KISS-Tree: Smart Latch-Free In-Memory Indexing on Modern Architectures

Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Growing main memory capacities and an increasing number of hardware threads in modern server systems led to fundamental changes in database architectures. Most importantly, query processing is nowadays performed on data that is often completely stored in main memory. Despite of a high main memory scan performance, index structures are still important components, but they have to be designed from scratch to cope with the specific characteristics of main memory and to exploit the high degree of parallelism. Current research mainly focused on adapting block-optimized B+-Trees, but these data structures were designed for secondary memory and involve comprehensive structural maintenance for updates.

In this paper, we present the *KISS-Tree*, a latch-free in-memory index that is optimized for a minimum number of memory accesses and a high number of concurrent updates. More specifically, we aim for the same performance as modern hash-based algorithms but keeping the order-preserving nature of trees. We achieve this by using a prefix tree that incorporates virtual memory management functionality and compression schemes. In our experiments, we evaluate the *KISS-Tree* on different workloads and hardware platforms and compare the results to existing in-memory indexes. The *KISS-Tree* offers the highest reported read performance on current architectures, a balanced read/write performance, and has a low memory footprint.

1. INTRODUCTION

Databases heavily leverage indexes to decrease access times for locating single records in large relations. On classic disk-based database systems, the B+-Tree [2] is typically the structure of choice. B+-Trees are suited for kinds of those systems because they are optimized for block-based disk accesses. However, modern server hardware is equipped with high capacities of main memory. Therefore, database architectures move from classic disk-based systems towards databases that keep the entire data pool (i.e., relations and indexes) in-memory and use secondary memory (e.g., disks) only for persistence. While disk block accesses constituted the bottleneck for disk-based systems, modern in-memory databases shift the memory hierarchy closer to the CPU and face the "memory wall" [13] as new bottleneck. Thus, they have to care for CPU data caches, TLBs, main memory accesses and access patterns. This essential change also affects indexes and forces us to design new index structures that are now optimized for these new design goals. Moreover, the movement from disks to main memory dramatically increased data access bandwidth and reduced latency. In combination with the increasing number of cores on modern hardware, parallel processing of operations on index structures imposes a new challenges for us, because the overhead of latches became a critical issue. Therefore, building future index structures in a latch-free way is essential for scalability on modern and future systems.

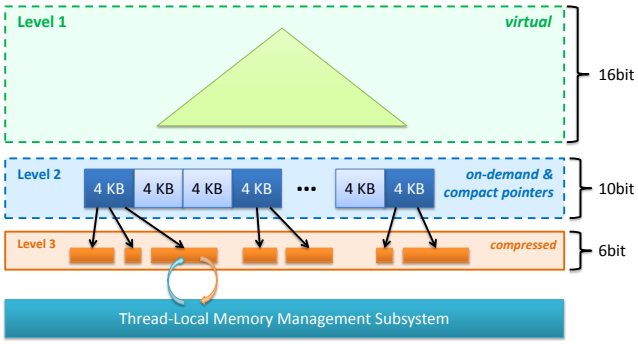
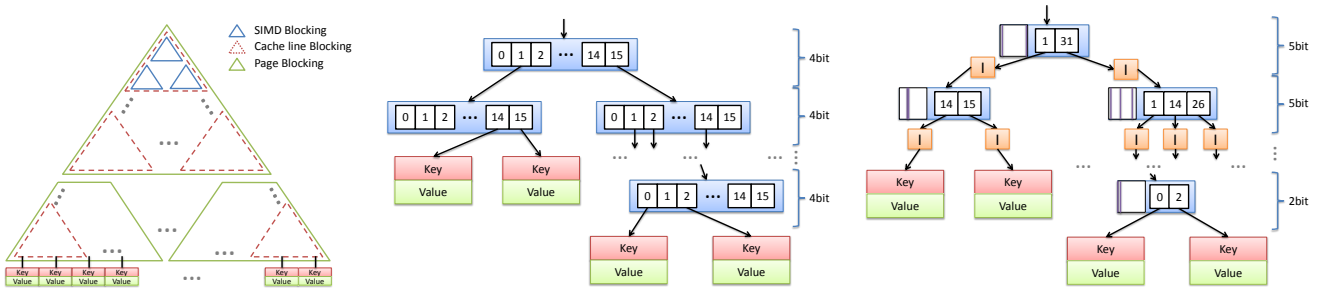


Figure 1: KISS-Tree Overview.

In this paper, we present the *KISS-Tree* [1] that is an order-preserving latch-free in-memory index structure for currently maximum 32bit wide keys and arbitrary values. It is up to 50% faster compared to the previously reported read performance on the same architecture[1] and exceeds its update performance by orders of magnitude. The *KISS-Tree* is based on the *Generalized Prefix Tree* [3] and advances it by minimizing the number of memory accesses needed for accessing a key's value. We achieve this reduction by taking advantage of the virtual memory management functionality provided by the operating system and the underlying hardware as well as compression mechanisms. Figure 1 shows an overview of the *KISS-Tree* structure. As shown, the entire

[1]Due to the unavailability of source code and binaries, we refer to the figures published in [5].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012) , May 21, 2012, Scottsdale, AZ, USA
Copyright 2012 ACM 978-1-4503-1445-9 ...$10.00.



(a) FAST. (b) Prefix Tree ($k' = 4$). (c) CTrie.

Figure 2: Existing In-Memory Index Structures.

index structure comprises only three tree levels. Each level leverages different techniques to find a trade-off between performance and memory consumption on the respective level.

2. RELATED WORK

In-memory indexing is a well investigated topic in database research since years. Early research mainly focused on reducing the memory footprint of traditional data structures and making the data structures cache-conscious. For example, the T-Tree [8] reduces the number of pointers of traditional AVL-trees [6] while the CSB+-Tree [11] is an almost pointer-free and thus cache-conscious variant of the traditional B+-Tree [2]. These data structures usually offer a good read performance. However, they struggle with problems when updates are performed by multiple threads in parallel. All block-based data structures require comprehensive balancing operations when keys are removed or added. This implicates complex latching schemes [7] that cause an operation to latch across multiple index nodes and hence blocks other read/write operations even if they work on different keys. Even if there is no node splitting/merging required, B+-Tree structures have to latch coarse-grained at page-level, which increases contention. Moreover, it is not possible to apply optimistic latch-free mechanisms, for instance atomic instructions, without the help of hardware transactional memory(HTM). To overcome this critical point, PALM [12] took a synchronous processing approach on the B+-Tree. Instead of processing single operations directly, the operations are buffered and each thread processes the operations for its assigned portion of the tree. This approach eliminates the need for any latches on tree elements, but still suffers from the high structural maintenance overhead of the B+-Tree.

The FAST approach [5] gears even more in the direction of fast reads and slow updates. FAST uses an implicit data layout, as shown in Figure 2(a), which gets along without any pointers and is optimized for the memory hierarchy of todays CPUs. The search itself is performed using a highly optimized variant of binary search. The fastest reported numbers for read performance on modern CPUs were published for this approach. However, FAST achieves only a very low update performance because each update requires a rebuild of the entire data structure.

The Generalized Prefix Tree [3] takes a different approach than the B+-Tree. Here, the path inside the tree does not depend on the other keys present in the index, it is only determined by the key itself. Each node of the tree consists of a fixed number of pointers; the key is split into fragments of an equal length k' where each fragment selects one pointer of a certain node. Starting from the most significant fragment, the path through the nodes is then given by following the pointers that are selected by the fragments, i.e., the i-th fragment selects the pointer within the node on level i. Figure 2(b) shows an exemplary prefix tree with $k' = 4$. The first tree level differentiates the first 4 bits of the key, the second level the next 4 bits, and so on. Prefix trees do not have to care for cache line sizes because there is only one memory access per tree level necessary. Also the number of memory accesses is limited by the length of the key, for instance, a 32bit key involves a maximum of 8 memory accesses. In order to reduce the overall tree size and the memory accesses, a prefix tree allows *dynamic expansion* and only unrolls tree levels as far as needed like shown on the left hand side of the example. Because of the *dynamic expansion*, the prefix tree has to store the original key besides the value in the content node at the end of the path. The characteristics of the prefix tree allow a balanced read/update performance and parallel operations, because there is not much structural maintenance necessary.

A weak point of the prefix tree is the high memory overhead that originates from sparely occupied nodes. The CTrie [10] removes this overhead by compressing each node. This is achieved by adding a 32bit bitmap (limits the k' to 5) to the beginning of each node, which indicates the occupied buckets in that node. Using this bitmap, the node only has to store the buckets in use. Due to the compression, nodes grow and shrink. Thus, the CTrie raises additional costs for updates, because growing and shrinking nodes requires data copying. To allow efficient parallel operations on the CTrie, it was designed latch-free. Synchronization is done via atomic compare-and-swap instructions, like it is possible in the basic prefix tree. However, because nodes and buckets inside nodes are moving as a result of the compression, the CTrie uses *I-nodes* for indirection as depicted in Figure 2(c). This indirection creates a latch-free index structure, but doubles the number of memory accesses per key.

Another approach was introduced by Burst Tries [4]. Burst Tries utilize existing index structures and combine them to a heterogeneous index. The index starts on the topmost levels as a Prefix Tree and changes over to another structure like the B+-Tree. This combination lets the Burst Trie take advantage from the individual characteristics of the base structures.

3. KISS-TREE STRUCTURE

The *KISS-Tree* is an in-memory index structure that is based on the generalized prefix tree and improves it in terms of reduced memory accesses and memory consumption. Figure 1 shows an overview of the *KISS-Tree*. Like in the generalized prefix tree, the 32bit key is split into fragments f_{level} that identify the bucket within the node on the corresponding level. While the prefix tree uses an equal fragment length on each tree level, the *KISS-Tree* splits the key into exactly three fragments, each of a different length. This results in three tree levels, where each level implements different techniques for storing the data. The fragment sizes for each level play an important role because of the individual characteristics of each level. In the following, we describe the characteristics of each level and the reason for the respective fragment size in detail.

3.1 Level 1–virtual level

The first level uses a fragment length of 16. Therefore, this level differentiates the 16 most significant bits of the key. The technique we deploy here is *direct addressing*. *Direct addressing* means that there is no memory access necessary to obtain the pointer to the node on the next level. Instead, the pointer is directly calculated from the fragment f_1. In order to make *direct addressing* possible, the next tree level has to store all nodes sequentially. The major advantage we gain from *direct addressing* is that we neither have to allocate any memory for the first level nor have to issue a memory access when descending the tree. Thus, we call this level the *virtual level*.

3.2 Level 2–on-demand level

The fragment size for the second level is set to 10. Thus, we use the next 10 bits of the key to determine the bucket inside a node on this level, which contains a pointer to the corresponding node on the next level. On this level, we deploy two techniques. The first technique is the *on-demand* allocation, provided by the platform and the operating system. Todays computers know two address spaces, which are the virtual and the physical address space. All applications that are running get their own virtual address space that is transparently mapped to the shared physical one. This mapping is done by a piece of hardware known as the memory management unit (MMU). The MMU uses the page directory that is maintained by the operating system to translate virtual addresses into physical addresses. If an application tries to access a virtual address that is either not mapped to physical memory or is write protected, the MMU generates an interrupt and invokes kernel routines to handle this interrupt. The kernel has two options to handle this interrupt. Either it terminates the application or physically allocates a new piece of memory. The second choice is called *on-demand* allocation. Here, we explicitly ask the kernel to allocate a large consecutive segment of memory in the virtual address space and do not allocate any physical memory at all. Thus, the second level is entirely virtually allocated at startup and does not consume any physical memory at this point of time. As soon as we write something to a node on the second level, the physical memory for this node is allocated by the operating system and actually consumes memory. The benefit we gain from the *on-demand* allocation is that we fulfill the requirements given by the first level (sequentially stored nodes) and actually do not have to

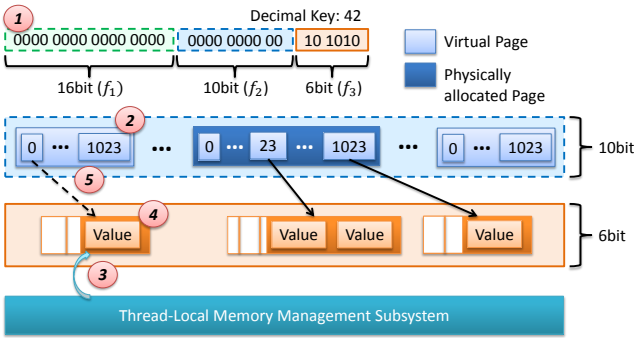


Figure 3: Update Operation on KISS-Tree.

waste the real physical memory for that. The critical point for *on-demand* allocation is the granularity of the memory mappings. For instance, if we write a 4Byte pointer on this level, the operating system has to allocate physical memory at the size of memory pages. The smallest available size for memory pages is currently 4KB on common architectures. Thus, when writing a small pointer, *on-demand* allocation has to map 4KB contiguous virtual address space to 4KB contiguous physical address space. To summarize, the complete second level is virtually allocated on index creation via one *mmap* call and does not claim any physical memory at this point of time. Only if a pointer is written to this level, the corresponding 4KB page gets physically allocated. Concurrent allocations of 4KB pages are solely handled by the operating system.

Because of the allocation granularity of 4KB, it makes sense to use 4KB nodes on this level. A node on this level consists of $2^{10} = 1024$ buckets because of the fragment length of 10. Thus, we need 4Byte pointers inside these buckets to get a node size of 4KB. Here we deploy our second technique called *compact pointer*. *Compact pointers* reduce standard 64bit pointer to 32bit pointer. The compaction is possible, because nodes on the third level are of one of the $2^6 = 64$ distinct sizes and the maximum number of child nodes of all nodes on the second level is 2^{26}. Thus, we are able to use the first 6 bits of a *compact pointer* for storing the size of the next node, which is later translated to an offset. The remaining 26 bits are used for storing the block number.

Using *compact pointers* for achieving a node size of 4KB on the second level has the effect that as soon as one pointer is written to a node, the entire node becomes physically allocated and consumes memory. The maximum number of nodes that are possible on this level are 2^{16}. This means that in the worst case scenario, the second level consumes 256MB of memory for 2^{16} keys. We investigate this issue of memory consumption further in Section 6.

3.3 Level 3–compressed nodes

On the last level, we use a fragment size of 6. Thus, the lowest 6 bits of the key determine the bucket inside a node on this level. Because the last level has the largest number of possibly present nodes (2^{26} at maximum), we apply a compression to the nodes on this level, which is similar to the compression scheme used for the CTrie. The compression works by adding a 64bit bitmap in front of each node. Because there is a maximum of $2^6 = 64$ buckets in a

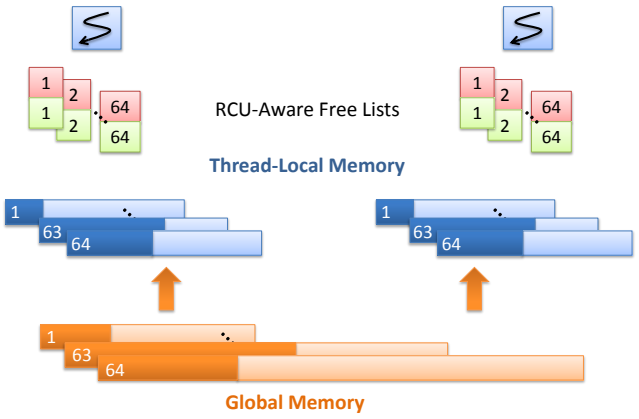


Figure 4: Memory Management.

node on the last level, the bitmap indicates which buckets are actually in use. Thus, we only have to store these buckets in use beyond the bitmap instead of storing all 64 buckets even if only a fraction of them has content. All buckets in a node on this level contain the value for the corresponding key. For duplicates and transactional isolation, it is also possible to store additional information as content.

Compression leads to nodes that shrink and grow over time. To handle this resizing and allowing latch-free compare-and-swap operations for updates, we use read-copy-updates (RCU) [9]. RCU creates copies of nodes and merges changes to this private copy. This allows readers to read consistent data from the original node, while the updated node is created. When the updated node becomes ready, we use an atomic compare-and-swap operation to exchange the corresponding pointer on the second level with a pointer to the new node. In order to allow arbitrary values and avoid lost updates, we do not make any in-place updates.

4. OPERATIONS

In this section, we describe how common index operations like updates and reads are processed on the *KISS-Tree*. The *KISS-Tree* also supports deletions and iterations, which are trivial to derive from the described operations.

4.1 Updates

We demonstrate the update respectively insertion of a key with the help of the *KISS-Tree* (8Byte Rids as values) depicted in Figure 3. At the beginning, the tree contains three key-value pairs and one physically allocated page on the second level. In this example, we insert the decimal key 42 into the *KISS-Tree*. The update operation starts with splitting the key into the three fragments f_1,f_2, and f_3 each of their respective length (step 1 in the figure). The first 16bit long fragment f_1 is used to identify the corresponding node on the second level (step 2). Because all nodes on the second level are of equal size and are sequentially stored, we can directly calculate the pointer to that node. Following the second step, we use the 10bit fragment f_2 to determine the bucket inside the second level node. Because the entire node is not physically allocated, the node contains only zero pointer[2], which indicates an empty bucket. The insert operation now remembers the zero pointer for the later compare-and-swap operation. Afterwards, we have to request a new node from the memory management subsystem that is able to store the nodes bitmap and exactly one value (step 3). In step 4, we prepare the new node for tree linkage. Therefore, we set the corresponding bit in the bitmap, which is determined by f_3. In our specific example, we have to set the 42th bit. Afterwards, we write the actual value after the bitmap. Finally, we try to exchange the pointer on the second level with the *compact pointer* to the new level three node using a compare-and-swap instruction (step 5). If the compare-and-swap is successful, the operation finished and the first 4KB node on level two is now backed with physical memory; otherwise the entire update operation has to repeat.

[2]The virtual memory is allocated via *mmap*, which initializes newly allocated pages for security reasons with zero

4.2 Reads

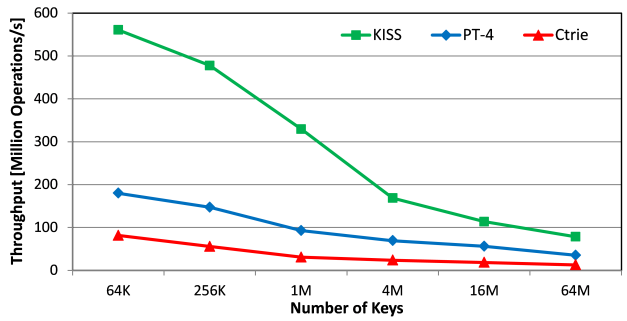
Assuming that the previously inserted decimal key 42 is now present in the *KISS-Tree* depicted in Figure 3, we now show how to read it again. We take f_1 to calculate the absolute address of the corresponding node on the second level and identifiy the bucket in this node using f_2. Because we read a non-zero pointer from this bucket, the bucket is in use and we translate the *compact pointer* to a 64bit pointer to the next node on the third level. To check whether the key is present, we test the 42th bit of the nodes bitmask. Because the bit is set, we apply a bitmask to the bitmap to unset all bits behind the 42th bit. Now we count the number of set bits using a population count instruction to obtain the bucket inside the compressed node, which contains the requested value.

4.3 Batched Reads

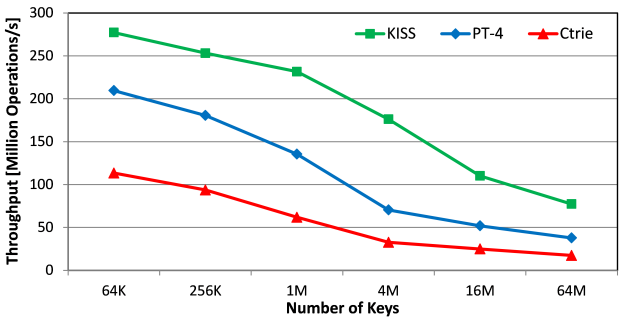
Especially on large *KISS-Trees* that do not fit into the CPU cache anymore, the memory latency plays an important role and decreases the performance of reads. In order to reduce the latency, we applied batched reads for latency hiding. With batched reads, each thread working on the tree executes multiple read operations at once. The number of simultaneous operations is the *batch size*. Executing multiple operations at once changes the way of processing. Instead of reading the nodes of level two and three alternately, the batched reads operation reads only the second level nodes at first, followed by the third level nodes. This gives us the advantage that nodes are potentially still in the cache when reading them multiple times. Moreover, we can issue prefetch instructions to bring the third level nodes in the cache while still reading level two nodes. The disadvantage of batching reads is an increased latency, which is traded for a higher throughput. However, certain database operations, like joins, are able to profit a lot from batch processing.

5. MEMORY MANAGEMENT

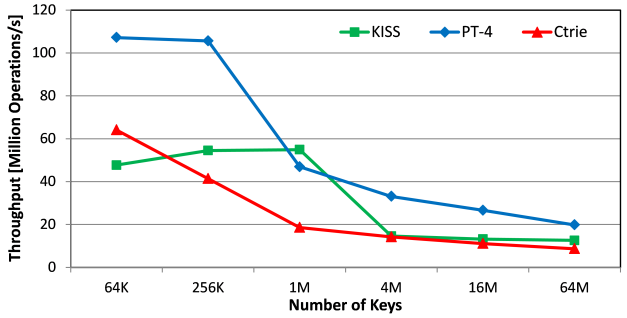
The memory management subsystem is tightly coupled to the *KISS-Tree*, because it is a critical component regarding the overall index performance. Especially the RCU mechanism deployed on the third level requires a responsive memory allocation. During the startup of the system, the memory management allocates consecutive virtual memory segments (physical memory is claimed on-demand) for each of the 64 node sizes possible on the third level as depicted at the bottom of Figure 4. Each segment consists of a maximum of 2^{26} blocks of the respective node size. We need the blocks of



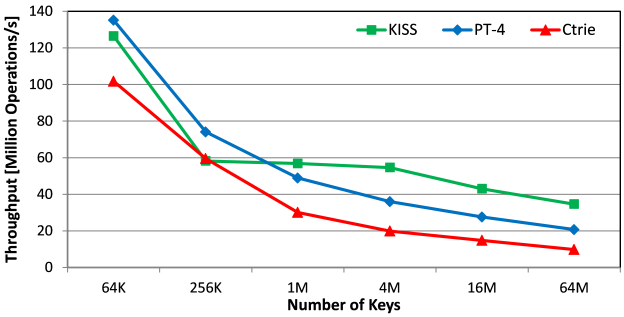
(a) Throughput for Reading Sequences.



(b) Throughput for Reading Uniform Data.



(c) Throughput for Updating Sequences



(d) Throughput for Updating Uniform Data

Figure 5: Read/Update Throughput for Sequences and Uniform Data.

each size to be in a row, because of the block-oriented compact pointers used on level two. In order to allow fast parallel access to the memory management subsystem, each thread allocates only a small number of blocks from each segment of the global memory and maintains them on its own (thread-local memory). This allows threads to administrate their memory independent from each other and removes synchronization bottlenecks. Only if a thread runs out of memory, it requests a new set of blocks from the global memory, which is synchronized via atomic operations. Moreover, every thread uses its own free lists for memory recycling. Those free lists have to be aware of the RCU mechanism, in order to prevent the recycling of a node that is still read by another thread. For our experiments, we used an active and an inactive free list per node size and thread. Freshly freed memory pieces are stored in the inactive free list and memory allocation requests are served by the active one. As soon as a *grace period* is detected, the memory management switches over to the inactive free list.

6. EVALUATION

In this section, we evaluate the *KISS-Tree* performance and memory usage for different workloads, tree sizes, update rates, and platforms. Moreover, we compare the results to the generalized prefix tree and the CTrie. We used 32bit keys and 64bit values as they are common for rids. All experiments, we conducted, were executed on an *Intel i7-2600 (3.4GHz clock rate, 3.8GHz max. turbo frequency, 4 cores, Hyper-Threading, 2 memory channels, 8MB LLC)* equipped with *16GB RAM (DDR3-1333)* running *Ubuntu Linux 11.10*.

In the first experiment, we compared the read/write throughput of the *KISS-Tree*, generalized prefix tree with

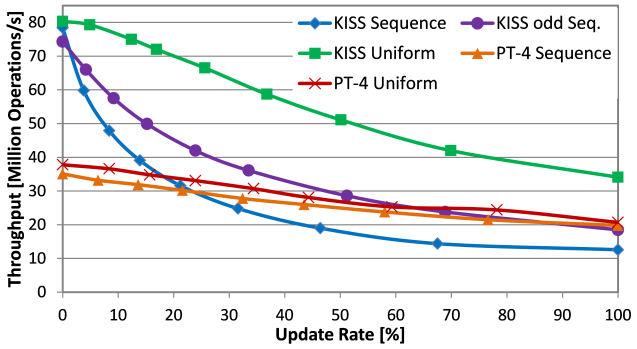


Figure 6: Throughput for different Update-Rates.

$k' = 4$ (PT-4), and CTrie for different tree sizes. We used all of the eight available hardware threads on the platform. For all experiments, we use a sequence and a uniformly distributed workload. Both workloads describe the upper respectively lower boundary for throughput and for instance, skewed distributions lie in between. The tree size reaches from 64 thousand (compute-bound) up to 64 million (memory-bound) keys present in the trees. All the three tree implementations are using batching for read operations as described in Section 4.3. Figure 5(a) shows the read throughput for sequences (keys are randomly picked from the sequence range). The *KISS-Tree* shows the highest throughput for small as well as for large trees. Figure 5(b) visualizes the throughput for uniformly distributes keys. Here, the *KISS-Tree* also shows the best performance, but the throughput for small trees is lower because the locality

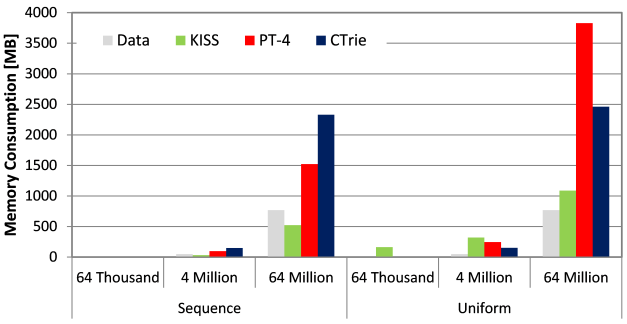


Figure 7: Comparison of Memory Consumptions.

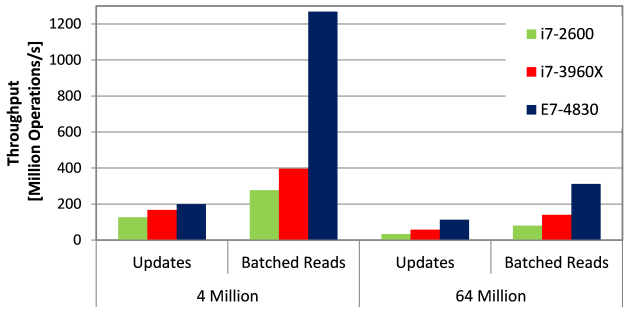


Figure 8: Throughput on different Platforms.

of data got worse. The main observation is that the *KISS-Tree* with the least memory accesses per read operation(2-3) has the highest throughput compared to the CTrie which requires up to 14 memory accesses. Figure 5(c) and Figure 5(d) show the update throughputs for both key distributions. The CTrie shows the worst results because of the high number of memory accesses per key. We measured the highest throughput for the PT-4, because it is able to do in-place updates where the *KISS-Tree* has to create copies of third level nodes. We see this effect when comparing the sequence workload to the uniform workload. With the sequence workload, all third level nodes contain 64 values that have to be copied for an update. The uniform workload on the other hand creates only sparely used nodes on the third level, which results in a better update performance. Moreover, we observe a lot of compare-and-swap failures on small trees for the sequence workload, because threads work on the same third level nodes. To summarize the first experiment, the *KISS-Tree* clearly shows the best read performance and an update performance that is able to keep pace with the CTrie and the PT-4.

With the second experiment, we investigate the overall throughput for workloads with different update rates on the *KISS-Tree* and the PT-4. Figure 6 shows the respective results. While the throughput of the PT-4 moves smoothly between 0% and 100% update rate, the behaviour of the *KISS-Tree* heavily depends on the key distribution. This is mainly caused by the RCU update mechanism, which has to copy entire nodes and so occupies the memory channels what also effects the other read operations. The one extreme is the uniform workload that does not require much overhead for copying nodes. Thus, the *KISS-Tree* is always faster than the PT-4. As worst case we identified the sequence

Processor	#threads	#channels	LLC(MB)	Freq.(GHz)
1x i7-2600	8	2	8MB	3.4
1x i7-3960X	12	4	15MB	3.3
4x E7-4830	4x16	4x4	4x24MB	2.13

Table 1: Evaluation Hardware

workload, where update operations always have to copy 64 values. Here, the overall throughput drops below the one of the PT-4 at an update rate of about 20%. To show the behavior between both extremes, we added the odd sequence key distribution, which only includes odd keys. This ends up in 32 values per third level node and a throughput that is better or equal to the PT-4.

The next experiment addresses the memory consumption of the *KISS-Tree*, PT-4, and CTrie for different key distributions and tree sizes. Figure 7 visualizes the measurements. For the sequence workload, the *KISS-Tree* turns out to have a very low memory consumption that is almost equal to the actual data size. The CTrie shows the highest memory usage, because the compression and the *I-Nodes* add additional overhead to the structure compared to the PT-4. When looking at the measurements for the uniform workload, the *KISS-Tree* wastes a lot of memory on small trees. This is caused by the second level nodes that are allocated at a 4KB granularity. For instance, a *KISS-Tree* that contains 4 million uniformly distributed keys uses 256MB on the fully expanded second level. This is the worst case scenario for memory consumption; with an increasing number of keys, the *KISS-Tree* starts to save a lot of memory compared to the other trees. The measurements for the uniform workload also show the worst case scenario for the uncompressed PT-4, which consists of very sparely used nodes. Here, the compression of the CTrie saves memory.

Finally, our last experiment investigates the performance of the *KISS-Tree* on the three different platforms listed in Table 1. Figure 8 contains the measurements for uniformly distributes keys. The experiment shows that the read performance scales with the total number of hardware threads and the clock rate. A problem we observed is the update performance on the massive parallel NUMA system. With 64 hardware threads updating a small tree, we have a high probability that threads try to update the same third level nodes. For this reason, updating a small *KISS-Tree* does not scale well on massive parallel hardware. As soon as the tree becomes larger, the scalability is given, because threads work on different nodes.

7. CONCLUSION AND FUTURE WORK

With the movement from disk-based database systems towards pure in-memory architectures, the design goals for index structures essentially changed. The new main optimizations targets are cache-awareness, memory access minimization and latch-freedom. Thus, classic disk-block optimized structures like B+-Trees are not suited for modern in-memory database architectures anymore. In this paper, we introduced the *KISS-Tree*, which addresses exactly these kinds of optimizations. With the *KISS-Tree*, we accomplished to reduce the number of memory accesses to 2–3 by deploying multiple techniques on the individual tree levels. The techniques are *direct addressing*, *on-demand allocation*,

compact pointer and *compression*. The evaluation revealed that the minimization of memory accesses is the most beneficial optimization for large indexes. In combination with latency hiding through batched reads, we were able to outperform existing tree-based in-memory index structures. Moreover, the *KISS-Tree* provides a high update rate, because it does not involve comprehensive structural maintenance and exploits parallelism with the help of optimistic compare-and-swap instructions. Regarding memory consumption, we showed that the *KISS-Tree* has very low indexing overhead, especially for large trees, and even consumes less memory than the actual data for sequential workloads.

Our future work will mostly focus on improving the update performance and adding support for larger keys. The critical point for the update performance is that we have to copy entire nodes for existing keys. This occupies the memory channels and increases the probability for the compare-and-swap to fail. To fix this issue, we will have to make changes in the second tree level. One possible solution for this problem would be the inclusion of version numbers into the compact pointers on the second level, so that each update changes the pointer. This way, we increase the update throughput that is independent of the workload.

The current main drawback of this *KISS-Tree* is, that it misses support for large integers or varchars. Therefore, we have to look for solutions to apply other techniques on the additionally required levels or consider combinations with other index structures. We expect those additional levels to deploy compression mechanisms for sparely occupied nodes as well as a vertical compression of node chains. Another important point for research is to push the performance on NUMA systems. NUMA architectures are completely different from SMP machines because there are additional costs for accessing memory on foreign sockets and especially the cache utilization on different sockets. Moreover, the cache-coherency has to be taken into account.

8. ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing”.

9. REFERENCES

[1] Dexter project. `http://wwwdb.inf.tu-dresden.de/dexter`.

[2] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes*, pages 245–262. Software pioneers, New York, NY, USA, 2002.

[3] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *BTW*, pages 227–246, 2011.

[4] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, Apr. 2002.

[5] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[6] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[7] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6:650–670, December 1981.

[8] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. VLDB ’86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[9] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems.

[10] A. Prokopec, P. Bagwell, and M. Odersky. Lock-free resizeable concurrent tries. *LCPC*, 2011.

[11] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD Rec.*, 29:475–486, May 2000.

[12] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.

[13] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

APPENDIX

In the appendix, we conducted as set of experiments that investigate the behavior of batched reads and scalability as well as a comparison to the CSB+-Tree.

A. BATCHED READS EVALUATION

In this section, we evaluate the performance of batched reads. Compared to standard read operations, batched reads process multiple read operations per thread at once to hide the latency of memory accesses. All of the following experiments were conducted on an *Intel i7-2600* processor, which corresponds to the machine used in Section 6.

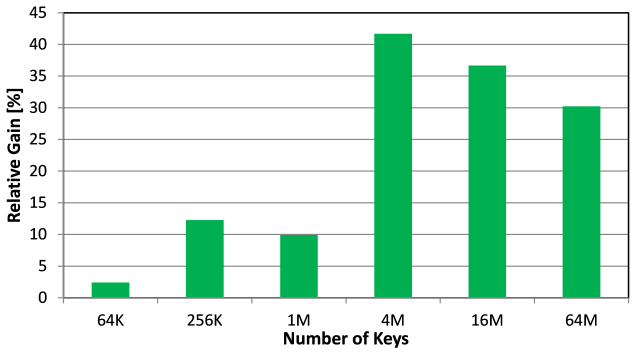


Figure 9: Relative Performance Gain through Batched Reads for Sequences.

In first two experiments, we measured the throughput of standard read operations and batched read operations for different tree sizes. Figure 9 shows the relative performance gain of batched reads over standard reads for sequential keys (keys randomly picked from the sequence range). We observe that this performance gain strongly depends on to size of the tree. A small tree, for instance, takes not much advantage from batched reads, because most of the data is in

the L1 cache of the CPU. Thus, we are unable to hide any latency. As soon as the L1 cache is exhausted, batched reads are able to hide latency between the L1 cache and the L2 cache. With more and more growing tree sizes, we hit the point where even the L2 cache and the LLC are exhausted. Thus, we have a high latency from the memory controller to the L1 cache, which gives us the highest performance gain for batched reads.

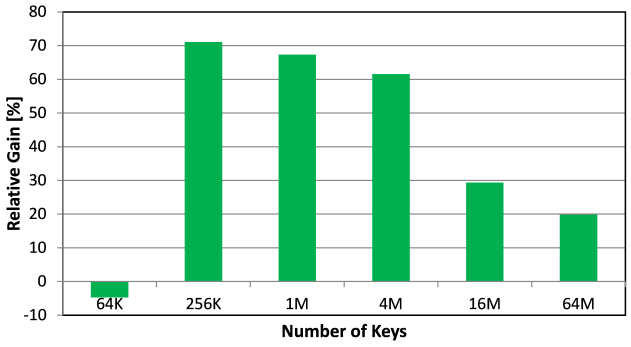


Figure 10: Relative Performance Gain through Batched Reads for Uniform Data.

Figure 10 visualizes the measurements for a uniform key distribution. On small trees that fit in the L1 cache of the CPU, we observe a performance loss by batching reads because prefetch instructions induce an additional overhead. When looking at trees of medium size, the performance gain is higher compared to the sequential keys. This is the case because prefetch instructions are unlikely to fetch same cache lines twice. Thus, batched reads are able to hide more latency.

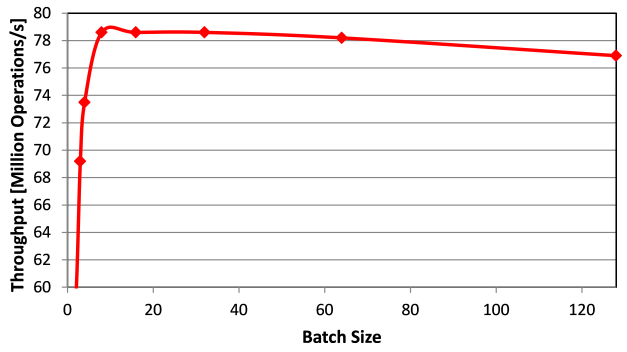


Figure 11: Throughput as a Function of Batch Size.

In the final experiment, we measured the throughput for different batch sizes. The batch size is the number of read operations that are simultaneously processed by a thread. The results are depicted in Figure 11. The experiment revealed that even small batch sizes dramatically increase the read throughput. With a batch size of eight, batched reads achieve the maximum throughput, which remains steady until it starts to slowly decrease with large batch sizes.

B. SCALABILITY

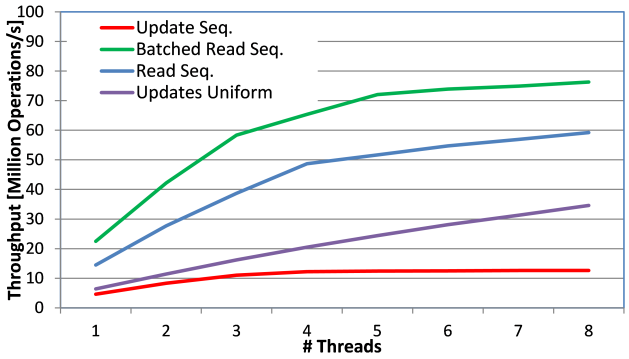


Figure 12: Scalability for Sequences (64M Keys).

To investigate the scalability of the *KISS-Tree*, we measured the throughput of different operations with a variable number of threads. The hardware we used for this experiment was an *Intel i7-2600*, which has four physical cores with Hyper-Threading (a total of 8 hardware threads) available. The experiment depicted in Fig. 12 revealed that a standard read operation scales with the number of physical cores. As soon as the physical cores are exhausted, the performance gain drops, because threads have to share cores. We observe another behavior for batched reads, which scale only well with the first three threads. This happens, because the batched operations have a better utilization of the memory controllers, which become the bottleneck with an increasing number of threads. The same effect can be observed, when comparing updates of sequential and uniform data. Because the sequential workload requires the copying of full nodes on the third level, the memory controllers become the limiting factor. However, the update operation of the uniform workload scales well with all available hardware threads.

C. COMPARISON TO THE CSB+-TREE

In this section, we compare the *KISS-Tree* to the CSB+-Tree implementation from [11]. This 32bit CSB+-Tree implementation neither implements concurrency control nor batched reads. Therefore, we measured the single-threaded throughput for simple uniformly distributed reads on an *Intel i7-2600*. The CSB+-Tree implementation performs about 3 million operations per second and thread with a tree size of 64 million keys. The *KISS-Tree* achieves 18.4 million operations per second and thread, which is more than 6 times faster.