Adapting Tree Structures for Processing with SIMD Instructions

Steffen Zeuch Frank Huber* Johann-Christoph Freytag
Humboldt-Universität zu Berlin
{zeuchste,huber,freytag}@informatik.hu-berlin.de

ABSTRACT

In this paper, we accelerate the processing of tree-based index structures by using SIMD instructions. We adapt the B^+-Tree and prefix B-Tree (trie) by changing the search algorithm on inner nodes from binary search to k-ary search. The k-ary search enables the use of SIMD instructions, which are commonly available on most modern processors today. The main challenge for using SIMD instructions on CPUs is their inherent requirement for consecutive memory loads. The data for one SIMD load instruction must be located in consecutive memory locations and cannot be scattered over the entire memory. The original layout of tree-based index structures does not satisfy this constraint and must be adapted to enable SIMD usage. Thus, we introduce two tree adaptations that satisfy the specific constraints of SIMD instructions. We present two different algorithms for transforming the original tree layout into a SIMD-friendly layout. Additionally, we introduce two SIMD-friendly search algorithms designed for the new layout.

Our adapted B^+-Tree speeds up search processes by a factor of up to eight for small data types compared to the original B^+-Tree using binary search. Furthermore, our adapted prefix B-Tree enables a high search performance even for larger data types. We report a constant 14 fold speedup and an 8 fold reduction in memory consumption compared to the original B^+-Tree.

1. INTRODUCTION

Since Bayer and McCreight introduced the B-Tree [3] in 1972, it has been adapted in many ways to meet the increasing demands of modern index structures to manage higher data volumes with an ever decreasing response time. The B-Tree combines a fixed number of data items in nodes and relate them in a tree-like manner. Each data item contains a key and its associated value. In the past, many variants of the original B-Tree envolved which differ, among other aspects, in the restrictions of allowed data items per node and the kind of data each node stores. The most widely used variant of the B-Tree is the B^+-Tree. The B^+-Tree distinguishes between leaf and branching nodes. While leaf nodes store data items to form a so-called *sequence set* [8], branching nodes are used for pathfinding based on stored key values. Thus, each node is either used for navigation or for storing data items, but not for both purposes like in the original B-Tree. One major advantage of the B^+-Tree is its ability to speedup sequential processing by linking leaf nodes to support range queries.

As a variant of the original B-Tree, Bayer et al. introduced the prefix B-Tree (trie) [4], also called digital search tree. Instead of storing and comparing the entire key on each level, a trie operates on parts of the key by using their digital representation. The keys are implicitly stored in the path from the root to the leaf nodes. When inserting a new key, the key is split into fix sized pieces and distributed among the different trie levels. Because the key length and the partial key length is defined during initialization statically, the height of a trie is invariant. The fixed height distinguishes a trie from other tree structures which grow and shrink dynamically. The fixed height of a trie changes the complexity of finding a key. Whereas finding a key in a B^+-Tree is $O(\log N)$, the worst-case time complexity of a trie is $O(1)$ for all operations, independent of the number of records in the trie. Furthermore, a trie may terminate the traversal above leaf level if a partial key is not present on the current level. Additionally, splitting keys allows prefix compression at different trie levels. However, a trie structure with its predefined parameters results in a more static structure compared to a dynamically growing and shrinking B^+-Tree. The existing trie-based structures are mainly used for string indexing [10]; however, indexing of arbitrary data types is also possible [7].

An important performance factor for all tree structures is the number of keys per node. As one node is usually mapped onto one page on secondary storage, one page is copied by one I/O operation into main memory. Thus, I/O operations are the most time-consuming steps in processing a B-Tree. Other steps that are CPU intensive are usually negligible in the presence of I/O operations. With a node as the transport unit within the storage hierarchy, it is important to realize that processing will be faster the more keys fit into one node. This observation has been true for the era of disk-based databases; it also holds nowadays for main-memory based databases. That is, the bottleneck between secondary storage and main memory has now been moved to a bottleneck between main memory and CPU caches [17].

*Current address: frank.huber@sap.com

10.5441/002/edbt.2014.10

The link between two levels of the storage hierarchy will be the bottleneck if the node size is greater than the amount of data that can be transferred in one step. Therefore, the node size in a disk-based database is determined by the I/O block size and in a main-memory database by the cacheline size [13, 14]. In this paper, we focus on main memory databases. We assume, the complete working set fits into main memory, and exclude I/O impact at all. For the remainder of this paper, we focus on the new bottleneck between main memory and CPU caches.

As mentioned before, a performance increase for a tree structure might result from storing more keys in one node. An increased node size is less applicable for optimization because the node size is mainly determined by the underlying hardware. Furthermore, the size of a pointer and the provided data types are also hardware specific. On the other hand, the number of keys and pointers within one node are adjustable. For example, the B^+-Tree moves associated values to leaf nodes. Therefore, the branching nodes are able to store more keys which accelerates the traversal speed by increasing the fanout. Other approaches for storing more keys in one node apply prefix or suffix compression techniques on each key [4]. The compression of stored pointers is also possible [18]. One disadvantage of compression is the additional computational effort. The question if compression overhead is rewarding for performance is beyond the scope of this paper.

Searching inside a node has been discussed extensively as well. Suggested search strategies range from sequential over binary to exploration search [8]. We contribute a new search strategy for inner node search based on the *k-ary search* algorithm. This algorithm uses single-instruction-multiple-data (SIMD) for comparing multiple data items in parallel [16]. We adapt the B^+-Tree and the prefix B-Tree structure for the k-ary search algorithm. The adapted B^+-Tree performs well on small data types, i. e., data types that use up to 16 bits for value representation. To improve k-ary search performance for larger data types, we also adapt the prefix B-Tree. Both tree adaptions make SIMD instructions applicable for tree-based index structures in modern database systems.

In the light of this discussion the contributions of this paper are as follows:

1. We adapt the B^+-Tree and the prefix B-Tree for SIMD usage by incorporating k-ary search.
2. We compare both adaptions and derive their suitability for different workloads.
3. We present a transformation and a search algorithm for a breath-first and depth-first data layout.
4. We contribute three algorithms for interpreting a SIMD comparison result.

The remainder of this paper is structured as follows. Section 2 covers preliminaries of our work. First, we discuss the SIMD chipset extension of modern processors and their opportunities. Furthermore, we outline the *k-ary search* idea as the foundation for our work. Sections 3 and 4 cover our adaption of a B^+-Tree (called *Segment-Tree*) and prefix B-Tree (called *Segment-Trie*) using k-ary search. The evaluation of our tree adaptions is presented in Section 5. In Section 6, we discuss related work. Finally, we conclude and mention future work in Section 7.

2. PRELIMINARIES

This section presents our basic approach of comparing two keys using SIMD instructions. We start in Section 2.1 by introducing SIMD as a modern chipset extension that enables parallel comparisons. Based on SIMD, we describe the *k-ary search* in Section 2.2 as our starting point for later B-Tree adaptions.

2.1 Single Instruction Multiple Data

With SIMD, one instruction is applied to multiple data items in parallel. This technique enables data-level parallelism that arises from concurrent processing of many independent data items. Two parameters determine the degree of parallelism. The *SIMD bandwidth* or size of a SIMD register in bits determines the number of bits that can be processed in parallel by one instruction. The *data type* defines the size of one data item in a SIMD register. Therefore, the data type limits the number of parallel processed data items. For example, a 128-bit SIMD register processes sixteen 8-bit or eight 16-bit data items with one instruction.

The SIMD instructions are chipset dependent and differ among various computer architectures. For example, Intel offers a wide range of arithmetical, comparison, conversion, and logical instructions [2]. We use SIMD comparison instructions for our tree adaptions. A SIMD comparison instruction splits a SIMD register into segments of fixed size. The segment size is determined by the used SIMD instruction, e. g., 8, 16, 32, or 64-bit. The comparison is performed for all segments in parallel with the corresponding segment in another SIMD register. For the remainder of this paper, we refer to one data item in a SIMD register as one *segment*. Notice, that data items must be stored in consecutive memory locations to be loaded in one SIMD load instructions.

We use SIMD comparison instructions to speedup the inner node search in a tree structure; the most time consuming operation. Therefore, we need to compare a search key v with a sorted list of keys inside a tree node. Following Schlegel et al. [16], our instruction sequence for comparing a search key with a sorted list of keys contains five steps:

1. Load keys segment-wise in register $R1$.
2. Load search key v in each segment of register $R2$.
3. Run pairwise comparison for each segment.
4. Save the result as a bitmask.
5. Evaluate the bitmask.

Unfortunately, SIMD instructions do not provide *conditional or branching statements* [2]. Since all operations are performed in parallel, there is no possibility to check individual values and branch to specific code. Therefore, the result of a comparison of two SIMD register is a *bitmask*. The bitmask indicates the relationship between the search key v and the list of keys. For the remainder of this paper, we use the greater-than relationship for comparisons. By evaluating the bitmask, we get a position in the sorted list of keys. This position indicates the first key that is greater-than the search key v. In a tree structure, this position identifies the pointer which leads to the next child node.

Our implementation of the aforementioned sequence for a 32-bit data type is illustrated in Figure 1. First, we load a list of keys into a 128-bit SIMD register by using

Table 1: Used SIMD instructions from Streaming SIMD Extensions 2 (SSE2).

SIMD instruction	Explanation
`__m128i _mm_load_si128 (__m128i *p)`	Loads a 128-bit value. Returns the value loaded into a variable representing a register.
`__m128i _mm_set1_epi32 (int i)`	Sets 4 signed 32-bit integer values to `i`.
`__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)`	Compares 4 signed 32-bit integers in `a` and 4 signed 32-bit integers in `b` for greater-than.
`__mm_movemask_epi8 (__m128i a)`	Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in `a` and zero extends the upper bits.

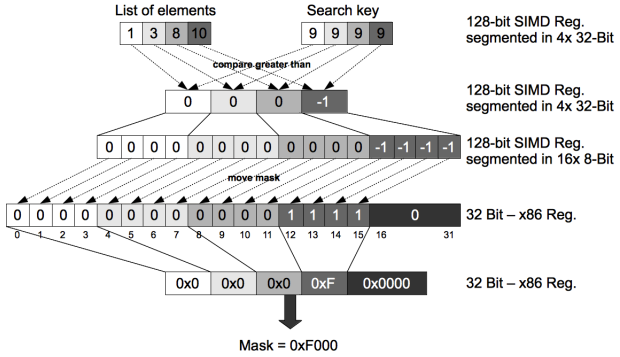

Figure 1: A sequence using SIMD instructions to compare a list of keys with a search key.

the `__mm_load_si128` instruction. After that, we load the search key $v = 9$ into each 32-bit segment of a second 128-bit SIMD register with `__mm_set1_epi32`. The pairwise greater-than comparison of each segment is executed by `__mm_cmpgt_epi32`. This instruction compares each 32-bit segment in both input registers and outputs -1 into the corresponding segment of a third 128-bit SIMD register if the key is greater than the search key, otherwise zero. To create a bitmask as the result of the comparison, we use `__mm_movemask_epi8` to extract the most significant bit from each 8-bit segment. The sixteen extracted bits are stored in the lower 16 bits of an x86 register. Unlike a SIMD register, a x86 register provides conditional and branching statements like *if*. Table 1 describes the used SIMD instructions with `__m128i` as a 128-bit SIMD data type.[1]

The resulting bitmask must be evaluated to determine the position of the search key within the sorted list of keys. We exploit a particular property of the *greater-than* comparison for the evaluation. When evaluating the bitmask linearly from left to right, the first key that is greater than the search key represents a *switch* point. After this point, all subsequent keys are greater than the search key and thus represented with a one in the bitmask. With this property in mind, we introduce three algorithms for bitmask evaluation. Notice, that the upper 16 bits are ignored for our evaluation. Algorithm 1 uses a loop to check if the least significant bit in each segment is set. For simplicity, we omit the case that the evaluation might terminate if we found the first greater key. In this case, we calculate the position assuming that only greater keys will follow. c denotes the number of segments in a SIMD register that is defined by the used data type and the SIMD bandwidth. Algorithm 2 implements a switch statement for each possible bitmask of a 32-bit segment size in a 128-bit SIMD register. Algorithm 3 uses the `popcnt` instruction to return the number of bits set in a register.

[1]http://msdn.microsoft.com/en-us/library/

Algorithm 1 Bit Shifting

```
mask ← bitmask
c ← number of segments
position ← 0
for i = 0 → c do
    position += mask & 0x01
    mask >>= c
end for
return c − position
```

Algorithm 2 Switch Case

```
mask ← bitmask
position ← 0
switch mask do
    case 0xffff
        position ← 0
        break
    case 0xfff0
        position ← 1
        break
    case 0xff00
        position ← 2
        break
    case 0xf000
        position ← 3
        break
return position
```

Algorithm 3 Popcnt

```
mask ← bitmask
c ← number of segments
shift ← 16/c
return c − _popcnt(mask)/shift
```

By evaluating the resulting bitmask $0xF000$ in Figure 1 using one of the three algorithms, we get the third position as a result. Therefore, the first key in the sorted list of keys that is greater than the search key v is located at position three. Note, the positioning starts at zero.

The aforementioned sequence utilizes four different SIMD instructions. The *load* and *set* instructions load keys in SIMD register. *Set* is a composite instruction containing one load instruction for moving a value into one segment and an additional instruction for copying the value to the other segments. The *comparison* instruction compares two SIMD register and the *movemask* instruction moves the resulting bitmask into a x86 register. Modern processors of Intels *Nehalem* or *Sandy Bridge* microarchitecture are able

to perform one SIMD load or comparison instruction in each CPU cycle resulting in one *cycle per instruction* (CPI) [2]. However, Intel does not provide CPI information for composite instructions. In our sequence, we perform the set instruction only once to load the search key. Therefore, we exclude the set instruction from the following considerations of a simplified runtime estimation on instruction level. We compare our SIMD sequence against the common approach using scalar instructions. First, the SIMD load and comparison instructions are as fast as similar scalar instructions operating on x86 registers. This leads to an increased *instructions per cycle* (IPC) rate because SIMD increases the number of parallel-executed instructions without introducing additional latency. However, the second step of evaluating the comparison result differs in terms of performed instructions. A sequence using scalar instructions performs conditional jumps depending on the status flags in the *EFLAGS* register. In contrast, our SIMD sequence performs one *movemask* instruction in two CPU cycles to extract a bitmask from the comparison result. After that, the bitmask is evaluated using one of the previously introduces bitmask evaluation algorithms. Section 5 will show, that despite the additional effort for bitmask evaluation, our SIMD sequence is still faster than a scalar instruction sequence.

Current SIMD extensions of modern processors support SIMD comparison instructions only for signed data types [2]. To use SIMD comparison instructions for unsigned data types, we implement a preceding subtraction by the maximum value of the signed data type. Therefore, we realign the unsigned value to a signed value. For example, the value zero of an 8-bit unsigned integer data type is realigned to -128. The value 256 is realigned to 127. With this preceding subtraction, we are able to use the signed SIMD comparison instructions for unsigned data types. The value must be realigned by insert and search operations.

2.2 k-ary Search

The k-ary search, introduced by Schlegel et al. [16], is based on binary search. The binary search algorithm uses the divide-and-conquer paradigm. This paradigm works iteratively over a sorted list of keys A by dividing the search space equally in each iteration. Thus, the algorithm first identifies the median key of a sorted list of keys. The median key serves as a separator that divides the search space in two equally sized sets of keys (so-called *partitions*). The left partition only contains keys that are smaller than the median key. The keys in the right partition are larger than the median key. After partitioning, the search key v is compared to the median key. The search terminates if the search key is equal to the median key. Otherwise, the binary search uses the left or right partition, depending on the greater-less relationship, as the input for the next iteration. In case of an empty partition, search key v is not in A and the search terminates. For an key count n, the complexity is logarithmic and performs $h = log_2 n$ iterations in the worst case and $h-(2^h-h-1)/n > h-2$ on average [16]. Figure 2 illustrates the binary search for $v = 9$ on a sorted list of 26 keys. The boxed keys form one partition.

While the binary search algorithm divides the search space into two partitions in each iteration, the k-ary search algorithm divides the search space into k partitions by using $k-1$ separators. We utilize our aforementioned SIMD sequence to create this increased number of partitions and

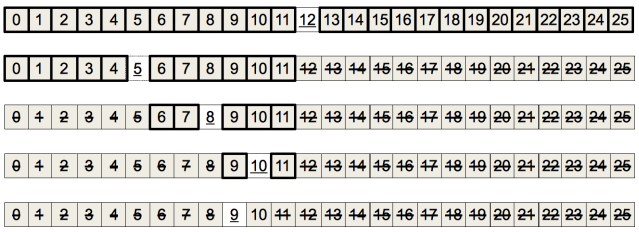

Figure 2: Binary search for key 9 and n = 26.

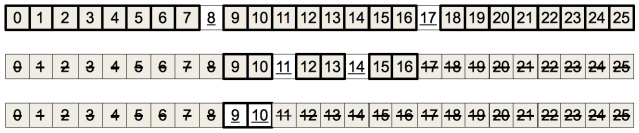

Figure 3: K-ary search for key 9, n = 26 and k = 3.

separators. As shown in Section 2.1, SIMD instructions are able to compare a list of keys with a search key in parallel. The number of parallel key comparisons depends on the data type and the available SIMD bandwidth. With parameter k, $k-1$ separator keys are compared in one iteration which increases the number of partitions to k. Figure 3 illustrates the same search as in Figure 2 now using k-ary search. The binary search compares only one key at a time with a search key; thus producing two partitions. In contrast, the k-ary search with $k = 3$ compares two keys in parallel with a search key and divides the search space into three partitions. As a result, the k-ary search terminates after three iterations while the binary search needs five iterations to find the search key. Generally, the k-ary search reduces the complexity to $O(log_k(n))$ compared to $O(log_2(n))$ for binary search. Assuming a commonly available SIMD bandwidth of 128-bit and a data type of 8-bit results in $k = 17$. With $k = 17$, sixteen 8-bit values are compared in parallel within one SIMD instruction. Therefore, the number of iterations is reduced by a factor of $\frac{log_2(n)}{log_k(n)} = log_2(k) \approx 4$. Table 2 illustrates the relationship between common data types and maximal supported k values in a 128-bit SIMD register.

The main restriction of SIMD instructions is their requirement for a sequential load of data. This requirement presupposes, that all keys that are loaded into one SIMD register with one SIMD instruction must be stored consecutively in main memory. Load or store instructions using scatter and gather operations that allows a load/store of keys from distributed memory locations are not supported on CPUs [2]. The keys in a sorted list are placed one key next to the other in linear order as shown in Figure 3. Therefore, the keys are placed depending on their relationship to each other, e. g., in ascending or descending order. This placement strategy is sufficient for binary search, but not amenable to k-ary search. In a linear sorted list of keys, possible separator keys are not placed in consecutive memory locations because sev-

Table 2: k values for a 128-bit SIMD register.

Data type	k value	Parallel comparisons
8-bit	17	16
16-bit	9	8
32-bit	5	4
64-bit	3	2

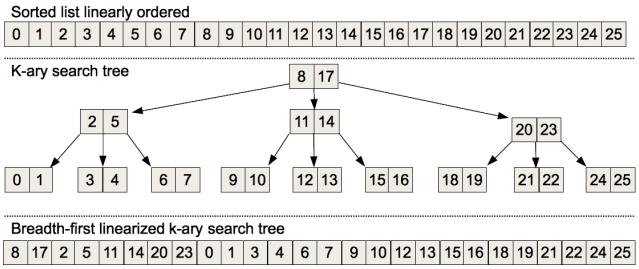

Figure 4: Breadth-first transformation overview.

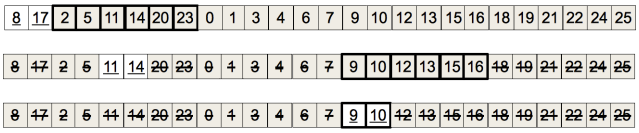

Figure 5: K-ary search for key 9 on a breadth-first linearized tree, n = 26 and k = 3.

eral keys fall in between. For example, the keys 8 and 17 in Figure 3 may be chosen as separators to partition the sorted list in three equally sized partitions. After that, the separators and the search key must be compared to determine the input for the next iteration. When storing the list of keys in linear order, the separator keys are not placed next to each other in main memory and thus cannot be loaded with one SIMD instruction. To overcome this restriction, Schlegel et al. [16] suggest to build a k-ary search tree from the sorted list of keys. They define a perfect k-ary search tree as: *"[. . .] every node – including the root node – has precisely $k-1$ entries, every internal node has k successors, and every leaf node has the same depth."*.

The k-ary search tree is a logical representation that must be transformed for storage in main memory or on secondary storage. For this transformation, Schlegel et al. propose to *linearize* the k-ary search tree. The linearization procedure transforms a sorted list of keys into a linearized k-ary search tree. Figure 4 summarizes the transformation process. As a result, both separator keys are placed side by side and thus can be loaded with one SIMD instruction. In Section 3.2, we present two algorithms that use depth-first search or breath-first search for this transformation. Figure 5 illustrates a k-ary search for search key $v = 9$ on a breadth-first linearized k-ary search tree. The white boxes show the separators and the block boxed the partitions. In the next section, we adapt the B^+-Tree for k-ary search usage.

3. THE SEGMENT-TREE

In this section, we present our *Segment-Tree (Seg-Tree for short)* that implements the k-ary search algorithm for inner node search in a B^+-Tree. In Section 3.1, we adapt the basic structure and show the implications for the traversal algorithm. Section 3.2 presents two algorithms for linearizing a sorted list of keys. In Section 3.3, we address the essential ability of supporting arbitrary sized search spaces. Finally, Section 3.4 analyzes the performance of the Seg-Tree.

3.1 Using k-ary Search in B$^+$-Trees

Our Seg-Tree uses the k-ary search algorithm for inner node search in a B^+-Tree. We consider each node as a k-ary search tree—an important aspect when updating a node. Thus, the effect of an update operation is limited to one node. This locality property eliminates the need for rebuilding the complete Seg-Tree for each update operation. The traversal across the nodes from the root to the leaves keeps unchanged compared to B^+-Trees. Furthermore, the split and merge operations in case of a node overflow or underflow are unaffected. Our approach changes the search method inside the nodes from commonly binary search to k-ary search. We store one array for n keys and one array for $n+1$ pointers inside each node.

For the rest of this paper, let D_m define a data type with at most m bits for representing its values; $|SIMD|$ denotes the SIMD bandwidth. Furthermore, let k denote the order of the k-ary search tree with $k = \frac{|SIMD|}{m} + 1$ pointers and $k-1$ keys in each node. The number of levels in the k-ary search tree is determined by $r = \lceil \log_k n \rceil$ with n being the number of keys in the sorted list. The maximum number of keys in one node is bound by $N-1$ with $N = k^r$.

Algorithms 4 and 5 implement the sequence of SIMD instructions for comparing a search key v with a sorted list of keys (see Section 2.1). Based on the linearization method, either one of these two algorithms can be used. Algorithm 5 performs a search on a breadth-first linearized list of keys in one Seg-Tree node. On each level of the k-ary search tree, $k-1$ keys are compared to a search key using SIMD instructions. The bitmask on each level is evaluated to a position using one of the bitmask evaluation Algorithms 1, 2, or 3. The resulting position will be incrementally built up during the search process and is additionally used to determine the offset for the next lower level. After the search on each level completes, a lookup into the pointer array using the returned position determines the path to the next node. Algorithm 4 is implemented in a similar way for searching on a depth-first linearized list of keys.

We refer to Figure 4 as a breadth-first linearized node in a Seg-Tree. The node contains $n = 26$ 64-bit keys. A $|SIMD|$ bandwidth of 128-bit leads to $k = 3$. The height of the k-ary search tree is determined by $r = \lceil \log_3 26 \rceil = 3$ and the maximum number of keys $N-1$ is 26 since $N = 3^3 = 27$. Consider a search for key $v = 9$ using Algorithm 5 within this node. R denotes a SIMD register containing the search key in each segment (Line 6). C denotes a SIMD register storing $k-1$ keys. First, the algorithm determines the key pointer $keyPtr$ in Line 8. Initially, $keyPtr$ points to the first key in the key array (Line 3). The algorithm loads $k-1$ keys via this pointer in a SIMD register (Line 11). During the first iteration, the node (8,17) is loaded and compared to search key $v = 9$ (Line 12). The resulting bitmask is evaluated in Line 13-14. The returned position 1 is added to $pLevel$ in Line 17. After that, we determine the base pointer for the next iteration in Line 18. The base pointer refers to the left most node on the next lower level (2,5). In the second iteration, we add an offset depending on $pLevel$ of the previous iteration to the base pointer in Line 8. This offsets the $keyPrt$ to the desired node (11,14) and the SIMD comparison sequence in Line 10-16 returns zero. For the last iteration, we set the base pointer to the left most node on the last level (9,10). This node represents the desired node and no offset must be added. The SIMD comparison sequence in Line 10-16 returns zero. Finally, we return $pLevel$ in Line 21. $pLevel = 9$ was incrementally built over all iterations and selects the first key in the Seg-Tree node that is greater than

search key $v = 9$. Note, that *pLevel* is equal to the search result of a binary search on the same list of keys. Therefore, the navigation to the next Seg-Tree node is similar to the original B^+-Tree navigation. In case of a branching node, we follow the pointer at position 9 to a child node on the next level that contains values smaller or equal to search key v. For a leaf node, we would perform an additional comparison for equality to located the associated value for search key v.

Generally, Algorithms 4 and 5 search on a linearized list of keys but returning the position as if the keys are in linear sorted order. Therefore, only the keys in the k-ary search tree must be linearized; pointers are left unchanged. Due to this important property, an update operation does not affect the pointer array. This property also impacts the transformation process in the next section. Figure 6 illustrates our Seg-Tree that rearranges keys inside nodes to enable an inner node search algorithm using SIMD instructions.

Algorithm 4 Depth-first search using SIMD

```
1: pLevel ← 0
2: subSize ← N-1
3: R ← set searchKey in each segment
4: keyPtr ← pointer to first key in key array
5: while subSize > 0 do
6:     pLevel ← pLevel * k
7:     subSize ← subSize − k − 1
8:     subSize ← subSize/k
9:     function searchSIMD(keyPtr, R)
10:        C ← load k − 1 keys from keyPtr
11:        cmp ← compare C and R for greater-than
12:        bitmask ← extract bitmask from cmp
13:        position ← evaluate bitmask
14:        return position
15:    end function
16:    keyPtr ← keyPtr + k − 1
17:    keyPtr ← keyPtr + subSize * position
18:    pLevel ← pLevel + position
19: end while
20: return pLevel
```

Algorithm 5 Breadth-first search using SIMD

```
1: pLevel ← 0
2: lvlCnt ← 1
3: keyPtr ← pointer to first key in key array
4: nextBasePtr ← keyPtr
5: endPtr ← keyPtr + key count
6: R ← set searchKey in each segment
7: while nextBasePtr < endPtr do
8:     keyPtr ← nextBasePtr + pLevel * (k − 1)
9:     pLevel ← pLevel * k
10:    function searchSIMD(keyPtr, R)
11:        C ← load k − 1 keys from keyPtr
12:        cmp ← compare C and R for greater than
13:        bitmask ← extract bitmask from cmp
14:        position ← evaluate bitmask
15:        return position
16:    end function
17:    pLevel ← pLevel + position
18:    nextBasePtr ← nextBasePtr + lvlCnt * (k − 1)
19:    lvlCnt ← lvlCnt * k
20: end while
21: return pLevel
```

The search Algorithms 4 and 5 perform one comparison operation on each k-ary search tree level. In contrast, the binary search algorithm has the possibility to perform less than $log_2 n$ iterations when the separator is placed on the searched key. One possible improvement might extend our search algorithms by an additional comparison for equality on each level. Therefore, instead of comparing both SIMD registers only for greater-than relationship, we additionally compare for equality. This additional comparison requires no further load instructions because the search key and the list of keys are already resident in the SIMD registers. However, the additional comparison result must be interpreted using expensive conditional branches with possibly no benefit. A benefit will only emerge, if the search key is equal to a key on an upper k-ary search tree level. In this case, the search may terminates above leaf level and comparisons below this level can be omitted. However, we expect no performance improvements for flat k-ary search trees.

3.2 Algorithms for Linearization

We examine two algorithms for linearizing keys in a Seg-Tree node. The first algorithm uses *breadth-first search* while the second algorithm uses *depth-first search* to determine the linearized key order. The *breadth-first search* transformation $P_{BF}(p_L)$ assigns each key in a sorted list $p_L = (0, \ldots, n-1)$ to a position in the linearized k-ary search tree $(0, \ldots, N-1)$. $N-1$ defines the maximum number of keys. Formula 1 calculates the offset recursively on each level of the k-ary search tree. The recursion starts on root level for $R = 0$ with $P_{BF}(p_L, 0)$ and terminates if the last level is reached. The division refers to an integer division without remainder and $S(R) = \lfloor \frac{N}{k^R+1} \rfloor$.

$$P_{BF}(p_L, R) = \begin{cases} \frac{p_L+1}{S(R-1)}(k-1) + \frac{(p_L+1) \bmod (S(R)k)}{S(R)} - 1, \\ \quad \text{if } (p_L+1) \bmod S(R) = 0, \\ P_{BF}(p_L, R+1) + k^R(k-1) \\ \quad \text{else.} \end{cases} \tag{1}$$

The *depth-first search* transformation formula $P_{DF}(p_L)$ is defined in Formula 2 and starts with $P_{DF}(p_L) = P_{DF}(p_L, 0)$.

$$P_{DF}(p_L, R) = \begin{cases} \frac{(p_L+1) \bmod S(R-1)}{S(R)} - 1 \\ \quad \text{if } (p_L+1) \bmod S(R) = 0, \\ P_{DF}(p_L, R+1) + (k-1) \\ + \frac{(p_L+1) \bmod S(R-1)}{S(R)}(S(R)-1) \\ \quad \text{else.} \end{cases} \tag{2}$$

In general, data manipulations require a reordering of existing keys. In case of an insert operation, a naive approach restores the linear order by sorting the list of keys first, before inserting the new key and linearizing the list again. This naive approach results in a possibly large reordering overhead. Therefore, the Seg-Tree is advantageous for workloads with few inserts. These workloads benefit from an accelerated search and the reordering overhead can be neglected. For workloads with high insert rates, like OLTP systems, the reordering overhead probably eliminates the speedup of an accelerated search.

Besides the naive approach, we identify two cases when we can avoid reordering of existing keys. Generally, inserting a new key into a linearized node that falls in between two existing keys requires a reordering of all existing keys. However, we can leverage a particular property in case of *continuous filling* with ascending key values. In this case,

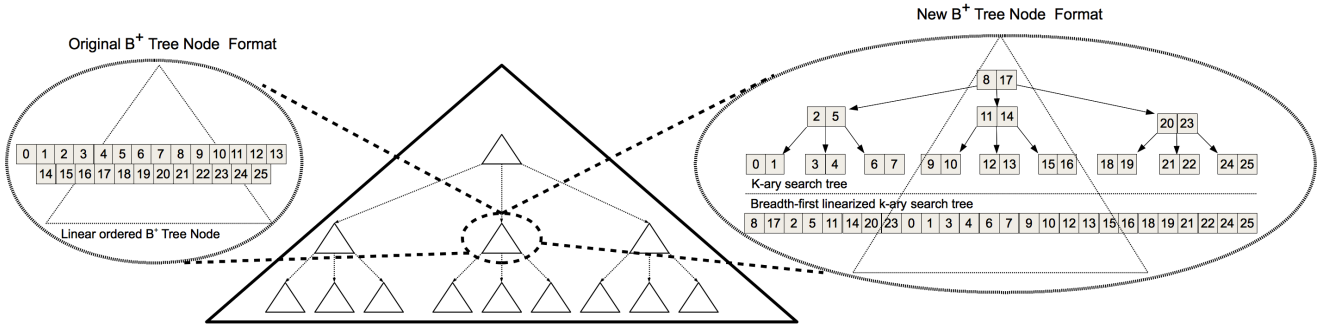

Figure 6: B$^+$-Tree node with linear order (left) and breadth-first linearized order (right).

the inserted key is guaranteed to be greater than all existing keys in the node; thus fall not in between two existing keys. Therefore, the positions of all existing keys remain unchanged and no reordering is necessary. The new key can be copied directly to its position in the linearized list of keys. The case of *initial filling* is a special case of continuous filling. In this case, a sorted data set will be inserted into an empty Seg-Tree in one batch.

Delete operations behave similar to insert operations. Except for a deletion from left to right (increasing values) and from right to left (decreasing values), every random deletion leads to a reordering operation. Update operations always require reordering due to their unpredictable modifications.

3.3 Arbitrary Sized Search Spaces

The key count in a B^+-Tree node is bound by the order o of the tree. One node contains at least o and at most $2o$ keys. A variable number of keys does not satisfy the requirements of a *perfect k-ary search tree* by Schlegel et al. [16]. A perfect k-ary search tree always contains $k^h - 1$ keys for some integer $h > 0$. A dynamically growing index structure is not capable of satisfying this static property. Therefore, the Seg-Tree must be able to build a k-ary search tree from less than $k^h - 1$ keys. Our SIMD sequence for searching requires only a multiple of $k - 1$ keys. In short, we must extend the k-ary search for an arbitrary number of keys.

Following Schlegel et al. [16], our approach extends the number of keys after linearization if necessary. At first, we identify S_{max} as the largest available key, i. e., the right most key in a sorted list of keys in ascending order. Next, we transform the sorted list into a linearized order as described in the previous section. Finally, all k-ary search tree nodes with less than $k - 1$ keys are replenished with the value of $S_{max} + 1$ until each node contains $k - 1$ keys. Figure 7 illustrates a list of 11 keys. To satisfy the property of $k - 1$ keys, we insert $S_{max} = 11$ three times in the k-ary search tree.

Our replenishment approach also affects the search strategy. A search for key v in a k-ary search tree must first check if $v > S_{max}$. If $v > S_{max}$ and the current node is the root or a leaf node, than the search terminates because v does not exist in the Seg-Tree. If $v > S_{max}$ for a branching node, the last pointer at position $n + 1$ must be traversed.

Appending S_{max} even affects the order of a Seg-Tree. In contrast to the original B^+-Tree, the order o of a Seg-Tree specifies no more the minimum and maximum key count in

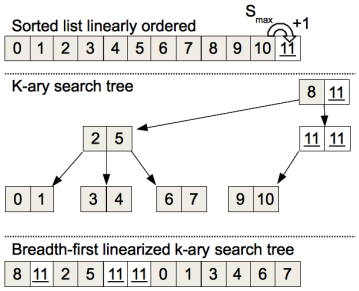

Figure 7: Linearization for an incomplete k-ary search tree.

each node. If the combination of k and o does not satisfy the condition of $k - 1$ keys, then the maximum and minimum key count in a Seg-Tree node are multiples of $k - 1$. For example, an order $o = 2$ leads to a minimum of two and a maximum of four keys per B^+-Tree node. With $k = 9$, $k - 1 = 8$ keys are needed for performing SIMD search. Therefore, a Seg-Tree node must store at least eight keys instead of four keys. Thus, our replenishment approach leads to a larger key count in the Seg-Tree nodes if the property of a multiple of $k-1$ keys per node is not satisfied. Our replenishment strategy represents a tradeoff between the ability to use SIMD instructions for searching and additional computational effort and memory consumption for storing keys in linearized order. The best node utilization is achieved by storing $k^h - 1$ keys per node. Schlegel et al. [16] suggest another approach for non perfect k-ary trees by defining a *complete tree* [16].

3.4 Seg-Tree Performance

The Seg-Tree performance depends on k-ary search. With larger data type sizes, the k-ary search slows down. Table 2 shows common data types and resulting k values for a commonly available 128-bit SIMD bandwidth. For an 8-bit data type and $k = 17$, the k-ary search compares 16 keys in parallel. For a 64-bit data type and the same SIMD bandwidth, the k-ary search compares only two keys in parallel. As a result, the 8-bit data type will perform better. Unfortunately, an 8-bit data type is less likely to be used—usually 32-bit or 64-bit data types are common. In contrast, k-ary search for common data types performs not as good as for small data types. This observation motivated us to develop the *Segment-Trie* to achieve 8-bit k-ary search performance on larger data types.

4. THE SEGMENT-TRIE

The *Segment-Trie (Seg-Trie for short)* enables the aforementioned performance advantages of k-ary search on small data types for a prefix B-Tree storing larger data types. Following Bayer et al. [4] and Boehm et al. [7], the L bit Seg-Trie is defined on data type D_m with length m bits as:

Definition Segment-Trie: Let Seg-Trie$_L$ be a balanced trie with $r = \frac{m}{L}$ levels $(E_0, \ldots, E_{r-1})$. Level E_0 contains exactly one node representing the root. Each node on each level contains one part of the key with length L (in bits), the *segment*. Each node contains n $(1 \leq n \leq 2^L)$ partial keys. One paritial key in one node on level E_i $(0 \leq i \leq r-2)$ points exactly to one node at level E_{i+1}. The nodes on level E_{r-1} contain just as many associated values as partial keys exist. The i-th pointer relates to the i-th partial key and vice versa.

Inserting a key into a Seg-Trie starts by disassembling the key. A key $S[b_{m-1} \ldots b_0]$ is split into r segments $S_0, \ldots, S_{r-1}$ of size L in bits. Each partial key $S_i[b_{L-1} \ldots b_0]$ is composed of $S[b_{(i+1)L-1} \ldots b_{iL}](0 \leq i \leq r-1)$. After disassembling, the segments are distributed among the different levels $E_0, \ldots, E_{r-1}$. The i-th segment S_i serves as partital key on level E_i.

The search for a key S navigates from the root node on level E_0 to a leaf node on level E_{r-1}. Therefore, S is split into $r = \frac{m}{L}$ segments; each segment will be compared on a different trie level. If a segment does not exist on level E_i, then the search key does not exist in the trie and the search terminates. If the search navigates down to the lowest level and the key exists in the leaf node, then the associated value is returned. Commonly associated values are sets of tuple ids or pointers to other data structures. As a variant of a trie, the major advantage of the Seg-Trie against tree structures is its reduced comparison effort resulting from non-existing key segments. If one key segment does not exist at one level, the traversal ends above leaf level. In contrast, a Seg-Tree will always perform the traversal to leaf level [3]. The insert and delete operations are defined similarly.

Suppose an 8-bit Seg-Trie (see Figure 8) storing two 64-bit keys $S_i[b_{L-1} \ldots b_0]$ and $K_i[b_{L-1} \ldots b_0]$. A Seg-Trie for a 64-bit data type is capable of storing up to 2^{64} keys. One 64-bit key is divided into eight 8-bit segments that are distributed over eight trie levels. Except the root level E_0, each level contains at most 256 nodes and each node points to at most 256 nodes on the next lower level. The nodes on leaf level store the associated value instead of pointers. Each node is able to represent the total domain for the segment data type, i. e., 256 values for 8-bit. Internally, the nodes store the partial keys in a linearized order. With commonly available 128-bit SIMD bandwidth, the keys inside the nodes are linearized using a 17-ary search tree and 16 keys can be compared in parallel. Each node maintains a k-ary search tree of two levels since $\lceil \log_{17} 256 \rceil = 2$. Therefore, an inner node search for a partial key requires two SIMD comparison operations; one for each k-ary search tree level. For simplicity, the nodes in Figure 8 show a k-ary search tree for 8 instead of 256 partial keys. A full traversal of a Seg-Trie with $k = 17$ from the root to the leaves takes at most $\lceil \log_{17} 2^{64} \rceil = 16$ comparison operations. In contrast, a trie using ternary search will perform $\lceil \log_3 2^{64} \rceil = 41$ comparison operations while a binary search trie performs $\lceil \log_2 2^{64} \rceil = 64$ comparison operations for the same number of keys.

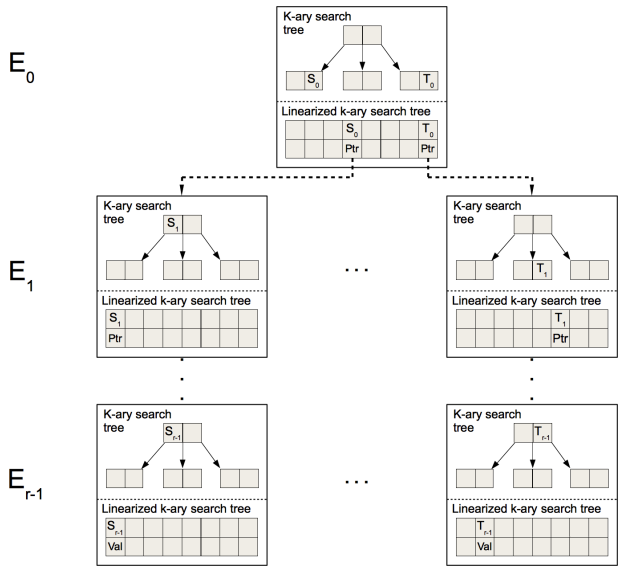

Figure 8: Segment-Trie storing two keys.

Additionally, an 8-bit Seg-Trie leads to an improved cacheline utilization. Compared to larger data types, the 8-bit Seg-Trie reduces the number of cache misses due to an increased ratio of keys per cacheline. Furthermore, the 8-bit data type offers the largest number of parallel comparison operations. Beyond that, the Seg-Trie offers three additional advantages. First, the corporate prefixes for keys leads to a compression. The Seg-Trie represents a prefix B-Tree on bit level; thus extending the already existing tries. Second, the fixed number of levels leads to a fixed upper bound for the number of search operations, page, and memory accesses. Third, each level stores a fixed partition of a key. Therefore, the reorganization following a data manipulation operations in one node is limited to this single node. The remaining trie remains unaffected.

The worst storage utilization for a Seg-Trie occurs when all keys are evenly distributed over the domain. Then, all upper nodes are completely filled; however, the nodes on lower levels contain only one key. This worst case utilization leads to a poor storage utilization due to sparsely filled nodes. One possible solution to overcome this problem is to swap the assignment of segments and levels. On the other hand, the best storage utilization is achieved when storing consecutive numbers like tuple ids. In this case, the Seg-Trie is evenly filled resulting in a high node utilization.

We identify three cases when no inner node search is necessary: 1) the node is empty, 2) the node contains only one key, and 3) the node is completely filled and contains all possible keys. The first case occurs only for an empty trie. In this case, the search key does not exist in the trie and the search terminates. A node that becomes empty due to deleting all partial keys will be removed. For the second case, if only one key is available in a node, we directly compare this key with the search key without performing a search. In the last case, the node is filled with all possible partial keys of the domain. Therefore, we directly follow the corresponding pointer for that partial key instead of performing a search. This transforms a node into a hash like structure with a constant-time lookup speed.

Following the idea of *expanding tries* by Boehm et al. [7] and *lazy expansion* by Leis et al. [12], we suggest to omit tree levels with only one key. Therefore, we create inner nodes only if they are required to distinguish between at least two lower nodes. This approach speeds up the search process and reduces the memory consumption for a Seg-Trie. We refer to this improvement as the *optimized Seg-Trie*. The optimized Seg-Trie stores only levels with at least two distinct keys. Suppose an 8-bit Seg-Trie storing 64-bit keys on eight levels. When filling the tree with consecutive keys starting from 0 to 255, the partial keys are only inserted into one leaf node. After initializing with zero, the seven nodes above leaf level remain unchanged and contain only one partial key throughout the entire key range $[0 \dots 255]$. Therefore, we suggest to omit the seven levels with only one partial key above leaf level. This reduces the memory consumption and speeds up the trie traversal. When inserting 256, the optimized Seg-Trie increases by one level and creates an additional node on the same level. The optimized Seg-Trie incrementally builds up the Seg-Trie starting from leaf level. To remember the prefixes of omitted level, we store them as an additional information inside the nodes. Other techniques for decreasing the height of a trie by reducing the number of levels are *Bypass Jumper Arrays* suggested by Boehm et al. [7] and *path compression* suggested by Leis et al. [12]. Both techniques are applicable for our Seg-Trie but currently not implemented. In the next section, we evaluate our Seg-Trie and Seg-Tree implementations.

5. EVALUATION

In this section, we experimentally evaluate our tree adaptions for different data types and data set sizes. At first, we describe our experimental setup. After that, we evaluate three algorithms for bitmask evaluation and choose one for the remaining measurements. Next, we evaluate the performance of our B^+-Tree (Seg-Tree) and trie (Seg-Trie) implementations using k-ary search. The original B^+-Tree serves as the baseline for our performance measurements.

5.1 Experimental Setup

All experiments were executed on a machine with an Intel Xeon E5520 processor (4 cores each 2,26 GHz and Intel Hyper Threading). Each core has a 32 KB L1 cache and a 256 KB L2 cache. Furthermore, all cores share a 8 MB L3 cache. The Xeon E5520 is based on Intel's *Nehalem* microarchitecture with a cacheline size of 128 byte and a SIMD bandwidth of 128 bit. The machine utilizes 8 GB of main memory with 32 GB/s maximum memory bandwidth. We use the Intel compiler with `O2` optimization flag and `SSE4` for SSE support on a Windows 7 64-bit Professional operating system.

We use a synthetically generated data set. For 8-bit and 16-bit data types, we generate key sequences for the entire domain of 256 and 65536 possible values, respectively. For 32-bit and 64-bit data types, we generate key sequences containing values in ascending order starting at zero. Initially, we load the entire data set into main memory. After that, we build the tree by creating the nodes using the configuration shown in Table 3. K results from a SIMD bandwidth of 128-bit and the chosen data type. N_L denotes the number of keys in the sorted list of keys and N_S denotes the number of keys in the linearized k-ary search tree of height r. N determines the maximum number of keys in one node. The memory consumption of one key consists of a key value and a pointer to the next node level. The size of a pointer on a 64-bit operating system is eight byte and the key size is determined by the chosen data type. To utilize the hardware prefetcher efficiently, we adjust the node size to be smaller than 4 KB. A node size smaller than 4 KB results in no cache miss due to crossing the 4 KB prefetch boundary [2]. Additionally, our node configuration builds a perfect k-ary search tree from k^r keys. Considering the prefetch boundary and perfect k-ary search tree property, we configure the nodes as shown in Table 3. The node size is calculated by $N_L + 1 * sizeof(pointer) + N_S * sizeof(data\ type)$. For example, each node for an 8-bit data type stores $N_L + 1 = 255$ 8-byte pointers and $N_S = 256$ 8-bit keys. We store the keys in one contiguous array. The *cache lines* column expresses how many cache lines are required to access each key in a node. It is calculated by $\frac{N_S * sizeof(data\ type)}{cacheline\ size}$. Using k-ary search, we need one comparison operation on each k-ary search tree level r. Therefore, we access at most r cache lines. Notice, that all nodes are completely filled. After building the tree, we measure the time for searching x keys in random order and calculate the average search runtime for one search operation. For the remainder of this paper, we define $x = 10,000$. To measure the runtime we use RDTSC (*Read time-stamp counter*) instructions to count the clock cycles between to points in time. All measurements are performed in a single thread. There is no output written to disk and the search result is not further processed.

Data type	k	N_L	N_S	r	N	Node size	Cache lines
8-bit	17	254	256	2	289	2296	2
16-bit	9	404	408	3	729	4056	7
32-bit	5	338	344	4	625	4096	11
64-bit	3	242	242	5	243	3880	16

Table 3: Node characteristics.

5.2 Bitmask Evaluation

As described in Section 2.1, our SIMD sequence compares two SIMD registers and outputs the result into a third SIMD register. The resulting bitmask in the third SIMD register must be evaluated to determine the relationship between the search key and the list of keys. For bitmask evaluation, we analyze three algorithms, i. e., *bit shifting*, *switch case*, and *popcount*. At first, all algorithms use the `movemask` instruction to create a 16-bit bitmask from the most significant bits in the result SIMD register and place the bitmask into the lower 16 bits of an x86 register. The algorithms differ in converting the 16-bit bitmask into a position in a sorted list of keys. Figure 9 shows the results for the three algorithms performing a search in an 8-bit Seg-Tree. The three categories *Single*, *5 MB* and *100 MB* represent the amount of data in the Seg-Tree. For the remainder of this paper, we refer to *Single* as a data set containing keys in one single node. With *5 MB* and *100 MB*, we refer to upper bounds for the data set size. The node count depends on the single node size and the upper bound (see Table 3).

The *popcount* algorithm achieves the best overall results; it is also independent of data set size. The main reason for its superiority is the elimination of 16 conditional branches; thus eliminating expensive pipeline flushes. The perfor-

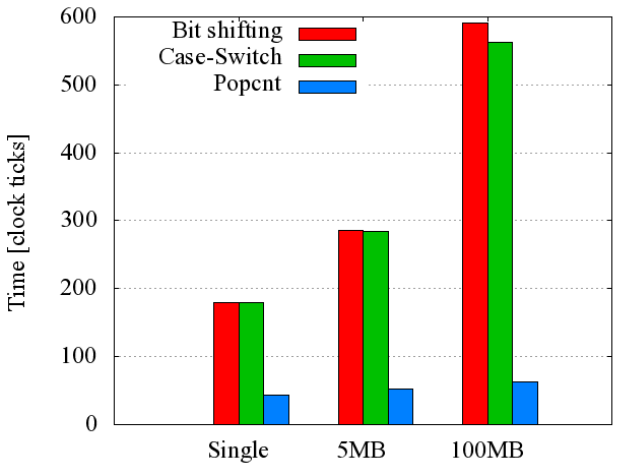

Figure 9: Evaluation of bitmask for 8-bit data type.

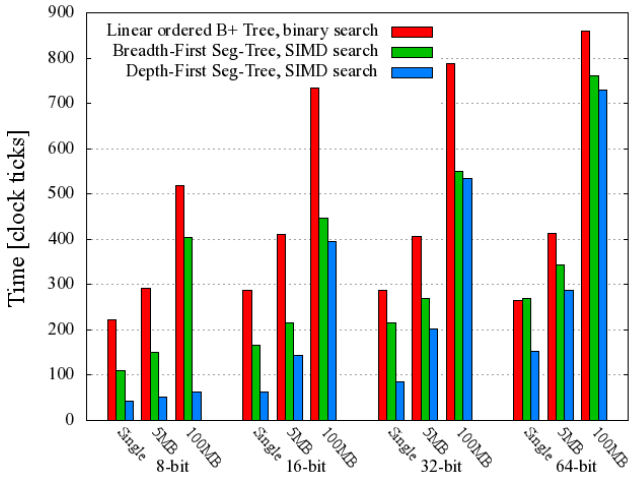

Figure 10: Evaluation of Seg-Tree.

mance improvements of k-ary search is mainly based on eliminating conditional branches. For larger data types, there are less conditional branches available which can be eliminated. Therefore, the decreasing number of conditional branches for larger data types leads to a decrease in k-ary search performance. The largest data type, i. e., 64-bit, performs only two conditional branches. Due to the overall best performance, we use the *popcount* algorithm for the following evaluation of our Seg-Tree and Seg-Trie implementation.

5.3 Evaluation Seg-Tree

We evaluate the Seg-Tree using four different integer data types (8-, 16-, 32-, and 64-bit) as keys and store three differently sized data sets (Single, 5 MB, 100 MB). Figure 10 shows the average runtime of one search operation in clock ticks using different inner node search algorithms. The red bar presents the original B^+-Tree using binary search. The Seg-Tree uses SIMD search on breadth-first (green bar) and depth-first (blue bar) linearized keys. The measurements show, that the depth-first search performs best in all configurations. Generally, the performance increases for smaller data types. This observation is independent of data set size and can be explained by two reasons. At first, for 8-bit data type values, 16 comparison operations can be performed in parallel while for 64-bit data type values, only two are possible. Second, small data type values lead to a better cache behavior due to an increased ratio of keys per cacheline. The k-ary search on 8-bit data type values outperforms the binary search nearly by a factor of eight even for large data set sizes.

For large data set sizes, the SIMD search performance on breadth-first and depth-first linearized keys is nearly similar, except for an 8-bit data type. For decreasing data set sizes, a Seg-Tree using depth-first linearized keys outperforms a Seg-Tree using breadth-first linearized keys. The cache hierarchy impacts the performance of both Seg-Trees and the B^+-Tree. For a single node, the node resides most likely in the L1 cache for each search operation. Therefore, the *Single* category illustrates the pure runtime for each search algorithm in a comparable way by excluding cache effects. For a 5 MB data set size, the entire data set will properly fits into the 8 MB L3 cache but not entirely in the 256 KB L2 cache. A random node select has a possibility to produce a L2 cache miss. The 100 MB data set fits in no cache level entirely; thus further increases the impact of cache misses. The computational effort for searching inside the nodes become more negligible with an increasing number of cache misses. The cache hierarchy becomes the bottleneck for larger data set sizes. Generally, the inner node search algorithms transform from a computation bound algorithm to a cache/memory bound algorithm for increasing data set sizes.

5.4 Evaluation Seg-Trie

We evaluate the Seg-Trie and optimized Seg-Trie against different Seg-Trees in Figure 11. The speedup refers to the original B^+-Tree using binary search. The optimized Seg-Trie implements the elimination of levels as mentioned in Section 4. The node configuration for the Seg-Tree is equal to the 64-bit data type configuration in Table 3. The Seg-Trie contains precisely eight levels and the optimized Seg-Trie contains at most eight levels. Each node follow the 8-bit data type configuration in Table 3. The depth of the tree in Figure 11 refers to the number of levels that are filled with keys. We vary the number of keys to fill the expected level count. For comparability reasons, all tree variants contain the same number of levels and keys. To achieve this, we skew the data for both Seg-Trie variants to produce the expected level count.

As shown in Figure 11, the performance of a Seg-Trie increases almost linearly with the depth of the tree. The performance is measured against a B^+-Tree using binary search. Instead of comparing a 64-bit search key with a 64-bit key on each level like the B^+-Tree using binary search, the Seg-Trie compares only one 8-bit part of the search key on each level. Additionally, an increase of tree depth by one for a Seg-Trie leads to no additional node comparison because a 64-bit Seg-Trie always searches above eight tree level. In contrast, the B^+-Tree using binary search must perform one additional node search for each additional tree level. Therefore, with increasing tree depth, the speedup of the Seg-Trie compared to the B^+-Tree using binary search increases almost linear.

The optimized Seg-Trie provides a constant speedup independent of tree depth. As mentioned in Section 4, the

optimized Seg-Trie omits levels with less than two distinct values. The depth of the tree in Figure 11 refers to the number of filled levels. Compared to a Seg-Trie, the optimized Seg-Trie requires one node comparison on each filled tree level. In contrast, a Seg-Trie always performs eight comparisons even for levels containing only one key. Therefore, the number of node comparisons for the optimized Seg-Trie increases for deeper trees. The speedup is constant because it is measured against the B^+-Tree using binary search. Each additional tree level adds one additional node to both tree variants. Therefore, the speedup remains unchanged. Suppose, we insert a 100 MB data set containing nearly 1.6 M keys in consecutive order (starting at zero) into a 64-bit optimized Seg-Trie. We need 21 bits out of the available 64 bits to represent the largest key representation (1,638,400). Therefore, the upper 43 bit are unused. The number of levels that can be omitted due to 43 unused bits depend on the size of the partial keys. In our example, we split a 64-bit key into eight parts. Therefore, the optimized Seg-Trie of depth three omits five out of eight levels, i. e., 40 bits. For a tree depth of eight, no levels are omitted and both Seg-Trie variants behave similar. The reduced number of levels leads to a reduced amount of memory transfers, a reduced possibility of cache misses and less computational effort.

The Seg-Tree using breadth-first linearization provides a constant speedup compared to a B^+-Tree using binary search and is independent of tree depth. However, the large data type leads to a small speedup. The Seg-Tree using depth-first linearization provides a similar improvement with same characteristics. Therefore, both lines overlap in Figure 11. Like the B^+-Tree using binary search, the Seg-Tree adds one node to the traversal path for each increase in tree depth. Therefore, the speedup remains constant.

The smallest data type that can currently be processed by the *SIMD Extensions* is 8-bit [2]. This restriction limits a further increase in tree depth. However, the optimized Seg-Trie and the Seg-Tree are independent of tree depth. The Seg-Tree performs poorly on large data types but increases its performance for smaller data types. The optimized Seg-Trie provides a constant 14 fold speedup independently of tree depth and an eight fold reduced memory consumption compared to the original B^+-Tree.

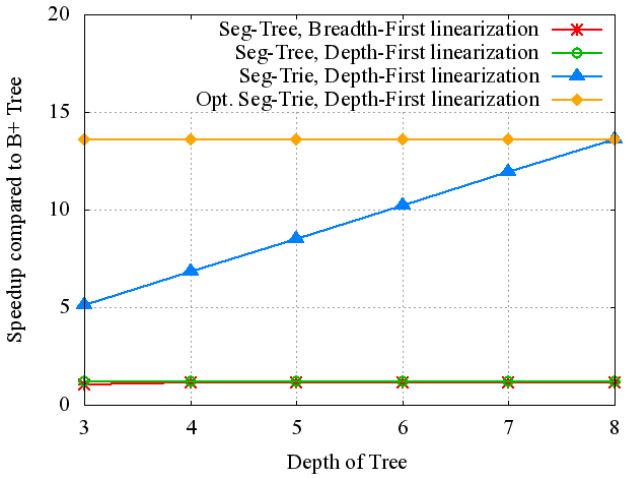

Figure 11: Evaluation Seg-Tree vs. Seg-Trie for 64-bit key.

6. RELATED WORK

Zhou and Ross examine the application of SIMD instruction to implement database operations [20]. They address sequential scans, aggregations, index operations, and joins. For our work, the index operations are the most important ones. The SIMD instructions are applied in three approaches. The first approach improves the binary search by expanding the number of elements in one iteration step. Instead of comparing one separator with the search key, they use the entire SIMD bandwidth. As a result, they include elements that are located besides the separator. The second approach is a sequential search using the entire SIMD bandwidth. Instead of comparing one element at a time, the second approach compares as many elements as fit into one SIMD register and proceed in a stepwise manner. The third approach combines both approaches in a so-called *hybrid search*. In contrast, our approach based on k-ary search that reorders the sorted list of elements. The k-ary search also increases the number of separators. Additionally, the k-ary search supports a distance between two separators that is wider than SIMD bandwidth.

Other research improves index structures like hashes using SIMD [15]. Kim et al. introduce *FAST*, a binary tree that is optimized for architectural features like page size, cache line size, and SIMD bandwidth of the underlying hardware [11]. They examined the impact of translation lookaside buffer (TLB) misses, last level cache (LLC) misses and memory latency on CPU and GPU. Furthermore, Kim et al. exploit thread-level and data-level parallelism on both CPUs and GPUs. They point out, that the tree size and the size of the LLC impacts the usability of CPU or GPU. In sum, the tree processing is computation bound on small trees which fit into LLC and bandwidth bound on trees larger than LLC size. Our evaluation shows similar results. Compared to our approach, they use an additional *lookup table* to evaluate the bitmask and navigate to the next child node. The data layout also differs between our Seg-Tree and Seg-Trie using k-ary search and the adapted binary search by Kim et al. [11]. They divide the tree in sub-trees to create a layout optimized for specific architectural features. In contrast, our approaches use k-ary search and our data layout is determined by breadth-first or depth-first search. Based on *FAST*, Yamamuro et al. introduce the *VAST-Tree*, a vector-advanced and compressed structure for massive data tree traversal [19]. By applying different compression techniques to different node levels, they achieve a more compact tree with higher traversal efficiency.

Leis et al. [12] introduce the *Adaptive Radix Tree* as an *ARTful* index for main memory databases. The ART tree uses four node types with different capacities inside the nodes depending on the number of keys. However, this approach uses SIMD instructions only for the search in one node type and for at most 16 keys. In comparison, our approach uses SIMD instructions independent of the number of keys and for each node size.

Graefe and Larson summarized several techniques for improving cache performance for B-Trees [9]. Furthermore, Bender et al. introduce a cache oblivious B-Tree [5] and a cache oblivious string B-Tree [6]. Rao and Ross introduce two cache conscious tree structure, the *Cache-Sensitive*

Search Trees (CSS-Tree) [13], and the *Cache Sensitive B^+-Tree (CSB+-Trees)* [14]. These tree variants construct the tree such that the keys are placed as cache-optimized as possible in terms of spatial or temporal locality. The trees differ in terms of knowing the main parameters of the memory hierarchy, i. e., they are cache *conscious*, or running best on an arbitrary memory hierarchy, i. e., they are cache *oblivious*. Besides these differences, all tree variants increase the cacheline utilization by changing the tree layout. In contrast, our approach constructs the tree in a way that enables SIMD usage for tree traversal. However, our layout modification increases the cacheline utilization as well. At first, our approach maximizes the cacheline utilization by sorting the keys such that separator keys are placed next to each other. Therefore, we compare k separators in parallel instead of two in case of the commonly used binary search. Second, our approach reduces the number of comparison operations inside the node from $log_2 n$ to $log_k n$. The decreased number of comparisons reduces the number of loaded cache lines. Furthermore, the number of accesses to different memory locations are reduced; thus increasing spatial locality.

Following the idea of a *prefix B-trees* by Bayer et al. [4], many trie variations were proposed. The *generalized trie* by Boehm et al. [7] exhibits the most similarities to our trie implementation using k-ary search. Both approaches partition a fix sized integer value and distribute it above different trie levels. However, the inner node search differs. Inside one node, the *generalized trie* maps the partial key to a position in an array of pointers. A node contains one pointer for each possible value of the partial key domain. In contrast, our Seg-Trie implementation performs a k-ary search in each node. Our implementation will store the same pointer array and an additional array for all possible key representation. For traversal, our implementation performs the k-ary search in each node with two comparison operations for an 8-bit data type.

7. CONCLUSION AND FUTURE WORK

This paper introduces the Seg-Tree and Seg-Trie which enable efficient SIMD usage for tree and trie structures. We show that SIMD instructions of modern processors are qualified to speed up tree-based index structures. Therefore, we make SIMD instructions applicable for tree based search algorithms in modern database systems. Based on k-ary search by Schlegel et al. [16], we investigate how to use this approach for a B^+-Tree and prefix B-Tree structure. We contribute two different linearization and search algorithms, the generalization to an arbitrary key count, and three algorithms for bitmask evaluation. The introduced Seg-Trie takes advantages of k-ary search for small data types and enables them for large data types. Furthermore, our optimized Seg-Trie provides a 14 fold speedup and an 8 fold reduced memory consumption compared to the original B^+-Tree. We emphasize, that the strength of a Seg-Trie arises from storing consecutive keys like tuple ids. On the other hand, if the keys are evenly distributed, the Seg-Trie needs further adjustments to enhance the storage utilization. As the SIMD bandwidth will increase in the future [1], index structures using SIMD instructions will further benefit by increased performance.

We plan to extend this work in two areas. First, this paper focuses on optimizing the Seg-Tree and Seg-Trie for single thread performance. In future work, we will investigate the impact of multi-threading, multi-core, and many-core architectures on different aspects of Seg-Tree and Seg-Trie processing. Especially, the impact of SIMD instructions on concurrently used index structures is an ongoing research task. Second, we plan to adapt the Seg-Trie and Seg-Tree for GPU processing. Compared to CPUs, GPUs support scatter and gather operations that allow a load/store of keys from distributed memory locations. Thus, we expect a reduced reordering effort.

8. REFERENCES

[1] *Intel Advanced Vector Extensions Programming Reference*. http://software.intel.com/en-us/avx/, 2008.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Workshop on Data Description, Access and Control*, 1970.

[4] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 1977.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *FOCS*, 2000.

[6] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string b-trees. In *PODS*, 2006.

[7] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *BTW*, 2011.

[8] D. Comer. Ubiquitous b-tree. *ACM Comp. Surv.*, 1979.

[9] G. Graefe and P.-A. Larson. B-tree indexes and cpu caches. In *ICDE*, 2001.

[10] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 2002.

[11] 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*, 2010.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[13] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, 1999.

[14] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD*, 2000.

[15] K. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.

[16] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN workshop*, 2009.

[17] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[18] R. E. Wagner. Indexing design considerations. *IBM Systems Journal*, 1973.

[19] T. Yamamuro, M. Onizuka, T. Hitaka, and M. Yamamuro. Vast-tree: a vector-advanced and compressed structure for massive data tree traversal. In *EDBT*, 2012.

[20] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, 2002.