Efficient In-Memory Indexing with Generalized Prefix Trees

Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer,
Dirk Habich, Wolfgang Lehner
TU Dresden; Database Technology Group; Dresden, Germany

Abstract: Efficient data structures for in-memory indexing gain in importance due to (1) the exponentially increasing amount of data, (2) the growing main-memory capacity, and (3) the gap between main-memory and CPU speed. In consequence, there are high performance demands for in-memory data structures. Such index structures are used—with minor changes—as primary or secondary indices in almost every DBMS. Typically, tree-based or hash-based structures are used, while structures based on prefix-trees (tries) are neglected in this context. For tree-based and hash-based structures, the major disadvantages are inherently caused by the need for reorganization and key comparisons. In contrast, the major disadvantage of trie-based structures in terms of high memory consumption (created and accessed nodes) could be improved. In this paper, we argue for reconsidering prefix trees as in-memory index structures and we present the *generalized trie*, which is a prefix tree with variable prefix length for indexing arbitrary data types of fixed or variable length. The variable prefix length enables the adjustment of the trie height and its memory consumption. Further, we introduce concepts for reducing the number of created and accessed trie levels. This trie is order-preserving and has deterministic trie paths for keys, and hence, it does not require any dynamic reorganization or key comparisons. Finally, the *generalized trie* yields improvements compared to existing in-memory index structures, especially for skewed data. In conclusion, the *generalized trie* is applicable as general-purpose in-memory index structure in many different OLTP or hybrid (OLTP and OLAP) data management systems that require balanced read/write performance.

1 Introduction

Index structures are core components of typical data management systems. While this area has been studied for a long time, many aspects need to be reconsidered in the context of modern hardware architectures. Due to the increasing main-memory capacity and the growing gap between CPU speed and main-memory latency [MBK00], especially, in-memory indexing gains in importance. The specific characteristics of in-memory indexing compared to disk-based indexing are that (1) pointer-intensive index structures with small node sizes can be preferred instead of page-based structures due to smaller block granularities of main memory, and that (2) the number of required key transformations and comparisons as well as efficient main memory management and cache consciousness are crucial influencing factors on the overall performance. For update-intensive in-memory indexing in the context of online transaction processing (OLTP), typically, tree-based or hash-based techniques are used, while tries are usually neglected.

All of those structures have their specific drawbacks. Tree-based structures require re-

organization (tree balancing by rotation or node splitting/merging) and many key comparisons compared to hash-based or trie-based techniques. Hash-based techniques heavily rely on assumptions about the data distribution of keys and they require reorganization (rehashing) as well. Tries are typically only designed for string operations, they often require dynamic reorganization (prefix splits), and they can cause higher memory consumption compared to tree- or hash-based structures. The disadvantages of tree-based and hash-based techniques are caused inherently by their structure. In contrast, the disadvantages of tries can be addressed appropriately. This optimization potential enables us to generalize existing trie structures in order to make them applicable for efficient in-memory indexing.

While tree-based and hash-based structures still have their application areas, we argue that a generalization of existing trie structures for in-memory indexing can, in particular with regard to a balanced read/write performance and for skewed data, achieve performance improvements compared to existing index structures. This trie generalization focuses on the goals of (1) trie-indexing for arbitrary data types, (2) order-preserving key storage, (3) deterministic trie paths (no need for dynamic reorganization, except leaf splits), and (4) efficient memory organization. Our primary contribution is the reconsideration and adaptation of prefix trees for efficient in-memory indexing. Furthermore, we make the following more concrete contributions, which also reflect the structure of this paper:

- First of all, in Section 2, we survey array-based, tree-based, hash-based and trie-based index structures and discuss their main drawbacks.
- Then, in Section 3, we describe the generalization of prefix trees. This includes the formal foundation and a physical realization of the *generalized trie*.
- Subsequently, in Section 4, we introduce the optimization techniques *bypass jumper array* and *trie expansion*. Furthermore, we provide insights into important read and write algorithms as well as additional memory optimization concepts.
- We present selected results of our experimental evaluation in Section 5.
- Finally, we conclude the paper and mention future work in Section 6.

2 Analysis of Existing Solutions

In order to give an overview of related index structures, we briefly survey the main categories of in-memory index structures but refer for more details to a comparison of basic in-memory structures [LC86] and a detailed time analysis, which also includes the number of required key comparisons [LDW09].

As the notation, we use the set of records R, where N denotes the number of records, with $N = |R|$. Each record $r_i \in R$ exhibits the structure $r_i = (k_i, \alpha_i)$, where k_i denotes the key and α_i denotes the associated information (payload). Furthermore, all records of R exhibit the same data type, where k with $k = |k_i|$ denotes the length of all keys in terms of the number of bits. If data types of variable length are used, the key length denotes the maximum key length. For example, a `VARCHAR(256)` results in $k = 2{,}048$.

Essentially, index structures are distinguished into the categories: (1) sorted arrays, (2) trees, (3) hash-based structures, and (4) prefix-trees (tries). In the following, we survey existing work according to these categories.

Sorted Arrays. The simplest index structure is an array of records. Unsorted arrays cause linear time complexity of $O(N)$. Hence, often sorted arrays are used in combination with *binary search* [Knu97]. It is known that the worst-case time complexity of binary search is $O(\log N)$. We have recently presented a *k*-ary search algorithm [SGL09] that uses SIMD instructions, yielding a significant improvement over binary search. However, the worst-case complexity is still $O(\log N)$. Although sorted arrays are advantageous for several application scenarios, they fall short on update-intensive workloads due to the need for maintaining the sorted order of records (moving records). In case of partitions of many duplicates, the shuffling technique [IKM07] (that moves as few as possible records of a partition) can minimize the costs for order maintenance. Another approach is to use sorted arrays with gaps according to a used fill factor such that only few tuples might be moved but the space requirements are increased. While a linked list would allow for more efficient updates, it is not applicable for binary search.

Tree-Based Structures. Typically, tree-based structures are used for efficient in-memory indexing. These structures are distinguished into unbalanced and balanced trees. The *binary tree* [Knu97] is an unbalanced tree where a node has at most two children. This structure can degenerate, causing more nodes to be accessed than for a balanced tree. In contrast, especially for in-memory indexing, balanced tree-structures are used. There is a wide variety of existing structures such as *B-trees* [BM72], *B^+-trees* [Jan95], *red-black-trees* [GS78], *AVL-trees* [Knu97] and *T-trees* [LC86]. With the aim of tree balancing, rules for node splitting/merging or tree rotation are used. Current research focuses on cache-conscious tree-structures for T-Trees [LSLC07] and B-Trees [CGMV02, RR99, RR00] and on exploiting modern hardware like SIMD instructions [ZR02] or architecture-aware tree indexing on CPUs and GPUs [KCS+10]. All of those balanced tree structures exhibit a logarithmic time complexity of $O(\log N)$ in terms of accessed nodes for all operations. Additionally, they require a total number of $O(\log N)$ key comparisons. This is especially important when indexing arbitrary data types such as `VARCHAR`.

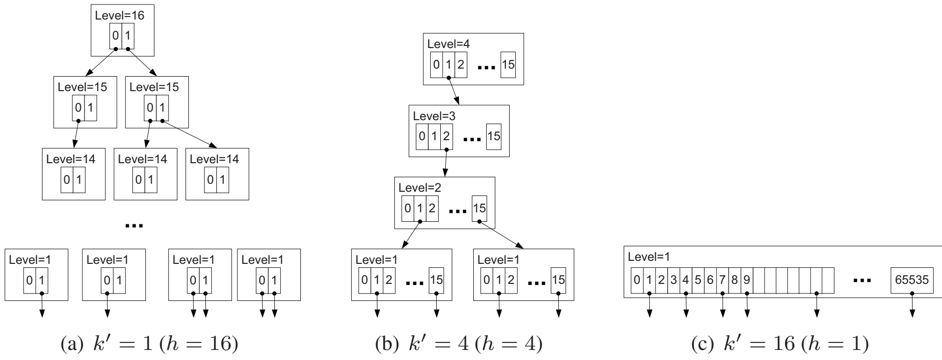
Hash-Based Structures. In contrast to tree-based structures, hash-based structures rely on a hash function to determine the slot of a key within the hash table (an array). Depending on the used hash-function, this approach makes assumptions about the data distribution of keys and can be order-preserving. For *chained bucket hashing* [Knu97], no reorganization is required because the size of the hash table is fixed. However, in the worst case, it degenerates to a linked list and thus, the worst-case search time complexity is $O(N)$. In contrast to this, there are several techniques that rely on dynamic reorganization such as *extendible hashing* [FNPS79], *linear hashing* [Lit80, Lar88], and *modified linear hashing* [LC86]. Current research focuses on efficiently probing multiple hash buckets using SIMD instructions [Ros07]. Due to reorganization, those structures exhibit a search time complexity of $O(1)$. However, additional overhead for dynamic hash table extension and re-hashing (worst case: $O(N)$) is required and thus it can be slower than tree-based structures. In conclusion, hash-based structures are advantageous for uniformly distributed keys rather than for skewed data.

Trie-Based Structures. The basic concept of prefix trees (tries) [Fre60]—also called digital search trees [Knu97]—is to use parts of a key k_i (with key length k) to determine the path within the trie. Prefix trees are mainly used for string indexing, where each node holds an array of references according to the used character alphabet [HZW02] and therefore they were neglected for in-memory indexing of arbitrary data types [LC86]. With that concept, the worst-case time complexity is $O(k)$ for all operations because, in general, the number of accessed nodes is independent of the number of records N (independent of the trie fill factor). However, the maximum number of indexable records is $N' = 2^k$. Compared to tree-based structures, more nodes are accessed because $k \geq \log N$, where $k = \log N$ only if $N' = N$. However, only partial key comparisons are required per node, where in the worst case, a single full key comparison is required in total for any operations. The most recognized trie structure is the *radix tree* [Knu97], where *radix tree*, *patricia trie* [Mor68] and *crit-bit tree* (critical bit) are used as synonyms. Essentially, these structures maintain a tree of string prefixes. Each node represents the string position, where the strings of the left and the right sub-trie differ. Thus, a node has at least two children and each edge can encode multiple characters [Mor68]. In contrast to traditional tries (using one character per trie level), this yields a significant string compression. Typically, this structure is only used for string indexing and efficient string operations. This data type restriction was addressed with the extension *Kart* (key alteration radix tree), where bit positions, rather than single character positions, are used for difference encoding. This allows for indexing arbitrary data types but due to the encoding of different prefix lengths within a trie, there is still the need for reorganization (arbitrary prefix splits). In conclusion, the advantage of trie structures is the good time complexity of $O(k)$ with only few (partial) key comparisons, which is especially useful when indexing strings. Existing trie-based structures are therefore mainly designed for those string operations rather than for indexing arbitrary data types.

Hybrid Structures. Furthermore, there are several hybrid index structures such as *prefix hash trees* [CE70] (trie and hash), *HAT-tries* [AS07] (trie and hash), *ternary search trees* [BS97] (trie and binary tree), *prefix B-trees* [BU77] (trie and B-tree), *partial keys* [BMR01] (trie and B-tree/T-tree), *J$^+$-trees* [LDW09] (trie and B-tree/T-tree), *CS-prefix-trees* [BHF09] (trie and CSB-tree), and *Burst-tries* [HZW02] (trie and arbitrary structure for containers). Interestingly, all of these here mentioned hybrid structures use to some extend trie-based concepts.

3 Generalized Prefix Trees

Due to the disadvantages of existing structures, we outline our so-called *generalized trie* as a new in-memory data structure. It is a generalization of prefix trees (tries) for indexing arbitrary data types of fixed and variable length in the form of byte sequences. The novel characteristic compared to existing trie-based structures is an assembly of known and some new techniques. In detail, we use (1) a prefix size of variable length, (2) a bypass structure for leading zeros, (3) dynamic, prefix-based trie expansion, and (4) optimizations for pointer-intensive in-memory indexing. In this section, we focus on the formal foundation of this trie generalization, the core operational concepts, and still existing problems.



(a) $k' = 1$ ($h = 16$) (b) $k' = 4$ ($h = 4$) (c) $k' = 16$ ($h = 1$)

Figure 1: Example Generalized Trie with $k = 16$

3.1 Formal Foundation

As the foundation of this work, we define the *generalized trie* as follows:

Definition 1 Generalized Trie: *The generalized trie is a prefix tree with variable prefix length of k' bits. We define that (1) $k' = 2^i$, where $i \in \mathbb{Z}^+$, and (2) k' must be a divisor of the maximum key length k. The generalized trie is then defined as follows:*

- *Given an arbitrary data type of length k and a prefix length k', the trie exhibits a fixed height $h = k/k'$.*
- *Each node of the trie includes an array of $s = 2^{k'}$ pointers (node size).*

The generalized trie is a non-clustered index with h levels. The root node describes level h, while nodes of level 1 are leaf nodes and point to the data items. Thus, leaf nodes are identified by the trie structure. The trie path for a given key k_i at level l is determined with the $(h-l)^{th}$ prefix of k_i. The single nodes of the trie are only created if necessary.

As a result, we have a deterministic prefix tree that indexes distinct keys of arbitrary data types without the need for (1) multiple key comparisons or (2) reorganization. The prefix length k' can be used to adjust the required space of single nodes and the number of accessed nodes for an operation. The following example shows the resulting spectrum.

Example 1 Configuration of Generalized Tries: *Assume the data type* `SHORT(2)` *with a key length of $k = 16$. On the one side, we can set $k' = 1$, where each node contains two pointers (bit set/ bit not set). This would result in a structure similar to a binary tree with a trie height $h = 16$. On the other side, we could set $k' = 16$, which results in a height $h = 1$ node containing an array of $65{,}536$ pointers to keys. In the latter case, the complete key determines the array index. Figure 1 illustrates this configuration possibility using $k' = 1$ (Figure 1(a), binary prefix tree), $k' = 4$ (Figure 1(b), well-balanced hierarchical configuration), and $k' = k = 16$ (Figure 1(c), a sorted array with gaps).*

Clearly, when configuring k', this is a trade-off between the number of accessed nodes and space consumption. An increasing k' will cause decreased node utilization and thus, in-

creased memory consumption, while the steps from the root to the leafs are reduced for all operations. Furthermore, the generalized trie has the following two important properties:

- *Deterministic Property*: Using a prefix length of k', a single node contains $s = 2^{k'}$ pointers. Each key has then only one path within the trie that is independent of any other keys. Due to these deterministic trie paths, only a single key comparison and no dynamic reorganization are required for any operations.
- *Worst-Case Time Complexity*: Based on the given key type of length k, the generalized trie has a time complexity of $O(h)$ and hence, has constant complexity in terms of the fill factor of the trie (the number of tuples N). Due to the dynamic configuration of k', we access at most $h = k/k'$ different nodes (where h however might be higher than $\log N$), and compare at most $O(1)$ keys for any operations.

In contrast to the original trie [Fre60], we use a variable prefix length (not single characters) and have a deterministic trie of fixed height. The fixed height allows the determination of leaves. If the root level is given by $h = k/k'$, leaves are only present at level 1.

For data types of variable length such as `VARCHAR`, the trie height is set to the maximum key length rounded up to a factor of k'. In order to ensure that leaves are only present at level 1, keys with length $k < h \cdot k'$ are *logically* padded with zeros at the end because padding at the beginning would lead to loosing the property of being order-preserving.

3.2 Physical Data Structure and Existing Problems

Based on the formal definition of the *generalized trie*, we now explain the `IXByte` that is a physical data structure realizing such a trie generalization. We describe the main data structure and its most important operational concepts.

The `IXByte` is designed as a hierarchical data structure in order to leverage the deterministic property of the generalized trie. A single index (`L0Item`) is created with the data type length as a parameter. In order to support also duplicates of keys, we use a further hierarchical structure of key partitions k_i (`L1Item`) and payloads α_i (`L2Items`). Then, the single keys are indexed as a generalized trie, while the payloads of all records associated with the same key are maintained as a list of `L2Items` within this `L1Item` partition. Furthermore, the node data structure used within the `IXByte` is defined as an array of $s = 2^{k'}$ `void*` pointers. We use generic `void*` pointers in order to refer to both inner nodes and `L1Items` (leaves), where the fixed maximum trie height h determines the leaves. The `IXByte` works on order-preserving byte sequences (big-endian) and therefore arbitrary data types can be supported as long as their keys can be transformed accordingly.

Example 2 `IXByte` Operations: *Assume an index of data type `SHORT(2)` with $k = 16$ and a prefix length of $k' = 4$. We insert the record $r_i = (107, value3)$, where the resulting key parts are illustrated at the left side of Figure 2 and by emphasized pointers. We start at the root node (level 4) and use the value of bits 0-3 as array index in order to determine the child pointer. We repeat this with bits 4-7 on level 3, and with bits 8-11 on level 2. Finally, on level 1, we use the value of bits 12-15 as our array index. We know that this*

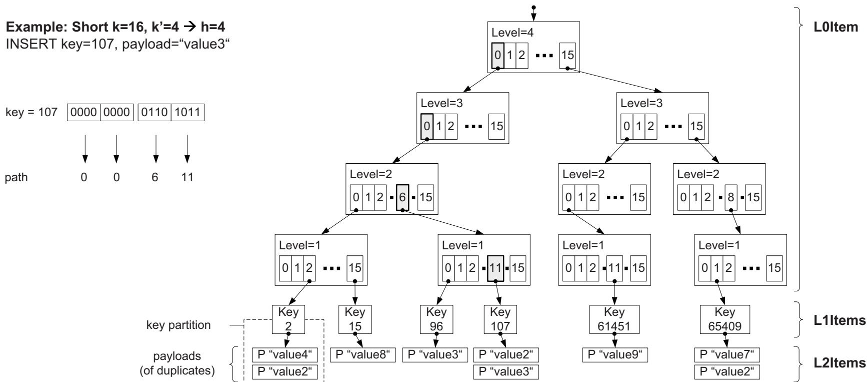


Figure 2: Example `IXByte` for $k = 16, k' = 4, h = 4$

pointer must reference an `L1Item` *(key partition). If no partition exists, we create a new one; otherwise, we just insert the payload into the partition.*

The *deterministic property*, where the trie path depends only on the key itself, implies that this structure is update-friendly because no reorganizations (prefix splits) are required. At the same time it also allows efficient lookups because no key comparisons are required. However, two typical problems arise when using such a trie structure:

Problem 1 Trie Height: *A problem of this general trie solution is the height of the trie (number of levels). For example, a* `VARCHAR(256)` *index with* $k' = 4$ *would result in a trie height* $h = 512$*. In this scenario, keys with a variable key length* k *smaller than* $h \cdot k'$ *are* logically *padded with zeros at the end in order to ensure an order-preserving deterministic structure. The advantage of tries—in the sense of constant time for all operations—might be a disadvantage because the constant number of nodes to be accessed is really high (*512*). This problem has two facets, namely the number of accessed trie nodes (operational) and the number of created nodes (memory consumption).*
Problem 2 Memory Consumption: *Due to the fixed trie height, an inserted key can cause the full expansion (creation of trie nodes) of all levels of the trie if no prefix can be reused. In the worst case, an insert causes the creation of* $h - 1$ *nodes. Furthermore, each node requires a space of* $2^{k'} \cdot size(ptr)$ byte*. For example, on a 64bit architecture, where a pointer* ptr *requires* 8 byte*, and with a prefix length of* $k' = 4$*, the node size is* 128 byte.

In the following, we tackle these problems. The trie height is reduced by the techniques *trie expansion* and *bypass jumper array*. There, *trie expansion* implicitly leads to decreased memory consumption as well. In addition, we also apply an explicit memory reduction by *pointer reduction* and *memory alignment* (structure compression).

4 Selected Optimization Techniques and Algorithms

In this section, we present selected optimization techniques that address the problems of the potentially large trie height and memory consumption. We also discuss selected algorithms.

4.1 Bypass Jumper Array

Approaches that reduce the large height of the trie (e.g., $h = 512$ for $k = 2048$ and $k' = 4$) are required, while preserving the properties of the generalized trie.

The core concept of the technique *bypass jumper array* is to bypass trie nodes for leading zeros of a key. In order to enable this, we (1) preallocate all direct 0-pointers (nodes where all parents are only referenced by 0-pointers) of the complete trie, and we (2) create a so-called jumper-array of size h, where each cell points to one of those preallocated nodes. Finally, we (3) use the jumper array to bypass higher trie levels if possible. We show the concept of this *bypass jumper array* using our running example:

Example 3 Bypass Jumper Array: *Recall Example 2. For this index of height $h = 4$ (Figure 3), we (1) preallocate all four direct 0-pointers, (2) create the jumper array c of size four, where cell c_j corresponds to the trie level l with $j = l$. Then, we (3) use the jumper array to jump directly to level $l = \lceil |k_i|/k' \rceil$ that is determined by counting leading zeros (clz). For example, when inserting the key $k_i = 107$, we can directly jump to the preallocated node at level 2.*

With regard to the applicability, we distinguish data types of fixed and variable length:

First, for data types with variable-length $|k_i|$ (e.g., `VARCHAR`), we change the index layout. Keys with a length smaller than $h \cdot k'$ are now *logically* padded with zeros at the beginning and thus, we loose the property of being order-preserving. However, if this is acceptable the benefit is significant. We assume that the key length $|k_i|$ is uniformly distributed in the interval $[1, k]$. As a result, the number of accessed trie nodes is reduced from h to $h/2$ in expectation. This causes significant performance improvements due to fewer accessed nodes (logical partitioning into h subtries but with prefix sharing). However, the time complexity is still $O(h)$.

Second, this technique also works for fixed-length data types such as `INT`, by counting leading zeros and without any index layout modification. Unfortunately, for uniformly distributed keys, the impact is lower because the probability of having a key length $|k_i| < k$ depends on k' with $P(\lceil |k_i|/k' \rceil < k/k') = 1/2^{k'}$. For example, if $k' = 4$, only $1/16$ of all keys would benefit at all from this technique. The probability of a specific length is given by $P\left(\lceil |k_i|/k' \rceil = k/k' - x\right) = (1/2^{k'})^x$, where x denotes the number of nodes that can be bypassed. However, numerical values are typically non-uniformly distributed and rather small (e.g., key sequences) and hence, they could also benefit from this technique.

In the description of algorithms, we will refer to this technique as `bypass_top_levels`.

4.2 Trie Expansion

Another reason for the huge trie height is that the index has a fixed height according to its data type and thus, each record is stored on level 1. This is also a reason for the huge memory consumption, because a single record can cause the creation of $h - 1$ new nodes.

Based on this problem, we investigated the dynamic *trie expansion*. The core concept is to defer the access to and creation of trie nodes during insert operations until it is required

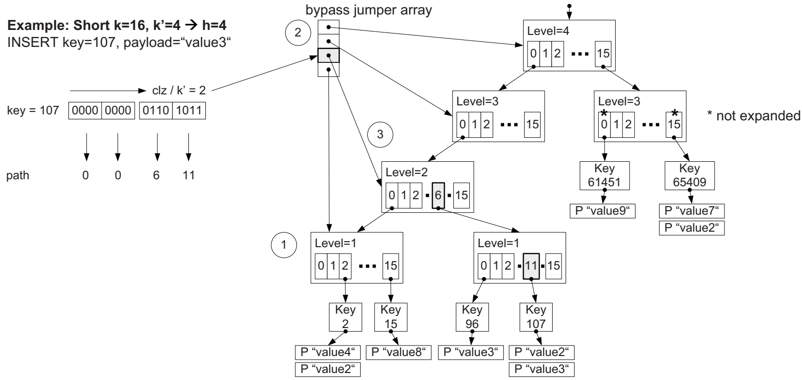


Figure 3: Example `IXByte` for $k = 16, k' = 4, h = 4$ with *bypass jumper array* and *trie expansion*

with respect to the *deterministic property*. In other words, a tuple can now be referenced on any level rather than only at the leaf nodes. It is possible to reference a record instead of an inner node if and only if there is only one record that exhibits a particular prefix (that is given by the position in the trie). A similar concept has already been used within the original algorithm [Fre60] for string indexing. For data types of variable length (e.g., `VARCHAR`), the logical padding with zeros at the end (or beginning, respectively) ensures the deterministic property even in case of trie expansion. We use the following example to illustrate the idea of *trie expansion*:

Example 4 Trie Expansion: *Recall our running example of a* `SHORT(2)` *index with* $h = 4$*. Figure 3 includes an example of the trie expansion. Consider the keys* $k_1 = 61{,}451$ *and* $k_2 = 65{,}409$*. If we insert* $k_2 = 65{,}409$*, we can refer to it on level 4 instead of on level 1 because so far, no other key with a first prefix of* $\texttt{1111}_2$ *exists. Further, if we insert* $k_1 = 61{,}451$*, we need to expand the third level for this sub-trie because* k_1 *and* k_2 *both share the same prefix* $\texttt{1111}_2$*. However, we do not need to expand any further because the second prefix of both keys (*$\texttt{0000}_2$ *and* $\texttt{1111}_2$*) differs. This still ensures the deterministic property because we only expand the deterministic trie path of keys if it is required.*

In contrast to the bypass jumper array, this technique does not only bypass existing nodes but influences the number of created nodes. While the dynamic *trie expansion* significantly reduces the trie height and therefore also the memory consumption for sparsely populated subtries, it requires changes of the trie node data structure. A node is now defined as an array of $s = 2^{k'}$ `void*` pointers and an array of $\lceil s/8 \rceil$ bytes to signal the expansion of a sub-trie. The ith bit of this byte array determines if the ith pointer references an inner node; otherwise, the pointer directly references a record or is `NULL`. This is required due to the generic pointers. However, the evaluation of this flag is a simple bit mask operation. In the algorithmic description, we refer to this as `isExpanded`. While so far, we have required $2^{k'} \cdot size(ptr)$ byte for a node, now a single node has a size of $size(node) = 2^{k'} \cdot size(ptr) + \lceil 2^{k'}/8 \rceil = 2^{k'} \cdot size(ptr) + \lceil 2^{k'-3} \rceil$. As an example, for $k' = 4$ without trie expansion, the node size was 128 byte; enabling trie expansion adds two more bytes to the node. The downside of this technique are costs for splitting and merging of nodes (but still no inner prefix splits) as well as an unaligned data structure

with the drawback that the node size is no longer a factor or divisor of the cache line size. However, due to the significant reduction of created nodes, the total memory consumption is significantly reduced, which improves the performance for all operations.

4.3 Memory Optimization

The solution presented so far still has the drawbacks of (1) creating many small data structures (many `malloc` calls), (2) a fairly high memory consumption, and (3) an unaligned data structure (expanded flags). Hence, we apply several memory optimization techniques.

First, we use *memory preallocation* in order to address the many small data structures. Each instance of an index includes a byte array of configurable size that is created using virtual memory upon index creation. We hold a pointer to this array (`mem_ptr`) and the allocation position *pos* as well as we use lists of free objects for reusing memory. We benefit because the operating system only maps physical pages of this array when they are accessed. Second, we reduce the memory consumption of trie nodes with the concept of *reduced pointers*, which store only the offset within the preallocated memory instead of the pointer to a certain memory position itself. The real pointer is then computed by the reference to the full array (`mem_ptr`) plus the given offset (reduced pointer), which reduces the required size of a pointer; for example (for a preallocated memory of less than 2 GB), from $8\,\mathrm{byte}$ to $4\,\mathrm{byte}$ on 64bit architectures. Third, cache-consciousness has significant influence on the overall performance. We align the trie node size to factors of the cache line size. Based on the concept of reduced pointers, the idea is to use the first bit of such a pointer as internal flag to indicate whether or not this pointer is an expanded node. Following this, we do not need any additional flags per node. As a result, the final structure of our trie node is defined as an array of $s = 2^{k'}$ `uint` reduced pointers resulting in a node size of $size(node) = 2^{k'} \cdot size(rptr) = 2^{k'} \cdot 4$. For example, using $k' = 4$, we get a node size of $64\,\mathrm{byte}$, which is equal to the cache line size of modern processors.

4.4 Algorithms

So far, we have mainly discussed structural properties of the `IXByte`; now, we focus on the operational aspects. The main operations are `get(key)` (point query), `getMin()`, `getNext(key1)` (scan), `insert(key,payload)`, and `delete(key,payload)`. Updates are represented by a delete/insert pair. It is worth mentioning that keys of arbitrary data types are converted into byte arrays such that all operations are only implemented once according to this byte representation. All algorithms to search and modify generalized tries include the optimization approaches *bypass jumper array* and *trie expansion*. We use the operations `get` and `insert` as illustrative example algorithms.

Algorithm 1 shows the `get` algorithm. A single index instance ix includes the root $trie$ node and the level of this root node (given by $h = k/k'$). First, the *bypass jumper array* is used to jump directly to the trie node of level $\lceil |k_i|/k' \rceil$ if required (line 2). Here, $level$ and $node$ are passed by reference and set accordingly. Then, the `get` algorithm mainly comprises a `while` loop (lines 3-10). For each iteration, we go one level down the trie,

Algorithm 1 get (non-recursive)

```
Require: index ix, key k_i, key length |k_i|
 1: node ← ix.trie,   level ← ix.level
 2: bypass_top_levels(node, level, |k_i|, ix)
 3: while (level ← level − 1) > 0 do
 4:    pos ← computePosition(level,k_i,|k_i|)
 5:    if ¬ isExpanded(node,pos) then
 6:       if node.ptr[pos].key = k_i then
 7:          return node.ptr[pos]
 8:       else
 9:          return NULL                          // null if no key or different key
10:    node ← node.ptr[pos]                       // go one level down
```

Algorithm 2 insert (non-recursive)

```
Require: index ix, key k_i, key length |k_i|
 1: node ← ix.trie,   level ← ix.level
 2: bypass_top_levels(node, level, |k_i|, ix)
 3: while (level ← level − 1) > 0 do
 4:    pos ← computePosition(level,k_i,|k_i|)
 5:    if ¬ isExpanded(node,pos) then
 6:       if node.ptr[pos] ≠ NULL then
 7:          entry ← node.ptr[pos]
 8:          if entry.key = k_i then
 9:             return entry
10:          while true do
11:             tmp ←createNode()                 // node prefix splitting
12:             node.ptr[pos1] ← tmp
13:             setExpanded(node,pos)
14:             node ← tmp,   level ← level − 1
15:             pos1 ← computePosition(level,entry.k_i,|k_i|)
16:             pos2 ← computePosition(level,k_i,|k_i|)
17:             if pos1 ≠ pos2 then
18:                node.ptr[pos1] ← entry
19:                return node.ptr[pos2] ←createL1(k_i,|k_i|)
20:             else
21:                pos ← pos1
22:       else
23:          return node.ptr[pos] ←createL1(k_i,|k_i|)
24:    node ← node.ptr[pos]                       // go one level down
```

where the loop terminates if $level = 0$. For each iteration, we first compute the position within the pointer array (`computePosition`, line 4). It is determined by the $(h - l)^{\text{th}}$ prefix of key k_i. If the pointer specified by *pos* is not expanded (see *trie expansion*), there must be a reference to a key, or the pointer is `NULL`. Hence, by checking for the right key (line 6), we could simply return this key partition or `NULL`. Otherwise (the pointer is expanded), there must be a reference to another trie node.

Similar to the get operation, Algorithm 2 uses the same core concept for the insert operation. After we have jumped to the lowest possible trie node in case it is applicable (line 2), the algorithm comprises a main while loop (lines 3-24). The structure of this loop is similar to the get operation. However, we maintain the dynamic *trie expansion* of sub-tries. If a pointer is not expanded and not NULL, we have reached a record. In case the keys are equivalent, we simply return the existing entry. Otherwise, we use an inner while loop (lines 10-21) to expand the sub-trie as long as it is required (leaf node splitting). Basically, we can stop splitting such nodes if we reach a level where the prefixes of both keys are different (lines 17-19). If the current pointer is already expanded, we follow this pointer and go one level down (line 24). Note that this splitting does not require any dynamic reorganization because we just go down the trie as long as it is required.

Both algorithms use the computePosition multiple times in order to compute the current prefix at each trie level. If many levels of the trie are expanded, it is advantageous to pre-compute all positions in advance in one run over the key.

The other algorithms work similar to the presented ones. For getMin, getNext, and delete, we additionally maintain a stack of parent nodes and current positions on each level in order to realize non-recursive algorithms. The getNext searches for the current key, and starting from this position, it returns the minimum key that is greater than this. Further, the delete searches a specific record and then deletes the given (key,value) pair and recursively collapse trie nodes if required.

The index can also store NULL keys, which is required if used as a secondary index. Therefore, an additional single key partition (L1Item) is referenced by the index instance. Furthermore, all algorithms include related NULL checks at the beginning.

4.5 Discussion of Related Work

The closest work compared with our approach is the *Prefix Hash Tree* [CE70], which was defined by Coffmann and Eve from a theoretical perspective. They use a hash function h_j to map the key k_i into a hash code b_i of fixed length j. Thereafter, the prefix hash tree uses prefix-based tree nodes that contain $s = 2^{k'}$ child references. In the case of the simplest hash function of $b_i = h_k(k_i) = k_i$, this is comparable to the generalized trie.

In contrast to this, we presented the *generalized trie* that does not rely on any hash function, i.e., it does not require related indirections for overflow lists of different keys and it can index keys of variable length. Furthermore, we explained the IXByte as a physical realization of the generalized trie including optimizations for the large trie height and memory consumption as well as the description of efficient algorithms using this data structure.

5 Experimental Evaluation

In this section, we present selected experimental results concerning the performance, scalability, memory consumption, and the comparison with existing index structures. We used synthetically generated data, a real data set as well as the MIT main-memory indexing

benchmark [Rei09]. In general, the evaluation shows the following results:

- *Data Types:* The IXByte achieves high performance for arbitrary data types, where an increasing key length causes only moderate overhead.
- *Skew Awareness:* It turns out that the IXByte is particularly appropriate for skewed data (sequence, real) because many keys share equal prefixes, while uniform data represents the worst case.
- *Prefix Length:* A variable prefix length (specific to our IXByte) of $k' = 4$ turned out to be most efficient and robust due to (1) a node size equal to the cacheline size ($64\,\mathrm{byte}$) and (2) a good trade-off between trie height and node utilization.
- *Comparison:* The IXByte shows improvements compared with a B+-tree and a T-tree on different data sets. Most importantly, it exhibits linear scaling on skewed data. Even on uniform data (worst case) it is comparable to a hash map.

As a result, the IXByte can be used as a general-purpose data structure because, especially for skewed in-memory data, it is more efficient than existing tree-based or hash-based solutions. The complete C source code (including all experiments) is available at wwwdb.inf.tu-dresden.de/dexter. Our core data structure IXByte is part of an overall index server, which is able to maintain an arbitrary number of indices (of different data types) in a multi-threaded fashion. Essentially, it implements the API, defined by the MIT main-memory benchmark [Rei09] including memory management, synchronization and transaction logging (transient UNDO-log). All reported results were measured from *outside* the index server—except the comparison with B+-trees.

5.1 Experimental Setting

As our test environment, we used a machine with a quad core Intel Core i7-920 processor (2.67GHz) and hyper-threading (two threads per core). It uses Fedora Core 14 (64bit) as operating system and 6GB of RAM are available. Furthermore, we used the GCC compiler with the following flags: -O3 -fPIC -lpthread -combine.

While the proposed IXByte is able to index arbitrary *data types*, here, we used only the types SHORT ($4\,\mathrm{byte}$), INT ($8\,\mathrm{byte}$) and VARCHAR(128) ($128\,\mathrm{byte}$). In order to evaluate the influence of skew and other data properties we used the following data sets:

- *Key Sequence* (best-case synthetic data): We generated a key sequence (highest skew) of N records with values $[1..N]$ in sorted order. For the data type VARCHAR, we convert this value into a string representation.
- *Uniform Data Distribution* (worst-case synthetic data): In opposite to the key sequence, we additionally generated N records with uniformly distributed keys with values $[1..2^k]$ in unsorted order. For VARCHAR(128), we first determined a random length in the interval $[1..128]$. For each position, we then picked a random character out of the alphabet of 52 printable characters.
- *Real Data:* Aside from the synthetically generated data, we also used the DBLP data set [Ley09] as a real data set. In more detail, we indexed all 2,311,462 distinct key attributes of this data set as a VARCHAR(128) index instance.

We evaluated the different operations `insert`, `get`, `getNext` (scan), and `delete`. Further, we varied the prefix length with $k' \in \{1, 2, 4, 8\}$. In addition, we compared our generalized prefix trie against other index structures: We used an unoptimized B+-tree [Avi09] and optimized it (memory optimizations) similar to our index structure in order to ensure a fair comparison. Furthermore, we used the MIT benchmark as well as the optimized hash map and T-tree implementations from the winner (Clement Genzmer) and another finalist (Cagri Balkesen) of the SIGMOD Programming Contest 2009.

5.2 Synthetically Generated Data

The first set of experiments uses the synthetically generated data sets *sequence* and *uniform*. We fixed a prefix length of $k' = 4$ as well as enabled *trie expansion* and the described memory optimizations. Then, we measured the execution time for the mentioned operations. Figure 4 shows the results of this set of experiments.

Experiment S.1: The first experiment uses the sequence data set. The goal was to evaluate the performance with an increasing number of records N. For each database size, this comprises N `inserts`, N point queries (`get`), a scan of N elements (`getNext`) as well as N `deletes`, where we used all keys of the generated sequence. We repeated this experiment for the three different data types. Figure 4 (first row) shows the measured results. First, `SHORT(4)` shows the best performance due to the lowest trie height ($h = 8$). Note that for sequences, the complete trie height is expanded. We further observe that `INT(8)` ($h = 16$) shows only a minor overhead, while for `VARCHAR` ($h = 256$), this overhead is higher. Those differences are caused by the trie height and additional memory requirements for the larger keys. From an operation type perspective, `delete` always has the worst performance, followed by `insert`, which is reasoned by memory management in addition to the search operations. Further, `getNext` is typically slower than `get`, because for a `getNext`, a similar point query is used in order to find the old key, and from there, it finds the next key, which was required by scans with concurrent updates. Interestingly, `getNext` is better than `get` for `VARCHAR` indices, which was reasoned by cache displacement on `get` due to the huge height for the `VARCHAR` indices. Furthermore, `get`, `insert`, and `delete` require additional overhead for key generation and copying of keys into the local memory of our index server. The technique *bypass jumper array* led to an improvement (not included in the figures) between 9% and 17% for the different operations. However, and most importantly, all operations show (1) a linear scaling according to the number of tuples (constant time for a single operation) due to the fixed number of accessed nodes and (2) a moderate overhead according to the key size.

Experiment S.2: For a second experiment, we repeated Experiment S.1 with the *uniform* data set but exactly the same configuration. Figure 4 (second row) illustrates the measured results of this experiment. We observe similar characteristics—with some exceptions—compared to the *sequence* data set. In general, the index is notably slower for uniform key distributions. This has different reasons. Recall that uniform data is the worst case regarding space requirements, while this is the best case with respect to the average number of accessed nodes per operation. The costs for allocating/loading more memory are higher than the benefit reached by the lower number of trie nodes. Due to uniformly generated

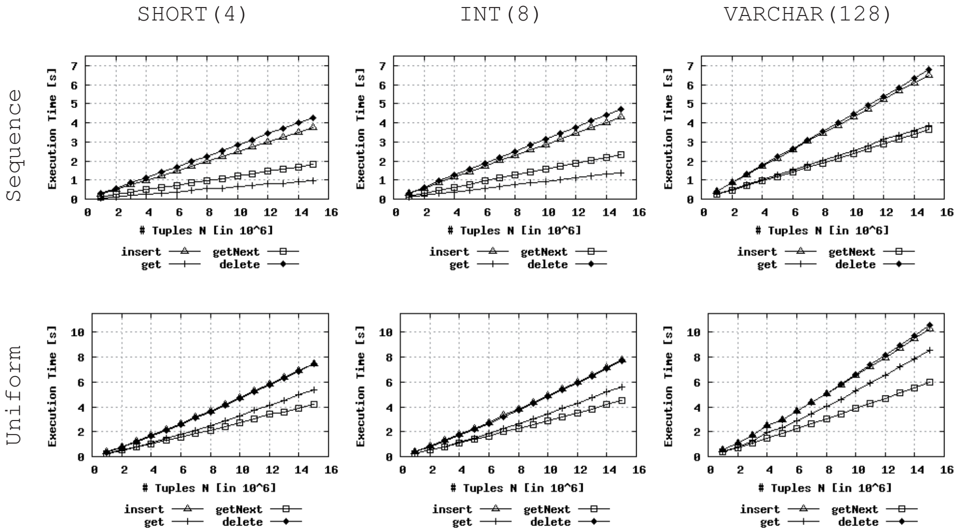


Figure 4: Basic Performance Results

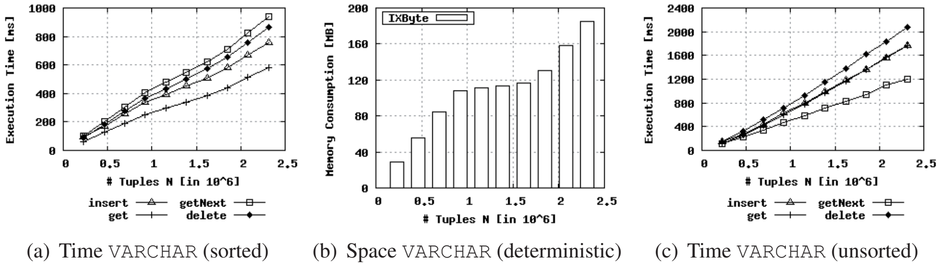
keys, each single operation (except the scan operation) accesses different nodes. As a result of uniform access and the higher memory consumption, many nodes are emitted faster from the cache. Another difference of uniform keys to key sequences is that the execution time increases slightly super-linear with an increasing number of tuples (logarithmic time for a single operation). This effect is caused by the logarithmically increasing number of accessed nodes with an increasing number of tuples. The second difference concerns the scan performance, where the `getNext` is faster than `get` for all data types. Essentially, this is a caching effect because the scan is the only operation that accesses the records in sequential order. Hence, the higher trie nodes are cached across multiple operations. The technique *bypass jumper array* lead to an improvement (not shown in the figures) between 4% and 7% for the different operations, which is lower than in experiment S.1 because the trie does not use the full height for uniform data and thus fewer nodes are bypassed.

As a result, our index shows good performance and an almost linear scaling for sequences (best-case) and uniform data (worst case). However, it is best suited for skewed data.

5.3 Real DBLP Data

The *sequence* and *uniform* data sets are extremes in terms of skew and thus give upper and lower bounds for the number of accessed nodes as well as required space. We now additionally use the real-world DBLP data set that lies within this spectrum.

Experiment R.1: As a preprocessing step, we extracted all distinct publication keys, which are concatenations of the type of publication, the conference/journal name, and the short bibtex key (e.g., `conf/sigmod/LarsonLZZ07`). We used a `VARCHAR(128)` index and truncated all keys that exceeded this key length (only a few). The number of distinct items was 2,311,462. Then, we inserted this data set (in sorted order) and evaluated



(a) Time VARCHAR (sorted) (b) Space VARCHAR (deterministic) (c) Time VARCHAR (unsorted)

Figure 5: Performance on Real Data Set (DBLP)

execution time and space requirements. Figures 5(a) and 5(b) show the results. We observe an almost linear scalability with increasing data size, similar to our other experiments. However, in contrast to sequences and uniform keys, there are two main differences. First, the getNext operation shows the worst performance compared to all other operations, while in the other experiments, it was usually faster than insert and delete, because many prefixes are shared such that insert and delete become more efficient. Similar to this, get is also slower than in other cases compared to the insert/delete functions. This effect is caused by long keys, where we need to access many nodes for each point query. Note that due to transactional requirements each getNext also includes such a point query, which reasoned that getNext was the slowest operation. Furthermore, we see three main parts. Up to 700,000 records, there is a linear scaling. Then, from 700,000 to 1,600,000 records, we see a slower increase. From there to the end, we observe a scaling similar to the first part. This is caused by the huge number of conferences that all share the same prefix (e.g., conf/sigmod/), which results in good compression. It is also worth to note that the performance is strongly correlated to the required memory.

Experiment R.2: As a comparison, we repeated the experiment with unsorted keys (see Figure 5(c)). Due to the deterministic property, the total memory consumption was equivalent to the previous experiment. In contrast to sorted keys, we observe that the performance was much lower due to cache displacement because created nodes are distributed over the memory. The getNext operation now performs best because it is the only one with a sequential access pattern with regard to the trie nodes. Most important, we still observe a linear scaling due to the good compression.

As a result, the trie works also well for real-world data. We conclude that the higher the skew, the higher the performance because we require less memory.

5.4 Comparison with Existing Structures

For comparison with existing structures, we used a general-purpose B^+-tree. We optimized the implementation with memory preallocation in order to achieve a fair comparison. For the IXByte, we used a prefix length of $k' = 4$, while we configured the B^+-Tree as a tree of order four (pointers per node [Knu97]) because experiments with different configurations showed that this leads to highest performance for this experimental setting.

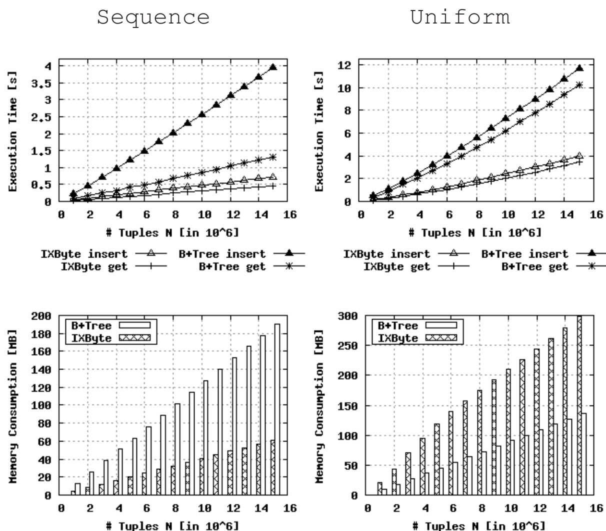


Figure 6: `SHORT(4)` Comparison `IXByte` and B+-Tree

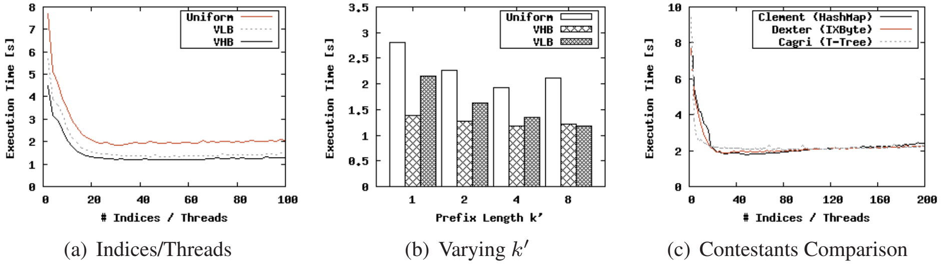
Experiment C.1: Due to the data type restriction of the used B+-tree, we used `SHORT(4)` only. We varied the number of tuples N and measured the execution time for `insert` and `get` as well as the memory consumption. The `delete` operation of the B+-tree was similar to the `insert` and a scan operation was not supported. We first used the *sequence* data set. In contrast to our other experiments, we measured the time for the operations of the core index structures rather than from outside the index server because the B+-tree does not implement the MIT main-memory benchmark API. Figure 6 (left column) shows the results. We yield an execution time improvement of up to factor 5 due to key comparisons and node splitting/merging within the B+-tree. Interestingly, the `IXByte` also requires three times less memory than the B+-tree. However, this is the best case for the prefix trie.

Experiment C.2: In addition, we repeated the experiment with the *uniform* data set. Figure 6 (right column) shows the results. The `IXByte` is still three times faster than the B+-tree. Both index structures are slower for this *uniform* data set. Due to different memory allocation behavior (uniform data is the worst case for `IXByte`), the B+-tree required only 50% of the memory that was required by `IXByte`.

Most importantly, the `IXByte` shows a linear scalability on skewed data with increasing number of tuples, while the B+-tree shows a super-linear scalability for both data sets.

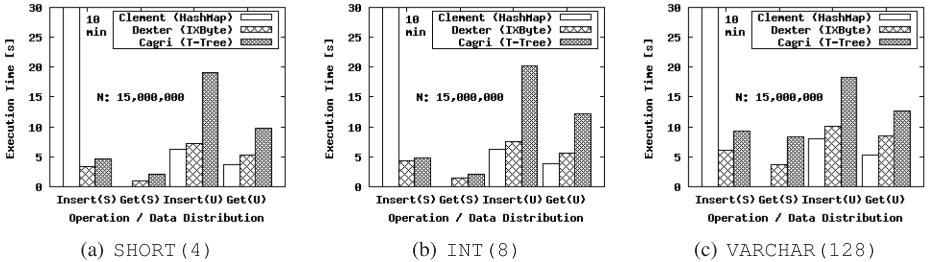
5.5 MIT Benchmark

With regard to repeatability, we provide the results of the MIT main-memory indexing benchmark [Rei09] that was designed during the SIGMOD Programming Contest 2009 (see Figure 7(c) for a comparison with the winning implementation, where the uniform data distribution is the best case for the hash map, while it is the worst case for our `IXByte`). The benchmark is designed with uniform key distributions but there are also test cases for skewed data—namely the test cases of varying low bits (VLB) and varying high bits (VHB). In general, the benchmark creates a random set of indices (of types `SHORT(4)`, `INT(8)`, and `VARCHAR(128)`) and runs a transaction workload in a concurrent fashion, where each transaction comprises between 5 and 200 single operations (point queries, range queries, inserts and deletes). With regard to the repeatability of the



(a) Indices/Threads (b) Varying k' (c) Contestants Comparison

Figure 7: MIT Main-Memory Indexing Benchmark



(a) `SHORT(4)` (b) `INT(8)` (c) `VARCHAR(128)`

Figure 8: Contestants Comparison for Different Operations and Data Sets

obtained benchmark results, we used the same seed (1468) for all sub-experiments. We conducted a set of experiments varying the benchmark parameters, which include the number of indices and threads `NUM_IX`, the number of initial inserts per index `NUM_INSERTS`, and the total number of transactions `NUM_TX` over all indices. The results are shown in Figure 7, where we measured the overall benchmark execution times.

Experiment M.1: We varied the number of indices and threads, respectively. There, we fixed `NUM_INSERTS`=4,000, `NUM_TX`=800,000 (contest configuration) and measured the execution time for the three different data distributions. The results are shown in Figure 7(a). We observe that the index performed best on the VHB benchmark because there, the varied bits are represented by the first two node levels. In contrast, for VLB we fully expanded a long path because the varied bits are represented by the last two node levels. However, for both benchmarks, many keys are shared such that they both exhibit lower execution time than the uniform benchmark; the differences are moderate. Although our test machine has only eight hardware threads, our index server scales pretty well with an increasing number of threads. Note that we reached a CPU utilization of 800% (up from a number of 25 threads).

Experiment M.2: Beside the benchmark parameters, we also evaluated the influence of the prefix length k' on the execution time. We fixed `NUM_IX`=50, `NUM_INSERT`=4,000, and `NUM_TX`=800,000 and varied the prefix length $k' \in (1, 2, 4, 8)$. Figure 7(b) shows the results of this second sub-experiment. Essentially, for different data distributions, different prefix lengths are advantageous. For example, while for uniform data (and VHB), the best prefix length is $k' = 4$, for VLB it is $k' = 8$ because there the index is fully expanded for a certain key range. In general, for skewed data (e.g., *sequence*), larger prefix lengths are advantageous because certain subtries are fully expanded.

Experiment M.3: Finally, we use the optimized hash map and T-tree implementations of the mentioned other contest finalists for a more detailed comparison regarding different operations (insert, get), data types (SHORT(4), INT(8), VARCHAR(128)) and data sets (sequence S and uniform U), where the results are shown in Figure 8. The hash map did not terminate within $10\,\mathrm{min}$ on the sequence data set such that these experiments were aborted. In contrast, we observe that our index performs best on sequence data, where the relative improvement over the T-tree is highest for VARCHAR due to the higher influence of the number of required key comparisons. On uniform data, the hash map performed best but our IXByte achieves only slightly worse performance and still outperforms the T-tree. In conclusions, the IXByte is especially beneficial in case of skewed data.

6 Conclusions

In-memory data structures are gaining importance but still exhibit certain disadvantages. While these are inherently given for tree-based and hash-based structures, we addressed the drawbacks of trie structures in order to make them applicable for in-memory indexing.

To summarize, we presented a generalized prefix tree (trie) for arbitrary data types that uses a variable prefix length. The major advantage is the deterministic property that eliminates the need for dynamic reorganization (prefix splits) and multiple key comparisons. Furthermore, we presented optimization techniques for reducing the number of accessed and created trie nodes (*bypass jumper array* and *trie expansion*). Additionally, we briefly discussed memory optimization techniques and efficient algorithms. In conclusion, the generalized trie is advantageous compared to existing solutions. The major benefit is an almost linear scalability with respect to the number of tuples, especially for skewed data.

Using this generalized trie, several further research issues arise, which include, for example, the (1) adaptive determination of the optimal prefix length k' during runtime, (2) a hybrid solution with different prefix lengths on different levels of a generalized trie (e.g., $\{16, 8, 4, 4\}$ for SHORT(4)), and (3) query processing on prefix trees, where database operations such as joins or set operations can exploit the deterministic trie paths.

References

[AS07] Nikolas Askitis and Ranjan Sinha. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC*, 2007.

[Avi09] Amittai Aviram. *B+ Tree Implementation*, 2009. http://www.amittai.com/prose/ bplustree.html.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1, 1972.

[BMR01] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD*, 2001.

[BS97] Jon Louis Bentley and Robert Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *SODA*, 1997.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1), 1977.

[CE70] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Commun. ACM*, 13(7), 1970.

[CGMV02] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B$\pm$Trees: optimizing both cache and disk performance. In *SIGMOD*, 2002.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.*, 4(3), 1979.

[Fre60] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3(9), 1960.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *FOCS*, 1978.

[HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2), 2002.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[Jan95] Jan Jannink. Implementing Deletion in B+-Trees. *SIGMOD Record*, 24(1), 1995.

[KCS$^+$10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.

[Knu97] Donald E. Knuth. *The art of computer programming Volume I. Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997.

[Lar88] Per-Åke Larson. Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval. *ACM Trans. Database Syst.*, 13, 1988.

[LC86] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*, 1986.

[LDW09] Hua Luan, Xiaoyong Du, and Shan Wang. Prefetching J$^+$-Tree: A Cache-Optimized Main Memory Database Index Structure. *J. Comput. Sci. Technol.*, 24(4), 2009.

[Ley09] Michael Ley. DBLP - Some Lessons Learned. In *VLDB*, 2009.

[Lit80] Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *VLDB*, 1980.

[LSLC07] Ig Hoon Lee, Junho Shim, Sang Goo Lee, and Jonghoon Chun. CST-Trees: Cache Sensitive T-Trees. In *DASFAA*, 2007.

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3), 2000.

[Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4), 1968.

[Rei09] Elizabeth G. Reid. Design and Evaluation of a Benchmark for Main Memory Transaction Processing Systems. Master's thesis, MIT, 2009.

[Ros07] Kenneth A. Ross. Efficient Hash Probes on Modern Processors. In *ICDE*, 2007.

[RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *SIGMOD*, 2000.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-Ary Search on Modern Processors. In *DaMoN*, 2009.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.