SIMD Acceleration for Main-Memory Index Structures – A Survey

Marten Wallewein-Eising, David Broneske, and Gunter Saake

University of Magdeburg,
Magdeburg, Germany
firstname.lastname@ovgu.de

Abstract. Index structures designed for disk-based database systems do not fulfill the requirements for modern database systems. To improve the performance of these index structures, different approaches are presented by several authors, including horizontal vectorization with SIMD and efficient cache-line usage.
In this work, we compare the adapted index structures Seg-Tree/Trie, FAST, VAST, and ART and evaluate the usage of SIMD within these. We extract important criteria of these adaptions and weight them according to their impact on the performance. As a result, we infer adaptions that are promising for our own index structure Elf.

1 Introduction

After decades of creating and improving index structures for disk-based database systems, nowadays even large databases fit into main memory. Since index structures like the B^+-tree or the radix tree have an important part in database systems to realize scan or range-based search operations, these index structures experienced many adaptions to fulfill the needs of modern database systems. Instead of overcoming the bottleneck of IO-operations from disk to RAM, the target of modern index structures is to improve the usage of CPU cache and processor architectures. Several index structures have already shown that the bottleneck from RAM to CPU can be overcome using Single Instruction Multiple Data (SIMD) [1] operations. These index structures include: the k-ary Search Tree (Seg-Tree) [2], Adapted Radix Tree (ART) [3], Fast Architecture Sensitive Tree (FAST) [4], and Vector-Advanced and Compressed Structure Tree (VAST) [5]. All approaches use SIMD only for key comparison within tree traversal and try to decrease the key size to fit more keys into one SIMD register. Therefore FAST and Seg-Tree only provide implementations for search algorithms. Despite several common features, all these structures have their special and unique optimizations. In order to allow other approaches to benefit from these highly-involved optimizations, we extract their optimizations in this work while exposing their specific benefit for other structures.

In this work we make the following contributions:

- We compare different optimizations of index structures to fulfill requirements of modern database systems

- We highlight the usage of SIMD and the cache-line adaptions in all approaches
- We state the performance impact of optimizations on the index structures
- We discuss the usefulness of optimizations for our own index structure Elf [6].

We organized the rest of the paper as follows. In Section 2, we give the preliminaries for SIMD in general and for the use in index structures. In Section 3, we present the different approaches of optimized index structures and compare them in Section 4. In Section 5, we name related work. In Section 6, we present our conclusion and describe future work.

2 SIMD-Style Processing

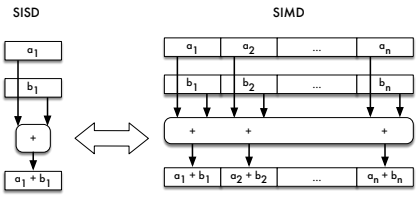

Fig. 1. Single Instruction Single Data (SISD) vs. Single Instruction Multiple Data (SIMD) processing.

A common approach of decreasing CPU cycles for algorithms is to adapt the algorithm to pipelining. While one instruction is executed, the next instruction is already fetched. This approach executes one instruction on one data item, called *Single Instruction Single Data (SISD)*. In contrast to execute one operation on one data item after another, the idea of *Single Instruction Multiple Data (SIMD)* is to execute a single instruction on multiple data items in parallel. In Figure 1, we show the difference between SIMD and SISD.

Modern CPUs have additional SIMD registers along with an additional instruction set adapted to process multiple data items in parallel. For example, using the Streaming SIMD Extensions 2 (SSE2) on the example in Figure 1, we can add 4 values in one clock cycle. After loading 4 different signed 32-bit integers in register a and 4 signed 32-bit integers in register b, we add them using _mm_add_epi32.

The main restriction of SIMD is that a sequential load of data is required. To load data into a SIMD register, the data has to be stored consecutively in main memory. Additionally, the size of SIMD registers is limited. Therefore processing data types of a common size of 64-bit and more lead to a small performance increase, since only few data items are processed with a single instruction.

Polychroniou et al. [7] show two general approaches to use SIMD, horizontal and vertical vector processing. They name the comparison of one search key to multiple other keys horizontal vectorization, whereas processing a different input key per vector lane is named vertical vectorization.

Since FAST, Seg-Tree, ART and VAST only use horizontal vectorization, we focus on this approach. For example, Zeuch et al. [2] use 128-bit SIMD registers and adjusted SIMD operations to load data into a register and to compare the data of one SIMD register with another. For example, a 128-bit SIMD register processes sixteen 8-bit or eight 16-bit data items with one instruction.

3 Optimized Main-Memory Tree Structures

In this section, we present the main concept of our multi-dimensional Elf index structure. Furthermore, we review previously proposed one-dimensional index structures Seg-Tree/Trie, FAST, ART, and VAST from which we want to extract important optimizations that could be applied to our Elf. We consider the adaptions made compared to the base index structure, the usage of SIMD, and the performance gain presented by the authors of the certain index structures.

3.1 Elf as Multi-Column Selection Index

Elf is a multi-dimensional tree-based index structure that was initially proposed to evaluate multi-column selection predicates (MCSPs, i.e., predicates that include several columns of one table) [6]. To exploit the properties of multi-dimensional data and MCSP queries, Elf features a special *conceptual design* and *memory layout*. In this section, we explain both features in order to give a broad overview on the basic functioning of Elf.

Design To illustrate the design of Elf, we use the data in Figure 2.(a) which has 4 columns and a *TID* to identify the tuple. Conceptually, the Elf indexes a table column by column where each column corresponds to a level in the tree. Each node of the Elf, called `DimensionList`, contains entries of the form $[Value, Pointer]$, where $Value$ is the column value of the subordinate tuples and $Pointer$ points to the subordinate `DimensionList`. The first level of the tree consists of one root node that contains every unique value of the first column – for instance, the values 0 and 1 of C_1 in the constructed Elf in Figure 2(b). According to these prefixes, data items can be assigned to one of these paths.

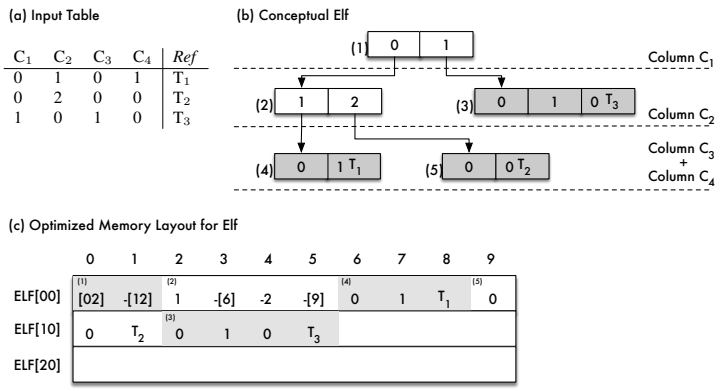

Fig. 2. Example data, Conceptual Elf, and Memory Layout of Elf

A `DimensionList` in a lower level holds all values of the column that have the prefix that is constructed following the path from the root to this `DimensionList`.

For example, a tuple stored in `DimensionList` (5) has the prefix 0 in the first dimension and 1 in the second dimension. Constructing the tree in this manner will span a tree with n levels where n corresponds to the number of indexed columns. However, the more dimensions or columns are indexed, the more sparsely populated (i.e., shorter) the `DimensionLists` are in deeper levels. The worst case is that we have a linked-list like tree, which we counter with the concept of `MonoLists` [6].

MonoList. Whenever, we encounter a path from a tree level t to n (where $t < n \wedge n =$ number of columns) without any fanout from t to n, we can construct a `MonoList`. In this case, a `MonoList` stores all values adjacent to each other and, thus, omits the pointers compared to the linked-list like case. In the Elf in Figure 2(b), `DimensionLists` (3),(4), and (5) are a `MonoList`, because there is no fan out since each of these `DimensionLists` correspond to only one tuple. The benefits of `MonoLists` are that it reduces storage costs and also leads to better performance due to less pointer chasing.

Memory layout The tree-based implementation using nodes and pointers yields a flexible structure for updates and insert, however, to achieve peak performance for analytical queries, a good memory layout is needed. In our experiments, our linearized Elf has a performance benefit of Factor 10 compared to the tree-based version. For linearizing the Elf into an array, we use a preorder traversal – i.e., we linearize the current node before storing the entries of the left subtree and afterwards follow all right subtrees.

In Figure 2(c), we present the corresponding linearized Elf. The first linearized `DimensionList` is only storing pointer to the underlying `DimensionLists`, because we create a hash map[1] here. The second linearized `DimensionList` stores values and pointers of the `DimensionList` (2), where values in brackets represent pointers into the array. In order to minimize the storage consumption of metadata even further, we omit to store length information of a `DimensionList`. Instead, we set the most significant bit (MSB) of the value to denote the end of a `DimensionList`. We visualize this with a negative value – for example, the -2 at offset 4 denotes the last entry of `DimensionList` (2). In contrast to that, a negative pointer means that we set the MSB of the pointer to indicate that the following `DimensionList` is a `MonoList`.

3.2 Seg-Tree and Seg-Trie

Zeuch et al. adapted the B^+-Tree by having a k-ary search tree as each inner node, called segment, and perform a k-ary search on each segment. In Figure 2, we show the adaption of nodes made by Zeuch et al. for Seg-Tree. The k-ary search bases on the binary search but divides the search space into k partitions with k-1 separators. Compared to binary search, the k-ary search reduces the

[1] The first list is not important for this paper. The interested reader is referred to the original publication [6]

complexity from $O(\log_2 n)$ to $O(\log_k n)$. Considering m as the maximum number of bits to represent a data type and $|SIMD|$ as the size of a SIMD-register, called SIMD bandwidth: $k = \frac{|SIMD|}{m}$ defines the number of partitions for the k-ary search.

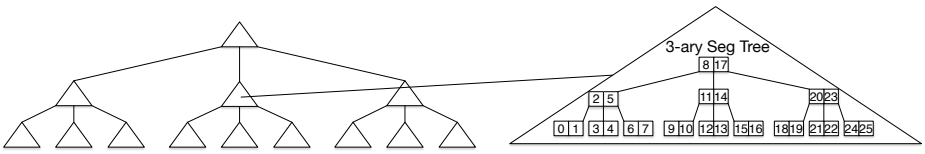

Fig. 3. Inner node format of Seg-Tree

SIMD Adaption. As mentioned before, each segment of the Seg-Tree is a k-ary search tree. To perform a k-ary search on a segment, Zeuch et al. liearize the elements of the segment. They show two algorithms for linearization using breadth-first search and depth-first search. Because of the condition $k = \frac{|SIMD|}{m}$, each partition of the k-ary search fits into a SIMD register and is compared to the search key. A perfect k-ary search tree contains $S_{max} = k^h - 1$ keys for an integer $h > 0$. The considered search algorithm only works for sequences with a multiple of $k - 1$ keys. In case of sequences with less than a multiple of $k - 1$ keys, they replenish the sequence with elements having the value $k_{max} + 1$ for the maximal key value k_{max} in the sequence. Consequently, the adapted search algorithm also works for sequences with less than a multiple of $k - 1$ keys.

Performance Improvement. The performance of Seg-Tree depends on k-ary search and horizontal vectorization. The smaller a key the more keys are compared parallel. Due to the relevance of 32 and 64-bit data types in modern systems, the k-ary search performance increases up to a factor of four for 32-bit types and two for 64-bit types.

Zeuch et al. also show the k-ary search on an adapted prefix trie *(trie for short)* called Seg-Trie. A trie is a search tree where nodes store parts of the key, called chunks. For example, a 32 bit key with a chunk size of 8 bit is stored in a trie with 4 levels. The Seg-Trie$_L$ is defined as a balanced trie where each node on each level contains one part of a key with L Bits. The tree has $r = \frac{m}{L}$ levels $(E_0, E_1, .., E_r)$, where m is the number of most bits to represent the data type. Similar to the Seg-Tree, each node is again designed as a k-ary search tree. Complete keys are stored in leaf nodes or are build by concatenating partial keys from the root node to a leaf node. This approach benefits of the separation of the keys in different levels of the tree. Consequently, they are smaller and more keys can be compared in parallel.

To perform a tree traversal on the Seg-Trie, the search key is split into r segments and each segment r_i is compared to the level E_i. If a matching partial

key is found in one node of E_i, the search continues at the referenced node for the partial key. If no match of the partial key is found, the Seg-Trie does not contain the search key and the search is finished. Consequently, the advantage of Seg-Trie against tree structures is the reduced comparison effort for non-existing key segments.

3.3 FAST

Kim et al. adapted a binary tree to optimize for architecture features like page size, cache-line size, and SIMD bandwidth called Fast Architecture Sensitive Tree [4]. In contrast to Seg-Trees, FAST is also adapted to disk-based database systems. Kim et al. show the performance increase because of decreasing cache misses and better cache-line usage. In order to optimize for architectural features, tree nodes are rearranged for hierarchical blocking. In Figure 3, we show an index tree blocked in three-level hierarchy introduced by Kim et al. They split the tree into a number of subtrees, each one fitting a page block. These sub-trees can be further split into cache-line-sized subtrees building the second level in the hierarchy. Each of these subtrees that fit a cache line can be further split into a number of nodes that fit a SIMD register. These nodes are laid out in a breadth-first fashion (the first key as root, the next keys as children on depth 2) representing a SIMD block.

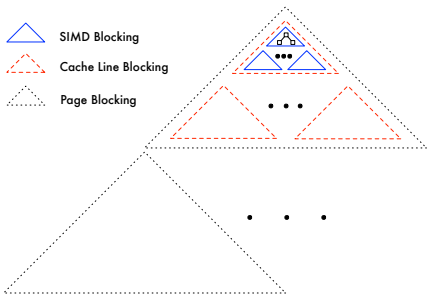

Fig. 4. Index tree blocked in three-level hierarchy: first-level page blocking, second-level cache-line blocking, third-level SIMD blocking of FAST. Adapted from Kim et al. [4].

SIMD Adaption. Kim et al. present implementations for building and traversing the tree adapted for CPU and also for GPU. Building up the tree, SIMD is used to computing the index for each set of keys within the SIMD-level block in parallel, achieving around 2X SIMD scaling as compared to the scalar code. With a Core i7 processor, the runtime of building a FAST tree with $64M$ tuples is less than 0.1 seconds. Traversing the tree, they compare one search key to multiple keys of the index structure. To use the complete bandwidth of cache and main memory within the search, blocks are loaded completely into associated memories from large blocks to small blocks. For a page block, at first the page is loaded into main memory. Then cache-line blocks are loaded one after another in the cache and for each cache-line block, the included SIMD blocks are loaded into the SIMD register. All keys of this SIMD block are compared with one SIMD instruction. After examining the bitmask as result fo the comparison, the corresponding next SIMD block is loaded (including the load of the surrounding larger blocks) until the key is found or the last level of the index structure is reached.

Performance Improvement. Kim et al. consider the search performance as queries per second. The CPU search performance on the Core i7 with 64M 32-bit (key, rowID) pairs is 5X faster than the best reported number [8], resulting in a throughput of 50M search queries per second. Considering larger index structures, the GPU performance increase exceeds the CPU performance increase, because TLB and cache misses grow up and the CPU search becomes bandwidth bound.

3.4 VAST

Yamamuro et al. extended FAST by building an index structure called Vector-Advanced and Compressed Structure Tree (VAST) [5]. They adapt the blocking and aligning structure of FAST and add compression of nodes along with improved SIMD usage.

In order to decrease the size of the index structure to better fit into main memory, they compress inner nodes. Inner nodes above a given threshold are compressed to 16 bit keys using lossy compression, while nodes in deeper levels are compressed to 8 bit with lossy compression. In the leaf nodes, Yamamuro et al. decrease the node size with the lossless compression algorithm *P4Delta*, which results in a good balance between compression ratio and decompression speed. To compensate errors that occur due to lossy compression, they present an algorithm for error correction. In a nutshell, they use prefix and suffix truncation to compress keys and calculate an offset Δw of the incorrect key to the correct key during tree traversal. If $\Delta w \neq 0$, VAST scans the leaf nodes sequentially until Δw becomes 0.

SIMD Adaption. Along with the other considered index structures, Yamamuro et al. compare multiple keys to one search key with SIMD in the tree traversal. Due to the key compression of nodes, VAST compares more keys in parallel than FAST. Additionally, they reduce branch misses with an adapted SIMD usage. They use addition and multiplication operations on the results of a SIMD key comparison, instead of conditional branches (if-then paths), to find the next node in tree traversal.

Performance Improvement. Due to lossy and lossless compression of the majority of nodes, Yamamuro et al. reach 95% less space consumption of the VAST compared to a binary tree or FAST. When considering an index with 2^{32} keys, they reach up to 6.0 and 1.24 times performance increase compared to a binary tree and FAST. Although errors occur due to lossy compression, the error correction does not take a major influence on the traversal speed.

3.5 ART

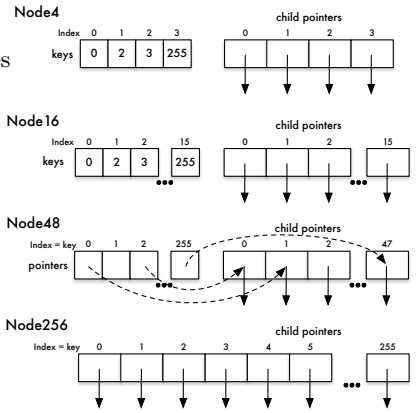

Fig. 5. Inner nodes of ART. The partial keys 0, 2, 3, and 255 are mapped to pointers of the subtrees. Adapted from Leis et al. [3]

Leis et al. adapted a radix tree for efficient indexing in main-memory database systems called Adaptive Radix Tree [3]. Similar to the Seg-Trie, the height of a radix tree depends on the chunk size of the keys stored in each node. ART divides keys into 8-bit chunks. They differentiate between inner nodes and leaf nodes and adapt each of them in a different way.

Instead of using a constant node size for each inner node, they present four types of nodes with different numbers of keys and children. In Figure 5, we show these node types containing keys that are mapped to subtrees. The types of nodes, sorted ascending by their size, are *Node4*, *Node16*, *Node48*, and *Node256*.

Node4: The smallest node type consists of one array with up to four sorted keys and another array with up to four children. The keys and pointers are stored at corresponding positions.

Node16: This node type consists of arrays of 16 keys and 16 children, storing keys and children analogue to *Node4*.

Node48: To avoid searching keys in many elements, this node type does not store the keys explicitly. Instead, an array with 256 elements is used. This array can be indexed with key bytes directly. It stores indexes into a second array with the size of 48 elements containing the pointers to child nodes.

Node256: The largest node type is simply an array of 256 pointers. Consequently, the next node can be found efficiently using a single lookup of the key byte in that array.

When the capacity of a node is exhausted due to insertion, it is replaced by a larger node type. When a node becomes underfull (e.g., due to key removal), it is replaced by a smaller node type.

For leaf nodes, Leis et al. use a mix of pointer and value slots in an array. If the value fits within the slot, they store it directly in the slot. Otherwise, a pointer to the value is stored. They tag each element with an additional bit indicating if a pointer or a value is stored.

SIMD Adaption. According to FAST and Seg-Tree, Leis et al. use SIMD within the tree traversal. They use horizontal vectorization, comparing the search key against multiple keys of a node. In contrast to Zeuch et al., using horizontal vectorization for each inner node, Leis et al. only compare the keys of nodes with type *Node16* in parallel. Therefor, they replicate the search key 16 times and compare these against all keys of nodes of type *Node16*.

In contrast to FAST and Seg-Tree, the goal of ART is also to reduce space consumption. Leis et al. use lazy expansion and path compression. The first technique, lazy expansion, is to create inner nodes only if they are required to

distinguish at least two leaf nodes. The second technique, path compression, removes all inner nodes that have only one child.

Performance Improvement. Since SIMD is only used in the tree search, we do not consider the performance increases of ART in insert and update operations. Leis et al. show, that looking up random keys using ART is faster than Seg-Tree and FAST, because ART has less cache misses and less CPU cycles per comparison. They consider the performance increases for dense and sparse keys, while ART works better with dense keys. Also they show that a span of 8 results in better performance than a smaller span.

4 Evaluation

In this section, we define criteria to compare the adaptions made in Seg-Tree, FAST, ART, and VAST to increase performance and show differences. These criteria will also show important optimizations that we should include in our own index structure Elf. We primarily focus on the usage of SIMD within the adapted index structures. We summarize the performance criteria and their impact on performance increase in Table 1 and show which index structure implements which of the criteria. In the following, we consider each criterium in detail before summarizing the comparison. For each criteria, we show how the index structure implements it. If the index structure does not implement the criteria, we consider if it is possible to implement it. Furthermore, we constitute why the adaptions cannot compared directly in view of performance.

4.1 Horizontal Vectorization

Segmenting the index structure is important to reach a better performance increase with SIMD instructions. Since the data for a SIMD operation has to be stored sequentially in main memory, a good segmenting and blocking strategy is necessary. In Section 2, we present vertical vectorization as alternative to horizontal vectorization. Due to the necessity of sequential data storage in main memory, comparing a different search key per vector lane is not applicable for SIMD instructions in tree traversal. All considered adaptions except Elf use horizontal vectorization in search operation for a single search key in the index structure. Hence, the usage of horizontal vectorization seems more promising than a vertical one.

4.2 Minimized key size

Minimizing the key size as much as possible speeds up the search performance, because more keys can be compared with a single SIMD instruction. Additionally, the smaller the keys are, the more keys fit into a cache line. Since tries or radix trees, depending on their span, have small chunks of the search key in each node, more keys are compared in parallel.

Criterium	Seg-Tree/ Trie	FAST	ART	VAST	Elf	Impact
Horizontal vectorization	x	x	x	x	-	high
Minimized key size	o	-	x	x	-	high
Adapted node sizes / types	-	x	-	x	-	low
Decreased branch misses	-	x	-	x	-	medium
Exploit cache lines using blocking and alignment	-	x	-	x	x	medium
Usage of Compression	o	-	x	x	x	medium
Adapt search algorithm for linearized nodes	x	-	-	-	x	low

Legend: x = implements the issue; o = partially implements the issue; - = does not implement the issue

Table 1. Comparison of the considered index structures based on extracted criteria and the impact on performance increase

Zeuch et al. minimize the key size for the Seg-Trie using a small chunk size. This leads to a bigger tree height, but the performance increase of comparing more keys in parallel leads to better search performance. For the Seg-Tree, they do not use adaptions to minimize the key size, therefore the performance increase with SIMD depends on the size of the used data type. Analogue to the Seg-Trie, Leis et al. use a small key size for ART to increase the performance speed up.

Kim et al. do not minimize the key size for FAST, therefore the performance increase with SIMD also depends on the size of the used data type. Yamamuro et al. use compression to minimize the key size of VAST, respectively to the layer. Since VAST is an extended Version of FAST, there is no need to check if this adaption is also applicable for FAST.

Elf currently does not feature a special compression mechanism of node entries. However, since the value range is usually known, compression will definitely pay out in this scenario. Especially in combination with SIMD acceleration, a magnitude of performance improvement is reasonable to expect.

4.3 Adapted node sizes and types

Index structures for disk-based database systems often adapt their node size to the page size of disk. In modern database systems, this adaption becomes obsolete. Consequently, the considered index structures adapt their node to other parameters.

Zeuch et al. adapted the B^+-Tree by having a k-ary search tree as each inner node. To perform k-ary search on the keys of a node, each node has the size of multiple of the SIMD-bandwidth k.

Kim et al. segment FAST into SIMD, cache, and page blocks. Each block contains multiple nodes, respectively to the block size. Since FAST is based on binary tree, every node contains still a single element and is not adapted in type or size. VAST uses the same hierarchical blocking and also does not adapt single nodes.

Leis et al. present different node types for inner nodes of ART. The node types differ in the number of keys and child nodes. Although Elf does not feature special node types, there are several possibilities to support those. In fact, large inner nodes could be represented as a k-ary search tree or even Seg-Trie.

4.4 Decreased branch misses

Evaluating comparisons using conditional branches can lead to branch misses, if the CPU prefetches the wrong branch. Consequently, decreasing branch misses improves search performance because less CPU cycles are needed. Yamamuro et al. use addition and multiplication operations in VAST to find the next node in their evaluation of the comparison result. This avoids conditional branching and therefore decreases branch misses.

Zeuch et al. use conditional branches in their search algorithms for the Seg-Tree and Seg-Trie, which can be replaced by the approach of VAST to decrease branch misses. Analogue, ART and Elf can be adapted using this approach to speed up the search performance.

4.5 Exploit cache lines using blocking and alignment

In main-memory database systems, the IO-bottleneck moves from being between disk and RAM further to being between RAM and cache. Consequently, an efficient cache-line usage is important to speed up the performance of index structures. An efficient cache-line usage is reached by blocking parts of index structures to cache-line size.

Zeuch et al. linearize keys to a multiple of the SIMD bandwidth with the Seg-Tree but do not adapt the node size to the cache-line size. Accordingly, an adaption of the inner nodes of Seg-Tree to the cache-line size could lead to better cache-line usage and consequently to better search performance. This is similar to Elf that also features a linearization of nodes for better cache-line usage

Kim et al. segmented FAST into SIMD, cache, and page blocks. They highly optimize FAST for efficient cache-line usage by blocking the index structure into blocks with the size of a cache line.

Leis et al. also do not make adaptions of ART in view of blocking to the cache-line size. Thus, a performance increase can be reached by adapting ART to cache-line size.

4.6 Usage of compression

Along with the fast growing amount of data stored in modern database systems, the size of index structures grows fast. Compression of nodes decreases the size

of index structures. Key compression leads to better search performance with SIMD, because more keys are compared in parallel.

Zeuch et al. do not use any node or key compression for Seg-Tree. Kim et al. also do not compress nodes or keys in FAST. Consequently, considering compression for Seg-Tree and FAST provides space for additional performance increase.

Yamamuro et al. use lossy block compression to decrease the key size of blocks. Consequently, they compare more keys with one SIMD instruction, whereas performance decrease of the occurring errors is smaller than the performance increase of the compression. Instead of compressing blocks, Seg-Trie and ART use path compression to decrease the tree height. Consequently, less nodes are passed to reach the matching leaf node, if it exists.

Leis et al. use path compression within ART. Path compression removes all inner nodes that have only one child to decrease the tree height and therefore increase search performance. Key compression is a possible adaption for ART to improve the search performance. This is also an idea that Elf exploits. We create `MonoLists` when there is no fan out in the lower levels anymore.

4.7 Adapt search algorithm for linearized nodes

Searching children in nodes with many keys can be speeded up with adapted search algorithms compared to linear search.

Zeuch et al. introduce k-ary search for the Seg-Tree and Seg-Trie, which performs in $O(log_n)$ compared to $O(n)$ of linear search, where k is the SIMD bandwidth. Consequently, less keys are compared in one node, whereas a linearised storage of the keys in the node is required.

For FAST and VAST, the k-ary search is not applicable because they are adapted from binary trees and have only one key in each node. Leis et al. use linear search in their adapted nodes of ART to find children. For the node type *Node256* k-ary search can speed up finding the correct child node. Similar to this, the sort order in nodes of Elf is not fixed. Hence, ART and Elf provide space to use k-ary search.

4.8 Summary

After considering each performance criteria in detail, we summarize the comparison and weight the mentioned criteria according to their performance increase.

As mentioned before, comparing a different search key per vector lane is not applicable for SIMD instructions in tree traversal. Therefore, all considered adaptions use horizontal vectorization, but evaluate the result of the SIMD comparison in different ways. Since horizontal vectorization with SIMD, depending on the key size, leads to direct performance increase up to a multiple, we assign the highest weight to it.

Due to the mentioned performance increase of horizontal vectorization, minimizing key size also leads to better search performance. We assign the second

highest weight to minimizing the key size, since more keys are compared in parallel. Because of the compression of keys, VAST reaches the highest performance increase by minimizing key size compared to the other index structures.

Compared to horizontal vectorization and minimizing key size, an efficient cache-line usage and avoiding branch misses lead to lesser performance increase. Although, both criteria decrease the number of CPU cycles, the performance increase does not reach a multiple. Therefore, we assign the third highest weight to efficient cache-line usage and to avoiding branch misses.

Afterwards, we assign the fourth highest weight to the usage of compression. With key compression, the performance increase of horizontal vectorization rises. Path compression decreases the height of index structures and consequently less nodes are passed during traversal.

We assign the lowest weight to the adaption of nodes and to the usage of an adapted search algorithm. The adaption of nodes does not lead directly to performance increase, but is the basis for horizontal vectorization and efficient cache-line usage. The usage of an adapted search algorithm, for example k-ary search, needs lesser comparisons. Nevertheless, it is not applicable for all of the considered index structures and it requires large node sizes to increase the performance.

Overall, this survey has shown that our index structure Elf already shares some characteristics with highly-tuned main-memory index structures. However, especially the most promising optimizations (horizontal vectorization, minimizing key sizes, and branch misses) are currently not used. Still, we have argued that all these optimizations are possible to be implement in future work.

5 Related Work

In main-memory database systems, a B^+-Tree does not fulfill the requirements of efficient cache-line usage. Therefore, it is important to adapt B^+-Trees to efficient cache-line usage. Rao et al. show improved cache-line usage for B^+-Trees [9]. Compression is used to decrease the index size to fill more keys into the cache line. Zukowski et al. show the performance increase of better cache-line usage with compression [10]. This adaption gives an alternative for us compared to the compression used in VAST.

Graefe et al. present and summarize several techniques for improving cache performance for B-Trees [11]. Furthermore, Bender et al. show a cache oblivious B-Tree [12] and a cache oblivious B-Tree for string data types [13]. Borodin et al. show improved page and cache-line usage in generalized search trees of PostgreSQL [14]. Other adapted index structures are shown by Rao et al., the Cache-Sensitive Search Trees (CSS-Tree) [9], and the Cache Sensitive B^+-Tree (CSB+-Trees) [15]. They construct the tree such that the keys are placed with the best cache optimization as possible in terms of spatial or temporal locality. Since we consider efficient cache-line usage as an important criteria for modern index structures, these adaptions gave useful ideas to measure performance increase due to efficient cache-line usage.

6 Conclusion and Future Work

In this work, we compare the adapted index structures Seg-Tree/Trie, FAST, VAST, and ART and compared their properties to our own index structure Elf. A special consideration was given here to the usage of SIMD. We extract important criteria of these adaptions and weight them according to their impact on the performance.

Due to direct performance increase of horizontal vectorization with SIMD up to a multiple, this is the highest weighted criteria. Minimizing the key size leads to the comparison of more keys in parallel and is therefore assigned with the second highest weight. Afterwards, we assign the efficient usage of the cache line and the decrease of branch misses the third highest weight. The other considered criteria, in descending order in view of performance increase, are the usage of compression, the adaption of node types and size, and the usage of an adapted search algorithm to find keys within nodes.

Since, currently, Elf does not feature most of these powerful optimizations of the considered index structures, we plan to integrate them into Elf.

References

1. Suaib, M., Palaty, A., Pandey, K.S.: Architecture of simd type vector processor. International Journal of Computer Applications **20**(4) (2011)
2. Zeuch, S., Huber, F., Freytag, J.c.: Adapting tree structures for processing with simd instructions. In: Proceedings of the International Conference on Extending Database Technology (EDBT), Citeseer (2014)
3. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: Proceedings of the International Conference on Data Engineering (ICDE), IEEE (2013) 38–49
4. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proceedings of the International Conference on Management of Data (SIGMOD), ACM (2010) 339–350
5. Yamamuro, T., Onizuka, M., Hitaka, T., Yamamuro, M.: Vast-tree: A vector-advanced and compressed structure for massive data tree traversal. In: Proceedings of the International Conference on Extending Database Technology (EDBT), ACM (2012) 396–407
6. Broneske, D., Köppen, V., Saake, G., Schäler, M.: Accelerating multi-column selection predicates in main-memory - the Elf approach. In: International Conference on Data Engineering (ICDE), IEEE (2017) 647–658
7. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of the International Conference on Management of Data (SIGMOD), ACM (2015) 1493–1508
8. Schlegel, B., Gemulla, R., Lehner, W.: K-ary search on modern processors. In: Proceedings of the International Workshop on Data Management on New Hardware (DaMoN), ACM (2009) 52–60
9. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: Proceedings of the International Conference on Very Large Databases (VLDB). Volume 99. (1999) 78–89

10. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: Proceedings of the International Conference on Data Engineering (ICDE), IEEE (2006) 59–59
11. Graefe, G., Larson, P.A.: B-tree indexes and CPU caches. In: Proceedings of the International Conference on Data Engineering (ICDE), IEEE (2001) 349–358
12. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. In: Proceedings of the Annual Symposium on Foundations of Computer Science, IEEE (2000) 399–409
13. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-oblivious string B-trees. In: Proceedings of the Symposium on Principles of Database Systems (PODS), ACM (2006) 233–242
14. Borodin, A., Mirvoda, S., Kulikov, I., Porshnev, S.: Optimization of memory operations in generalized search trees of PostgreSQL. In: International Conference: Beyond Databases, Architectures and Structures (BDAS), Springer (2017) 224–232
15. Rao, J., Ross, K.A.: Making b+-trees cache conscious in main memory. In: ACM SIGMOD Record. Volume 29., ACM (2000) 475–486