

SIMD Acceleration for Index Structures

Marten Wallewein-Eising
Otto-von-Guericke University
Magdeburg, Germany
marten.wallewein-eising@st.ovgu.de

Abstract—

- **summary:**

Give short an overview of SIMD and modern index structures

Explain what are the problems of the “old” index structures made for disk-based database systems

Explain which approaches were made to adapt index structures to modern systems and what they have in common and what are differences

- **Why is this work important:**

Give a state of current development of the index structures

Collect common approaches to adapt other index structures TODO: ReThink

- **K-ary search trees, FAST, VAST and ART compared**

- **Contribution: What are important approaches used by different implementations to adapt index structures to modern systems**

*Index Terms—*SIMD, index

I. INTRODUCTION

After decades of creating and improving index structures for disk-based database systems, nowadays even large databases fit into the main memory. Since index structures like the B^+ -tree [TODO: ref] or the radix tree [TODO: ref] have an important part in database-systems to realise SCAN or range-based search operations, these index structures experienced many adaptations 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.

In this paper we compare different approaches to adapt index structures to use Single Instruction Multiple Data (SIMD) [1] operations and to better overcome the bottleneck from RAM to CPU cache. We consider the K-ary Search Tree (Seg-tree) [TODO: ref], Adapted Radix Tree (ART) [4], Fast Architecture Sensitive Tree (FAST) [6], and Vector-Advanced and Compressed Structure Tree (VAST) [5]. As the authors of VAST-Tree show, important causes for increased calculation time are cache misses and branch mispredictions. To overcome branch mispredictions and to decrease CPU cycles, SIMD [TODO: ref] is used in modern index structures for tree traversal. The authors of the k-ary search show how to use SIMD to compare multiple keys in one CPU cycle. To decrease cache misses, the authors of FAST and ART show how to adapt index structures to the cache line size.

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. We consider the approaches VAST and ART make to implement operations like update and insert und name ideas to use SIMD for them. Consequently, with this work we make the following contributions:

- We compare different adaptations of index structures to fulfill requirements of modern database systems
- We highlight the usage of SIMD and the cache line adaptations in all approaches
- We show openings for adaption for other approaches to use SIMD

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 analyse the different approaches of adapted index structures and evaluate the comparison in section 4. In section 5 we name related work. In section 5 we present our conclusion und describe future work in section 7.

II. PRELIMINARIES

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. In contrast to execute one operation on one data after another, the idea of SIMD is to execute a single instruction on multiple data. The authors of Rethinking SIMD Vectorization for In-Memory Databases [TODO: ref] show two general approaches to use SIMD in in-memory databases, 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. TODO: Exact difference?

Since FAST, Seg-Tree, ART and VAST only use horizontal vectorization, we focus on this approach. Modern CPUs have additional SIMD registers along with an additional instruction set adapted to process multiple data with a single instruction. For example, the authors of Seg-Tree [3] 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. A 128-bit SIMD register processes sixteen 8-bit or eight 16-bit data items with one instruction. In Table 1 TODO: Insert Table! we show a comparison of key size and the number of keys that can be processed parallel with one SIMD instruction.

The main restriction of SIMD instructions 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.

III. ADAPTED TREE STRUCTURES

A. Seg-Tree

The authors of Seg-Tree 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. 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 complexity from $O(\log 2n)$ to $O(\log kn)$. They consider m as the most bits to represent a datatype and $|SIMD|$ as the size of SIMD-register, called SIMD bandwidth. Furthermore, $k = \frac{|SIMD|}{m}$ defines the count of partitions for the k-ary search.

As mentioned before, each segment of the Seg-Tree is a k-ary search tree. To perform a k-ary search on a segment, the authors of Seg-Tree make a linearization of the segment. They show two algorithms for linearization, 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 can be 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 works for sequences with less than a multiple of $k - 1$ keys. **TODO: Fix: In figure 1, we show the adaption of nodes made by the authors of Seg-Tree. Todo: Figure!**

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

The authors of Seg-Tree also show the k-ary search on an adapted prefix tree (*trie for short*) [TODO: ref] called Seg-Trie. A trie is a search tree where each node stores a part of the key. Each node is again designed as a k-ary search tree. **TODO: Useful?** 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, the compare keys are smaller and more keys can be compared parallel. The Seg-Trie_L is defined as a balanced trie where each node on each level contains one part of the key with L Bits. The tree has $r = \frac{m}{L}$ levels (E_0, E_1, \dots, E_r), where m is the number of most bits to represent the data type.

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 from non-existing key segments.

B. FAST

C. ART

D. VAST

IV. EVALUATION

In common:

- SIMD instructions used to compare the search key with multiple keys of the index
- Segmenting tree to blocks for a better usage of cache lines, save the data of the nodes in an adapted way
- The keys should be as short as possible to compare more keys in one step and to decrease the passed data to the cache line
- Each approach improves the tree traversal

Differences:

- Node compression in VAST, Path compression in ART and K-ary seg trie
- FAST and K-ary trees readonly to improve traversal, ART and FAST adapt insert too
- FAST uses and K-ary trees will use GPU calculation instead of CPU

Why performance can not be compared in a useful way... performance issues:

- Seg-Tree:

V. RELATED WORK

TODO:

- ART and VAST compared to FAST??
- Ideas and implementations of the adapted trees already in III...
- KD-Tree with SIMD

VI. CONCLUSION

VII. FUTURE WORK

Open questions, use SIMD for tree creation/updates instead of only for traversal

REFERENCES

- [1] Mohammad Suaib, Abel Palaty and Kumar Sambhav Pandey, "Architecture of SIMD Type Vector Processor" in International Journal of Computer Applications (0975 - 8887) Volume 20 No.4, April 2011.
- [2] Jingren Zhou and Kenneth A. Ross "Implementing Database Operations Using SIMD Instructions" in ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA
- [3] Steffen Zeuch, Frank Huber and Johann-Christoph Freytag "Adapting Tree Structures for Processing with SIMD Instructions" in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece
- [4] Viktor Leis, Alfons Kemper and Thomas Neumann "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases"
- [5] Takeshi Yamamuro, Makoto Onizuka, Toshio Hitaka, and Masashi Yamamuro "VAST-Tree: A Vector-Advanced and Compressed Structure for Massive Data Tree Traversal" in EDBT 2012, March 26-30, 2012, Berlin, Germany.
- [6] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt and Pradeep Dubey "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs" in SIGMOD10, June 6-11, 2010, Indianapolis, Indiana, USA.