HOT: A Height Optimized Trie Index for Main-Memory Database Systems

Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht
University of Innsbruck, Austria
firstname.lastname@uibk.ac.at

Viktor Leis
Technische Universität München, Germany
leis@in.tum.de

ABSTRACT

We present the Height Optimized Trie (HOT), a fast and space-efficient in-memory index structure. The core algorithmic idea of HOT is to dynamically vary the number of bits considered at each node, which enables a consistently high fanout and thereby good cache efficiency. The layout of each node is carefully engineered for compactness and fast search using SIMD instructions. Our experimental results, which use a wide variety of workloads and data sets, show that HOT outperforms other state-of-the-art index structures for string keys both in terms of search performance and memory footprint, while being competitive for integer keys. We believe that these properties make HOT highly useful as a general-purpose index structure for main-memory databases.

CCS CONCEPTS

• **Information systems → Data access methods**; *Main memory engines*;

KEYWORDS

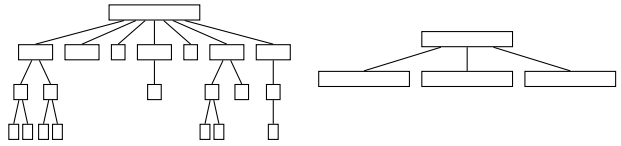
height optimized trie, main memory, index, SIMD

ACM Reference Format:
Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3183713.3196896

1 INTRODUCTION

For many workloads, the overall performance of main-memory database systems depends on fast index structures. At the same time, a large fraction of the total main memory is often occupied by indexes [30]. Having fast *and* space-efficient index structures is therefore crucial.

While in disk-based database systems B-trees are prevalent, some modern in-memory systems (e.g., Silo [26] or HyPer [13]) use trie structures (e.g., Masstree [22] or ART [18]). The reason for this preference is that, in main memory, well-engineered tries often outperform comparison-based structures like B-trees [1, 3, 18, 30]. Furthermore, unlike hash tables, tries are order-preserving and therefore support range scans and related operations. Nevertheless, even recent trie proposals have weaknesses that preclude optimal performance and space consumption. For example, while ART can achieve a high fanout and therefore high performance on integers, its average fanout is much lower when indexing strings. This lower fanout usually occurs at lower levels of the tree and is caused by sparse key distributions that are prevalent in string keys.



(a) Adaptive Radix Tree **(b) Height Optimized Trie**

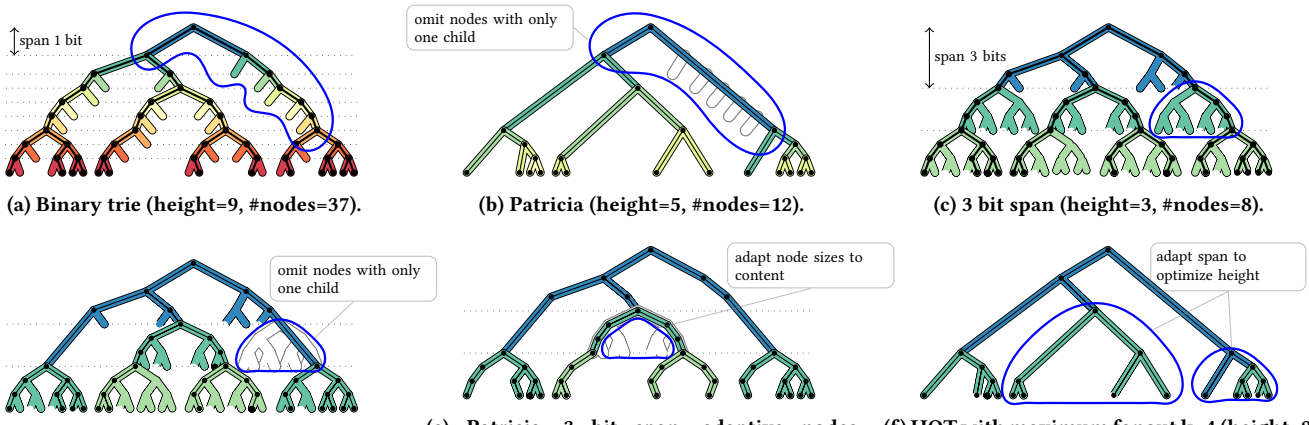
Figure 1: Illustration of adaptive radix tree and height-optimized trie storing sparsely distributed keys. Whereas the average node fanout for the adaptive radix tree decreases at lower levels of the tree, HOT retains a consistently high fanout and therefore has a smaller overall height.

In this work, we present the Height Optimized Trie (HOT), a general-purpose index structure for main-memory database systems. HOT is a balanced design that efficiently supports all operations relevant for an index structure (e.g., online updates, point and range lookups, support for short and long keys, etc.), but is particularly optimized for space efficiency and lookup performance. For string data, the size of the index is generally significantly smaller than the string data itself.

While HOT incorporates many optimizations used in modern trie variants, its salient algorithmic feature is that it achieves a high average fanout for arbitrary key distributions. In contrast to most tries, the number of bits considered at each node (sometimes called span or alphabet) is not fixed, but is adaptively chosen depending on the data distribution. This enables a consistently high fanout and avoids the sparsity problem that plagues other trie variants. As a result, space consumption is reduced and the height of the tree is minimized, which improves cache efficiency. This height-reducing effect is schematically depicted in Figure 1.

Besides using this novel algorithmic approach, the layout of nodes is carefully engineered for good performance on modern CPUs. The node representation is optimized for cache efficiency and allows for efficient, SIMD-optimized search. Our evaluation is based on the YCSB benchmark, and a wide variety of workloads and data distributions. We compare HOT with B-trees, Masstree, and ART, which are state-of-the-art, order-preserving, in-memory index structures. The experimental results show that HOT generally outperforms its competitors in terms of performance and space consumption, for both short integers as well as long strings. These

SIGMOD'18, June 10–15, 2018, Houston, TX, USA



(a) Binary trie (height=9, #nodes=37). **(b) Patricia (height=5, #nodes=12).** **(c) 3 bit span (height=3, #nodes=8).**

(d) Patricia, 3 bit span (height=3, #nodes=6). **(e) Patricia, 3 bit span, adaptive nodes (height=3, #nodes=6).** **(f) HOT with maximum fanout k=4 (height=2, #nodes=4).**

Figure 2: Trie optimizations. Nodes on the same level have the same color. Discriminative bits are shown as black dots.

properties make HOT particularly well suited as an index for in-memory database systems and, more generally, for string-intensive applications. Our implementation of HOT is publicly available under the ISC license at `https://github.com/speedskater/hot`.

The rest of the paper is organized as follows. We start by describing important background and related work on tries in Section 2. Section 3 introduces the high-level algorithms for insertion and discusses how the tree structure is dynamically organized such that the overall tree height is optimized. The physical node layout and the data-parallel operations are then described in Section 4. Section 5 presents a scalable synchronization protocol for multi-core CPUs. After presenting our experimental evaluation in Section 6, we summarize the paper in Section 7.

2 BACKGROUND AND RELATED WORK

The growth of main memory capacities has led to the development of index structures that are optimized for in-memory workloads (e.g., [4, 14, 20, 24, 25, 28, 31]). **Tries**, in particular, have proven to be highly efficient on modern hardware [1, 3, 16, 18, 22, 27, 29, 30]. Tries are tree structures where all descendants of a node share a common prefix and the children of a node are searched using the binary representation of the remaining bits. In a binary trie, for example, at each node one bit of the key determines whether to proceed with the left or right child node. While binary tries are conceptually simple, they do not perform well due to their large tree heights (e.g., a height of 32 for 4 byte integers). Prior research focused on reducing the height of trie structures. In the following, we discuss and graphically illustrate some of the most relevant approaches in this area. Each trie depicted in Figure 2 stores the same 13 keys, all of which are 9 bits long. Compound nodes are surrounded by solid lines and are colored according to their level in the respective tree structure. Dots in the figures represent either leaf values or bit positions in compound nodes which are used to distinguish between different keys. In Figure 2a, a binary trie is depicted. The subsequent Figures 2b-2f illustrate different optimizations, which we discuss in the following.

Figure 2b shows a binary **Patricia** trie [23], which reduces the overall tree height by omitting all nodes with only one child[1]. The resulting structure resembles a full binary tree, where each node either is a leaf node or has exactly two children. While this optimization often reduces the tree height (e.g., from 9 to 5 in our example), the small fanout of 2 still yields large tree heights.

To reduce the height, many trie structures consider more than 1 bit at each node, i.e., they **increase the span**. For a given span s, this is generally implemented using an array of 2^s pointers in each node. The Generalized Prefix Tree [3], for example, uses a span of 4 bits reducing the overall tree height by a factor of 4 in comparison to a binary trie. The downside of a larger span is increased space consumption for sparsely distributed keys (e.g., long strings) as most of the pointers in the nodes will be empty and the actual fanout is typically much smaller than the optimum (2^s). The resulting tree structures therefore remain vulnerable to large tree heights and wasted space. These problems can be observed in Figure 2c, which depicts a trie with a span of 3 bits. Its average fanout is only 2.5, which is considerably smaller than the optimum of 8. Also note that, while the Patricia optimization can be applied to tries with a larger span, it becomes less effective (though may still be worthwhile). As Figure 2d shows, when applied to the trie depicted in Figure 2c with a span of 3 bits, the Patricia optimization saves only two nodes and does not reduce the maximum tree height.

One fairly effective approach for addressing the shortcomings of larger spans is to dynamically **adapt the node structure**. The Adaptive Radix Tree (ART) [18], for example, uses a span of 8 bits, but avoids wasting space by dynamically choosing more compact node representations (instead of always using an array of 256 pointers). Hence, adaptive nodes reduce memory consumption and enable the use of a larger span, which increases performance through better cache efficiency. However, even with a fairly large span of 8 bits, sparsely distributed keys result in many nodes with a very small fanout at lower levels of the tree. The concept of adaptive

[1] As a result of the Patricia optimization, keys are not necessarily stored fully in the trie and every key must therefore be available at its corresponding leaf node. For main-memory database systems, this is usually the case because the leaf node will store a reference to the full tuple (including the key).

nodes is depicted in Figure 2e, which adds adaptive nodes to the trie of Figure 2d. It clearly shows that using adaptive nodes successfully limits the issue of memory consumption in case of sparsely distributed data. However, it also shows that, in our example, adaptive nodes do not have an impact on the effective node fanout and the overall tree height.

Having surveyed the different approaches to reduce the overall tree height of trie-based index structure, we conclude that all optimizations depicted in Figure 2 combine multiple nodes of a binary trie into a compound node structure, such that the height of the resulting structure is reduced and the average node fanout is increased. Moreover, these approaches choose the criteria to combine multiple binary trie nodes, namely the span representing the bits considered per node, independently of the data stored. Therefore, the resulting fanout, memory consumption and access performance heavily depend on the data actually stored.

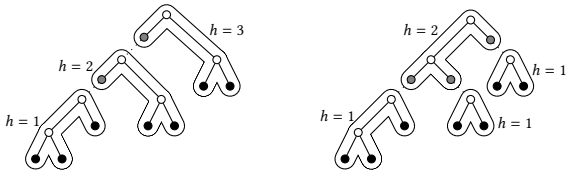
In this work, we propose the **Height Optimized Trie (HOT)**. HOT combines multiple nodes of a binary *Patricia* trie into *compound nodes* having a maximum node fanout of a predefined value k such that the height of the resulting structure is optimized. Thus, *each node uses a custom span* suitable to represent the discriminative bits of the combined nodes. Moreover, *adaptive node sizes* are used to reduce memory consumption and non-discriminative bits are ignored (i.e., skipped during traversal) like in a Patricia trie. Figure 2f shows a Height Optimized Trie with a maximum node fanout of $k = 4$ that has 4 compound nodes and an overall height of 2 to store the same 13 keys as the other trie structures.

While all data structures discussed so far are "pure tries", a number of **hybrid data structures** that combine a trie with some other data structure have also been proposed. For example, the BURST-Trie [10] and HAT-Trie [1] use binary trees and hash tables respectively for sparsely populated areas. Both data structures achieve fairly high performance for string workloads, but are limited in terms of memory consumption and access performance in case of integer- or densely distributed keys. Another hybrid structure is Masstree [22], which uses a large span of 64 bits and B-trees as its internal node structure. This solves the sparsity problem at the cost of relying more heavily on comparison-based search, which is often slower than the bitwise trie search. HOT, in contrast, is a pure trie and solves the sparsity problem by using a varying span. The Bit Tree [6] is primarily a B-tree that uses discriminative bits at the leaf level. This optimization is done to save space on disk and the data structure is not optimized for in-memory use cases.

3 THE HEIGHT OPTIMIZED TRIE

The optimizations discussed in the previous section combine the nodes of a binary trie into compound nodes with a higher fanout. The most important optimization is to increase the span of each node. However, in current data structures, the span is a static, fixed setting (e.g., 8 bits) that is set globally without taking the actual keys stored into account. As a result, both the performance and memory consumption can strongly vary for different data sets.

Consider, for example, a trie with span of 8 bits storing 1 million 64-bit integers. For monotonic integers (i.e., 1 to 1,000,000), almost all nodes are full, the average fanout is close to the maximum of 256, and, as a result, performance as well as space consumption is



(a) Optimized for # of nodes **(b) Optimized for height**

Figure 3: Different ways of combining binary nodes into compound nodes, which are annotated with their height h.

optimal. For integers randomly drawn from the full 64-bit domain, on the other hand, many nodes at lower levels of the tree will only be sparsely filled. Strings are also generally sparsely distributed, with genome data representing nucleic acids using a single-byte character (A, C, G, T) being an extreme case. Using a fixed span, sparse distributions have a low average fill factor, which negatively affects performance. Also, as most nodes are at lower levels, space consumption is high.

To solve the problem of sparsely-distributed keys, we propose to set the span of each node adaptively depending on the data distribution. Thus, dense key regions (e.g., near the root) will have a smaller span than sparse regions (e.g., at lower levels), and a consistently high fanout can be achieved. Instead of having a fixed span and data-dependent fanout as in a conventional trie, HOT features a data-dependent span and a fixed maximum fanout k.

3.1 Preliminaries: k-Constrained Tries

A crucial property of HOT is that every *compound node* represents a binary Patricia trie with a fanout of up to k. As can be observed in Figure 2b, a binary Patricia trie storing n keys has exactly $n - 1$ inner nodes. A HOT compound node therefore only needs to store at most $k - 1$ binary inner nodes (plus up to k pointers/leaves).

For a given parameter k, there are multiple ways of combining binary nodes into compound nodes. Figure 3 shows two trees with a maximum fanout $k = 3$ storing the same data. While the tree shown in Figure 3a reduces the total number of compound nodes, the tree shown in Figure 3b is usually preferable, as it minimizes the overall tree height. In the figure and in our implementation every compound node n is associated with a height $h(n)$, such that $h(n)$ is the maximum height of its compound child nodes + 1. Based on this definition, the overall tree height is the height of the root node. More formally, assuming a node n has $n.m$ child nodes, $h(n)$ can be defined as

$$h(n) = \begin{cases} 1, & n.m = 0 \\ \max_{i=1}^{n.m}(h(n.child[i])) + 1, & \text{else.} \end{cases}$$

Creating k-constrained nodes in a way that minimizes the overall tree height is analogous to partitioning a full binary tree into disjoint subtrees, such that the maximum number of partitions along a path from the root node to any leaf node is minimized. For static trees, Kovács and Kis [17] solved the problem of partitioning trees such that the overall height and cardinality are optimized. In this paper, we present a dynamic algorithm, which is able to preserve the height optimized partitioning while new data is inserted.

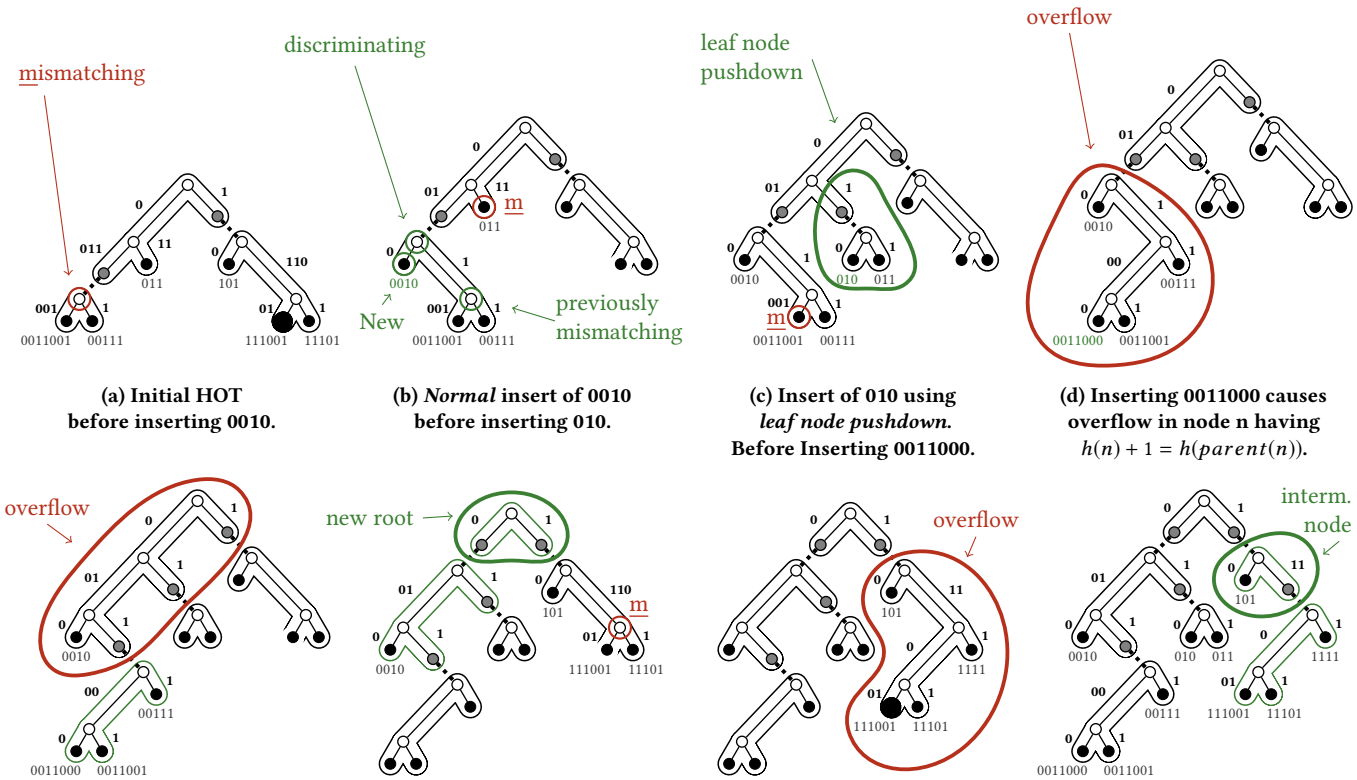


Figure 4: Step-by-step example inserting the keys 0010, 010, 0011000, and 1111 into a HOT with a maximum fanout of $k = 3$.

To avoid confusion between binary nodes and compound nodes, in the rest of the paper, we use the following terminology: whenever we denote a node in a binary Patricia trie we use the term *BiNode*. In all other cases, the term *node* stands for a compound node. In this terminology, a node contains up to $k - 1$ BiNodes and up to k leaf entries.

3.2 Insertion and Structure Adaptation

Similar to B-trees, the insertion algorithm of HOT has a normal code path that affects only a single node and other cases that perform structural modifications to the tree. As the deletion operation is algorithmically analogous, we focus on the insertion operation in the remainder.

In the following, we describe the different cases by successively inserting four keys into a HOT structure with a maximum node fanout of $k = 3$. The initial tree is shown in Figure 4a. Insertion always begins by traversing the tree until the node with the mismatching BiNode is found. The *mismatching BiNode* for the first key to be inserted, 0010, is shown in Figure 4a.

In the **normal** case, insertion is performed by locally modifying the BiNode structure of the affected node. More precisely, and as shown in Figure 4b, a new *discriminating BiNode*, which discriminates the new key from the keys contained in the subtree of the mismatching BiNode, is created and inserted into the affected node. The normal case is analogous to inserting into a Patricia tree. However, because nodes are k-constrained, the normal case is only applicable if the affected node has less than k entries.

The second case is called **leaf-node pushdown** and involves creating a new node instead of adding a new BiNode to an existing node. If the mismatching BiNode is a leaf *and* the affected node is an inner node ($h(n) > 1$), we replace the leaf with a new node. The new node consists of a single BiNode that distinguishes between the new key and the previously existing leaf. In our running example, this case is triggered when the key 010 is inserted into the tree shown in Figure 4b. Leaf-node pushdown does not affect the maximum tree height as can be observed in Figure 4c: Even after leaf-node pushdown, the height of the root node (and thus the tree) is still 2.

An overflow happens when neither leaf-node pushdown nor normal insert are applicable. As Figure 4d shows, such an invalid intermediate state occurs after inserting 0011000.

There are two different ways of resolving an overflow. Which method is applicable depends on the height of the overflowed node in relation to its parent node. As Figure 4e illustrates, one way to

resolve an overflow is to perform **parent pull up**, i.e., to move the root BiNode of the overflowed node into its parent node. This approach is taken when growing the tree "downwards" would increase the tree height, and it is therefore better try to grow the tree "upwards". More formally, parent pull up is triggered when the height of the overflowed node n is "almost" the height of its parent: $h(n) + 1 = h(parent(n))$. By moving the root BiNode, the originally overflowed node becomes k-constrained again, but its parent node may now overflow—this indeed happens in the example shown in Figure 4e. Overflow handling therefore needs to be recursively applied to the affected parent node. In our example, because the root node is also full, overflow is eventually resolved by creating a new root, which is the only case where the overall height of the tree is increased. Thus, similar to a B-tree, the overall height of HOT only increases when a new root node is created.

The second way to handle an overflow is **intermediate node creation**. Instead of moving the root BiNode of the overflowed node into its parent, the root BiNode is moved into a newly created intermediate node. Intermediate node creation is only applicable if adding an additional intermediate node does not increase the overall tree height, which is the case if: $h(n) + 1 < h(parent(n))$. In our example, this case is triggered when the key 1111 is inserted into the tree shown in Figure 4g. As can be seen in Figure 4g, the overflowed node n has a height of 1 and its parent has a height of 3. Thus, there is "room" above the overflowed node and creating an intermediate node does not affect the overall height, as can be observed in the tree shown in Figure 4h.

Based on the insertion operation we designed an analogous deletion operation consisting of the following three cases mirroring its insertion counterparts in the following. A normal deletion, modifying a single node, compensates normal insert or leaf-node pushdown. Underflow handling by merging two nodes or integrating a link to a direct neighbor corresponds to the the overflow handling strategies leaf-node pushdown or intermediate node creation.

To summarize the insertion operation, there are four cases that can happen during an insert operation. A normal insert only modifies an existing node, whereas leaf-node pushdown creates a new node. Overflows are either handled using parent pull up or intermediate node creation.

These four cases are also visible in Listing 1, which shows the full insertion algorithm.

Listing 1: Structure-adapting insertion algorithm.

```
1  insert(hot, key):
2    n = traverse hot for key
3    m = traverse n until mismatch
4    if (isLeafEntry(m) and h(n) > 1):
5      # leaf node pushdown
6      l = createNode(m, key)
7      n̂ = replaceNode(n, m, l)
8    else:
9      d = createBiNode(m, key)
10     n̂ = replaceBiNode(n, m, d)
11     handleOverflow(n̂)

13 handleOverflow(n):
14   if (not isFull(n))
15     # normal path
16     return
17   n̂ = split(n)
18   p = parentNode(n)
19   if (height(n̂) == height(p)):
20     # parent pull up
21     e = createBiNode(n̂[0], n̂[1])
22     p̂ = replaceBiNode(p, n, e)
23     handleOverflow(p̂)
24   else
25     # intermediate node creation
26     p̂ = replaceNode(p, n, n̂)
```

3.3 Properties of Height Optimized Tries

HOT is a pure trie structure, i.e., every node represents a prefix of the key. Nevertheless, it has similarities with comparison-based multi-way structures, which have to perform $log_2(n)$ key comparisons. Like in B-trees, in HOT the maximum fanout of each node is bounded and both structures strive to reduce the overall (maximum) tree height by dynamically distributing the data and nodes. Another similarity is that the height of both structures only increases when a new root node is created.

But there are also major differences. Whereas the theoretical properties in terms of tree height and access performance for B-trees are well known, this is currently not the case for HOT. A common property of most tries not shared by comparison-based trees is that any given set of keys results in the same structure, regardless of the insertion order. Based on experiments, we conjecture that this deterministic structure applies to HOT as well.

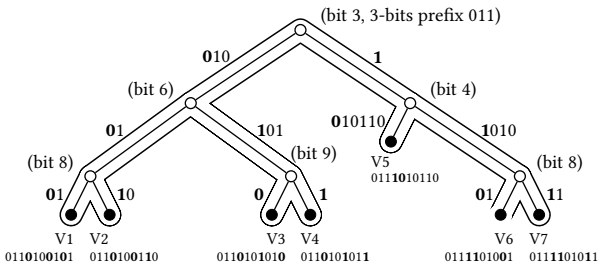
4 NODE IMPLEMENTATION

The algorithms in Section 3 have been presented in a fairly high-level way: We did not specify k, did not discuss how nodes are physically organized, and did not show how operations within nodes are performed. In this section, we fill these missing details by presenting a design for general-purpose in-memory indexing with a focus on space efficiency and lookup performance. Let us note that other designs, e.g., optimized for other workloads or disk-based storage, would also be possible based on the algorithms presented in Section 3.

4.1 Overview

In principle, one could organize each HOT node (i.e., each k-constrained binary Patricia trie) as a pointer-based binary trie structure. However, this approach would waste much space for pointers and would be very inefficient (as all operations would require traversing this binary trie, which results in cache misses as well as control or data dependencies). To make HOT space-efficient *and* fast, a compact representation that can be searched quickly is required.

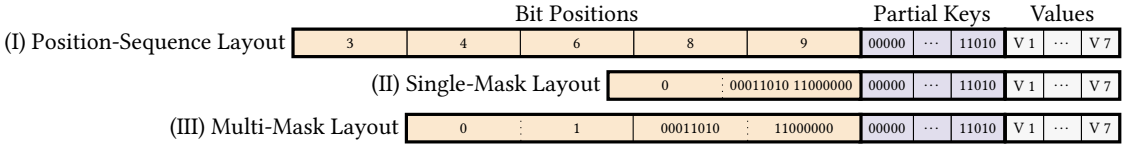
The key idea behind our node layout is to linearize a k-constrained trie to a compact bit string that can be searched in parallel using SIMD instructions. To achieve this, we store the discriminative bits of each key consecutively. Consider, for example, the trie shown in



(a) Example Binary Patricia Trie. Discriminative bits along a path are typeset in bold.

Raw Key	Bit Positions	Partial Key (dense)	Partial Key (sparse)
0 1 1 **0** 1 0 **0** 1 **0** 1	{3, 6, 8}	**0** 1 **0 0** 1	**0** 0 **0 0** 0
0 1 1 **0** 1 0 **0** 1 **1** 0	{3, 6, 8}	**0** 1 **0 1** 0	**0** 0 **0 1** 0
0 1 1 **0** 1 0 **1** 0 1 **0**	{3, 6, 9}	**0** 1 **1** 1 **0**	**0** 0 **1** 0 **0**
0 1 1 **0** 1 0 **1** 0 1 **1**	{3, 6, 9}	**0** 1 **1** 1 **1**	**0** 0 **1** 0 **1**
0 1 1 **1 0** 1 0 1 1 0	{3, 4}	**1 0** 0 1 0	**1 0** 0 0 0
0 1 1 **1 1** 0 1 0 **0** 1	{3, 4, 8}	**1 1** 1 **0** 1	**1 1** 0 **0** 0
0 1 1 **1 1** 0 1 0 **1** 1	{3, 4, 8}	**1 1** 1 **1** 1	**1 1** 0 **1** 0

(b) Raw and partial keys for the trie in (a).



(c) Three different variations of a node representing a linearized version of the trie shown in (a). All representations consist of three parts: the bit positions, the partial keys, and the values (tuple identifiers or node pointers). We show 3 representations for bit positions: (I) stores them naively as a sequence of positions, (II) uses a single bit mask, (III) uses multiple bit masks.

Figure 5: Illustration of how HOT encodes a Binary Patricia Trie.

Figure 5a, which consists of 7 keys and has the discriminative bit positions $\{3, 4, 6, 8, 9\}$. The 5 discriminative bits for each key form the *partial keys (dense)* and are shown in Table 5b. By storing the partial keys consecutively, we can search all keys in parallel using SIMD operations instead of traversing the corresponding trie. As will be described in Section 4.4, to improve insertion performance we actually use as slightly improved version of partial keys called *sparse partial keys*.

Another important design decision is to set the maximum fanout k to 32, which is large enough to benefit from CPU caches and small enough to support fast updates. As a result, a node can have up to 31 bit positions (which is always sufficient to distinguish between 32 keys). Note that 32 is an upper bound, and fewer bits are often sufficient (e.g., because the node has only 14 entries). On the other hand, partial keys need to be aligned to enable fast SIMD operations. Therefore, on a physical level, partial keys are stored in one of 3 representations, namely as an array of 8-bit, 16-bit, or 32-bit values. For each node, the smallest possible layout is chosen.

Besides the partial keys, each node must also store the corresponding bit positions (i.e., $\{3, 4, 6, 8, 9\}$ in our example). The most obvious way to do this would be to store this set as a sequence in an array, as shown in Figure 5c (I). The problem with this approach is that it would slow down search: Before the actual data-parallel key comparison could be performed, one would have to sequentially extract bits from the search key bit-by-bit to form the comparison key. Note that key extraction is done for every node encountered during tree traversal and is therefore critical for performance.

To speed up key extraction, we therefore utilize the *PEXT* instruction from the BMI2 instruction set, which extracts bits specified by a bit mask from an integer. Thus, as shown in Figure 5c (II), we represent the bit positions as a bit mask (and an initial byte position). This layout can be used whenever the smallest and largest bit positions are close to each other (less than 64 bit positions difference). Otherwise, we use the multi-mask layout, which is illustrated in Figure 5c (III). It breaks up the bit positions into multiple 8-bit masks, each of which is responsible for an arbitrarily byte. Again, PEXT can be used to efficiently extract the bits contained in multiple 8-bit key portions in parallel using this layout. As with the partial keys, for any node the most compact representation for the bit positions is chosen.

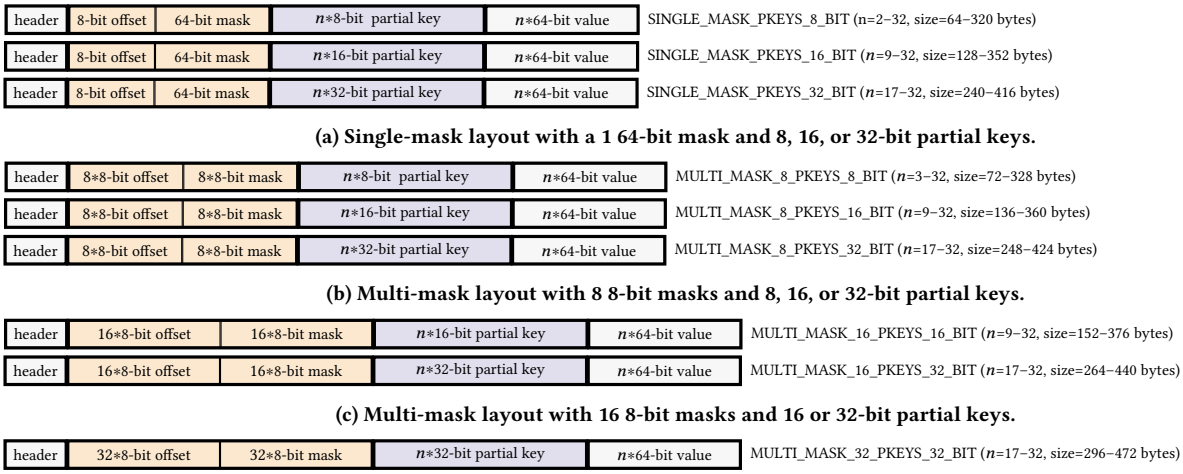
To summarize, the node layout has two dimensions of adaptivity. The first dimension is the size of the partial keys (8, 16, or 32 bits), and the second dimension is the representation of the bit positions (single-mask or multi-mask). In both cases we choose representations that adapt to the data distribution at hand. Furthermore, all representations allow exploiting modern hardware instructions (data-parallel comparisons, PEXT).

4.2 Physical Node Layout

Figure 6 shows the 9 physical node layouts supported by HOT. As the figure shows, each node layout consists of a (1) node header, (2) bit positions, (3) partial keys, and (4) values.

Each node starts with a header section containing the *height* of its subtree, a bit mask describing the *used entries* and a *lock* in the synchronized version.

To store the bit positions 4 different layouts are used. The *single-mask layout* is shown in Figure 6a and consists of an 8-bit *offset* and a 64-bit *mask*. The offset determines the starting byte position from which the partial key is extracted using the mask. Besides the single-mask layout, there are 3 *multi-mask layouts* that use multiple



(a) Single-mask layout with a 1 64-bit mask and 8, 16, or 32-bit partial keys.

(b) Multi-mask layout with 8 8-bit masks and 8, 16, or 32-bit partial keys.

(c) Multi-mask layout with 16 8-bit masks and 16 or 32-bit partial keys.

(d) Multi-mask layout with 32 8-bit masks and 32-bit partial keys.

Figure 6: The physical node layouts.

byte offsets and multiple 8-bit masks. These 3 layouts (Figure 6b, 6c, 6d) differ only in the number of offset/mask pairs (8, 16, or 32).

The partial keys are stored as an array of 8-bit, 16-bit, or 32-bit entries. Note that some bit-position/partial-key layout combinations cannot occur, which is why there are not 12 but 9 node layouts. For example, the 32-entry multi-mask layout shown in Figure 6d implies that there are more than 16 discriminative bits and that therefore, neither 8-bit nor 16-bit partial keys would suffice. Each node also stores 64-bit *values*, which are either pointers to other nodes or tuple identifiers. We distinguish between a pointer and a tuple identifier using the most-significant bit, which is otherwise always 0.

Depending on the number and the distribution of the discriminative bits, we always choose the smallest of the 9 node layouts. Furthermore, for a node storing n entries we allocate exactly n partial keys and n values, and use copy-on-write when modifying nodes. Besides saving space, copy-on-write also helps concurrent operations (cf., Section 5). Using arbitrary node sizes (between 2 and 32) takes the concept of adaptive nodes [18] to its logical conclusion.

4.3 Lookup

Listing 2 shows the (slightly simplified) code for lookup, which traverses the tree until a leaf node containing a tuple identifier is encountered. As a last step, the key corresponding to that tuple is loaded from the database and compared to the search key (line 7). This is necessary because lookup in a Patricia trie may otherwise yield a false positive.

The "heavy lifting" of a lookup operation is done by the `retrieveResultCandidates` function (lines 12–25), which, given a node and the search key, performs the actual intra-node search consisting of two steps: (1) extracting the (dense) partial key from the search key, and (2) comparing the node's (sparse) partial keys with it. The implementation of both steps depends on the node layout. For this reason `retrieveResultCandidates` consists of a `switch` statement over all node layouts (lines 13–24). Each case then consists of calling the appropriate extraction (`extract*`) and search primitives (`searchPartialKeys*`) (e.g., lines 14–16).

To illustrate the extraction of a key's discriminative bits, we show code for single-mask (`extractSingleMask`, lines 27–30) and multi-mask 8 extraction (`extractMultiMask8`, lines 31–38), both of which use the PEXT instruction. The AVX2-based search is shown for 8-bit partial keys (`searchPartialKeys8`, lines 42–49). The remaining primitives are implemented analogously.

4.4 Insertion

Supporting efficient insertion operations is an important aspect of any index structure. In the context of HOT, insertion performance depends on a fast method to insert new keys into its nodes representing linearized k-constrained tries.

As described in Section 4.1, each node consists of a set of partial keys. The most obvious approach to create partial keys, are so-called *dense partial keys*. These dense partial keys are formed by extracting all discriminative bit positions for each key, e.g., 5 bit positions (3,4,6,8,9) for the example in Table 5b. While search and deletion operations on dense partial keys can be implemented efficiently, inserting a new key into a node can be slow. The reason is that a new key may yield a new discriminative bit and may therefore require resolving this new discriminative bit for all keys already stored in that node. For instance, inserting the key 0110101101 into the binary Patricia trie of Figure 5a would result in bit 7 becoming a new discriminating bit and thus, a new bit position. All existing dense partial keys would have to be extended to 6 bits length and therefore, new bits would have to be determined by loading the existing keys, which would obviously slow down insertion. To overcome this shortcoming, we use a slightly modified version of partial keys, which we call *sparse partial keys*. The difference to dense partial keys is that for sparse partial keys, only those discriminative bits are extracted that correspond to inner BiNodes along the path from the root BiNode and that *all other bits are set to 0*. Thus, sparse partial key bits set to 0 are intentionally left undefined. In case of a deletion this allows to remove unused discriminative bits. To illustrate the difference between dense and sparse partial

Listing 2: HOT lookup.

```
TID lookup(Node* root, uint8_t* key) {
  Node* node = root;
  while (!isLeaf(node)) {
    uint32_t candidates = retrieveResultCandidates(node, key);
    node = node->value[clz(candidates)];
  }
  if (!isEqual(loadKey(getTid(node)), key))
    return INVALID_TID; // key not found
  return getTid(node);
}

uint32_t retrieveResultCandidates(Node* node, uint8_t* key) {
  switch (getNodeType(node)) {
    case SINGLE_MASK_PKEYS_8_BIT:
      uint32_t partialKey = extractSingleMask(node, key);
      return searchPartialKeys8(node, partialKey);
    case MULTI_MASK_8_PKEYS_8_BIT:
      uint32_t partialKey = extractMultiMask8(node, key);
      return searchPartialKeys8(node, partialKey);
    ...
    case MULTI_MASK_32_PKEYS_32_BIT:
      uint64_t partialKey = extractMultiMask32(node, key);
      return searchPartialKeys32(node, partialKey);
  }
}

uint32_t extractSingleMask(SMaskNode* node, uint8_t* key) {
  uint64_t* keyPortion = (uint64_t*) (key + node->offset)
  return _pext_u64(*keyPortion, node->mask);
}
uint32_t extractMultiMask8(MMask8Node* node, uint8_t* key) {
  uint64_t keyParts = 0;
  //load all 8-bit mask into a single 64-bit mask
  for (size_t i=0; i < node->numberMasks; ++i)
    ((uint8_t*) keyParts)[i] = key[node->offsets[i]];
  //SIMD approach to extract multiple 8-bit masks in parallel
  return _pext_u64(keyParts, node->mask);
}
uint32_t extractMultiMask16(MMask16Node* node, uint8_t* key)...
uint32_t extractMultiMask32(MMask32Node* node, uint8_t* key)...

int searchPartialKeys8(Node* node, uint32_t searchKey) {
  __m256i sparsePKeys = _mm256_loadu_si256(node->partialKeys8);
  __m256i key = _mm256_set1_epi8(searchKey);
  __m256i selBits = _mm256_and_si256(sparsePKeys, key);
  __m256i complyKeys = _mm256_cmpeq_epi8(selBits, sparsePKeys);
  uint32_t complyingMask = _mm256_movemask_epi8(complyKeys);
  return bit_scan_reverse(complyingMask & node->usedKeysMask);
}
int searchPartialKeys16(Node* node, uint32_t partialKey) ...
int searchPartialKeys32(Node* node, uint32_t partialKey) ...
```

keys, we show both in Table 5b for the trie in Figure 5a. Based on this definition of sparse partial keys, we now describe how new keys can be inserted into a HOT node.

Before the actual insertion, a search operation is issued checking whether the key to insert is already contained. If the thereby retrieved key does not match the search key, the mismatching bit position is determined. In contrast to a traditional binary Patricia trie, explicitly determining the corresponding mismatching BiNode is impossible, as explicit representations for BiNodes do not exist in linearized k-constrained tries. Instead, we directly determine all leaf entries contained in the subtree of the mismatching BiNode and denote these entries as the *affected entries*. Using SIMD instructions, we therefore mark all partial keys that have the same prefix up to the mismatching bit position as the initially matching false positive partial key as affected. Next, if the mismatching bit position is not contained in the set of the node's discriminative bit positions, all sparse partial keys are recoded using a single PDEP instruction to create partial keys containing also the mismatching bit position. For instance, to add bit position 7 to the sparse partial keys depicted in Figure 5a, the `_pdep_u32(existingKey, 0b111011)` instruction is executed for each key.

To directly construct the new (sparse) partial key representation, we exploit the fact that it shares a common prefix up to the mismatching bit with the affected entries. Therefore, to obtain the new (sparse) partial key, we copy this prefix and set the mismatching bit accordingly. As the bit at the mismatching bit position discriminates the new key from the existing keys in the affected subtree, the affected partial keys' mismatching bits are set to the inverse of the new key's mismatching bit. Finally, again depending on the mismatching bit, the newly constructed partial key is inserted either directly in front or after the affected entries.

4.5 Optimizations

To improve the performance of search as well as insert operations, we prefetch the first 4 cache lines of a node. Furthermore, we encode the node type within the least-significant bits of each node pointer. These two optimizations allow overlapping two time-consuming operations, namely loading the node data, which can trigger a cache miss, and resolving the node type, which may otherwise suffer from a branch misprediction penalty.

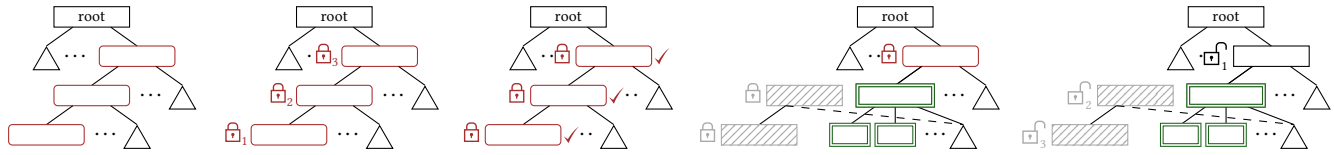
5 SYNCHRONIZATION PROTOCOL

Besides performance and space efficiency, scalability is another crucial feature of any index structure. A scalable synchronization protocol is therefore necessary to provide efficient concurrent index accesses. In the following, we present such a protocol for HOT.

Traditionally, index structures used fine-grained locking and lock coupling to provide concurrent accesses to index structures [7, 8]. However, it has been shown that using such fine grained locks for reading and writing has a huge impact on the overall system performance and does not scale well [19]. Therefore, different approaches based on the concept of lock-free index structures or write-only minimal locks have been proposed [15, 19–21]

Lock-free data structures often use a single compare-and-swap (CAS) operation to atomically perform updates. Therefore, it is tempting to assume that HOT—using a copy-on-write approach and swapping a single pointer per insert operation—would be lock-free by design. However, using a single compare-and-swap (CAS) operation does not suffice to synchronize write accesses to HOT. If two insert operations are issued simultaneously, it is possible that inserts are lost. If one insert operation replaces a node N with a new copy N', while the other insert operation replaces a child node C of N with a new copy C', it might occur that in the final tree, node C is a child of N', whereas C' is a child node of the now unreachable node N.

Although the combination of copy-on-write and CAS is not enough to synchronize HOT, it is a perfect fit for the Read-Optimized



(a) Determine affected nodes. (b) Lock affected nodes bottom-up. (c) Validate locked nodes. (d) Perform insertion and mark modified as obsolete. (e) Unlock all affected nodes.

Figure 7: Step-by-step example of HOT's synchronization protocol. The example shows an insertion operation resulting in a parent pull up. The three affected nodes are marked red with rounded corners, the newly created nodes are marked green with a double border and all modified and therefore obsolete nodes are marked gray and filled with a line pattern.

Write EXclusion (ROWEX) synchronization strategy [19]. ROWEX does not require readers to acquire any locks and hence, they can progress entirely wait-free (i.e., they never block and they never restart). Writers, on the other hand, do acquire locks, but only for those nodes that are actually modified during an insert operation. Writers also have to ensure that the data structure is valid at any point in time because locks are ignored by readers. As a result, the lookup code (cf, Listing 2) remains unaffected, and in the following we only describe update operations.

Modification operations (e.g., insert, delete) are performed in five steps, which are illustrated in Figure 7 and explained in the following: (a) During tree traversal, all nodes that need to be modified are determined (and placed in a stack data structure). We denote these nodes as the *affected nodes*. (b) For each of the affected nodes a lock is acquired—in bottom-up order to avoid deadlocks. (c) A validation phase then checks whether any of the affected nodes is obsolete, i.e., have not been removed in the meantime. In case any of the locked nodes is invalid, the operation is restarted (after unlocking all previously locked nodes). (d) If the validation is successful, the actual insert operation is performed. Nodes replaced by new nodes (due to copy-on-write) are marked as obsolete. (e) Finally, all locks are released (in top-down order).

The crucial part in HOT's synchronization implementation is to determine the set of affected nodes, as a single modification operation can affect multiple nodes. Analogously to the insertion operation, we distinguish 4 different approaches to determine the set of affected nodes (cf. Section 3.2). In case of a normal insert the set of affected nodes consists of the node containing the mismatching BiNode and its parent node. For the other three cases (i) *leaf-node pushdown*, (ii) *parent pull up*, and (iii) *intermediate node creation* the set of affected nodes is determined as follows: (i) In case of a leaf-node pushdown, the set of affected nodes solely consists of the node containing the mismatching BiNode. If an overflow occurs, all ancestor nodes of this node are traversed, and added to the set of affected nodes until either (ii) in case of a parent pull up, a node with sufficient space or the root node is reached or (iii) in case of an intermediate node creation, a node n fulfilling $height(parent(n)) >= height(n)$ is reached. Finally, the direct parent of the last accessed node is added.

Another critical aspect of HOT's synchronization strategy is marking nodes as obsolete instead of directly reclaiming the nodes' memory. This reclamation strategy has two advantages. On the one hand, the obsolete marker allows concurrent writers to detect whether one of the currently locked nodes has been replaced in the meantime (and restart the operation). On the other hand, readers do not need any locks to deal with concurrent writes. Whenever writers modify a currently read node, the reader is able to finish the lookup—on the now obsolete—node. To actually reclaim the memory of obsolete nodes HOT uses an epoch based memory reclamation strategy[9], which frees the memory of obsolete nodes whenever no more reader or writer accesses the corresponding nodes.

6 EVALUATION

In the following, we experimentally evaluate HOT and compare it with other state-of-the-art in-memory index structures. We first describe the experimental setup before presenting our results, which focus on the following four areas: (i) performance, (ii) memory consumption, (iii) scalability, and (iv) tree height.

6.1 Experimental Setup

Most experiments were conducted on a workstation system with an Intel i7-6700 CPU, which has 4 cores and is running at 3.4 GHz with 4 GHz turbo frequency (32 KB L1, 256 KB L2, and 8 MB L3 cache). The scalability experiments were conducted on a server system with an Intel i9-7900X CPU, which has 10 cores and is running at 3.3 GHz with 4.3 GHz turbo frequency (32 KB L1, 1 MB L2, and 8 MB L3 cache). Both systems are running Linux and all code was compiled with GCC 7.2.

We compare the following state-of-the-art index structures:

- *ART:* The Adaptive Radix Tree (ART) [18], which is the default index structure of HyPer [30]. It features variable sized nodes and selectively uses SIMD instruction to speed up search operations.
- *Masstree:* Masstree [22] is a hybrid B-Tee/trie structure used by Silo [26].
- *BT*: The STX B$^+$-Tree[2] represents a widely used cache-optimized B$^+$ Tree (e.g., [12]) and hence, a baseline for a comparison-based approach. The default node size is 256 bytes which in the case of 16 bytes per slot (8 bytes key + 8 bytes value) amounts to a node fanout of 16.
- *HOT:* Our C++14 implementation of HOT using AVX2.

We use the publicly available implementations of ART, the STX B$^+$-Tree, and Masstree. We do not compare against hash tables, as these do not support range scans and related operations.

We use 64-bit pointers as tuple identifiers to address and resolve the actually stored values and keys. In case the stored values only

[2]https://github.com/bingmann/stx-btree

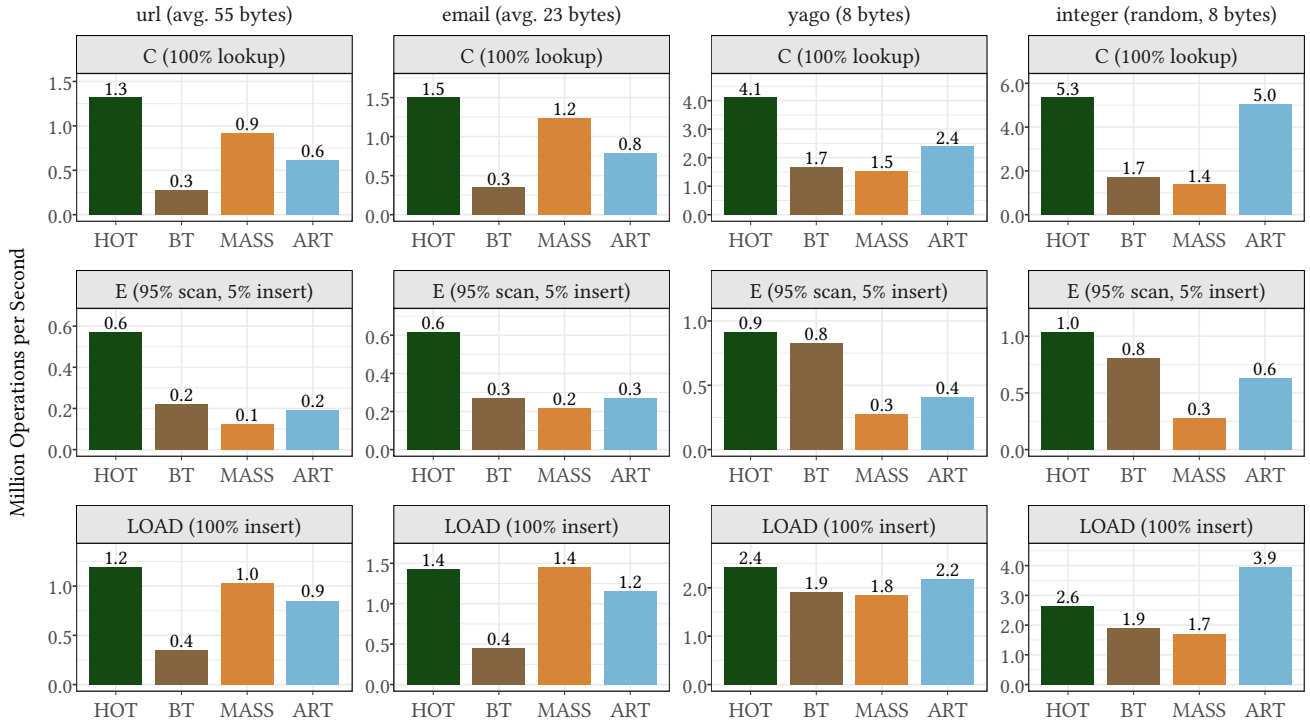


Figure 8: Throughput using different data sets in million operations per second for read-only workload C, scan-heavy workload E, and the insert-only load phase.

consist of fixed sized keys up to 8 byte length (e.g., 64-bit integers), those keys are directly embedded in their tuple identifiers.

For our workload, we rely on the work of Zhang et al. [30], who proposed an index micro-benchmark adapted from the YCSB framework [5]. We therefore base our work on the available implementation of their workload generator[3] and add support to jointly configure multiple workloads. To foster reproducibility and repeatability, we make the extended workload generator available online[4].

Our *benchmark configurations* correspond to the six YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (read-only), D (latest-read, 95% read, 5% insert), E (95% range-scan accessing up to 100 elements, 5% insert) and F (50% read, 50% read-modify-write) [5]. Each workload is separated into two phases: the load phase inserts 50 million keys in random order into the index structure to evaluate; in the transaction phase, 100 million operations specified by the workload to be evaluated are executed. Each benchmark configuration is created in two variants: using a Zipfian and a uniform distribution to select the records to operate on. Each of these benchmark configurations is created for four different data sets, two string data sets representing long keys and two 8 byte short key data sets:

- *url:* The url data set consists of a total of 72,701,109 distinct URLs, which we collected from the 2016-10 DBPedia data set [2], where we removed all URLs that are longer than 255 characters.
- *email:* 30 byte long email addresses originating from a real-world email data set. We cleansed the data set by removing invalid email addresses or emails solely consisting of numbers or special characters.
- *yago:* 63-bit wide triples of the Yago2 data set [11]. The triples are compound keys, where the lowest 26 bits are used for the object id, bits 27 to 37 store predicate information and bits 38 to 63 are used for subject information.
- *integer:* uniformly-distributed 63-bit random integers.

Overall, the 6 workloads (A, B, C, D, E, F), 4 data sets (url, email, yago, integer) and 2 operation distributions (uniform, Zipf) amount to a total of 48 different benchmark configurations. For each of the 48 benchmark configurations, performance metrics, statistics regarding the overall memory consumption of the respective index structures and the distribution of leaf values in relation to the depth of occurrence are collected.

6.2 Performance

In this section, we present the results of the runtime performance evaluation. Here, we mainly discuss the results of the uniformly distributed lookup-only workload C, the scan-heavy (95% scan, 5% insert) workload E, consisting of short range scans accessing up to 100 entries and an insert only workload consisting of the load phase, which is identical for all workloads. The results of these three workloads is depicted in Figure 8. For each combination of workload and data set, the figure shows a bar plot illustrating the number of million operations per second (mops) executed by each of the evaluated index structures. As the remaining workloads A, B, D, and F only represent combinations of the workloads presented in

[3]https://github.com/huanchenz/index-microbench
[4]https://github.com/speedskater/index-microbench

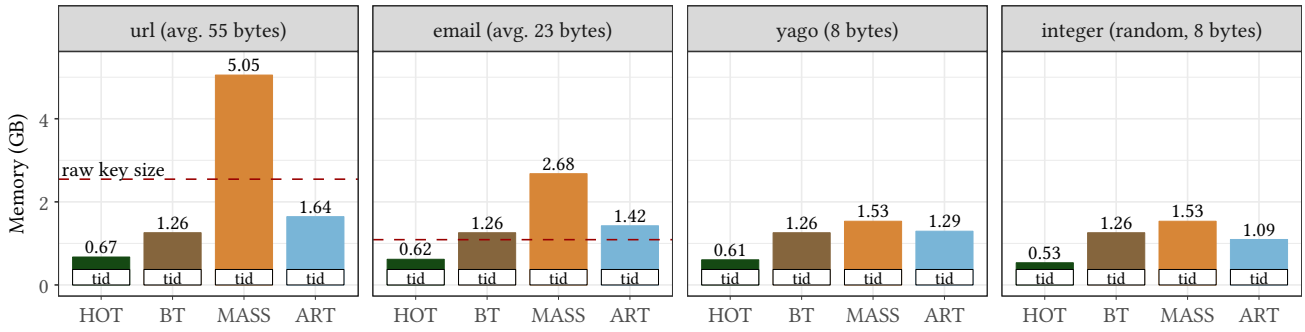


Figure 9: Memory consumption in gigabytes. The lower part of each bar, labeled with tid, represents the space required to store the raw tuple identifiers (0.37 GB). For the short-key data sets, the space required to store the raw keys is equal to the space required to store the raw tuple identifiers. For the string-based data sets, the dashed red line marks the space required to store the raw keys (url: 2.55 GB, email: 1.09 GB).

Figure 8 and the performance results for the Zipfian distributed operations are similar to the uniformly distributed ones, we omit those results for the remainder of this section. However, for completeness, these additional results are depicted in the Appendix A.

For the lookup-only workload C, HOT achieves an at least 25% higher throughput across all evaluated data sets compared to the other evaluated index structures. For the scan heavy workload E, consisting of short range scans accessing up to 100 elements, HOT has the highest throughput for all key distributions. In case of the URL data set, HOT's throughput is 200% higher than for the other data structures. For the insert only workload, the measured performance characteristics are similar to the read operations discussed in context of workload C. An exception is the insertion throughput for the integer data set, where ART achieves a 50% higher insertion throughput compared to HOT. However, for all other data set none of the evaluated index structures is able to process a higher amount of insertion operations than HOT. Having discussed the raw runtime characteristics of the evaluated index structures, we draw the conclusion that HOT is able to achieve consistently high performance regardless of the evaluated workload or data set, which makes it highly promising as a general purpose index structure for main memory databases. The other evaluated index structures achieve peak performance for specific workloads, but are not able to provide consistently high throughput for all workloads and data sets.

6.3 Space Consumption

To evaluate HOT's memory efficiency, we rely on the benchmark configurations presented in Section 6.1. As the data sets and therefore, the memory consumption is identical for all workloads, we present the memory consumption measured after the load phase of the read-only workload C. To measure the memory consumption of ART and the B-Tree, we add custom code to their implementations that allows computing the memory consumption without impacting the runtime behavior of the respective data structures. For Masstree, we use its allocation counters to measure the space consumption. To ensure a fair comparison we do not take the memory required to store Masstree's tuples into account, as the space required to represent the raw tuples is not considered for any of the evaluated data structures.

Figure 9 reports the memory consumption for all measured data structures. The overall height of each bar represents the total amount of memory required (in GB) to store the given data set in the respective index structure. The white lower part of each bar represents the minimum amount of memory necessary to store the raw 8-byte tuple identifiers (tid) of the indexed values. As all data sets contain 50 million entries, the memory required to store those raw tuple identifiers is the same for all workloads (0.37 GB). Additionally, the raw size of the stored keys is marked with a dashed red line for the two textual data sets (email: 1.09 GB, url: 2.55).

Figure 9 shows that for all four data sets, HOT outperforms the other index structures in terms of memory consumption. The evaluation shows that the space consumption of trie based index structures increases for long and non-uniformly distributed keys. While in the worst case of long textual keys Masstree's space consumption increases by 230% and Art's space consumption by 51%, HOT's space consumption only increases by 26% in comparison to the integer data set. Thereby, HOT is the only trie based index structure, which for all data sets has a space consumption which is substantially below the space consumption of the B-Tree. Due to the B-Tree's design and the decision to use tids to resolve keys longer than 8 bytes, the amount of space required is the same (1.26 GB) for all data sets. However, for all data sets, it requires at least 88% more space than HOT's worst-case space consumption measured for the url data set. Moreover, HOT is the only index structure which for both textual data sets requires less space than the actual raw keys (email: 43%, url: 74%).

To conclude, besides featuring superior memory consumption in comparison to the other evaluated data structures, HOT has a very stable memory footprint, which for all evaluated data sets lies between 11.4 and 14.4 bytes per key.

6.4 Scalability

Besides evaluating HOT's single-threaded performance, we evaluated HOT in terms of its scalability. For each of the data sets described in Section 6.1, we execute a workload consisting of 50 million randomly distributed insert operations, followed by 100

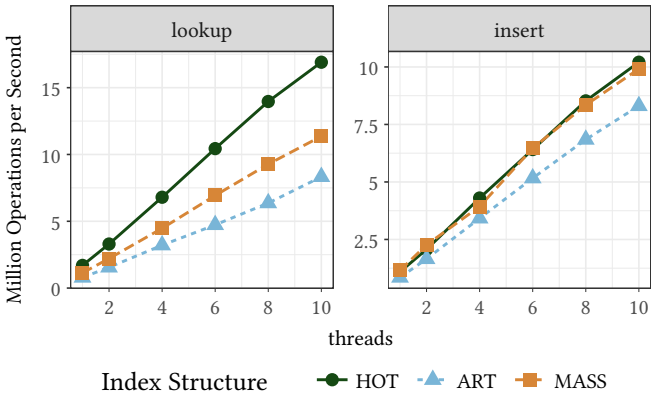


Figure 10: Scalability on the 50M urls data set.

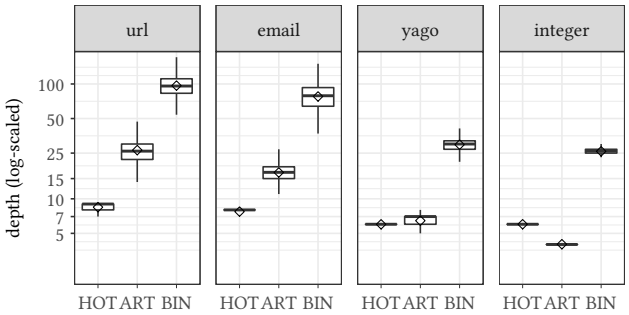


Figure 11: Depth distribution of leaves for HOT, ART, and binary Patricia trie (BIN). Diamonds are the mean.

million uniformly distributed random lookups. Each workload is executed seven times for thread counts between one and ten, with ten representing the maximum physical core count of the server used to run the evaluation. To prevent outliers, the median throughput of the seven executed runs is considered for the comparison.

We conduct this experiment for the synchronized versions of Masstree, ART (using the ROWEX synchronization protocol) and HOT. In contrast to the previously conducted single threaded experiments these variations of the evaluated index structures support concurrent modifications. Therefore, due to lack of synchronization, we omit the STX B-Tree for the scalability evaluation.

As all evaluated index structures, achieve a near linear speedup we depict the absolute performance numbers for insert and lookup operations only for the url data set in Figure 10. For all other data sets the speedups vary slightly between the evaluated data structures. For instance, the mean speedups for all lookup operations are 9.96 for HOT, 9.91 for ART, and 10.1 for Masstree. The respective mean speedups for the insert operations are 9.00 for HOT, 9.51 for ART, and 7.87 for Masstree.

From these experiments, we conclude that besides featuring excellent single threaded performance, HOT's synchronization protocol achieves almost linear scalability.

6.5 Tree Height

To better understand the performance and space consumption numbers reported above, we now present the depth distribution of leaf values, which is a measure of how balanced a tree is.

Specifically, we are interested in the effect of HOT's algorithm to combine multiple BiNodes into compound nodes. We therefore compare the depth distribution of HOT with the depth distribution of the "pure trie structures" (ART and binary Patricia trie). As existing trie structures are vulnerable to deep structures in case of long or non-uniformly distributed keys (cf. Section 2), we analyze HOT's depth distribution for all data sets (url, email, yago and integer). The results are illustrated in Figure 11.

The results show that for textual data sets (email and url) HOT is able to reduce the mean depth of leaf entries up to 68% in comparison to ART and up to an order of magnitude in contrast to the binary Patricia trie. Even for non-uniformly distributed short keys represented by the yago data set, HOT achieves the lowest mean depth. Only for the integer data set consisting of uniformly distributed random keys, ART's maximal node fanout results in a lower mean depth (HOT: 6.0, ART: 4.02).

To conclude, we observe that HOT achieves a lower depth distribution than other trie structures—regardless of the data set and its distribution. While, the worst case mean depth distribution of the other evaluated trie structures can be several times higher than in the best case (ART 560%, BIN 270%), HOT's depth distribution in the worst case is only 42% higher than in its best case. Only in the case of uniformly distributed short keys, HOT's mean depth distribution is larger when compared with trie structures having a higher maximum node fanout like ART. We therefore argue that HOT fulfills the requirements of a general purpose index structure, achieving a consistently low depth distribution of leaves regardless of the key size or distribution when compared with existing trie structures in our experiments.

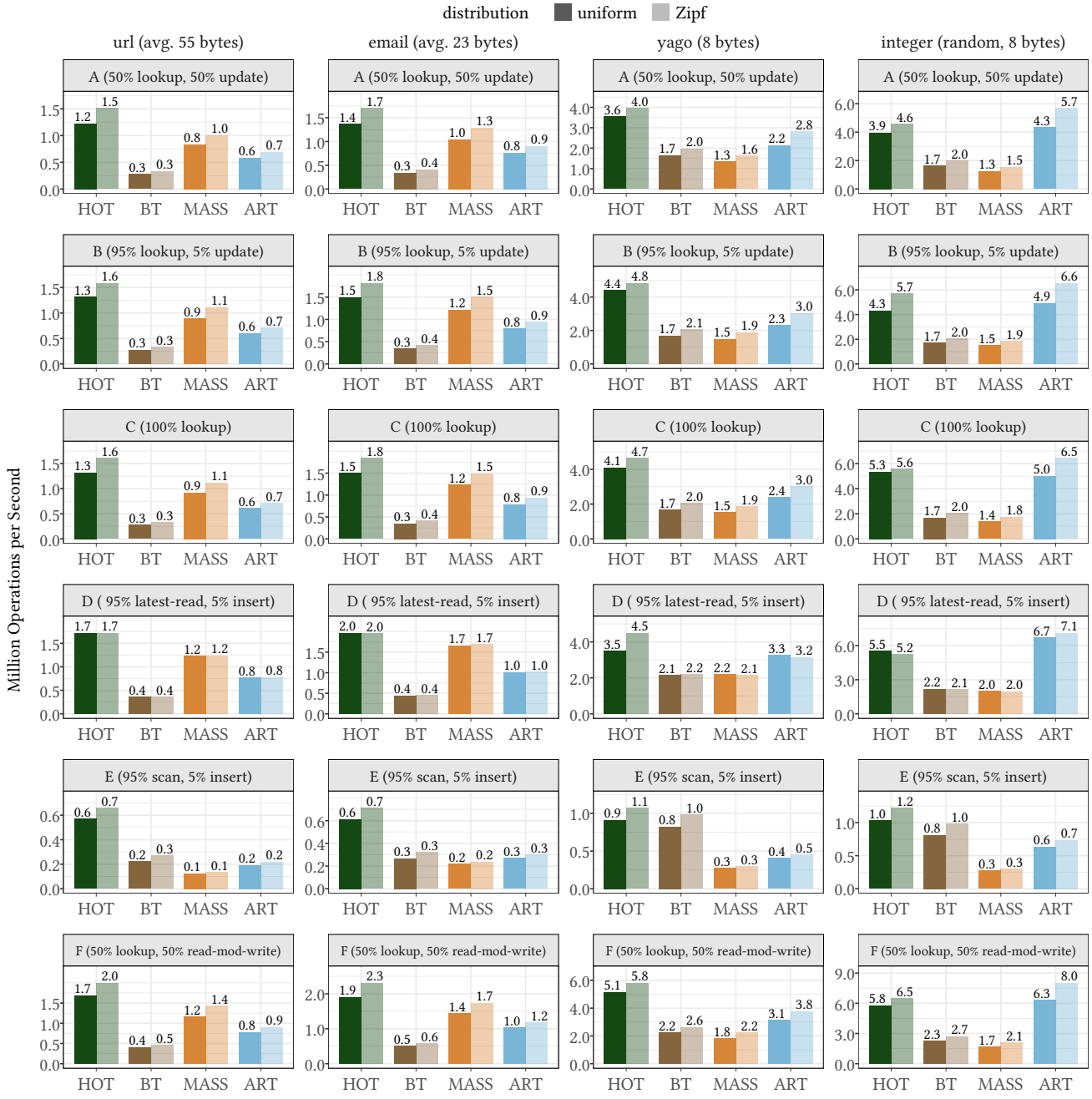
7 SUMMARY

We presented the Height Optimized Trie (HOT), which is a novel index structure that adjusts the span of each node depending on the data distribution. In contrast to existing trie structures, this enables a consistently high fanout for arbitrary key distributions. Furthermore, HOT's compact node layout enables efficient search using SIMD operations.

Our experimental results show that HOT is 2x as space efficient as its state-of-the-art competitors (B-trees, Masstree and ART), that it generally outperforms them in terms of lookup and scan performance and that it features the same linear scalability. These properties make HOT a highly promising index structure for main-memory database systems.

In future work we aim to proof the theoretical properties of HOT including the worst-case height properties and the deterministic characteristics of the overall structure. Further, we plan to investigate methods to establish higher node fanouts, thereby reducing the overall tree height even further.

A ADDITIONAL WORKLOADS



REFERENCES

[1] N. Askitis and R. Sinha. HAT-trie: a cache-conscious trie-based data structure for strings. *Proceedings of the thirtieth Australasian conference on Computer science - Volume 62*, pages 97–105, 2007.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference*, pages 722–735, 2007.

[3] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Proceedings of the 14th BTW conference on Database Systems for Business, Technology, and Web*, pages 227–246, 2011.

[4] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 235–246, 2001.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[6] D. E. Ferguson. Bit-tree: A data structure for fast file processing. *Communications of the ACM*, 35(6):114–120, June 1992.

[7] G. Graefe. A survey of b-tree locking techniques. *ACM Transactions on Database Systems*, 35(3):16:1–16:26, July 2010.

[8] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[9] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[10] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, Apr. 2002.

[11] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. In *Proceedings of the 20th International Conference Companion on World Wide Web*, pages 229–232, 2011.

[12] 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. In *Proceedings of the VLDB Endowment*, pages 1496–1499, Aug. 2008.

[13] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, April 2011.

[14] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 339–350, 2010.

[15] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706, 2015.

[16] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart Latch-free In-memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 16–23, 2012.

[17] A. Kovács and T. Kis. Partitioning of trees for minimizing height and cardinality. *Information Processing Letters*, 89(4):181–185, 2004.

[18] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering*, pages 38–49, 2013.

[19] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN, 2016.

[20] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering*, pages 302–313, April 2013.

[21] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.*, 8(11):1298–1309, July 2015.

[22] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 183–196, 2012.

[23] D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 10 1968.

[24] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[25] B. Schlegel, R. Gemulla, and W. Lehner. k-Ary Search on Modern Processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 52–60, 2009.

[26] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[27] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, 2018.

[28] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. F. Wong. Parallelizing skip lists for in-memory multi-core database systems. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering*, pages 119–122, April 2017.

[29] H. Zhang, D. G. Andersen, M. Kaminsky, A. Pavlo, H. Lim, V. Leis, and K. Keeton. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, 2018.

[30] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1567–1581, 2016.

[31] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.