The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann

Fakultät für Informatik
Technische Universität München
Boltzmannstrae 3, D-85748 Garching
<lastname>@in.tum.de

***Abstract*—Main memory capacities have grown up to a point where most databases fit into RAM. For main-memory database systems, index structure performance is a critical bottleneck. Traditional in-memory data structures like balanced binary search trees are not efficient on modern hardware, because they do not optimally utilize on-CPU caches. Hash tables, also often used for main-memory indexes, are fast but only support point queries.**

To overcome these shortcomings, we present ART, an adaptive radix tree (trie) for efficient indexing in main memory. Its lookup performance surpasses highly tuned, read-only search trees, while supporting very efficient insertions and deletions as well. At the same time, ART is very space efficient and solves the problem of excessive worst-case space consumption, which plagues most radix trees, by adaptively choosing compact and efficient data structures for internal nodes. Even though ART's performance is comparable to hash tables, it maintains the data in sorted order, which enables additional operations like range scan and prefix lookup.

I. Introduction

After decades of rising main memory capacities, even large transactional databases fit into RAM. When most data is cached, traditional database systems are CPU bound because they spend considerable effort to avoid disk accesses. This has led to very intense research and commercial activities in main-memory database systems like H-Store/VoltDB [1], SAP HANA [2], and HyPer [3]. These systems are optimized for the new hardware landscape and are therefore much faster. Our system HyPer, for example, compiles transactions to machine code and gets rid of buffer management, locking, and latching overhead. For OLTP workloads, the resulting execution plans are often sequences of index operations. Therefore, index efficiency is the decisive performance factor.

More than 25 years ago, the T-tree [4] was proposed as an in-memory indexing structure. Unfortunately, the dramatic processor architecture changes have rendered T-trees, like all traditional binary search trees, inefficient on modern hardware. The reason is that the ever growing CPU cache sizes and the diverging main memory speed have made the underlying assumption of uniform memory access time obsolete. B$^+$-tree variants like the cache sensitive B$^+$-tree [5] have more cache-friendly memory access patterns, but require more expensive update operations. Furthermore, the efficiency of both binary and B$^+$-trees suffers from another feature of modern CPUs: Because the result of comparisons cannot be predicted easily, the long pipelines of modern CPUs stall, which causes additional latencies after every second comparison (on average).

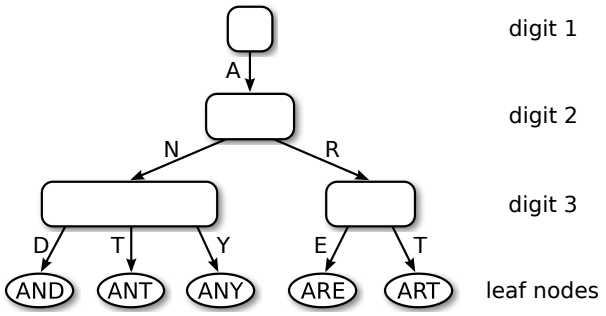

Fig. 1. Adaptively sized nodes in our radix tree.

These problems of traditional search trees were tackled by recent research on data structures specifically designed to be efficient on modern hardware architectures. The k-ary search tree [6] and the Fast Architecture Sensitive Tree (FAST) [7] use data level parallelism to perform multiple comparisons simultaneously with Singe Instruction Multiple Data (SIMD) instructions. Additionally, FAST uses a data layout which avoids cache misses by optimally utilizing cache lines and the Translation Lookaside Buffer (TLB). While these optimizations improve search performance, both data structures cannot support incremental updates. For an OLTP database system which necessitates continuous insertions, updates, and deletions, an obvious solution is a differential file (delta) mechanism, which, however, will result in additional costs.

Hash tables are another popular main-memory data structure. In contrast to search trees, which have $O(\log n)$ access time, hash tables have expected $O(1)$ access time and are therefore much faster in main memory. Nevertheless, hash tables are less commonly used as database indexes. One reason is that hash tables scatter the keys randomly, and therefore only support point queries. Another problem is that most hash tables do not handle growth gracefully, but require expensive reorganization upon overflow with $O(n)$ complexity. Therefore, current systems face the unfortunate trade-off between fast hash tables that only allow point queries and fully-featured, but relatively slow, search trees.

A third class of data structures, known as trie, radix tree, prefix tree, and digital search tree, is illustrated in Figure 1.

These data structures directly use the digital representation of keys instead of hashing or comparing keys. The underlying idea is similar to a thumb-index found in many alphabetically ordered dictionary books: The first character of a word can directly be used to jump to all words starting with that character. In a computer, this process can be repeated with the next characters until a match is found. As a consequence of this process, all operations have $O(k)$ complexity where k is the length of the key. In the era of extremely large data sets, when n is growing faster than k, having a time complexity independent of n is very attractive.

In this work, we present the adaptive radix tree (ART) which is a fast and space-efficient in-memory indexing structure specifically tuned for modern hardware. While most radix trees require to trade off tree height versus space efficiency by setting a globally valid fanout parameter, ART adapts the representation of every individual node, as exemplified in Figure 1. By adapting each inner node *locally*, it optimizes *global* space utilization and access efficiency at the same time. Nodes are represented using a small number of efficient and compact data structures, chosen dynamically depending on the number of child nodes. Two additional techniques, path compression and lazy expansion, allow ART to efficiently index long keys by collapsing nodes and thereby decreasing the tree height.

A useful property of radix trees is that the order of the keys is not random as in hash tables; rather, the keys are ordered bitwise lexicographically. We show how typical data types can be reordered efficiently to support all operations that require the data to be ordered (e.g., range scan, prefix lookup, top-k, minimum, and maximum).

This work makes the following contributions:

- We develop the adaptive radix tree (ART), a fast and space-efficient general-purpose indexing structure for main-memory database systems.
- We prove that the space consumption per key is bounded to 52 bytes, even for arbitrarily long keys. We show experimentally, that the space consumption is much lower in practice, often as low as 8.1 bytes per key.
- We describe how common built-in data types can be stored in radix trees while retaining their order.
- We experimentally evaluate ART and other state of the art main-memory index structures, including the most efficient search tree proposals.
- By integrating ART into the main-memory database system HyPer and running the TPC-C benchmark, we prove its superior end-to-end performance in a "real-life" transaction processing application.

The rest of this paper is organized as follows. The next section discusses related work. Section III presents the adaptive radix tree and analyzes its space consumption. In Section IV we introduce the concept of binary-comparable keys and show how common built-in types can be transformed. Section V describes experimental results including a number of micro benchmarks and the TPC-C benchmark. Finally, Section VI concludes and discusses future work.

II. Related Work

In disk-based database systems, the B$^+$-tree [8] is ubiquitous [9]. It retrieves large blocks from disk to reduce the number of accesses. Red-black trees [10], [11] and T-trees [4] were proposed for main-memory database systems. Rao and Ross [5] showed that T-trees, like all binary search trees, suffer from poor cache behavior and are therefore often slower than B$^+$-trees on modern hardware. As an alternative, they proposed a cache conscious B$^+$-tree variant, the CSB$^+$-tree [12]. Further cache optimizations for B$^+$-trees were surveyed by Graefe and Larson [13].

Modern CPUs allow to perform multiple comparisons with a single SIMD instruction. Schlegel et al. [6] proposed k-ary search which reduces the number of comparisons from $\log_2 n$ to $\log_K n$ where K is the number of keys that fit into one SIMD vector. In comparison with binary trees, this technique also reduces the number of cache misses, because K comparisons are performed for each cache line loaded from main memory. Kim et al. extended this research by proposing FAST, a methodology for laying out binary search trees in an architecture sensitive way [7]. SIMD, cache line, and page blocking are used to optimally use the available cache and memory bandwidth. Additionally, they proposed to interleave the stages of multiple queries in order to increase the throughput of their search algorithm. FAST trees and the k-ary search trees are pointer-free data structures which store all keys in a single array and use offset calculations to traverse the tree. While this representation is efficient and saves space, it also implies that no online updates are possible. Kim et al. also presented a GPU implementation of FAST and compared its performance to modern CPUs. Their results show that, due to the higher memory bandwidth, GPUs can achieve higher throughput than CPUs. Nevertheless, the use of GPUs as dedicated indexing hardware is not yet practical because memory capacities of GPUs are limited, communications cost with main memory is high, and hundreds of parallel queries are needed to achieve high throughput. We, therefore focus on index structures for CPUs.

The use of tries for indexing character strings has been studied extensively. The two earliest variants use lists [14] and arrays [15] as internal node representations. Morrison introduced path compression in order to store long strings efficiently [16]. Knuth [17] analyzes these early trie variants. The burst trie is a more recent proposal which uses trie nodes for the upper levels of the tree, but switches to a linked list once a subtree has only few elements. The HAT-trie [18] improves performance by replacing the linked list with a hash table. While most research focused on indexing character strings, our goal is to index other data types as well. Therefore, we prefer the term radix tree over trie because it underscores the similarity to the radix sort algorithm and emphasizes that arbitrary data can be indexed instead of only character strings.

The Generalized Prefix Tree was proposed by Boehm et al. [19] as a general-purpose indexing structure. It is a radix tree with a fanout of 16 and was a finalist in the SIGMOD

Programming Contest 2009. The KISS-Tree [20] is a more efficient radix tree proposal with only three levels, but can only store 32 bit keys. It uses an open addressing scheme for the first 16 bits of the key and relies on the virtual memory system to save space. The second level, responsible for the next 10 bits, uses an array representation and the final level compresses 6 bits using a bit vector. The idea of dynamically changing the internal node representation is used by the Judy array data structure which was developed at Hewlett-Packard research labs [21], [22].

Graefe discusses binary-comparable ("normalized") keys, e.g. [23], as a way of simplifying and speeding up key comparisons. We use this concept to obtain meaningful order for the keys stored in radix trees.

III. Adaptive Radix Tree

This section presents the adaptive radix tree (ART). We start with some general observations on the advantages of radix trees over comparison-based trees. Next, we motivate the use of adaptive nodes by showing that the space consumption of conventional radix trees can be excessive. We continue with describing ART and algorithms for search and insertion. Finally, we analyze the space consumption.

A. Preliminaries

Radix trees have a number of interesting properties that distinguish them from comparison-based search trees:

- The height (and complexity) of radix trees depends on the length of the keys but in general not on the number of elements in the tree.
- Radix trees require no rebalancing operations and all insertion orders result in the same tree.
- The keys are stored in lexicographic order.
- The path to a leaf node represents the key of that leaf. Therefore, keys are stored implicitly and can be reconstructed from paths.

Radix trees consist of two types of nodes: Inner nodes, which map partial keys to other nodes, and leaf nodes, which store the values corresponding to the keys. The most efficient representation of an inner node is as an array of 2^s pointers. During tree traversal, an s bit chunk of the key is used as the index into that array and thereby determines the next child node without any additional comparisons. The parameter s, which we call *span*, is critical for the performance of radix trees, because it determines the height of the tree for a given key length: A radix tree storing k bit keys has $\lceil k/s \rceil$ levels of inner nodes. With 32 bit keys, for example, a radix tree using $s = 1$ has 32 levels, while a span of 8 results in only 4 levels.

Because comparison-based search trees are the prevalent indexing structures in database systems, it is illustrative to compare the height of radix trees with the number of comparisons in perfectly balanced search trees. While each comparison rules out half of all values in the best case, a radix tree node can rule out more values if $s > 1$. Therefore, radix trees have smaller height than binary search trees for $n > 2^{k/s}$. This relationship is illustrated in Figure 2 and assumes that keys

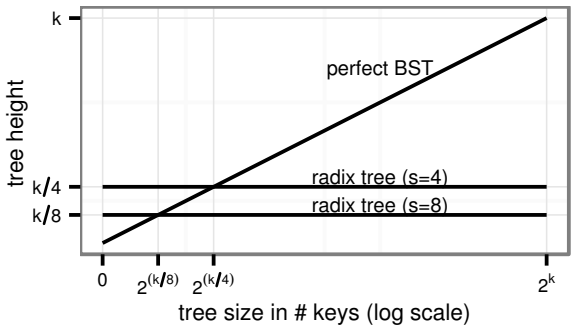

Fig. 2. Tree height of perfectly balanced binary search trees and radix trees.

can be compared in $O(1)$ time. For large keys, comparisons actually take $O(k)$ time and therefore the complexity of search trees is $O(k \log n)$, as opposed to the radix tree complexity of $O(k)$. These observations suggest that radix trees, in particular with a large span, can be more efficient than traditional search trees.

B. Adaptive Nodes

As we have seen, from a (pure lookup) performance standpoint, it is desirable to have a large span. When arrays of pointers are used to represent inner nodes, the disadvantage of a large span is also clear: Space usage can be excessive when most child pointers are null. This tradeoff is illustrated in Figure 3 which shows the height and space consumption for different values of the span parameter when storing 1M uniformly distributed 32 bit integers. As the span increases, the tree height decreases linearly, while the space consumption increases exponentially. Therefore, in practice, only some values of s offer a reasonable tradeoff between time and space. For example, the Generalized Prefix Tree (GPT) uses a span of 4 bits [19], and the radix tree used in the Linux kernel (LRT) uses 6 bits [24]. Figure 3 further shows that our adaptive radix tree (ART), *at the same time*, uses *less space* and has *smaller height* than radix trees that only use homogeneous array nodes.

The key idea that achieves both space and time efficiency is to adaptively use different node sizes with the same, relatively large span, but with different fanout. Figure 4 illustrates this

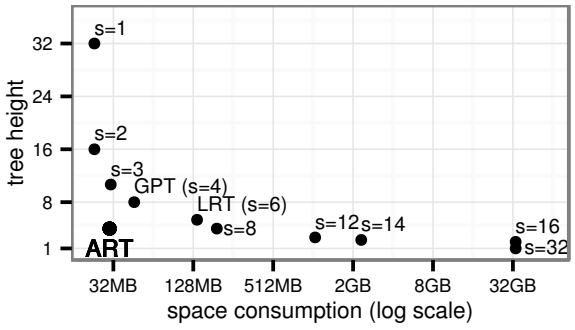

Fig. 3. Tree height and space consumption for different values of the span parameter s when storing 1M uniformly distributed 32 bit integers. Pointers are 8 byte long and nodes are expanded lazily.

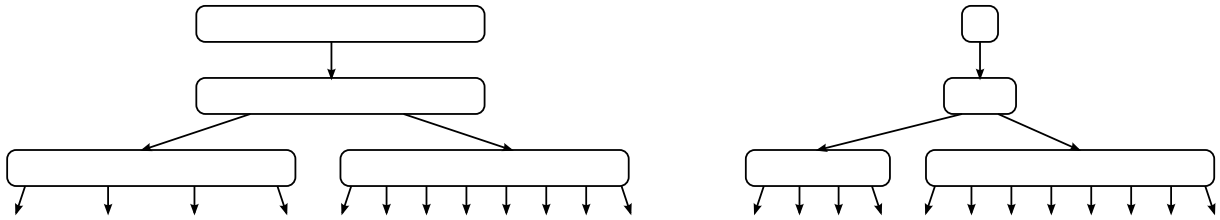

Fig. 4. Illustration of a radix tree using array nodes (left) and our adaptive radix tree ART (right).

idea and shows that adaptive nodes do not affect the structure (i.e., height) of the tree, only the sizes of the nodes. By reducing space consumption, adaptive nodes allow to use a larger span and therefore increase performance too.

In order to efficiently support incremental updates, it is too expensive to resize nodes after each update. Therefore, we use a small number of node types, each with a different fanout. Depending on the number of non-null children, the appropriate node type is used. When the capacity of a node is exhausted due to insertion, it is replaced by a larger node type. Correspondingly, when a node becomes underfull due to key removal, it is replaced by a smaller node type.

C. Structure of Inner Nodes

Conceptually, inner nodes map partial keys to child pointers. Internally, we use four data structures with different capacities. Given the next key byte, each data structure allows to efficiently find, add, and remove a child node. Additionally, the child pointers can be scanned in sorted order, which allows to implement range scans. We use a span of 8 bits, corresponding to partial keys of 1 byte and resulting a relatively large fanout. This choice also has the advantage of simplifying the implementation, because bytes are directly addressable which avoids bit shifting and masking operations.

The four node types are illustrated in Figure 5 and are named according to their maximum capacity. Instead of using a list of key/value pairs, we split the list into one key part and one pointer part. This allows to keep the representation compact while permitting efficient search:

Node4: The smallest node type can store up to 4 child pointers and uses an array of length 4 for keys and another array of the same length for pointers. The keys and pointers are stored at corresponding positions and the keys are sorted.

Node16: This node type is used for storing between 5 and 16 child pointers. Like the `Node4`, the keys and pointers are stored in separate arrays at corresponding positions, but both arrays have space for 16 entries. A key can be found efficiently with binary search or, on modern hardware, with parallel comparisons using SIMD instructions.

Node48: As the number of entries in a node increases, searching the key array becomes expensive. Therefore, nodes with more than 16 pointers do not store the keys explicitly. Instead, a 256-element array is used, which can be indexed with key bytes directly. If a node has between 17 and 48 child pointers, this array stores indexes into a second array which contains up to 48 pointers. This indirection saves space in comparison to 256 pointers of 8 bytes, because the indexes only require 6 bits (we use 1 byte for simplicity).

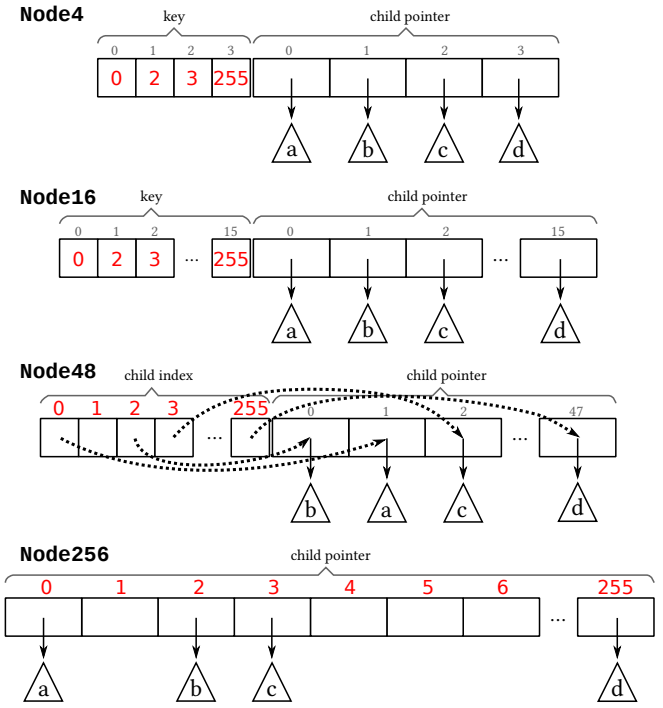

Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

Node256: The largest node type is simply an array of 256 pointers and is used for storing between 49 and 256 entries. With this representation, the next node can be found very efficiently using a single lookup of the key byte in that array. No additional indirection is necessary. If most entries are not null, this representation is also very space efficient because only pointers need to be stored.

Additionally, at the front of each inner node, a header of constant size (e.g., 16 bytes) stores the node type, the number of children, and the compressed path (cf. Section III-E).

D. Structure of Leaf Nodes

Besides storing paths using the inner nodes as discussed in the previous section, radix trees must also store the values associated with the keys. We assume that only unique keys are stored, because non-unique indexes can be implemented by appending the tuple identifier to each key as a tie-breaker.

The values can be stored in different ways:

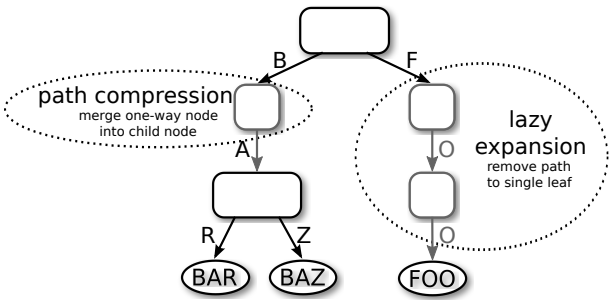

Fig. 6. Illustration of lazy expansion and path compression.

```
1  search (node, key, depth)
2  if node==NULL
3    return NULL
4  if isLeaf(node)
5    if leafMatches(node, key, depth)
6      return node
7    return NULL
8  if checkPrefix(node,key,depth)!=node.prefixLen
9    return NULL
10 depth=depth+node.prefixLen
11 next=findChild(node, key[depth])
12 return search(next, key, depth+1)
```

Fig. 7. Search algorithm.

- Single-value leaves: The values are stored using an additional leaf node type which stores one value.
- Multi-value leaves: The values are stored in one of four different leaf node types, which mirror the structure of inner nodes, but contain values instead of pointers.
- Combined pointer/value slots: If values fit into pointers, no separate node types are necessary. Instead, each pointer storage location in an inner node can either store a pointer or a value. Values and pointers can be distinguished using one additional bit per pointer or with pointer tagging.

Using single-value leaves is the most general method, because it allows keys and values of varying length within one tree. However, because of the increased tree height, it causes one additional pointer traversal per lookup. Multi-value leaves avoid this overhead, but require all keys in a tree to have the same length. Combined pointer/value slots are efficient and allow to store keys of varying length. Therefore, this method should be used if applicable. It is particularly attractive for secondary database indexes which store tuple identifiers with the same size as pointers.

E. Collapsing Inner Nodes

Radix trees have the useful property that each inner node represents a key prefix. Therefore, the keys are implicitly stored in the tree structure, and can be reconstructed from the paths to the leaf nodes. This saves space, because the keys do not have to be stored explicitly. Nevertheless, even with this implicit prefix compression of keys and the use of adaptive nodes, there are cases, in particular with long keys, where the space consumption per key is large. We therefore use two additional, well-known techniques that allow to decrease the height by reducing the number of nodes. These techniques are very effective for long keys, increasing performance significantly for such indexes. Equally important is that they reduce space consumption, and ensure a small worst-case space bound.

With the first technique, lazy expansion, inner nodes are only created if they are required to distinguish at least two leaf nodes. Figure 6 shows an example where lazy expansion saves two inner nodes by truncating the path to the leaf "FOO". This path is expanded if another leaf with the prefix "F" is inserted. Note that because paths to leaves may be truncated, this optimization requires that the key is stored at the leaf or can be retrieved from the database.

Path compression, the second technique, removes all inner nodes that have only a single child. In Figure 6, the inner node storing the partial key "A" was removed. Of course, this partial key cannot simply be ignored. There are two approaches to deal with it:

- Pessimistic: At each inner node, a variable length (possibly empty) partial key vector is stored. It contains the keys of all preceding one-way nodes which have been removed. During lookup this vector is compared to the search key before proceeding to the next child.
- Optimistic: Only the count of preceding one-way nodes (equal to the length of the vector in the pessimistic approach) is stored. Lookups just skip this number of bytes without comparing them. Instead, when a lookup arrives at a leaf its key must be compared to the search key to ensure that no "wrong turn" was taken.

Both approaches ensure that each inner node has at least two children. The optimistic approach is particularly beneficial for long strings but requires one additional check, while the pessimistic method uses more space, and has variable sized nodes leading to increased memory fragmentation. We therefore use a hybrid approach by storing a vector at each node like in the pessimistic approach, but with a constant size (8 bytes) for all nodes. Only when this size is exceeded, the lookup algorithm dynamically switches to the optimistic strategy. Without wasting too much space or fragmenting memory, this avoids the additional check in cases that we investigated.

F. Algorithms

We now present the algorithms for search and updates:

Search: Pseudo code for search is shown in Figure 7. The tree is traversed by using successive bytes of the key array until a leaf node or a null pointer is encountered. Line 4 handles lazy expansion by checking that the encountered leaf fully matches the key. Pessimistic path compression is handled in lines 7 and 8 by aborting the search if the compressed path does not match the key. The next child node is found by the `findChild` function, which is shown in Figure 8. Depending on the node type the appropriate search algorithm is executed: Because a `Node4` has only 2-4 entries, we use

```
findChild (node, byte)
1  if node.type==Node4 // simple loop
2    for (i=0; i<node.count; i=i+1)
3      if node.key[i]==byte
4        return node.child[i]
5    return NULL
6  if node.type==Node16 // SSE comparison
7    key=_mm_set1_epi8(byte)
8    cmp=_mm_cmpeq_epi8(key, node.key)
9    mask=(1<<node.count)-1
10   bitfield=_mm_movemask_epi8(cmp)&mask
11   if bitfield
12     return node.child[ctz(bitfield)]
13   else
14     return NULL
15 if node.type==Node48 // two array lookups
16   if node.childIndex[byte]!=EMPTY
17     return node.child[node.childIndex[byte]]
18   else
19     return NULL
20 if node.type==Node256 // one array lookup
21   return node.child[byte]
```

Fig. 8. Algorithm for finding a child in an inner node given a partial key.

a simple loop. For a `Node16`, the pseudo code shows an SIMD implementation using SSE instructions, which allow to compare 16 keys with one instruction *in parallel*. First, the searched key is replicated (line 7) and then compared to the 16 keys stored in the inner node (line 8). In the next step, a mask is created (line 9), because the node may have less than 16 valid entries. The result of the comparison is converted to a bit field and the mask is applied (line 10). Finally, the bit field is converted to an index using the count trailing zero instruction (line 12). Alternatively, binary search can be used if SIMD instructions are not available. Lookup in a `Node48` is performed by first checking if the `childIndex` entry is valid, and then returning the corresponding pointer. A `Node256` lookup consists of only a single array access.

Insert: The pseudo code is shown in Figure 9. The tree is traversed using the recursive call in line 29, until the position for the new leaf is found. Usually, the leaf can simply be inserted into an existing inner node, after growing it if necessary (lines 31-33). If, because of lazy expansion, an existing leaf is encountered, it is replaced by a new inner node storing the existing and the new leaf (lines 5-13). Another special case occurs if the key of the new leaf differs from a compressed path: A new inner node is created above the current node and the compressed paths are adjusted accordingly (lines 17-24). We omit some helper functions for lack of space: `replace` substitutes a node in the tree by another node, `addChild` appends a new child to an inner node, `checkPrefix` compares the compressed path of a node with the key and returns the number of equal bytes, `grow` replaces a node by a larger node type, and `loadKey` retrieves the key of a leaf from the database.

Bulk loading: When an index is created for an existing relation, the following recursive algorithm can be used to speed up index construction: Using the first byte of each key the key/value pairs are radix partitioned into 256 partitions and an

```
insert (node, key, leaf, depth)
1  if node==NULL // handle empty tree
2    replace(node, leaf)
3    return
4  if isLeaf(node) // expand node
5    newNode=makeNode4()
6    key2=loadKey(node)
7    for (i=depth; key[i]==key2[i]; i=i+1)
8      newNode.prefix[i-depth]=key[i]
9    newNode.prefixLen=i-depth
10   depth=depth+newNode.prefixLen
11   addChild(newNode, key[depth], leaf)
12   addChild(newNode, key2[depth], node)
13   replace(node, newNode)
14   return
15 p=checkPrefix(node, key, depth)
16 if p!=node.prefixLen // prefix mismatch
17   newNode=makeNode4()
18   addChild(newNode, key[depth+p], leaf)
19   addChild(newNode, node.prefix[p], node)
20   newNode.prefixLen=p
21   memcpy(newNode.prefix, node.prefix, p)
22   node.prefixLen=node.prefixLen-(p+1)
23   memmove(node.prefix,node.prefix+p+1,node.prefixLen)
24   replace(node, newNode)
25   return
26 depth=depth+node.prefixLen
27 next=findChild(node, key[depth])
28 if next // recurse
29   insert(next, key, leaf, depth+1)
30 else // add to inner node
31   if isFull(node)
32     grow(node)
33   addChild(node, key[depth], leaf)
```

Fig. 9. Insert algorithm.

inner node of the appropriate type is created. Before returning that inner node, its children are created by recursively applying the bulk loading procedure for each partition using the next byte of each key.

Delete: The implementation of deletion is symmetrical to insertion. The leaf is removed from an inner node, which is shrunk if necessary. If that node now has only one child, it is replaced by its child and the compressed path is adjusted.

G. Space Consumption

Even though servers with terabytes of RAM are readily available, main memory is still a precious resource. Therefore, index structures should be as compact as possible. For radix trees, the space consumption depends on the distribution of the stored keys. Dense keys (e.g., integers ranging from 1 to n) are the best case, and can be stored space efficiently, even using a large span and without adaptive nodes. When the keys are sparse on the other hand, many pointers of inner nodes are null, which results in wasted space. Skewed keys cause some nodes to contain mostly null pointers, and other nodes to be densely packed. Adaptive nodes solve these problems and ensure that any key distribution is stored compactly by locally optimizing the space consumption at each node.

We now analyze the worst-case space consumption per key, taking into account adaptive nodes, lazy expansion, and path compression. For the following analysis, we assume that pointers are 8 bytes long and that each node has a 16 byte

TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

Type	Children	Space (bytes)
Node4	2-4	$16 + 4 + 4 \cdot 8 = 52$
Node16	5-16	$16 + 16 + 16 \cdot 8 = 160$
Node48	17-48	$16 + 256 + 48 \cdot 8 = 656$
Node256	49-256	$16 + 256 \cdot 8 = 2064$

TABLE II
WORST-CASE SPACE CONSUMPTION PER KEY (IN BYTES) FOR DIFFERENT RADIX TREE VARIANTS WITH 64 BIT POINTERS.

	$k = 32$	$k \to \infty$
ART	43	52
GPT	256	∞
LRT	2048	∞
KISS	>4096	NA.

header storing the node type, the number of non null children, and the compressed path. We only consider inner nodes and ignore leaf nodes because leaves incur no space overhead if combined pointer/value slots with tagged pointers are used. Using these assumptions, the resulting space consumption for each inner node type is shown in Table I.

Think of each leaf as providing x bytes and inner nodes as consuming space provided by their children. If each node of a tree has a positive budget, then that tree uses less than x bytes per key. The budget of an inner node is the sum of all budgets of its children minus the space consumption of that node. Formally, the budget $b(n)$ of a node n with the child nodes $c(n)$ and the space consumption $s(n)$ is defined as

$$b(n) = \begin{cases} x, & \text{isLeaf}(n) \\ \left(\sum_{i \in c(n)} b(i)\right) - s(n), & \text{else.} \end{cases}$$

Using induction on the node type, we now show that $b(n) \geq 52$ for every ART node n if $x = 52$: For leaves, the statement holds trivially by definition of the budget function. To show that the statement holds for inner nodes, we compute a lower bound for $\sum_{i \in c(n)} b(i)$ using the induction hypothesis and the minimum number of children for each node type. After subtracting the corresponding space consumption, we obtain a lower bound on the budget of each node type. In all four cases it is greater than or equal to 52, which concludes the induction. To summarize, we have shown that for $x = 52$ it is not possible to construct an adaptive radix tree node that has a negative budget. As a consequence, the worst-case space consumption is 52 bytes for any adaptive radix tree, even for arbitrarily long keys. It can also be shown analogously that with six node types[1], the worst-case space consumption can be reduced to 34 bytes per key. As we will show in Section V-D, in practice, the space consumption is much smaller than the worst case, even for relatively long keys. The best case of 8.1 bytes, however, does occur quite frequently because surrogate integer keys are often dense.

[1]The Node4 type is replaced by the new node types Node2 and Node5 and the Node48 type is replaced by the new Node32 and Node64 types.

Let us close with a comparison to other radix trees which is summarized in Table II. Because the Generalized Prefix Tree and the Linux kernel radix tree do not use path compression, the number of inner nodes is proportional to the length of the keys. Therefore, the worst-case space consumption per key is not bounded. Furthermore, even for short keys, both data structures have a much higher worst-case space consumption than ART because they do not use adaptive nodes. The worst-case space consumption of the KISS-Tree is over 4KB per key, and occurs, for example, with the unsigned integer keys $\{i \cdot 2^{16} \,|\, i \in \{0, 1, \ldots, 2^{16} - 1\}\}$.

IV. CONSTRUCTING BINARY-COMPARABLE KEYS

An important aspect in choosing an indexing structure is whether or not the data is stored in sorted order. The sorted order traversal of an index structure facilitates the implementation of efficient ordered range scans and lookups for minimum, maximum, top-N, etc. By default, only comparison-based trees store the data in sorted order, which resulted in their prevalence as indexing structures for database systems. While the use of order-preserving hashing has been proposed to allow hash table elements to be sorted, it is not common in real-world systems. The reason is that for values from unknown distributions, it is very hard to come up with functions that spread the input values uniformly over the hash table while preserving the input order.

Keys stored in radix trees are ordered bitwise lexicographically. For some data types, e.g., ASCII encoded character strings, this yields the expected order. For most data types this is not the case. For example, negative two-complement signed integers are lexicographically greater than positive integers. However, it is possible to obtain the desired order by transforming the keys. We call the values resulting from such a transformation binary-comparable keys. If only binary-comparable keys are used as keys of a radix tree, the data is stored in sorted order and all operations that rely on this order can be supported. Note that no modifications to the algorithms presented in the previous section are necessary. Each key must simply be transformed to a binary-comparable key before storing it or looking it up.

Binary-comparable keys have other use cases. Just as this concept allows to replace comparison-based trees with radix trees, it allows to replace comparison-based sorting algorithms like quicksort or mergesort with the radix sort algorithm which can be asymptotically superior.

A. Definition

A transformation function $t : D \to \{0, 1, \ldots, 255\}^k$ transforms values of a domain D to binary-comparable keys of length k if it satisfies the following equivalences ($x, y \in D$):

- $x < y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) < 0$
- $x > y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) > 0$
- $x = y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) = 0$

The operators $<$,$>$,$=$ denote the usual relational operators on the input type while memcmp_k compares the two input vectors component wise. It returns 0 if all compared values

are equal, a negative value if the first non-equal value of the first vector is less than the corresponding byte of the second vector, and otherwise a positive value.

For finite domains, it is always possible to transform the values of any strictly totally ordered domain to binary-comparable keys: Each value of a domain size n is mapped to a string of $\lceil \log_2 n \rceil$ bits storing the zero-extended rank minus one.

B. Transformations

We now discuss how common data types can be transformed to binary-comparable keys.

a) Unsigned Integers: The binary representation of unsigned integers already has the desired order. However, the endianness of the machine must be taken into account when storing the value into main memory. On little endian machines, the byte order must be swapped to ensure that the result is ordered from most to least significant byte.

b) Signed Integers: Signed two-complement integers must be reordered because negative integers are ordered descending and are greater than the positive values. An b bit integer x is transformed very efficiently by flipping the sign bit (using $x \operatorname{XOR} 2^{b-1}$). The resulting value is then stored like an unsigned value.

c) IEEE 754 Floating Point Numbers: For floating point values, the transformation is more involved, although conceptually not difficult. Each value is first classified as positive or negative, and as normalized number, denormalized number, NaN, ∞, or 0. Because these 10 classes do not overlap, a new rank can easily be computed and then stored like an unsigned value. One key transformation requires 3 `if` statements, 1 integer multiplication, and 2 additions.

d) Character Strings: The Unicode Collation Algorithm (UCA) defines complex rules for comparing Unicode strings. There are open source libraries which implement this algorithm and which offer functions to transform Unicode strings to binary-comparable keys[2]. In general, it is important that each string is terminated with a value which does not appear anywhere else in any string (e.g., the 0 byte). The reason is that keys must not be prefixes of other keys.

e) Null: To make a null value binary comparable, it must be assigned a value with some particular rank. For most data types, all possible values are already used. A simple solution is to increase the key length of all values by one to obtain space for the null value, e.g., 4 byte integers become 5 bytes long. A more efficient way to accommodate the null value is to increase the length only for some values of the domain. For example, assuming null should be less than all other 4 byte integers, null can be mapped to the byte sequence 0,0,0,0,0, the previously smallest value 0 is mapped to 0,0,0,0,1, and all other values retain their 4 byte representation.

f) Compound Keys: Keys consisting of multiple attributes are easily handled by transforming each attribute separately and concatenating the results.

[2]The C/C++ library "International Components for Unicode" (http://site.icu-project.org/), for example, provides the `ucol_getSortKey` function for this purpose.

V. Evaluation

In this section, we experimentally evaluate ART and compare its performance to alternative in-memory data structures, including comparison-based trees, hashing, and radix trees. The evaluation has two parts: First, we perform a number of micro benchmarks, implemented as standalone programs, with all evaluated data structures. In the second part, we integrate some of the data structures into the main-memory database system HyPer. This allows us to execute a more realistic workload, the standard OLTP benchmark TPC-C.

We used a high-end desktop system with an Intel Core i7 3930K CPU which has 6 cores, 12 threads, 3.2 GHz clock rate, and 3.8 GHz turbo frequency. The system has 12 MB shared, last-level cache and 32 GB quad-channel DDR3-1600 RAM. We used Linux 3.2 in 64 bit mode as operating system and GCC 4.6 as compiler.

As contestants, we used

- a B$^+$-tree optimized for main memory (Cache-Sensitive B$^+$-tree [CSB]),
- two read-only search structures optimized for modern x86 CPUs (k-ary search tree [kary], Fast Architecture Sensitive Tree [FAST]),
- a radix tree (Generalized Prefix Tree [GPT]), and
- two textbook data structures (red-black tree [RB], chained hash table [HT] using MurmurHash64A for 64-bit platforms [25]).

For a fair comparison, we used source code provided by the authors if it was available. This was the case for the CSB$^+$-Tree [26], k-ary search [27], and the Generalized Prefix Tree [28]. We used our own implementation for the remaining data structures.

We were able to validate that our implementation of FAST, which we made available online [29], matches the originally published numbers. To calibrate for the different hardware, we used the results for k-ary search which were published in the same paper. Our implementation of FAST uses 2 MB memory pages, and aligns all cache line blocks to 64 byte boundaries, as suggested by Yamamuro et al. [30]. However, because FAST and k-ary search return the rank of the key instead of the tuple identifier, the following results include one additional lookup in a separate array of tuple identifiers in order to evaluate a meaningful lookup in the database context.

We had to use 32 bit integers as keys for the micro benchmarks because some of the implementations only support 32 bit integer keys. For such very short keys, path compression usually increases space consumption instead of reducing it. Therefore, we removed this feature for the micro benchmarks. Path compression is enabled in the more realistic second part of the evaluation. In contrast to comparison-based trees and hash tables, the performance of radix trees varies with the distribution of the keys. We therefore show results for *dense* keys ranging from 1 to n (n denotes the size of the tree in # keys) and *sparse* keys where each bit is equally likely 0 or 1. We randomly permuted the dense keys.

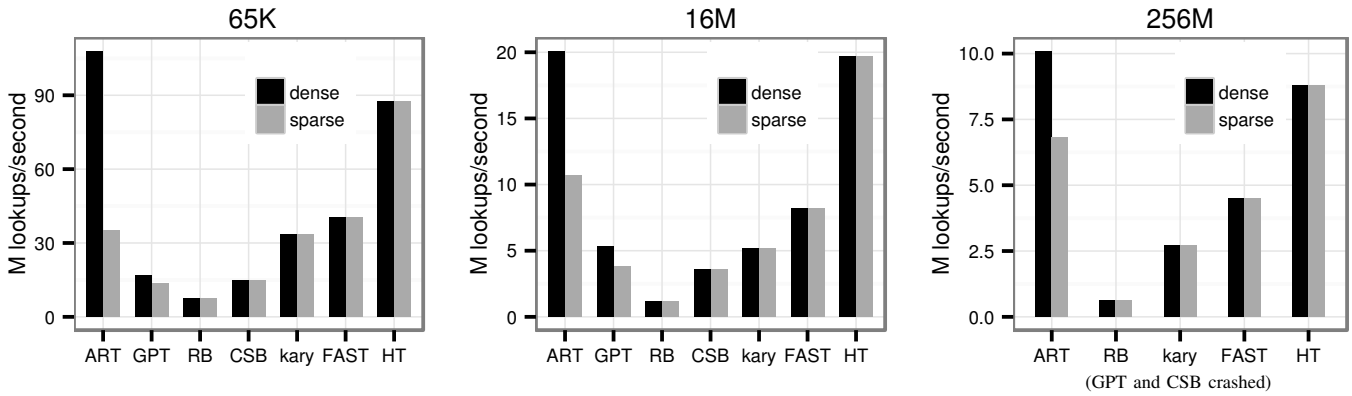

Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

A. Search Performance

In our first experiment, we measured the performance of looking up random, existing[3] keys. Figure 10 shows that ART and the hash table have the best performance. ART is more than twice as fast as GPT, a radix tree with half the span and therefore twice the height. The red-black tree is the slowest comparison-based tree, followed by the CSB$^+$-tree, k-ary search, and finally FAST. Even though FAST does not support updates and is optimized for modern architectures, it is slower than ART and the hash table. The relative performance of the data structures is very similar for the three index sizes. This indicates that the fact that small indexes (65K) are about 10 times faster than large indexes (256M) is mostly caused by caching effects, and not by the asymptotic properties of the data structures.

To better understand these results, consider Table III, which shows performance counters per random lookup for the three fastest data structures (ART, FAST, and the hash table). With 16M keys, only parts of the index structures are cached, and lookup is memory bound. The number of cache misses is similar for FAST, the hash table, and ART with sparse keys. With dense keys, ART causes only half as many cache misses because its compact nodes can be cached effectively. In small trees, the lookup performance is mostly determined by the number of instructions and branch mispredictions. While ART has almost no mispredicted branches for dense keys, sparse keys lead to around 0.85 mispredicted branches per lookup, which occur during node type dispatch. Dense keys also require less instructions, because finding the next child in a `Node256` requires no computation, while the other node types result in more effort. FAST effectively avoids mispredictions which occur with ordinary search trees, but requires a significant number of instructions (about 5 per comparison) to achieve this. The hash table has a small number of mispredictions which occur during collision handling.

So far, lookups were performed one query at a time, in a single thread. The goal of the next experiment was to find the maximum achievable throughput using multiple unsynchronized threads. Besides using multiple threads, it has been shown that throughput can be improved by interleaving multiple tree traversals using software pipelining [7]. This technique exploits modern superscalar CPUs better, but increases latency, and is only applicable if multiple queries are available (e.g., during an index-nested loop join). FAST benefits most from software pipelining (2.5x), because its relatively large latencies for comparisons and address calculations can be hidden effectively. Since ART performs less calculations in the first place, its speedup is smaller but still significant (1.6x-1.7x). A chained hash table can be considered a tree of only two levels (the hash table array, and the list of key/value pairs), so the speedup is relatively small (1.2x). Nevertheless, Figure 11 shows that even with 12 threads and 8 interleaved queries per thread, ART is only slightly slower than FAST for sparse keys, but significantly faster for dense keys.

[3]Successful search in radix trees is slower than unsuccessful search.

TABLE III
PERFORMANCE COUNTERS PER LOOKUP.

	65K			16M		
	ART (d./s.)	FAST	HT	ART (d./s.)	FAST	HT
Cycles	40/105	94	44	188/352	461	191
Instructions	85/127	75	26	88/99	110	26
Misp. Branches	0.0/0.85	0.0	0.26	0.0/0.84	0.0	0.25
L3 Hits	0.65/1.9	4.7	2.2	2.6/3.0	2.5	2.1
L3 Misses	0.0/0.0	0.0	0.0	1.2/2.6	2.4	2.4

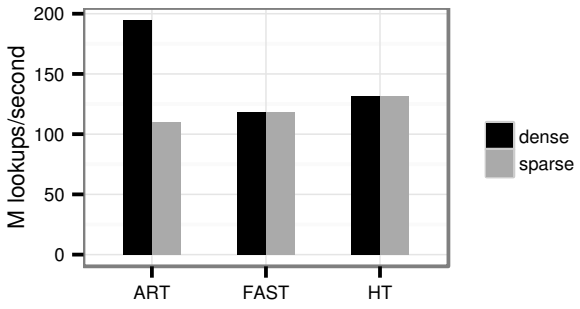

Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).

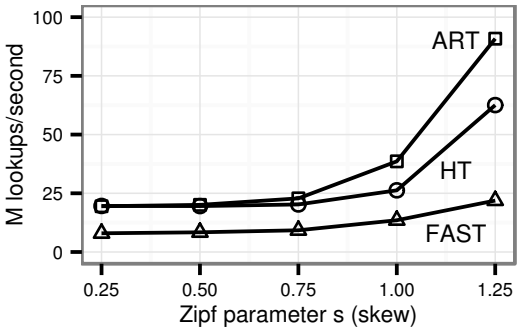

Fig. 12. Impact of skew on search performance (16M keys).

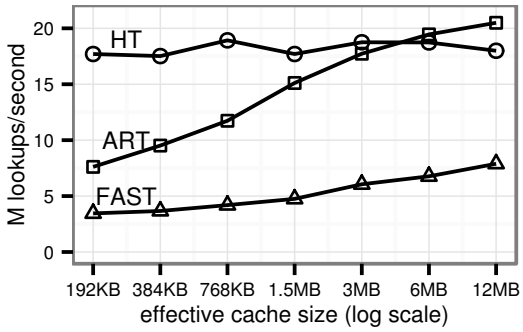

Fig. 13. Impact of cache size on search performance (16M keys).

B. Caching Effects

Let us now investigate caching effects. For modern CPUs, caches are extremely important, because DRAM latency amounts to hundreds of CPU cycles. Tree structures, in particular, benefit from caches very much because frequently accessed nodes and the top levels are usually cached. To quantify these caching effects, we compare two tree structures, ART (with dense keys) and FAST, to a hash table.

Random lookup, which we performed so far, is the worst case for caches because this access pattern has bad temporal locality. In practice, skewed access patterns are very common, e.g., recent orders are accessed more often than old orders. We simulated such a scenario by looking up Zipf distributed keys instead of random keys. Figure 12 shows the impact of increasing skew on the performance of the three data structures. All data structures perform much better in the presence of skew because the number of cache misses decreases. As the skew increases, the performance of ART and the hash table approaches their speed in small, cache resident trees. For FAST the speedup is smaller because it requires more comparisons and offset calculations which are not improved by caching.

We now turn our attention to the influence of the cache size. In the previous experiments, we only performed lookups in a single tree. As a consequence, the entire cache was utilized, because there were no competing memory accesses. In practice, caches usually contain multiple indexes and other data. To simulate competing accesses and therefore effectively smaller caches, we look up keys in multiple data structures in a round-robin fashion. Each data structure stores 16M random, dense keys and occupies more than 128MB. Figure 13 shows that the hash table is mostly unaffected, as it does not use caches effectively anyway, while the performance of the trees improves with increasing cache size, because more often-traversed paths are cached. With $\frac{1}{64}$th of the cache (192KB), ART reaches only about one third of the performance of the entire cache (12MB).

C. Updates

Besides efficient search, an indexing structure must support efficient updates as well. Figure 14 shows the throughput when

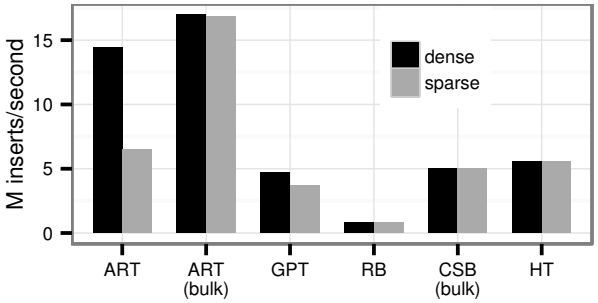

Fig. 14. Insertion of 16M keys into an empty index structure.

inserting 16M random keys into an empty structure. Although ART must dynamically replace its internal data structures as the tree grows, it is more efficient than the other data structures. The impact of adaptive nodes on the insertion performance (in comparison with only using `Node256`) is 20% for trees with 16M dense keys. Since the space savings from adaptive nodes can be large, this is usually a worthwhile trade off. In comparison with incremental insertion, bulk insertion increases performance by a factor of 2.5 for sparse keys and by 17% for dense keys. When sorted keys, e.g., surrogate primary keys, are inserted, the performance of ordered search trees increases because of caching effects. For ART, 50 million sorted, dense keys can be inserted per second. Only the hash table does not benefit from the sorted order because hashing randomizes the access pattern.

FAST and the k-ary search tree are static data structures that can only be updated by rebuilding them, which is why they were not included in the previous experiment. One possibility for using read-only data structures in applications that require incremental updates is to use a delta mechanism: A second data structure, which supports online updates, stores differences and is periodically merged with the read-only structure. To evaluate the feasibility of this approach, we used a red-black tree to store the delta plus FAST as the main search structure, and compared it to ART (with dense keys) and a hash table. We used the optimal merging frequency between

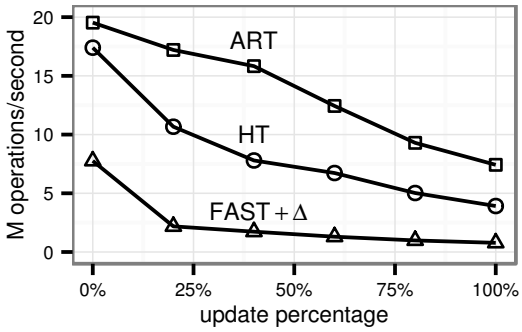

Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

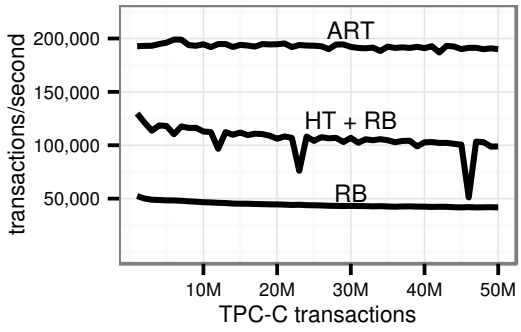

Fig. 16. TPC-C performance.

FAST and the delta which we determined experimentally. Our workload consisted of random insertions, deletions, and lookups in a tree of about 16M elements. Figure 15 shows the results for varying fractions of lookups versus insertions and deletions. As the fraction of lookups decreases, the performance of FAST+delta degrades because of the additional periodic $O(n)$ merging step.

D. End-to-End Evaluation

For the end-to-end application experiment, we used the main-memory database system HyPer. One of its unique characteristics is that it very efficiently supports both transactional (OLTP) and analytical (OLAP) workloads at the same time [3]. Transactions are implemented in a scripting language which is compiled to assembler-like LLVM instructions [31]. Furthermore, HyPer has no overhead for buffer management, locking, or latching. Therefore, its performance critically depends on the efficiency of the index structures used. For each index, HyPer allows to determine which data structure is used. Originally, a red-black tree and a hash table were available. We additionally integrated ART, including the path compression and lazy expansion optimizations. We further implemented the key transformation scheme discussed in Section IV for all built-in types so that range scan, prefix lookup, minimum, and maximum operations work as expected.

The following experiment uses TPC-C, a standard OLTP benchmark simulating a merchandising company which manages, sells, and distributes products. It consists of a diverse mix of select statements (including point queries, range scans, and prefix lookups), insert, and delete statements. While it is considered a write-heavy benchmark, 54% of all SQL statements are queries [32]. Our implementation omits client think-time and uses a single partition with 5 warehouses. We executed the benchmark until most of the available RAM was exhausted. As index configurations we used ART, a red-black tree, and a combination of a hash table and a red-black tree. It is not possible to use hash tables for all indexes because TPC-C requires prefix-based range scans for some indexes.

Figure 16 shows that the index structure choice is critical for HyPer's OLTP performance. ART is almost twice as fast as the hash table/red-black tree combination and almost four times as fast as the red-black tree alone. The hash table improved performance significantly over the red-black tree alone, but introduced unacceptable rehashing latencies which are clearly visible as spikes in the graph.

TABLE IV
MAJOR TPC-C INDEXES AND SPACE CONSUMPTION PER KEY USING ART.

#	Relation	Cardinality	Attribute Types	Space
1	item	100,000	int	8.1
2	customer	150,000	int,int,int	8.3
3	customer	150,000	int,int,varchar(16),varchar(16),TID	32.6
4	stock	500,000	int,int	8.1
5	order	22,177,650	int,int,int	8.1
6	order	22,177,650	int,int,int,int,TID	24.9
7	orderline	221,712,415	int,int,int,int	16.8

Let us turn our attention to the space consumption of the TPC-C benchmark which we measured at the end of the benchmark runs. In order to save space, HyPer's red-black tree and hash table implementations do not store keys, only tuple identifiers. Using the tuple identifiers, the keys are loaded from the database on demand. Nevertheless, and although ART may use more space per element in the worst case, ART used only half as much space as the hash table and the red-black tree. More detailed insights into space consumption can be obtained by considering the structural information for each major index and its corresponding space consumption per key, which is shown in Table IV. Index 3 uses most space per key, as it stores relatively long strings. Nevertheless, its space consumption stays well below the worst case of 52 bytes. Indexes 1, 2, 4, and, 5 use only 8.1 bytes per key because they store relatively dense integers. Indexes 6 and 7 fall in between these extremes.

The results of our final experiment, shown in Figure 17, measure the impact of path compression and lazy expansion on the average tree height. By default, the height of a radix tree equals the length of the key (in bytes for ART). For example, the height of index 3 would be 40 without any optimizations. Path compression and lazy expansion reduce the average height to 8.1. Lazy expansion is particularly effective with long strings (e.g., index 3) and with non-unique

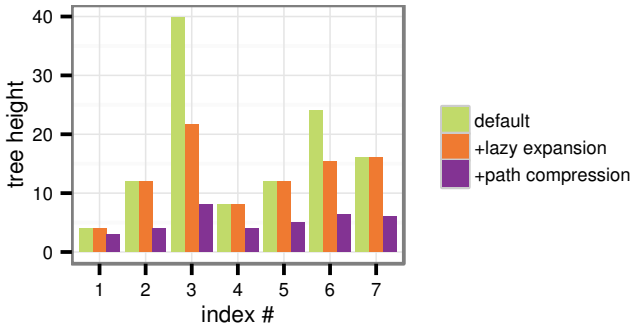

Fig. 17. Impact of lazy expansion and path compression on the height of the TPC-C indexes.

indexes mostly containing unique values because the appended 8 byte tuple identifier can be truncated (e.g., index 6). Path compression helps with long strings (e.g., index 3) and with compound indexes of dense integers which share a common prefix (e.g., indexes 2, 4, 5, and, 7). The impact of the two optimizations on space consumption is similar to the impact on height which is why we do not show it separately. To summarize, except for short integers, path compression and lazy expansions are critical for achieving high performance and small memory consumption with radix trees.

VI. Conclusions and Future Work

We presented the adaptive radix tree (ART), a fast and space-efficient indexing structure for main-memory database system. A high fanout, path compression, and lazy expansion reduce the tree height, and therefore lead to excellent performance. The worst-case space consumption, a common problem of radix trees, is limited by dynamically choosing compact internal data structures. We compared ART with other state-of-the-art main-memory data structures. Our results show that ART is much faster than a red-black tree, a Cache Sensitive B$^+$-Tree, and GPT, another radix tree proposal. Even the architecture sensitive, read-only search tree FAST, which is specifically designed for modern CPUs, is slower than ART, even without taking updates into account. Of all the evaluated data structures only a hash table was competitive. But hash tables are unordered, and are therefore not suitable as general-purpose index structures. By integrating ART into the main-memory database system HyPer and executing the TPC-C benchmark, we demonstrated that it is a superior alternative to conventional index structures for transactional workloads.

In the future, we intend to work on synchronizing concurrent updates. In particular, we plan to develop a latch-free synchronization scheme using atomic primitives like compare-and-swap. Another idea is to design a space-efficient radix tree which has nodes of equal size. Instead of dynamically adapting the fanout based on the sparseness of the keys, the number of bits used from the key should change dynamically, while the fanout stays approximately constant. Such a tree could also be used for data stored on disk.

References

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," *PVLDB*, vol. 1, 2008.

[2] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *SIGMOD Record*, vol. 40, no. 4, 2012.

[3] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.

[4] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *VLDB*, 1986.

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

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

[7] 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.

[8] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *SIGFIDET*, 1970.

[9] D. Comer, "Ubiquitous B-tree," *ACM Comp. Surv.*, vol. 11, no. 2, 1979.

[10] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, 1972.

[11] L. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *IEEE Annual Symposium on the Foundations of Computer Science*, vol. 0, 1978.

[12] J. Rao and K. A. Ross, "Making B+ trees cache conscious in main memory," in *SIGMOD*, 2000.

[13] G. Graefe and P.-A. Larson, "B-tree indexes and CPU caches," in *ICDE*, 2001.

[14] R. De La Briandais, "File searching using variable length keys," in *western joint computer conference*, 1959.

[15] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, September 1960.

[16] D. R. Morrison, "PATRICIA-practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, 1968.

[17] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*, 1998.

[18] N. Askitis and R. Sinha, "Engineering scalable, cache and space efficient tries for strings," *The VLDB Journal*, vol. 19, no. 5, 2010.

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

[20] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, "KISS-Tree: Smart latch-free in-memory indexing on modern architectures," in *DaMoN workshop*, 2012.

[21] D. Baskins, "A 10-minute description of how judy arrays work and why they are so fast," http://judy.sourceforge.net/doc/10minutes.htm, 2004.

[22] A. Silverstein, "Judy IV shop manual," http://judy.sourceforge.net/doc/shop_interm.pdf, 2002.

[23] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, 2006.

[24] J. Corbet, "Trees I: Radix trees," http://lwn.net/Articles/175432/.

[25] A. Appleby, "MurmurHash64A," https://sites.google.com/site/murmurhash/.

[26] J. Rao, "CSB+ tree source," http://www.cs.columbia.edu/~kar/software/csb+/.

[27] B. Schlegel, "K-ary search source," http://wwwdb.inf.tu-dresden.de/team/staff/dipl-inf-benjamin-schlegel/.

[28] M. Boehm, "Generalized prefix tree source," http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/core-indexing-structure-and-techniques/.

[29] V. Leis, "FAST source," http://www-db.in.tum.de/~leis/index/fast.cpp.

[30] 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.

[31] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *PVLDB*, vol. 4, 2011.

[32] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier, "Fast updates on read-optimized databases using multi-core CPUs," *PVLDB*, vol. 5, 2011.