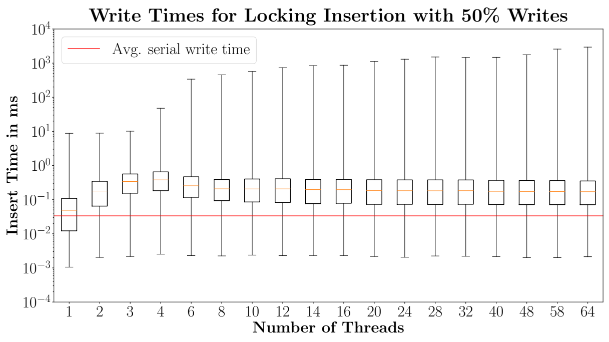


(b) Write Times

Figure A.1: Locking Parallel Insertion Times With 25% Writes

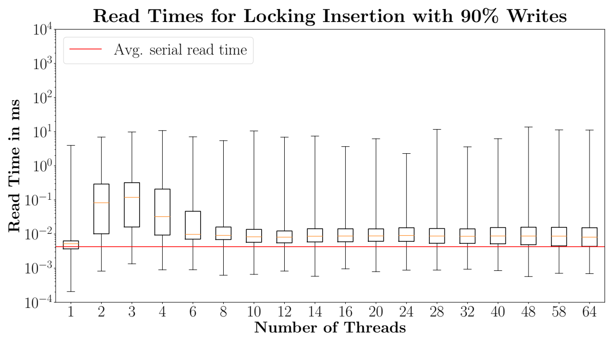


(a) Read Times

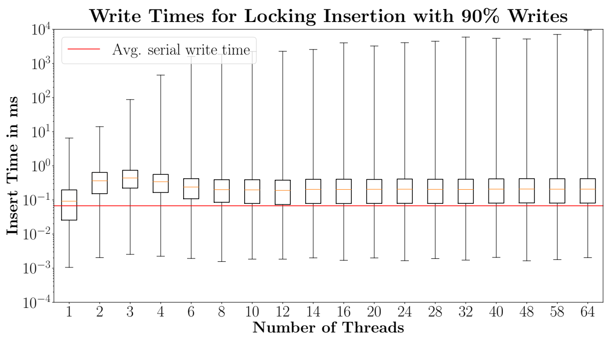


(b) Write Times

Figure A.2: Locking Parallel Insertion Times With 50% Writes



(a) Read Times



(b) Write Times

Figure A.3: Locking Parallel Insertion Times With 90% Writes

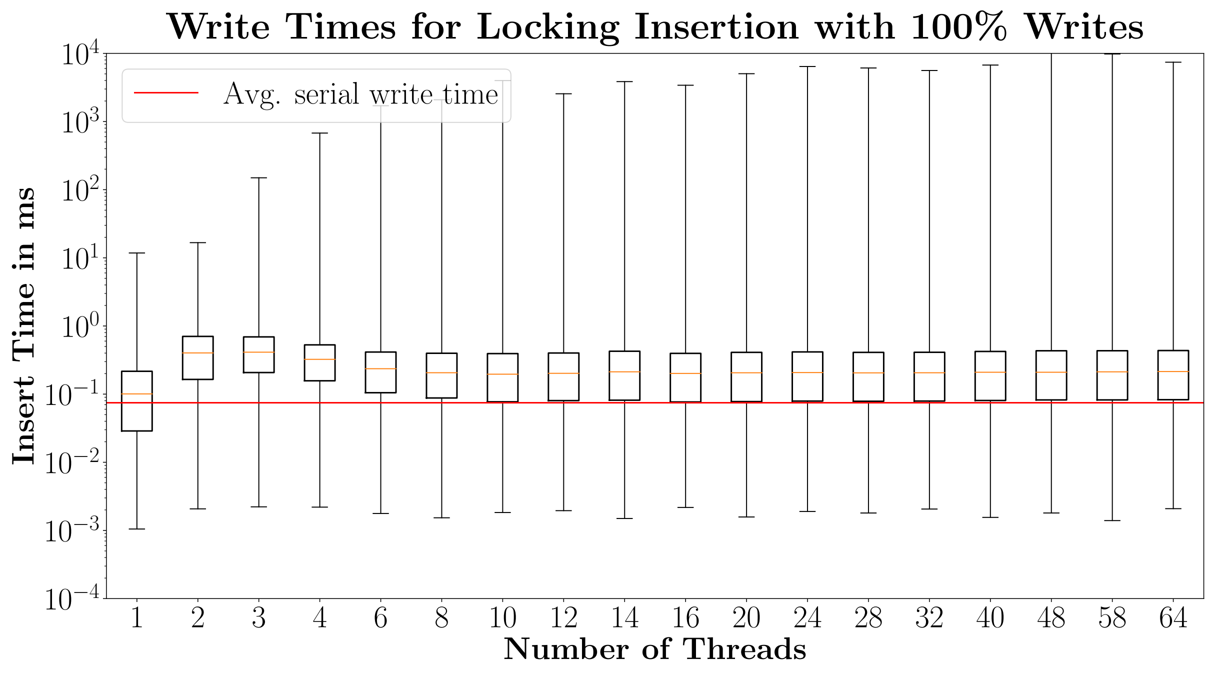
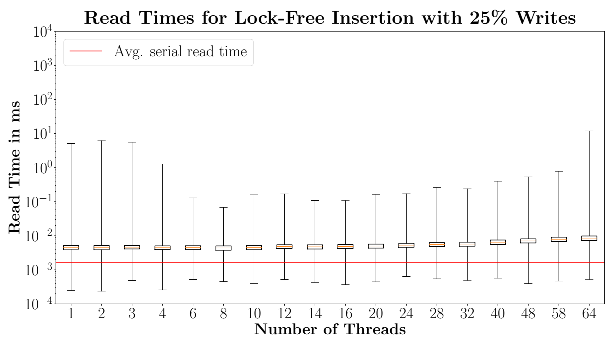
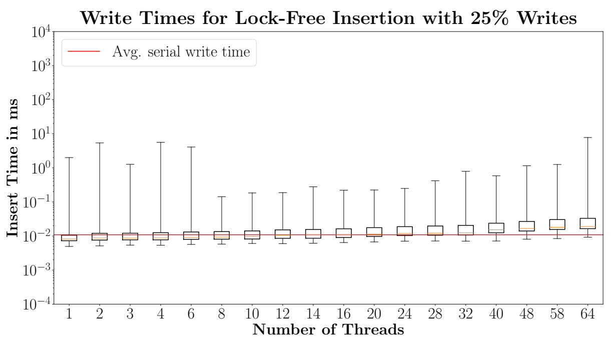


Figure A.4: Locking Parallel Insertion Times Write Only

B Lock-Free Insert Benchmarks

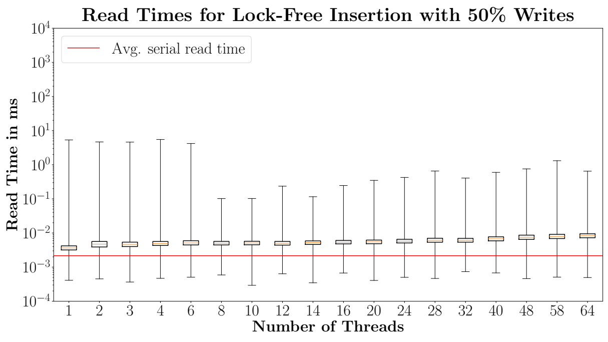


(a) Read Times

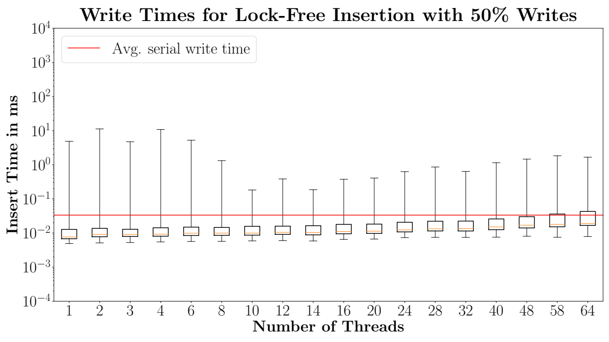


(b) Write Times

Figure B.1: Lock-Free Parallel Insertion Times With 25% Writes

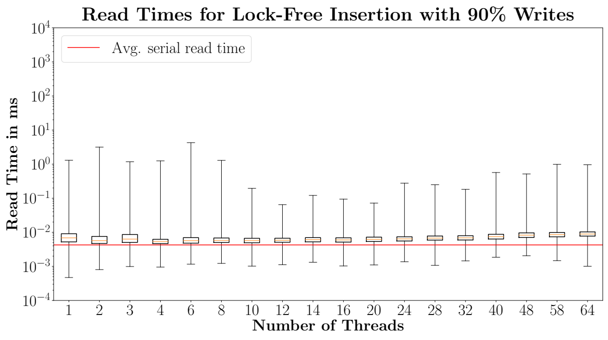


(a) Read Times

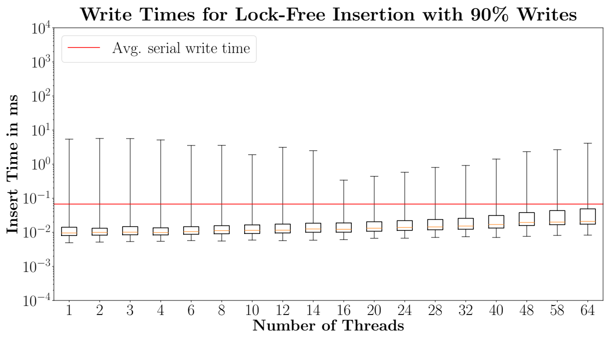


(b) Write Times

Figure B.2: Lock-Free Parallel Insertion Times With 50% Writes



(a) Read Times



(b) Write Times

Figure B.3: Lock-Free Parallel Insertion Times With 90% Writes

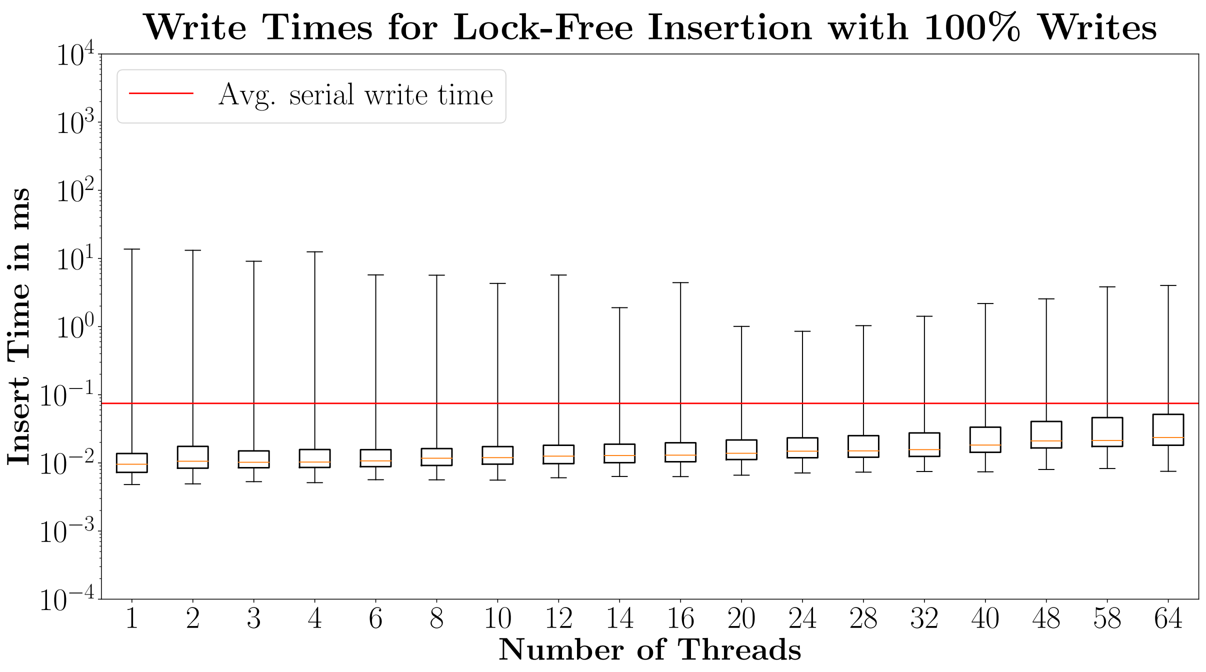
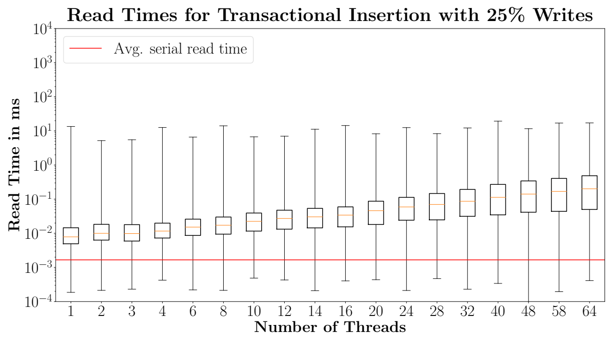
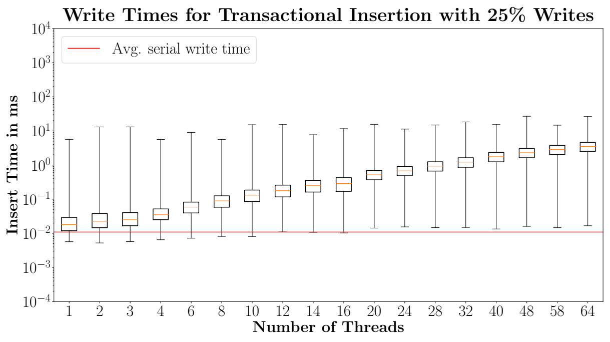


Figure B.4: Lock-Free Parallel Insertion Times Write Only

C Transactional Insert Benchmarks

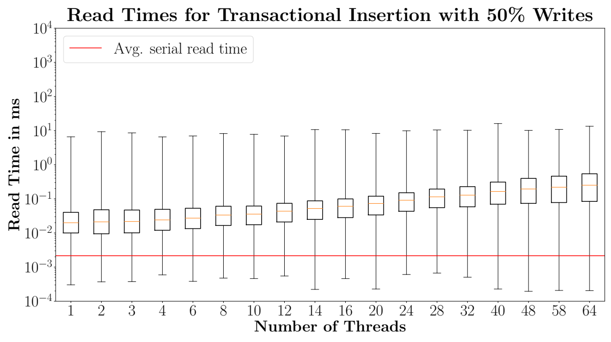


(a) Read Times

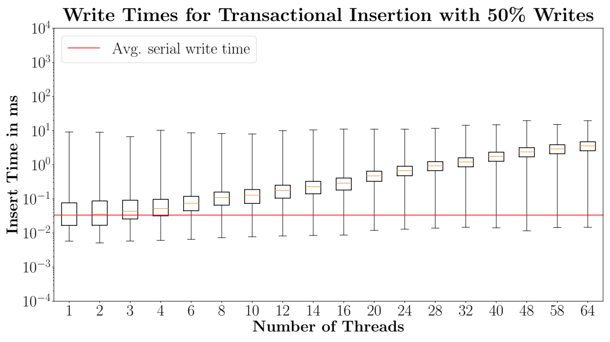


(b) Write Times

Figure C.1: Transactional Parallel Insertion Times With 25% Writes

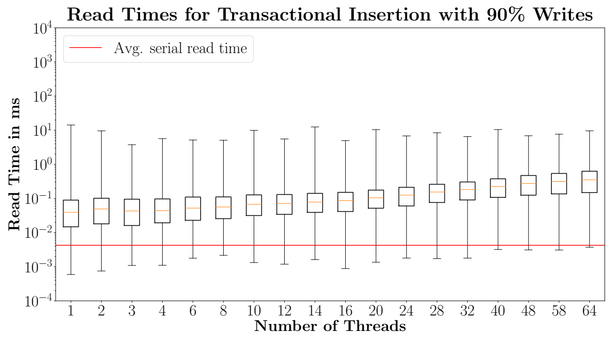


(a) Read Times

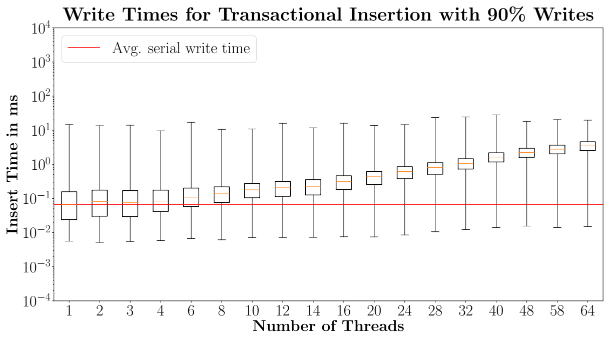


(b) Write Times

Figure C.2: Transactional Parallel Insertion Times With 50% Writes



(a) Read Times



(b) Write Times

Figure C.3: Transactional Parallel Insertion Times With 90% Writes

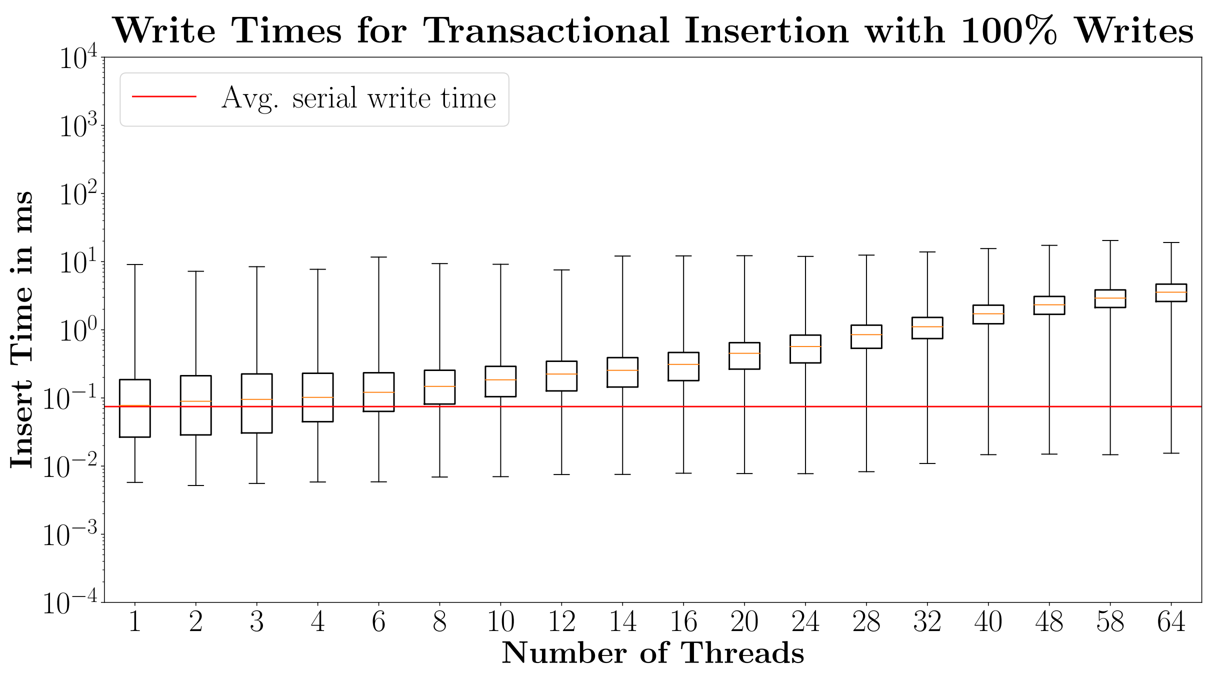


Figure C.4: Transactional Parallel Insertion Times Write Only

Bibliography

Bayer, R., & Schkolnick, M. (1977). Concurrency of Operations on B-trees. *Acta Informatica*, *9*(1), 1–21. doi:10.1007/BF00263762

Boncz, P. A., Kersten, M. L., & Manegold, S. (2008). Breaking the Memory Wall in MonetDB. *Communications of the ACM*, *51*(12), 77–85. doi:10.1145/1409360.1409380

Braginsky, A., & Petrank, E. (2012). A Lock-free B+Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (pp. 58–67). doi:10.1145/2312005.2312016

Broneske, D., Köppen, V., G., & Schäler, M. (2017). Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (pp. 647–658). doi:10.1109/ICDE.2017.118

Brown, T., & Avni, H. (2012). Range Queries in Non-blocking k-ary Search Trees. In *Principles of Distributed Systems* (pp. 31–45). doi:10.1007/978-3-642-35476-2_3

Chatterjee, B. (2017). Lock-free Linearizable 1-Dimensional Range Queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking* (9:1–9:10). doi:10.1145/3007748.3007771

Chatterjee, B., Walulya, I., & Tsigas, P. (2018). Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree. In *Proceedings of the 19th International Conference on Distributed Computing and Networking* (11:1–11:10). doi:10.1145/3154273.3154307

Council, T. P. P. (2014). *TPC benchmark H (decision support)*. Retrieved from http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

Dechev, D., Pirkelbauer, P., & Stroustrup, B. (2006). Lock-free Dynamically Resiable Arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems* (pp. 142–156). doi:10.1007/11945529_11

Dechev, D., Pirkelbauer, P., & Stroustrup, B. (2010). Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (pp. 185–192). doi:10.1109/ISORC.2010.10

Detlefs, D. L., Martin, P. A., Moir, M., & Steele, G. L., Jr. (2001). Lock-free Reference Counting. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (pp. 190–199). doi:10.1145/383962.384016

Dijkstra, E. W. (1962). Over de sequentialiteit van procesbeschrijvingen. *Transactions by Martien van der Burgt and Heather Lawrence*. Retrieved from http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF

Felber, P., Fetzer, C., & Riegel, T. (2008). Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 237–246). doi:10.1145/1345206.1345241

Feldman, S., Valera-Leon, C., & Dechev, D. (2016). An Efficient Wait-Free Vector. *IEEE Transactions on Parallel and Distributed Systems*, *27*(3). doi:10.1109/TPDS.2015.2417887

Ferguson, C., & Korf, R. E. (1988). Distributed Tree Search and Its Application to Alpha-beta Pruning. In *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence* (pp. 128–132). Saint Paul, Minnesota: AAAI Press.

Flynn, M. (2011). Flynn's Taxonomy. In *Encyclopedia of Parallel Computing* (pp. 689–697). Springer US.

Fraser, K. (2004). *Practical lock-freedom*. University of Cambridge, Computer Laboratory. Retrieved from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

Friedman, D. P., & Wise, D. S. (1978). Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, (4), 289–296.

Gaede, V., & Günther, O. (1998). Multidimensional Access Methods. *ACM Computing Surveys*, *30*(2), 170–231. doi:10.1145/280277.280279

Harris, T. L. (2001). A Pragmatic Implementation of Non-blocking Linked-lists. In *Distributed Computing* (pp. 300–314). doi:10.1007/3-540-45414-4_21

Herlihy, M. [M.], Luchangco, V., & Moir, M. (2003). Obstruction-free synchronization: double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems* (pp. 522–529). doi:10.1109/ICDCS.2003.1203503

Herlihy, M. [Maurice]. (1993). A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, *15*(5), 745–770. doi:10.1145/161468.161469

Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, *12*(3), 463–492. doi:10.1145/78969.78972

Herlihy, M. [Maurice], & Moss, J. E. B. (1993). Transactional Memory: Architectural Support for Lock-free Data Structures. *ACM SIGARCH Computer Architecture News*, *21*(2), 289–300. doi:10.1145/173682.165164

Hillis, W. D., & Steele, G. L., Jr. (1986). Data Parallel Algorithms. *Communications of the ACM*, *29*(12), 1170–1183. doi:10.1145/7902.7903

Horn, D. (2005). Stream Reduction Operations for GPGPU Applications. In M. Pharr & R. Fernando (Eds.), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)* (Chap. 36, pp. 573–589). Addison-Wesley Professional.

IBM. (1983). *IBM System/370 Extended Architecture — Principles of Operation*. Retrieved from http://bitsavers.trailing-edge.com/pdf/ibm/370/princOps/SA22-7085-0_370-XA_Principles_of_Operation_Mar83.pdf

Intel, C. (2019a). *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z*. Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Intel, C. (2019b). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1:Basic Architecture*. Retrieved from https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf

Kardaras, M., Siakavaras, D., Nikas, K., Goumas, G., & Koziris, N. (2018). Fast Concurrent Skip Lists with HTM. In *11th International Symposium on High-Level Parallel Programming and Application*. Retrieved from http://www.cslab.ece.ntua.gr/~knikas/files/papers/hlpp2018.pdf

Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., … Dubey, P. (2010). FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (pp. 339–350). doi:10.1145/1807167.1807206

Ladner, R. E., & Fischer, M. J. (1980). Parallel Prefix Computation. *Journal of the ACM*, *27*(4), 831–838. doi:10.1145/322217.322232

Lev, Y., & Maessen, J.-W. (2008). Split Hardware Transactions: True Nesting of Transactions Using Best-effort Hardware Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 197–206). doi:10.1145/1345206.1345236

Levandoski, J. J., Lomet, D. B., & Sengupta, S. (2013). The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE)* (pp. 302–313). doi:10.1109/ICDE.2013.6544834

Mckenney, P., Kleen, A., Krieger, O., Russell, R., Sarma, D., & Soni, M. (2001). Read-Copy Update. In *AUUG Conference Proceedings 2001*.

Michael, M. M. (2004). Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, *15*(6), 491–504. doi:10.1109/TPDS.2004.8

Nguyen, H. (2007). Gpu gems 3. (First, Chap. 39). Addison-Wesley Professional.

Plattner, H. (2009). A Common Database Approach for OLTP and OLAP Using an In-memory Column Database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (pp. 1–2). doi:10.1145/1559845.1559846

Pugh, W. (1990). Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 668–676. doi:10.1145/78973.78977

Rao, J., & Ross, K. A. (2000). Making B+- Trees Cache Conscious in Main Memory. *SIGMOD Record*, *29*(2), 475–486. doi:10.1145/335191.335449

Saake, G., Sattler, K., & Heuer, A. (2011). Datenbanken - Implementierungstechniken, 3. Auflage. (Chap. 9, p. 441). MITP.

Shavit, N., & Touitou, D. (1997). Software transactional memory. *Distributed Computing*, *10*(2), 99–116. doi:10.1007/s004460050028

Spear, M. F., Dalessandro, L., Marathe, V. J., & Scott, M. L. (2009). A Comprehensive Strategy for Contention Management in Software Transactional Memory. *ACM SIGPLAN Notices*, *44*(4), 141–150. doi:10.1145/1594835.1504199

Sprenger, S., Schäfer, P., & Leser, U. (2018). Multidimensional Range Queries on Modern Hardware. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management* (4:1–4:12). doi:10.1145/3221269.3223031

Sprenger, S., Schäfer, P., & Leser, U. (2019). BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads. In *22nd International Conference on Extending Database Technology* (pp. 169–180).

Sprenger, S., Zeuch, S., & Leser, U. (2017). Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs. In *Data Management on New Hardware* (pp. 1–17). Springer International Publishing.

Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., & Schaffner, J. (2009). SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proceedings of the VLDB Endowment*, *2*(1), 385–394. doi:10.14778/1687627.1687671

Zäschke, T., Zimmerli, C., & Norrie, M. C. (2014). The PH-tree: A Space-efficient Storage Structure and Multi-dimensional Index. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (pp. 397–408). doi:10.1145/2588555.2588564

Statement of Authorship / Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und ausschließlich unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.
Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder einer anderen Prüfungsbehörde vorgelegt oder noch anderweitig veröffentlicht.

Unterschrift	Datum