Height Optimized Tries

ROBERT BINNA, EVA ZANGERLE, MARTIN PICHL, and GÜNTHER SPECHT,
University of Innsbruck, Austria
VIKTOR LEIS, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

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. For a fixed maximum node fanout, the overall tree height is minimal and its structure is deterministically defined. Multiple carefully engineered node implementations using SIMD instructions or lightweight compression schemes provide compactness and fast search and optimize HOT structures for different usage scenarios. Our experiments, 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.

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

Additional Key Words and Phrases: Height optimized trie, main memory, index structure, SIMD

ACM Reference format:
Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Trans. Database Syst.* 47, 1, Article 3 (April 2022), 46 pages.
https://doi.org/10.1145/3506692

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 [40]. 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 [35] or HyPer [19]) use trie structures (e.g., Masstree [29] or Adaptive Radix Tree (ART) [25]). The reason for this preference is that, in main memory, well-engineered tries often outperform comparison-based structures like B-trees [2, 9, 25, 40]. Furthermore, unlike hash tables, tries are order-preserving and therefore support range scans and related operations. Nevertheless, even recently proposed trie structures 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.

Authors' addresses: R. Binna, E. Zangerle, M. Pichl, and G. Specht, University of Innsbruck, Department of Computer Science, Technikerstraße 21a, 6020 Innsbruck, Austria; emails: rbinna@gmail.com, eva.zangerle@uibk.ac.at, martin.pichl@pichl.com, guenther.specht@uibk.ac.at; V. Leis, Friedrich-Alexander Universität Erlangen-Nürnberg, Germany; email: viktor.leis@fau.de.

0362-5915/2022/04-ART3 $15.00
https://doi.org/10.1145/3506692

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 it 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, hence, improving cache efficiency. In this work, we are actually able to prove that the height of HOT structures is minimal and the overall structure is deterministically defined regardless of the insertion order of the stored keys.

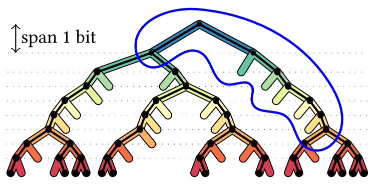
Based on this novel algorithmic approach, we propose different node layouts that tailor HOT's structure to different requirement. The proposed layouts of nodes are carefully engineered for good performance on modern CPUs and minimal memory requirement. The node representations are optimized for cache efficiency and apply efficient, SIMD-optimized search whenever it is beneficial. The different node layouts are extensively evaluated on a wide variety of workloads and data distributions. We compare our HOT variants with B-trees, Masstree, and ART, which are state-of-the-art, order-preserving, in-memory index structures. The experimental results show that the HOT structures generally outperform its competitors in terms of performance and space consumption, for both short integers as well as long strings. These properties make HOT particularly well suited as an index for in-memory database systems and, more generally, for string-intensive applications. The SIMD optimized implementation of our fastest HOT node layout is publicly available under the ISC license at `https://github.com/speedskater/hot`.

The rest of the article 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 the theoretical properties of the algoriths are analyzed and proven in Section 4. A number of possible node layouts are then described in Section 5. Section 6 presents a scalable synchronization protocol for multi-core CPUs. Finally, after presenting the experimental evaluation in Section 7, we summarize the article in Section 8.

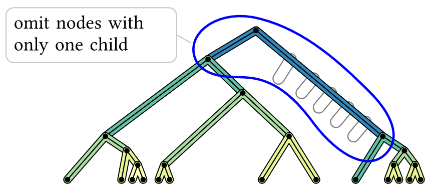
This article is based on a conference paper [6], which we extend with additional material that was previously submitted as part of a PhD thesis [7]: First, in Section 3, we introduce a detailed explanation of the deletion operation. Second, in Section 4, we introduce and prove three theoretical properties of HOT: (I) minimal height, (II) deterministic structure, and (III) recursive design. Third, in Section 5.2, we confirm the assumption mentioned in the conference paper that HOT structures can be geared toward different usage scenarios by using different physical node layouts. To this end, we introduce the new family of hierarchical node layouts. We show that by trading off access performance for memory efficiency, space consumption can be significantly reduced. In addition, we provide a discussion on the general applicability of the presented synchronization protocol in Section 6, an upper bounds estimation of HOT's space consumption in Section 3, and a discussion on possible disk-optimized node layouts in Section 5.3.

2 BACKGROUND AND RELATED WORK

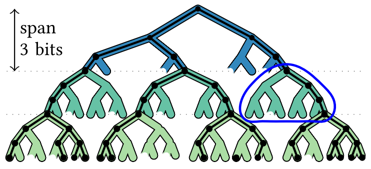
The growth of main memory capacities has led to the development of index structures that are optimized for in-memory workloads (e.g., References [10, 20, 27, 33, 34, 37, 42]). **Tries**, in particular, have proven to be highly efficient on modern hardware [2, 9, 22, 25, 29, 36, 39–41]. Tries are tree



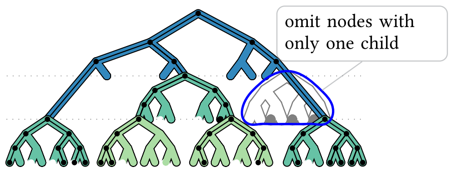
(a) Binary trie (height=9, #nodes=37).



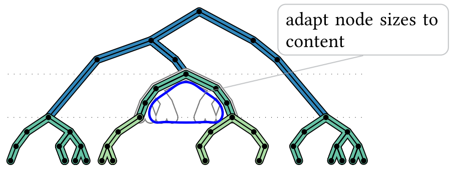
(b) Patricia (height=5, #nodes=12).



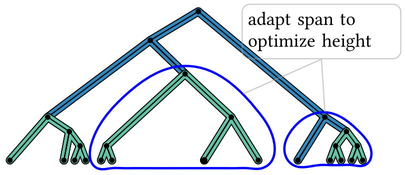
(c) 3 bit span (height=3, #nodes=8).



(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).

Fig. 1. Trie optimizations. Nodes on the same level have the same color. Discriminative bits are shown as black dots.

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. In the remainder of this article, we denote this particular bit as the discriminative bit. 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 1 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 1(a), a binary trie is depicted. The subsequent Figures 1(b)–1(f) illustrate different optimizations, which we discuss in the following.

Figure 1(b) shows a binary **Patricia** trie [30], 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 (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

[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).

in each node. The Generalized Prefix Tree [9], 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 1(c), 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 1(d) shows, when applied to the trie depicted in Figure 1(c) 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 ART [25], 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 nodes is depicted in Figure 1(e), which adds adaptive nodes to the trie of Figure 1(d). 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 1 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 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 1(f) 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 [16] and HAT-Trie [2] 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 [29], 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 [11] 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.

Besides the logical high-level design, also the physical layout of individual nodes that are geared toward the underlying hardware has a major impact on the performance of tree-based index structures. In particular, the trend toward multi-layer cache hierarchies, long CPU pipelines, and wider SIMD instruction sets paved the way to many innovations in the area of main-memory index structures.

Overcoming the disparity between cache and memory access latency is similar to bridging the gap between main memory and disk access latencies [14]. Hence, cache-conscious index structures use similar techniques as disk optimized index structures but are geared toward cache-line size instead of page size [32, 33]. Other optimization techniques for cache-conscious index structures are lightweight compression [5, 8, 14], separation of key and pointer information [4, 25, 29], and reduction of the amount of wasted space [40].

To reduce latencies caused by branch misprediction, the general approach is to avoid branches. For instance, search operations on fixed span tries omit node internal branches by using partial keys to directly address matching child pointers [9, 25].

Another approach to reduce the number of branch mispredictions is to use SIMD optimized multiway-search operations [34, 42]. Instead of comparing a single value at a time, SIMD instructions allow one to compare multiple values in parallel. To increase the number of comparisons per SIMD instruction, index structures typically use compressed keys [20] or partial keys in the comparison [25, 38].

So far, all these SIMD optimizations use k-array search to either compare compressed or partial keys. In contrast, our *Linearized Node Layouts* uses SIMD instructions and a sparse key representation to optimistically search linearized binary patricia tries.

3 THE HEIGHT OPTIMIZED TRIE

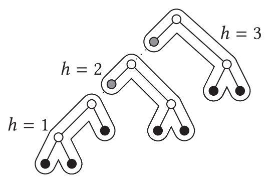
The optimizations discussed in the previous section combine the nodes of a binary trie into compound nodes with a higher fanout. The most important optimization is to increase the span of each node. However, in current data structures, the span is a static, fixed setting (e.g., 8 bits) that is set globally 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 a 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 also close to optimal. For integers randomly drawn from the full 64-bit domain, however, many nodes at lower levels of the tree are only 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.

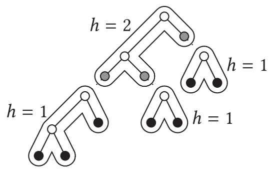
To solve the problem of sparsely-distributed keys, we propose to set the span of each node adaptively depending on the data distribution. Thus, dense key regions (e.g., near the root) have a smaller span than sparse regions (e.g., at lower levels), and a consistently high fanout can be achieved. Instead 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 1(b), 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).



(a) Optimized for # of nodes



(b) Optimized for height

Fig. 2. Two ways of combining binary nodes into compound nodes, annotated with height h.

For a given parameter k, there are multiple ways of combining binary nodes into compound nodes. Figure 2 shows two trees with a maximum fanout $k = 3$ storing the same data. While the tree shown in Figure 2(a) reduces the total number of compound nodes, the tree shown in Figure 2(b) 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

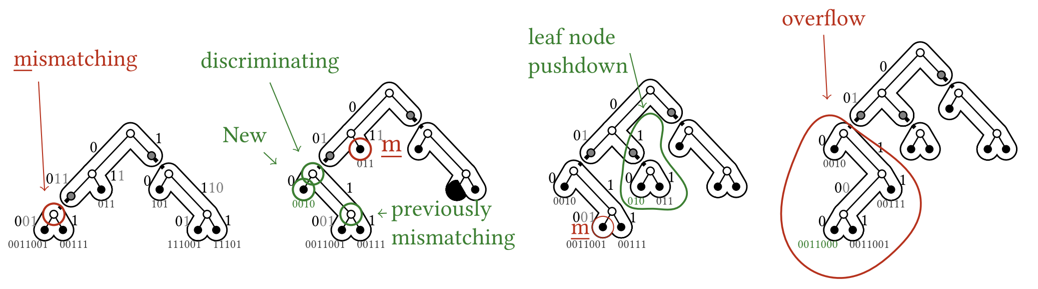
$$h(n) = \begin{cases} 1 & if\, n.m = 0, \\ \max_{i=1}^{n.m}(h(n.child[i])) + 1 & \text{else.} \end{cases}$$

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 [23] solved the problem of partitioning trees such that the overall height and cardinality are optimized. In this article, we present a dynamic algorithm, which is able to preserve the height optimized partitioning while new data is inserted.

To avoid confusion between binary nodes and compound nodes, in the remainder of the article, 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.

Before we describe the insertion and deletion algorithms along the lines of the examples shown in Figures 3 and 4, we introduce the following definitions:

- **Discriminating Bits:** In a binary Patricia, trie each BiNode corresponds to a bit position of the stored keys. As the value of the bit at this position discriminates the key to be either stored in the left subtree (value 0) or the right subtree (value 1), we denote this bit position as the discriminating Bit.
- **Search Path:** For a given key and a given binary Patricia trie, we define the *search path* as the path from the root to a leaf node where the value of the discriminating bit of the search key matches the edges taken.
- **Result Candidate:** By definition, binary Patricia tries omit all BiNodes that have only one child. Hence, search operations are executed optimistically, and we denote the leaf node at the end of the search path as the *result candidate*.
- **Mismatching Bit:** For a given binary Patricia trie P and a key k, that is not contained in K, we define the mismatching Bit $mbit$ to be the most significant bit where $result\_candidate(P, k)[mbit] \neq k[mbit]$ holds.
- **Mismatching BiNode:** For a given binary Patricia trie P and a key $k \notin P$, we define the *mismatching BiNode mb* to be the topmost BiNode on a key's search path with $discriminating\_bit(mb) > mbit$.

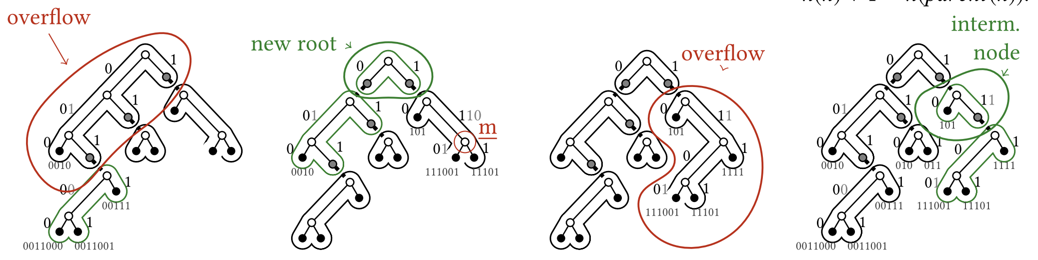


(a) Initial HOT before inserting 0010.

(b) *Normal* insert of 0010 before inserting 010.

(c) Insert of 010 using *leaf node pushdown.* Before Inserting 0011000.

(d) Inserting 0011000 causes overflow in node n having $h(n) + 1 = h(parent(n))$.



(e) Handling overflow by *parent pull up* (propagates overflow to parent node).

(f) Overflow resolution by *parent pull up* resulting in *new root node creation.* Before inserting 1111.

(g) Inserting 1111 causes overflow in node n having $h(n) + 1 < h(parent(n))$.

(h) Overflow resolution by *intermediate node creation.*

Fig. 3. Step-by-step example inserting the keys 0010, 010, 0011000, and 1111 into a HOT with a maximum fanout of $k = 3$. Please note that the gray parts of the shown subkey labels are only displayed for convenience but are not physically stored in the structure.

- **Discriminating BiNode:** For a given binary Patricia trie P and a key $k \notin P$, we define the *discriminating BiNode* to be the BiNode that is inserted as a direct parent of the mismatching BiNode along the key's search path. The new BiNode's discriminating bit is the mismatching bit and its second child is a leaf node corresponding to the new key.

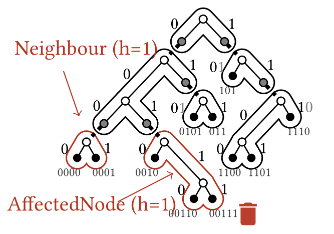
3.2 Insertion, Deletion, and Structure Adaptation

Similar to B-trees, the insertion and deletion algorithms of HOT have a normal code path that affects only a single node and other cases that perform structural modifications to the tree. In the remainder of this section, we will describe these cases in more detail.

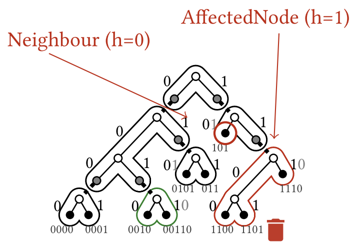
3.2.1 Insertion Operation. In the following, we describe the different cases by successively inserting four keys into a HOT structure with a maximum node fanout of $k = 3$.

The initial tree is shown in Figure 3(a). 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 3(a).

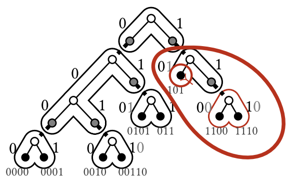
In the **normal** case, insertion is performed by locally modifying the BiNode structure of the affected node. More precisely, and as shown in Figure 3(b), a new *discriminating BiNode*, which discriminates the new key from the keys contained in the subtree of the mismatching BiNode, is created and inserted into the affected node. The normal case is analogous to inserting into a



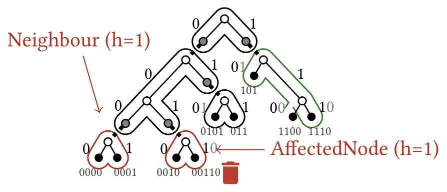
(a) Initial HOT before deleting 00111.



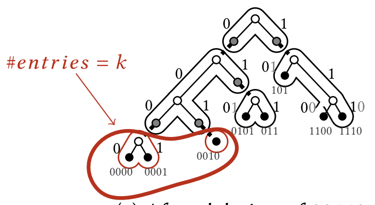
(b) *Normal* deletion of 00111 before deleting 1101.



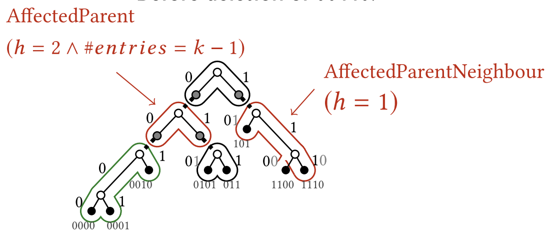
(c) After deletion of 1101. Before *simple BiNode pull down*



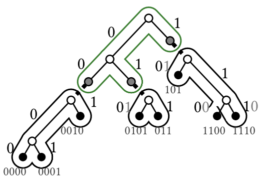
(d) After *simple BiNode pull down*. Before deletion of 00110.



(e) After deletion of 00110. Before *node merge*.



(f) After *node merge*. Before *simple BiNode pull down*



(g) Final tree after *simple BiNode pull down*, which in this case reduces the overall tree height by 1.

Fig. 4. Step-by-step example deleting the keys 00111, 1101, and 00110 from HOT with a maximum fanout of $k = 3$. Please note that the gray parts of the shown subkey labels are only displayed for convenience but are not physically stored in the structure.

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 ($h(n) > 1$), then we replace the leaf with a new node. The new node consists of a single BiNode that distinguishes 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 3(b). Leaf-node pushdown does not affect the maximum tree height as can be observed in Figure 3(c): 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 3(d) 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 3(e) 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 3(f). 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 3(g). As can be seen in Figure 3(g), 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 3(h).

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.

```
insert(hot, key):
   n = traverse hot for key
   m = traverse n until mismatch
   if (isLeafEntry(m) and h(n) > 1):
      # leaf node pushdown
      l = createNode(m, key)
      n̂ = replaceNode(n, m, l)
   else:
      d = createBiNode(m, key)
      n̂ = replaceBiNode(n, m, d)
      handleOverflow(n̂)
```

```
handleOverflow(n):
   if (not isFull(n))
      # normal path
      return
   n̂ = split(n)
   p = parentNode(n)
   if (height(n̂) == height(p)):
      # parent pull up
      e = createBiNode(n̂[0], n̂[1])
      p̂ = replaceBiNode(p, n, e)
      handleOverflow(p̂)
   else
      # intermediate node creation
      p̂ = replaceNode(p, n, n̂)
```

Listing 1. Structure-adapting insertion algorithm.

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.

3.2.2 Deletion Operation. To maintain an optimized overall tree height, it is crucial to not only incrementally maintain an optimized height in case of insertion operations but also in the case of deletion operations. In contrast to the insertion algorithm, the deletion algorithm only requires two kinds of structural modifications.

We describe these different cases by sequentially removing three keys (00111, 1101, and 00110) from an existing HOT structure with a maximum node fanout of $k = 3$. The initial tree is depicted in Figure 4(a). Each deletion operation starts by traversing the tree until the entry to delete is found. We denote the node containing the entry to delete as the *affected node*. Depending on its height and the number of entries in its neighbor node, we distinguish three different cases.

Before we describe the three different kinds of HOT's deletion operations, we introduce our definition of an underflow in HOT, which can occur by removing an entry from a node: An underflow occurs whenever the total number of entries of two sibling nodes is less or equal than the maximum node fanout k. An underflow cannot occur when an *affected node's* sibling is a BiNode.

A normal deletion is executed, whenever removing the entry to delete with its associated *discriminating* BiNode, does not yield an underflow. Removing the key 00111 from the initial tree shown in Figure 4(a) is an example of a normal deletion resulting in the tree depicted in Figure 4(b). A normal deletion on a node, that contains only two entries, represents an edge case as the affected hot node is transformed into a leaf node. Thus, it is the inverse of the leaf node pushdown operation.

To resolve an underflow, we distinguish two scenarios. First, the affected node's neighbor has a smaller height. Second, the affected node's neighbor has the same height.

In the first case, we pull the parent BiNode down into the affected node. We therefore call this ***simple BiNode pull down***. Such a simple BiNode pull down is applied when deleting the entry with the key 1101 from the tree in Figure 4(b). This operation is depicted in a two-step process. First, the entry to delete is removed, yielding the tree of Figure 4(c). The two nodes affected by the resulting underflow are highlighted in this Figure. Next, we execute ***simple parent pull down***, which results in the tree shown in Figure 4(d).

In case the nodes affected by the underflow both have the same height a ***node merge*** is applied. This node merge is the inverse of the parent pull up operation known from the insertion operation and combines the three parts, parent BiNode, left sibling, and right sibling into a single node. A node merge is illustrated in Figure 4(f), which combines the remaining part of the affected node and its neighbor after deleting 00110 from the tree depicted in Figure 4(d).

In contrast to insertion operations, where only parent pull up may recurse up the tree, both underflow resolution strategies simple BiNode pull down as well as node merge may affect their parent node. Therefore, both operations may potentially recurse up the tree until the overall tree height is reduced. The reason is that both operations reduce the number of entries in the affected node's parent node. Thus, a potential underflow may occur in the parent node, which again must be resolved by one of the possible underflow resolution strategies. This is shown in Figure 4(g), which shows the result of executing a simple BiNode pull down after the node merge shown in Figure 4(f).

In summary, HOT's deletion algorithm supports three different cases. The normal deletion is used when no underflow occurs and only affects a single node. In the case of an underflow and depending on the height of the affected node's sibling, either a simple BiNode pulldown or a node merge is used. To illustrate under what conditions each of these three cases is used, we show the complete deletion algorithm in Listing 2.

3.3 Upper Bound for Space Consumption

In the following, we provide an upper bound for HOT's memory consumption per key. We focus on HOT's high-level structure and assume that individual node layouts provide an upper space bound for storing a single entry inside a compound node. We denote this node level upper space bound for storing a single entry as e.

This upper bound e allows us to focus on HOT's high-level structure that is defined as a tree over compound nodes. By doing so, we can omit the internal structure of compound nodes, but

```
delete(hot, key):
   n = traverse hot for key
   m = traverse n until leaf entry
   if (getKey(m) == key):
      n̂ = delete(n, key)
      handleUnderflow(n̂,m,s)

handleUndeflow(n,m,s):
   s = determineSiblingNode(n)
   if(height(n) < height(s) or (count(n) + count(s)) > max_fanout):
      # no underflow handling necessary
      return

   p = parentNode(n)
   pBi = determineParentBiNode(p, key)
   if (height(n) > height(m)):
      # simple BiNode pulldown
      n̂ = pulldownBiNode(n, pBi)
   else: # (height(n) == height(m))
      # node merge
      n̂ = mergeNodes(n, s, pBi)
   p̂ = removeBiNode(p, pBi)
   handleUnderflow(p̂)
```

Listing 2. Structure-adapting deletion algorithm.

rather interpret the compound nodes as lists of entries sorted by their keys. In this model, each entry has size e, and we distinguish two types of entries: Leaf entries store the actual values, while boundary entries represent pointers to other compound nodes.

Based on this abstraction, we can determine an upper bound for the space required to store a single key, by determining how many boundary entries are stored in relation to leaf entries, in the worst case. We denote this upper space bound to store a single key as sp in the remainder. To derive the upper space bound for a single entry, we construct a scenario where the number of boundary nodes in relation to the number of leaf nodes is maximal. We assume that a HOT structure with a maximum fanout of $k = 2$ and its internal nodes only containing boundary entries represent such a scenario (Figure 5).

As full binary trees contain more leaf nodes than inner nodes, we can trivially infer that the number of boundary entries is less than the number of leaf entries. From this, we can deduce that the upper bound for the space required per key in a HOT structure with a maximum fanout of $k = 2$ is $sp < 4e$.

Next, we provide a proof that the same upper space bound of $sp < 4e$ holds for all HOT structures regardless of the chosen maximum fanout k. The proof is based on the hypothesis that the structure's underlying binary Patricia trie contains more leaf entries than boundary entries.

In the base case of a single leaf node that only contains leaf entries, this assumption holds trivially. For the step case, we have to consider each of the four cases of HOT's insertion algorithm:

- *Normal Insert*: This case only adds a single leaf entry, the proof assumption trivially holds.
- *Leaf Node Pushdown*: As Leaf node pushdown inserts exactly one leaf and one boundary entry, the proof assumption holds as well.

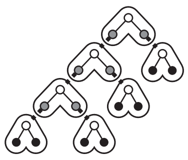


Fig. 5. HOT structure with a maximum fanout of $k = 2$.

- *Parent Pull Up, Intermediate Node Creation*: In the case of parent pull up and intermediate node creation, we have to distinguish two cases. First, only a single boundary node is created. As a new leaf entry is created at the same time the assumption holds. Second, two new boundary nodes are created. This assumes that each of these boundary nodes points to a compound node with at least two entries. Each of these entries is either a leaf node or an internal node having two subtrees. According to the induction hypothesis, each of these two subtrees contains more leaf than boundary entries. Hence, the subtree rooted at the BiNode connected to these newly created boundary nodes also contains more leaf than boundary entries.

After presenting the upper space point to store a single key in HOT to be $sp < 4e$, we will use this finding in Section 7 to provide an upper space bound for each of the evaluated HOT structures. For this purpose, we parametrize the high-level upper bound estimate, with the node layout specific upper space bounds for entries in these node layouts (cf. Section 5) and implementation-specific parameters like the used encodings or the maximum key length.

4 THEORETICAL PROPERTIES OF HEIGHT OPTIMIZED TRIES

HOT is a pure trie structure, i.e., every node represents a prefix of the key. Nevertheless, it shares similarities with comparison-based multi-way structures that perform $log_2(n)$ key comparisons. Like in B-trees, HOT bounds the maximum fanout of each node, 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. While the theoretical properties in terms of tree height and access performance for B-trees are well known, these types of theoretical properties are traditionally not available for trie structures. In this section, we first introduce and then prove three interesting theoretical properties of HOT:

(I) *Determinism*: For the same set of keys, regardless of their insertion order, the resulting HOT will have the same structure and the same set of compound nodes.
(II) *Minimum height*: For a given set of keys and a maximum fanout k, no set of compound nodes can be found such that that the height of the resulting tree is lower than the height of a HOT with the same maximum fanout k and the same set of keys.
(III) *Recursive structure*: Each subtree of a HOT structure is itself a HOT structure. Thus, in combination with property (II), this implies that each subtree of a HOT structure has also minimal height.

4.1 Proof Idea and Challenges

Before we formally prove the three properties, let us briefly describe the proof idea and challenges. The key idea of the proof is that the implicit partitioning created by HOT's compound nodes over the underlying binary Patricia trie is equivalent to the minimum height partitioning as produced by the tree partitioning algorithm of Kovács and Kis [23]. To prove this equivalence, we need to define a mapping between the HOT compound nodes and the partitions created by the algorithm of Kovács and Kis. To this end, our proof has to consider the following two aspects: (i) hoisted nodes in HOT represent individual partitions in the minimum height partitioning algorithm by Kovács, (ii) we need to parameterize the algorithm by Kovács and Kis with a weight function and a so-called Knapsack constraint, such that the partitions are size constraint with a predefined maximum fanout constraint. To address (i) the proof introduces a modification of HOT called sHOT, which does not collapse single entry compound nodes into their boundary nodes but keeps them separate. This in itself poses the challenge of coming up with an alternative incremental insertion algorithm and a subsequent proof that the resulting sHOT and HOT structures are equivalent and a bijective transformation between sHOT and HOT exists. We will address this later. To address (ii) the proof exploits that each binary Patricia trie is a full binary tree and that each full binary tree with n leaf nodes has $n-1$ inner nodes. From this, we derive the parametrization of the algorithm of Kovács and Kis, which we denote as *Static Minimum Height Partitioning*.

Before we can prove the equivalence of SMHP and sHOT, we need to ensure that SMHP provides the three postulated HOT properties. While the minimum height has already been proved by Kovács and Kis [23], the recursive nature and determinism need to be inferred by us. The recursive nature of the algorithm can be trivially inferred from SMHP as the algorithm works bottom-up and partitions higher up in the tree do not impact partitions closer to the leaf level. That SMHP deterministically leads to identical partitioning for the same set of keys, can be inferred from the properties of the underlying binary Patrica trie that regardless of the insertion order lead to the same binary tree structure.

The next part of the proof is to show that SMHP and sHOT create equivalent partitionings for the same underlying structure. The challenge in this part of the proof is that SMHP is a static algorithm that takes a statically defined tree as an input and creates the optimal partitioning as an output. However, HOT and sHOT are incremental algorithms that iteratively insert new values into an existing tree structure over compound nodes. To prove that the static approach SMHP and the incremental insertion algorithm of sHOT indeed create an equivalent partitioning, we have to apply induction on the number of entries. Thereby, we need to prove that inserting a new value into a HOT structure is equivalent to an SMHP structure over the same underlying data. To do so, we exploit that according to the induction hypothesis an SMHP structure exists that is equivalent to the sHOT structure before the insertion. The actual proof is then to show that each of the four insertion cases leads to a partitioning that is equivalent to an SMHP over the same underlying data. The trick here is to map each of the four insertion cases to a local modification on the underlying binary Patricia trie and to use this local modification to infer the SMHP structure after the insertion.

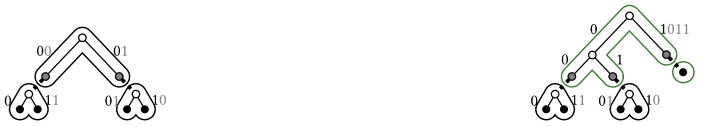
The final step is to prove that the same properties apply also to HOT. The technique that we apply for this is to (i) show that we can transform each sHOT into an equivalent HOT structure that preserves the proven properties and (ii) that inserting a value into a HOT structure is identical to first inserting the same value into an equivalent sHOT structure and then transforming it into a HOT structure. The key idea for (i) is to show that each single entry node always has a sibling node and hoisting single entry nodes do not affect the fanout of its parent or the sibling node. As neither the parent node nor its sibling is affected the tree height and the remaining compound nodes are not affected. For (ii) we again apply induction over the number of elements for all four cases of the insertion algorithm.

After presenting the proof idea, we show the actual proof in the following subsections:

(1) In Section 4.2, we introduce a simplified variant of HOT called *sHOT*, which by omitting the hoisting of leaf nodes has a similar structure as partitionings generated by the algorithm of Kovács and Kis [23].

$$\begin{tikzcd}\boxed{\text{SMHP}} \arrow[r, "\text{add bounding nodes}"] & \boxed{\text{sHOT}} \arrow[r, "\text{hoist leaf entries}"] & \boxed{\text{HOT}}\end{tikzcd}$$

Fig. 6. Relationship between SMHP, sHOT, and HOT and how to transform partitionings created by SMHP into HOT structures.



(a) Initial HOT before inserting 1011. (b) *Intermediate insert* of 1011.

Fig. 7. Example of an intermediate insert of the key 1011 into an sHOT with a maximum fanout of $k = 3$.

(2) Then, Section 4.3 introduces a parameterized version of the minimum height partitioning algorithm by Kovács and Kis [23] that is called *Static Minimum Height Partitioning (SMHP).* For each partition it satisfies a predefined maximum fanout constraint k.
(3) Next, in Section 4.4, we prove that SMHP is a deterministic algorithm and the created partitionings are of recursive structure.
(4) We then leverage these properties of SMHP to prove in Section 4.5 that the partitionings created by sHOT are equivalent to the ones created by SMHP.
(5) Finally, in Section 4.6, we prove that, except for the hoisting of leaf nodes, sHOT and HOT are equivalent and share the three HOT-properties.

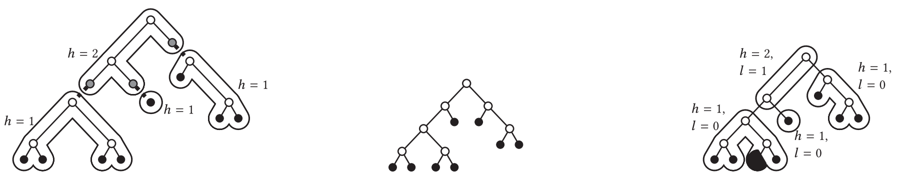
The flow of the proof and the relationship between SMHP, sHOT, and HOT is shown in Figure 6.

4.2 Simplified HOT (sHOT)

We introduce a simplified version of HOT called *sHOT* that stores leaf entries in leaf nodes only. This restriction has implications on the insertion algorithm and therefore, sHOT's insertion algorithm differs from the insertion algorithm of HOT that is presented in Section 3.2. Both algorithms feature a "normal" and three "special" cases. Because sHOT does not contain any leaf entries in non-leaf nodes, sHOT does not support the special case of leaf node pushdown. Instead, it introduces *intermediate insert*, which is applicable whenever the mismatching BiNode is contained in an intermediate node. An example for an intermediate insert is show in Figure 7. In addition to *intermediate insert*, sHOT uses modified versions of the two overflow handling strategies parent pull up and intermediate node creation that ensure that intermediate nodes will never contain leaf entries. In contrast, for both overflow handling strategies sHOT plainly splits overflown nodes without hoisting single entry nodes into their parent nodes. Thus, single entry nodes are created whenever one of the overflown node's left or right subtree contains only a single entry.

4.3 Static Minimum Height Partitioning

The algorithm by Kovács and Kis [23] partitions a weighted tree structure such that for each partition the intrapartition weight of the vertexes contained in a single partition satisfies a predefined so-called "knapsack" constraint W and the induced tree defined over the resulting partitioning is of minimum height (by Lemma 1 from Reference [23]). Their algorithm assigns each vertex v a level $l(v)$ and an intrapartition weight $rw(v)$ such that connected subgraphs consisting of nodes with the same level form individual partitions. Initially, each leaf vertex v_l is labeled with level $l(v_l) = 0$ and its intrapartition weight $rw(v_l)$ is equal to its weight $w(v_l)$. For the definition of the labeling functions $l(v)$ and $rw(v)$ the algorithm defines the helper function $cl_{max}(v) = max_{u \in child(v)} l(u)$,



(a) sHOT (b) Binary Patricia trie (c) SMHP

Fig. 8. Two equivalent representations of the same underlying binary Patricia trie. Both representations, sHOT as well as SMHP, apply a maximum fanout constraint. The sHOT representation restricts the maximum fanout to three, whereas the SMHP restricts the maximum intrapartition weight of its vertices to two. In both representations the partitions/compound nodes are annotated with the height of the corresponding subtree. Additionally, each partition generated by SMHP is labeled with the corresponding level of the partition. The grey boundary entries, which link individual sHOT nodes together, are missing in the case of SMHP.

which for a given vertex v returns the maximum level of any of its child nodes. Additionally, we define the helper function $cmax(v) = \{v \in child(v) | l(v) = cl_{max}(v)\}$, which returns the set of all child nodes of a vertex v having the same level as $cl_{max}(v)$. The level $l(v)$ of a non-leaf vertex v depends on the intrapartition weight and levels of its children and is defined as follows:

$$l(v) = \begin{cases} cl_{max}(v) & \text{if } w(v) + \sum_{u \in cmax(v)} rw(u) \leq W, \\ cl_{max}(v) + 1 & \text{else.} \end{cases}$$

For all non-leaf vertices v, the intrapartition weight $rw(v)$ is the sum of the weights of all its descendant vertices within the same partition. More formally the intrapartition weight $rw(v)$ of a single non-leaf vertex v is defined as follows:

$$rw(v) = w(v) + \sum_{u \in cmax(v) \wedge l(v) = l(u)} rw(u).$$

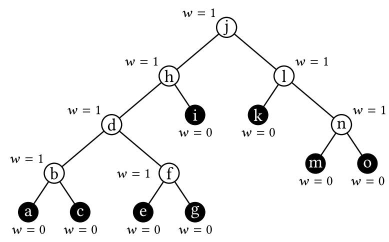
Our weight function $w(v)$ assigns the weight zero to all leaf entries and one to each BiNode. This weight function implies that only the weights of BiNodes are taken into account:

$$w(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf,} \\ 1 & \text{otherwise.} \end{cases}$$

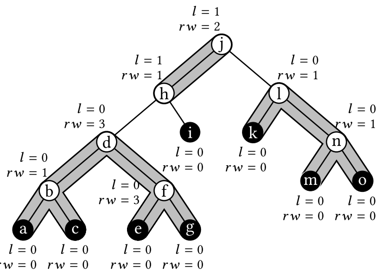
We denote the constraint, which is called the knapsack constraint in the original paper by Kovács and Kis [23], the *maximum intrapartition weight* constraint. We set this constraint to equal $k - 1$, where k is the maximum node fanout of an equivalent sHOT.

Figure 9 shows an example of an SMHP partitioning. The underlying binary Patricia trie is depicted in Figure 9(a) (on the left), while the final partitioning is shown in Figure 9(b) (on the right). Nodes in the underlying binary Patricia trie are labeled with their weights and nodes in the final partitioning are labeled with the derived intrapartition weights and node levels.

According to the SMHP algorithm the leaf nodes in the example have a weight of 0 and internal nodes a weight of 1. The figure also shows that internal nodes can only form a partition with their child nodes if the sum of their weights and their child's intrapartition weights is less than the *maximum intrapartition weight* constraint of three. For instance, node b trivially gets assigned a level $l(b) = 0$ and $rw(1)$ accordingly as $w(h) + rw(d) + (i) = 1$, and therefore, these nodes are part of the same partition (a, b, c, d, e). On the contrary, node h gets assigned a level of $l(h) = cl_{max}(h) + 1 = 1$ as $w(h) + rw(d) + (i) = 4)$ would violate the *maximum intrapartition weight* constraint of 3. Hence, h does not extend the partition of its child node d but is the leaf node of a new partition (h, j).



(a) A binary Patricia trie with assigned node weights. Leaf nodes have a weight of $w = 0$ and non-leaf nodes have a weight of $w = 1$.



(b) An SMHP representation of the binary Patricia trie shown in 9a. Each node is labeled with its intra partition weight and level.

Fig. 9. An example showing the construction for an SMHP for the same binary Patricia trie as shown in Figure 8. The *maximum intrapartition weight* constraint $W = 3$ is chosen to limit the maximum fanout to 4. To depict how the algorithm works, each node is annotated with its weight, the derived intrapartition weights and the levels used to form the individual partitions.

In the remainder, we call the combination of the minimum height partitioning algorithm by Kovács and Kis, our custom weight function w and the maximum intrapartition weight constraint $k - 1$ ***static minimum height partitioning*** **(SMHP)**. Even though sHOT as well as SMHP partition binary Patricia tries, both algorithms use different representations for the actual partitions. Figures 8(a)–8(c) illustrate the differences.

In the context of sHOT, individual compound nodes represent individual partitions. While both sHOT as well as SMHP assign each vertex to a partition, sHOT adds boundary entries to link individual partitions together. Additionally, Kovács and Kis [23] originally defined the height of individual partitions to be equal to their level l starting with level 0 assigned to leaf partitions. In contrast, the height of leaf nodes in sHOT is 1. To compare SMHP and sHOT, we (i) ignore the boundary entries of sHOT and (ii) redefine the height of an SMHP to be $height(p) = l(p) + 1$.

4.4 Properties of Static Minimum Height Partitionings

Before we actually prove the properties of sHOT, we first show that the function $rw(v)$ calculating the intrapartition weight of individual vertices is equal to counting the number of descendant BiNodes contained in the same partition. We then use this finding to show that each partition generated by SMHP satisfies the same maximum fanout k as nodes in sHOT. Afterwards, we show that SMHP deterministically generates the same partitioning for the same keys and that the structure of an SMHP is recursively defined.

To prove that the intrapartition weight $rw(v)$ of individual vertices is equal to counting the number of descendant BiNodes in the same partition, we apply induction on the number of BiNodes contained in a partition p with root vertex r. For a subtree containing $n + 1$ entries, we assume in the induction step that the left subtree contains o BiNodes and the right subtree contains k BiNode. So in total both subtrees contain $o + k = n$. By the definition of the intrapartition weight, the intrapartition weight of the root node is $rw(r) = o + k + 1 = n + 1$, which is exactly the number of BiNodes in the considered partition.

As the maximum value of the intrapartition weight function for a single partition is limited by $k - 1$, we have shown that each partition generated by SMHP contains at most $k - 1$ BiNodes, which is equivalent to satisfying the maximum fanout constraint of sHOT, namely, a single partition contains at most k boundary or leaf nodes.

Next showing that SMHP is deterministically defined and has a recursive structure is trivial. As the algorithm of SMHP is deterministic and the structure of a binary Patricia trie for a given set of keys is independent of the actual insertion order, SMHP over a binary Patricia trie and a given set of keys is also independent of the actual insertion order and therefore, deterministically defined. As the intrapartition weights and levels of vertices only depend on the weights of its descendants, we can infer that each subtree of the induced tree over a partitioning created by SMHP is identical to an SMHP over the same underlying vertices.

4.5 Equivalence of Static Minimum Height Partitioned Patricia Tries and sHOT

We prove that, when applied on the same set of keys, sHOT and SMHP result in an equivalent structure. Hence, we define that a compound node generated by sHOT is equivalent to a partition generated by SMHP if and only if all vertices of the original binary Patricia trie that are contained in the respective partition are also contained in the corresponding compound node. Based on this definition a static minimum height partitioning P and a corresponding sHOT H are equivalent if for all partitions $p \in P$ a compound node $n \in H$ exists, which is equivalent to p and for all nodes $n \in H$ a partition $p \in P$ exists, which is equivalent to n. This implies that whenever a partitioning P and an sHOT H are equivalent, the height of the induced subtrees over the partition/nodes is also the same. In Figure 8, we depict this notion of equivalence by showing a binary Patricia trie, the corresponding HOT and an equivalent static minimum height partitioned tree.

For the actual proof that an sHOT for a set of keys K is equivalent to an SMHP structure defined over a binary Patricia trie for the same set of keys K, we apply induction on the number of keys n. For trees that have at most k entries both SMHP as well as sHOT trivially construct a single entity containing all k entries. SMHP denotes this entity as partition whereas sHOT calls it a compound node.

For the induction step, i.e., trees that contain more than k entries, we prove each case of sHOT's insertion algorithm separately. We exploit that the structure of a binary Patricia trie for the same set of keys is deterministically defined and inserting a new key into an existing binary Patricia trie is only a *local modification* that inserts two new vertices. According to the definitions introduced in Section 3.1, these two vertices are the *discriminating BiNode* (bi_{dis}) and the leaf vertex corresponding to the new key (v_{new}).

To determine whether sHOT's insertion algorithm results in the same partitioning as inserting the same key x into the original underlying binary Patricia trie and then creating an SMHP, we exploit the properties of the local modification in the following way. First, for each of sHOT's insertion operation types, we determine the affected partition of an SMHP, which is equivalent to the corresponding sHOT before the insertion of x. Next, we exploit the effects of the local modification to infer the SMHP after the insertion. Finally, we check whether the resulting partitioning is equivalent to the sHOT structure after the insertion of x.

Before we address the individual cases of the insertion algorithm, we observe that for both SMHP and sHOT the placement of the v_{new} does by definition neither affect the intrapartion weight nor the level or node assignment of other BiNodes. Hence, we only have to consider the placement of the *discriminating BiNode* in the remainder of the proof.

- **Normal Insert and Intermediate Insert:** According to the induction hypothesis none of the vertices v previously contained in the affected partition has an intrapartition weight of $rw(v) > k - 2$. Therefore, by applying the local modification on the underlying binary Patricia trie neither vertices that are previously contained in the affected partition nor the new *discriminating BiNode* receive an intra partition weight larger than $k - 1$. As no other partitions are affected by this local modification the resulting structure is identical to an sHOT structure after applying the corresponding normal insert operation.
- **Intermediate Node Creation:** An intermediate node creation for sHOT (cf. Section 4.2) is applicable whenever the affected node n contains k entries, and the height of its parent

node is $h(n_{parent}) \geq h(n_{aff}) + 1$. According to the induction hypothesis, we infer that the intrapartition weight of the affected partition is $rw(p_{aff}) = k - 1$ and that the level of the affected partition's parent is $l(p_{par}) > l(p_{aff}) + 1$. We further infer that the level of the affected partition's sibling is $level(p_{sib}) = level(p_{par}) - 1 > l(p_{aff})$. Hence, after the local modification either the previous partition root or the new mismatching BiNode would have an intrapartition of k. By the definition of SMHP it will therefore form a new partition with a level $l(newPartition) = l(p_{aff}) + 1$ and an intrapartition weight $rw(newPartition) = 1$. However, it will not affect any other partition and therefore the resulting SMHP is equivalent to an sHOT after applying the equivalent intermediate node creation.

- **Parent Pull Up:** Parent pull up for sHOT is applied whenever the affected node contains k entries (cf. Section 3.2), and the height of the parent node is $h(n_{parent}) = h(n_{aff})+1$. According to the induction hypothesis, we infer that the intrapartition weight of the affected partition is $rw(p_{aff}) = k - 1$ and the level of its parent partition is $l(p_{par}) = l(p_{aff}) + 1$.

  Analogously to intermediate node creation either the previous root of the affected partition or the new mismatching BiNode would become the new root of the affected Partition. As this potential new root would have an intrapartition weight of k and therefore violate the knapsack constraint, it gets an intrapartition weight of 1 and a level of $l(p_{aff}) + 1$ assigned and therefore becomes a part of the previous parent partition. However, this affects the intrapartition weights of the parent partition. If the parent partition previously contained less than $k - 1$ BiNodes, then the knapsack constraint is satisfied for all vertices in the previous parent partition. If this is not the case, then the parent pull up affects the grandparent partition. This can potentially ripple up the tree until the root partition is reached and a new root partition is formed. Again this behavior is identical with the definition of sHOT and therefore the resulting sHOT is equivalent with an SMHP over the same set of keys.

By proving that the sHOT structure is always equivalent to an SMHP over the same set of keys, we have also shown that the properties proven for SMHP also hold for sHOT:

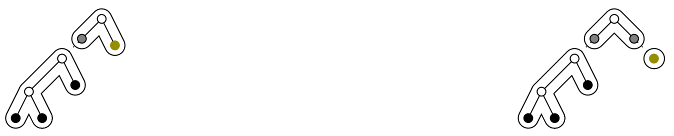
- The tree is of *minimum height*.
- The algorithm produces a *deterministic* tree *independent of the insertion order* of the keys.
- The generated node structure is *recursive*, such that the height of each node is minimal.

4.6 Equivalence of sHOT and HOT

In Section 4.5, we have shown that sHOT produces an equivalent partitioning as the minimum height partitioning algorithm by Kovács et al. [23]. Now, we show that all properties that hold for sHOT hold for HOT as well. To do this, we first show that a deterministic transformation between sHOT and HOT exists that preserves the properties previously shown for sHOT. Then, we show that directly inserting a value into HOT is equivalent to first inserting the same value into an equivalent sHOT and then transforming the resulting sHOT into HOT.

To show that a deterministic transformation between sHOT and HOT exists, we first introduce the *minimum cardinality constraint*. It ensures that for both sHOT and HOT each BiNode bi_i with $h(b_i) > 1$ has enough descending BiNodes such that its descendants cannot be merged without violating the maximum fanout constraint. This constraint trivially holds, as BiNodes of $h(b_i) > 1$ are only generated by parent pull up operations. By design parent pull up operations are only applicable if the number of entries in a single node would otherwise exceed the maximum fanout constraint k. This is also not affected by intermediate insert in the case of sHOT, as the mismatching BiNode also adheres to the minimum cardinality constraint.

To transform an sHOT structure into a HOT structure, it is sufficient to replace all boundary nodes that are connected to single entry leaf nodes with the leaf entry itself. An example of this transformation is depicted in Figure 10.



(a) HOT stores leaf entries in the same node as its parent discriminating BiNode

(b) sHOT stores leaf entries in leaf nodes only

Fig. 10. A HOT and an sHOT representation for the same binary Patricia trie and a maximum fanout of $k = 3$. Differences between both representations are highlighted in green.

As this replacement does neither affect the fanout nor the BiNode assignment of the leaf entry's previous parent node, the minimum cardinality constraint still holds and the overall tree height is not affected. Hence, the tree height of the resulting HOT is equal to the height of the original sHOT.

To finally show the equivalence between HOT and sHOT, we have to show that inserting a new key into an existing HOT is identical to first inserting the key into an equivalent sHOT and then transforming it into a HOT by hoisting all single element leaf nodes into their parent nodes. The proof is again done by induction on the number of keys n. In the base case up to k entries both structures consist of a single node and therefore are trivially equal.

In the step case, we again have to consider multiple cases:

- Inserting a new key into HOT that neither affects nor creates a hoisted leaf entry. In this case the resulting HOT and sHOT are again trivially equivalent.
- Inserting a new key into HOT that results in a new hoisted entry. This only happens as part of node splits while handling overflows using parent pull up or intermediate node creation. In contrast, in the equivalent case of sHOT the single element subtree is not hoisted but instead an explicit single entry node is created. If the resulting sHOT is transformed into a HOT structure, then the newly created single entry node is converted into the same hoisted node as directly created during the overflow handling in the equivalent HOT structure. As no other boundary entries or single entry leaf nodes are affected the resulting HOT and sHOT structures are equivalent.
- Inserting a new key into HOT that modifies a hoisted entry. This happens if a leaf node push down converts the previously hoisted entry into a new leaf node of cardinality two. Inserting the same key into an equivalent sHOT structures converts the single entry node that previously corresponded to the hoisted entry of the HOT structure into the same two value node. Since no other boundary entries or single entry leaf nodes are affected in this case either, the resulting HOT and sHOT structures are equivalent again.

By proving the equivalence of HOT and sHOT, we have also proven the equivalence of HOT and SMHP. Thus, we have proven the three core properties of HOT:

(I) **Determinism:** For the same set of keys, regardless of the insertion order, the compound node structure generated by HOT is always the same.

(II) **Minimum Height:** No partitioning of a binary Patricia trie under a maximum fanout constraint k can be found such that the height of the induced tree structure is lower.

(III) **Recursive Structure:** Each subtree of HOT is identical to a HOT structure that only contains those keys, which are contained in the respective subtree.

5 NODE IMPLEMENTATION

The Height Optimized Trie introduced in Section 3 is a trie-based index structure that uses a novel algorithm to minimize the overall tree height of binary Patricia trie structures by combining

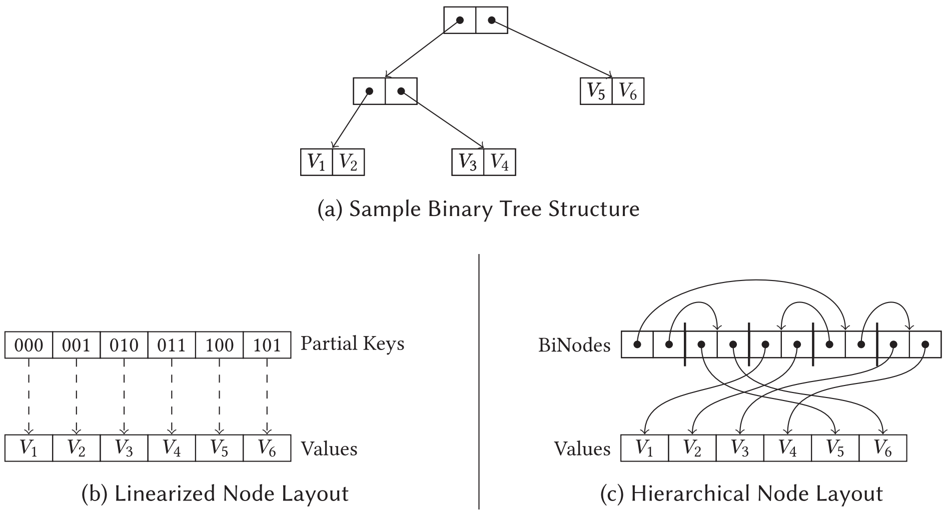

V_5 V_6
V_1 V_2
V_3 V_4
(a) Sample Binary Tree Structure
000 001 010 011 100 101
Partial Keys
V_1 V_2 V_3 V_4 V_5 V_6
Values
(b) Linearized Node Layout
BiNodes
Values
V_1 V_2 V_3 V_4 V_5 V_6
(c) Hierarchical Node Layout

Fig. 11. An example of a linearized and a hierarchical node layout for the same binary tree structure.

multiple BiNodes into k-constrained compound nodes. In Section 4, we showed that the achieved tree height is optimal with regards to a previously chosen constraint k. However, we have not discussed the actual physical layout of these compound nodes in memory. To benefit from a Height Optimized Trie, the use of a fast and space-efficient compound node layout is crucial. In this section, we therefore introduce different memory layouts for k-constraint tries.

In this article, we particularly focus on node layouts geared toward the use in main memory database systems. 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.

An important aspect when designing structures for main memory systems is to improve cache locality and to restrict the number of cache misses. A well-known technique that improves cache locality is to separate keys and pointers in nodes [4, 14, 25]. We apply this technique to all our memory layouts by dividing a node's memory into two contiguous chunks of memory. The first part contains the actual structure of the k-constrained trie used for searching and the second part consists of pointers to child nodes or leaf entries in key order. We denote the first part of each node as the *structural area* and the latter as the *data area*. Accordingly, the search operations for all node types consist of two stages. First, the index of the entry in the data area is determined by searching the structural area. Second, the corresponding entry in the data area is fetched.

As this overall scheme applies to all our node designs, we focus on the different structural representations. We classify our node designs according to their search operations into two categories. Nodes in the first category are searched using linear search or equivalent data-parallel SIMD operations and are geared toward high access performance. Nodes in the second category are searched by traversing a hierarchical structure and are geared toward high space efficiency. Hence, we denote node layouts falling into the first category as *linearized node layouts* and node layouts falling into the second category as *hierarchical node layouts*. Figure 11 illustrates the difference between a hierarchical and a linearized node layout for a sample binary tree structure.

Although the various hierarchical and linear node layouts use different approaches to represent a k-constrained binary Patricia trie, they share two design decisions. All our node layouts are designed to be as space-efficient as possible. Hence, instead of reserving spare memory for in-place updates, we use a copy-on-write approach for updates. Besides saving space, applying copy-on-write techniques allows for concurrent operations (cf. Section 6). As we target contemporary 64-bit hardware architectures, all node layouts store 64-bit values. These values are either pointers to other nodes or tuple identifiers. We use the most significant bit to distinguish between a pointer and tuple identifiers. In the case that stored values require less than 64-bits, we embed the value directly in the tuple identifier.

In the remainder of this section, we first introduce the linearized node layout and its SIMD-based optimizations. Second, we introduce the hierarchical node layout and different variations thereof.

We will conduct an extensive evaluation of the individual node layouts with regards to access performance and memory consumption in Section 7.2.

5.1 Linearized Node Layout: Access Optimized

In principle, one could organize each HOT node (i.e., each k-constrained binary Patricia trie) as a pointer-based hierarchical binary trie structure. Although—as we will see in the next Section 5.2—such hierarchical structures can support highest space efficiency, traversing these hierarchical structures leads to insufficient access performance, due to control and data dependencies and also to the raw number of instructions required.

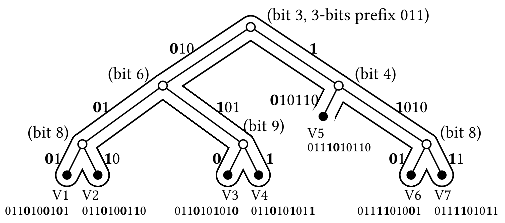
However, to make HOT both space-efficient *and* fast, a compact representation that can be searched quickly is required. This is addressed by our linearized node layouts. The key idea of these layouts is to convert a k-constrained trie to a compact bit string that can be searched in parallel using SIMD instructions.

Before we dive into the actual physical layouts, we will first describe our technique that we use to convert binary Patricia tries into compact linearized bit strings. Next, we present how the instruction sets of modern hardware can be leveraged to search such a linearized representation efficiently. Finally, we present two different variations of such hardware accelerated linearized node layouts with the goal to further improve performance and memory consumption.

5.1.1 Linearizing Binary Patricia Tries. The challenge of linearizing binary Patricia tries lies in creating an implicit representation of the trie structure, which can be searched by linear scan operations or equivalent data-parallel operations. Our approach to creating such an implicit representation is to store the path information of each key in the original binary Patricia trie.

As each BiNode represents the position of a single discriminative bit, we can represent the path of a single key by extracting its discriminative bits along the path and storing them consecutively. To create a uniform representation of these extracted partial keys, we extract all the discriminative bits of the same binary Patricia trie structure and not only the bits along the key's path. We call this specific definition of partial keys *dense*. To interpret the extracted partial keys, we also need to store the positions of the used discriminative bits. Consider, for example, the trie shown in Figure 12(a), which consists of seven keys and has the discriminative bit positions $\{3, 4, 6, 8, 9\}$. The five discriminative bits for each key form the *partial keys (dense)* and are shown in Tab. 12(b). Searching this implicit binary Patricia trie represented by its dense partial keys is straight-forward. First, we extract the discriminative bits of the search key. Second, we find the partial key that is equal to the search key's partial key.

However, using dense partial keys to represent a binary Patricia trie has a major caveat. 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 12(a) 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, we would have



(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)

Fig. 12. Illustration of linearizing a binary Patricia trie.

```
Leaf* search(uint8_t const * key) const {
  uint densePartialKey = extractBits(discriminativeBitPostions, key);
  uint leafCandidateIndex = 0;
  // start at index one because the first sparse partial key is always trivially zero
  for(uint i=1; i < numberOfKeys; ++i) {
    // check if the sparse partial key is a potential match candidate
    if(densePartialKey & sparsePartialKey[i] == sparsePartialKey[i]) {
      leafCandidateIndex = i;
    }
  }
  return leafs[leafCandidateIndex]
}
```

Listing 3. Lookup in a linearized binary Patricia trie.

to determine bit 7 for the existing partial keys by loading the corresponding actual keys, which would slow down insertion. To overcome this shortcoming, we use a slightly modified partial key representation, which we call *sparse partial keys*. The difference to dense partial keys is that for sparse partial keys, only those discriminative bits that correspond to inner BiNodes along the path from the root BiNode are extracted and all other bits are set to 0. Thus, sparse partial key bits set to 0 are intentionally left undefined. In the case of a deletion, this allows us to remove unused discriminative bits. To illustrate the difference between dense and sparse partial keys, we show both in Figure 12(b) for the trie in Figure 12(a).

As zero bits in sparse partial keys have ambiguous semantics, we have to adapt the search algorithm. For instance, key 0111010110 shown in row 5 of Figure 12(b) yielding the dense partial key 10010 does not match its corresponding sparse partial key for the binary Patricia trie depicted in Figure 12(a). To solve this issue, we developed the search algorithm shown in Listing 3. First, the algorithm extracts the dense partial key corresponding to the provided search key. Next, the algorithm checks whether the bits set to one in the sparse partial key also have the value one in the dense partial key. Finally, we select the largest sparse partial key where only these bits are set. While inserting a new key into a normal binary Patricia trie structure is straightforward, inserting a new key into a linearized binary Patricia trie operating solely on the linearized representation requires a novel, more sophisticated algorithm. Hence, we describe how such an insertion operation works in the following. 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, then 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 a linearized binary Patricia tries. Instead, we directly determine all

```
int searchPartialKeys8(Node* node, uint32_t densePartialSearchKey) {
  // loading all sparse partial keys
  __m256i sparsePKeys = _mm256_loadu_si256(node->partialKeys8);
  // broacasting the search key into a SIMD register
  __m256i key = _mm256_set1_epi8(densePartialSearchKey);
  // determine all bits of the partial search keys
  // that are set to one in the dense partial key
  __m256i selBits = _mm256_and_si256(sparsePKeys, key);
  // determine all sparse partial keys that have only those bits set
  __m256i complyKeys = _mm256_cmpeq_epi8(selBits, sparsePKeys);
  // create a bit mask of all matching sparse partial keys
  uint32_t complyingMask = _mm256_movemask_epi8(complyKeys);
  // determine the index of the largest matching key
  return bit_scan_reverse(complyingMask & node->usedKeysMask);
}
```

Listing 4. Searching multiple sparse partial keys in parallel using SIMD.

leaf entries contained in the subtree of the mismatching BiNode and denote these entries as the *affected entries*. To do so, we 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, then all sparse partial keys have to be recoded to create partial keys containing also the mismatching bit position. 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.

5.1.2 Fixed-sized Linearized Node Layout. So far, searching a linearized node layout requires linear search. In this section, we show that by using SIMD operations, we can achieve constant inter-node search behavior. In addition, we show that also the extraction of partial keys can be improved by using bit manipulation instruction set extensions of modern hardware. As all partial keys in the resulting node layout have a fixed length, we call it ***Fixed-sized Linearized node Layout (FLL)***.

The key idea to improve over linear search in the FLL is to exploit the data-parallel instructions of modern CPU architectures to search all of a node's sparse partial keys in parallel. Given an extracted dense partial key, the SIMD-based algorithm works as follows. First, the dense partial key is broadcast to all slots of a single SIMD register. Second, we load multiple sparse partial keys into a SIMD register. Third, we determine all sparse partial keys that only match the dense partial key's discriminative bits in parallel. Next, we store the result of the match operation in a bit mask. Each bit of the bit mask corresponds to one of the node's sparse partial keys and each set bit represents a matching partial key. Finally, determining the index of the sparse partial key is done by using a single bit scan reverse CPU instruction. The pseudo-code for searching 8-bit sparse partial keys is shown in Listing 4.

Until now, we have not detailed how a search key's discriminative bits are actually stored and how partial search keys are extracted. The most obvious way is to store the set of a node's discriminative bits in an array, as shown in Figure 13(I). The downside of this approach is that it slows

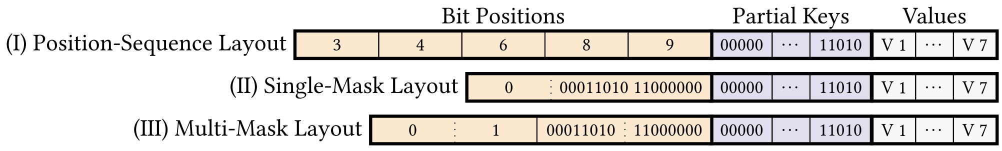


Fig. 13. Three different variations of a node representing a linearized version of the trie shown in Figure 12. All representations consist of three parts: the bit positions, the partial keys, and the values (tuple identifiers or node pointers). We show three representations for bit positions: (I) stores them naively as a sequence of positions, (II) uses a single bit mask, and (III) uses multiple bit masks.

down access performance: Before the actual data-parallel key comparison can be performed, we 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 and is therefore critical for performance.

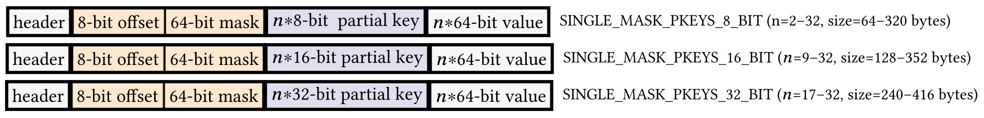
To speed up key extraction, we propose two layouts that utilize the *PEXT* instruction of the BMI2 instruction set. The PEXT instruction extracts bits specified by a bit mask from a given integer. Thus, as shown in Figure 13(II), we represent the bit positions as a bit mask (and an initial byte position). This layout can be used whenever the lowest and highest bit positions are close to each other (less than 64 bit positions difference). In case the range of the discriminative bit positions is larger, the multi-mask layout can be used, which is illustrated in Figure 13(III). It breaks up the bit positions into multiple 8-bit masks, each of which is responsible for an arbitrary byte. Again, PEXT is used to efficiently extract the bits contained in multiple 8-bit key portions in parallel using this layout.

Besides speeding up the extraction of partial keys, FLL also uses the bit manipulation instruction set BMI2 to improve insert and deletion performance. While insertion operations use the PDEP instruction to recode partial keys when inserting a new discriminative bit, its counterpart, the PEXT instruction is used by deletion operation to recode partial keys when deleting a superfluous discriminative bit. For instance, to add bit position 7 to the sparse partial keys depicted in Figure 12(a), the `_pdep_u32(existingKey, 0b111011)`[2] instruction is executed for each key.

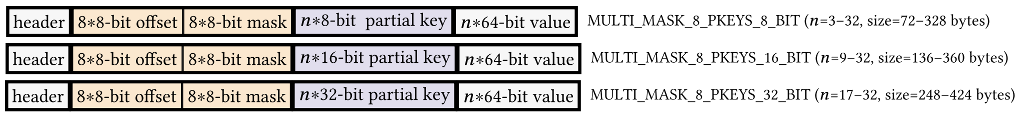
5.1.3 Adaptive Linearized Node Layout. Although the fixed-sized linearized node layout presented in the previous section exploits the instruction sets of modern hardware to provide constant search performance, it has one downside. Namely, to represent tries with a maximum fanout of k it always uses partial keys of length $k-1$ bits. However, depending on the dataset stored, only a small fraction of these k bits may be required to represent all of a node's discriminative bits. In the best case, where a node's keys form a single dense region only $log_2(k)$ bits are necessary to represent the node's sparse partial keys (cf. Section 5.2.3 for the definition of dense regions). We introduce the ***Adaptive Linearized node Layout (ALL)*** to take advantage of keysets that can be represented by less than $k-1$ discriminative bits. The ALL chooses the smallest data type per node capable of storing the node's sparse partial keys, instead of choosing the data type based on the worst-case partial key length. Besides reducing the memory footprint this optimization has two major advantages resulting in improved performance too. First, shorter partial keys require fewer memory accesses, and therefore, fewer cache misses. Second, using smaller data types increases parallelism, because more data items can be processed in a single SIMD instruction.

As the used hardware platform has a direct impact on the width of its SIMD registers and hence, on the number of partial keys that can be loaded in parallel, we have to decide on the available SIMD instruction sets in advance to design the actual SIMD-based index structure. Without loss

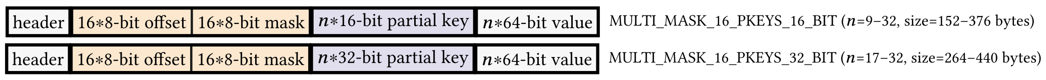
[2]https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-pdep-u32-64.


header | 8-bit offset | 64-bit mask | n∗8-bit partial key | n∗64-bit value
SINGLE_MASK_PKEYS_8_BIT (n=2–32, size=64–320 bytes)
header | 8-bit offset | 64-bit mask | n∗16-bit partial key | n∗64-bit value
SINGLE_MASK_PKEYS_16_BIT (n=9–32, size=128–352 bytes)
header | 8-bit offset | 64-bit mask | n∗32-bit partial key | n∗64-bit value
SINGLE_MASK_PKEYS_32_BIT (n=17–32, size=240–416 bytes)

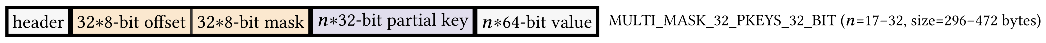
(a) Single-mask layout with a 1 64-bit mask and 8, 16, or 32-bit partial keys.


header | 8∗8-bit offset | 8∗8-bit mask | n∗8-bit partial key | n∗64-bit value
MULTI_MASK_8_PKEYS_8_BIT (n=3–32, size=72–328 bytes)
header | 8∗8-bit offset | 8∗8-bit mask | n∗16-bit partial key | n∗64-bit value
MULTI_MASK_8_PKEYS_16_BIT (n=9–32, size=136–360 bytes)
header | 8∗8-bit offset | 8∗8-bit mask | n∗32-bit partial key | n∗64-bit value
MULTI_MASK_8_PKEYS_32_BIT (n=17–32, size=248–424 bytes)

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


header | 16∗8-bit offset | 16∗8-bit mask | n∗16-bit partial key | n∗64-bit value
MULTI_MASK_16_PKEYS_16_BIT (n=9–32, size=152–376 bytes)
header | 16∗8-bit offset | 16∗8-bit mask | n∗32-bit partial key | n∗64-bit value
MULTI_MASK_16_PKEYS_32_BIT (n=17–32, size=264–440 bytes)

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


header | 32∗8-bit offset | 32∗8-bit mask | n∗32-bit partial key | n∗64-bit value
MULTI_MASK_32_PKEYS_32_BIT (n=17–32, size=296–472 bytes)

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

Fig. 14. The physical node layouts.

of generality, however, similar approaches can also be transferred to other hardware architectures. In our particular case, we focus on the widest SIMD instruction set, which is broadly available. For end-user systems, this is currently (as of January 2020) the AVX2 instruction set with a market share of approximately 75%.[3]

As the AVX2 instruction set has 256-bit wide SIMD registers and is able to load multiple 8-bit, 16-bit, 32-bit, and 64-bit wide data words into a single SIMD register, we choose the maximum number of entries k per node to be 32. This decision has three reasons. First, in the case of the 8-bit data type, we are able to load and process all 32 items in a single SIMD register simultaneously. Second, in the worst case of requiring 32-bits per partial key, only two cache lines need to be accessed, and traditionally adjacent cache lines tend to be automatically prefetched by the CPU. Third, even in the worst case of using 32-bit per partial key, the amount of instructions needed to search the partial keys is quadrupled.

Based on the chosen maximum fanout, we choose two dimensions of adaptivity, depending on a node's discriminative bits, to adapt the node layout to the key set stored. First, depending on the range of a node's discriminative bits, we choose between a (I) single-mask layout and three (II) multi-mask layouts. These three multi-mask layouts (Figures 14(b), 14(c), and 14(d)) differ only in the number of offset/mask pairs (8, 16, or 32). The second dimension of adaptivity chooses the data types used for storing partial keys. To enable fast SIMD operations these partial keys need to be aligned. Therefore, we support partial keys to be stored in one of three representations, namely, as an array of 8-bit, 16-bit, or 32-bit values.

Combining all options for both dimensions results in 12 distinct node types. However, only nine of these combinations can occur in practice. For example, the 32-entry multi-mask layout shown in Figure 14(d) implies that there are more than 16 discriminative bits and that therefore, neither 8-bit nor 16-bit partial keys would suffice. The final nine supported node layouts are shown in Figure 14. An optimization we apply for the ALL is to store the type of each node within the least-significant bits of each node's pointer. This allows us to overlap two time-consuming operations, namely, loading a node's data, which can trigger a cache miss, and resolving the node type, which

[3]https://web.archive.org/web/20200103040011/http://store.steampowered.com/hwsurvey/.

```
TID2 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)) // Validate the candidate to prevent false positives
    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;
  for (size_t i=0; i < node->numberMasks; ++i) //load all 8-bit mask into a single 64-bit mask
    ((uint8_t*) keyParts)[i] = key[node->offsets[i]];
  return _pext_u64(keyParts, node->mask); //extract multiple 8-bit masks in parallel
}
uint32_t extractMultiMask16(MMask16Node* node, uint8_t* key)...
uint32_t extractMultiMask32(MMask32Node* node, uint8_t* key)...
int searchPartialKeys8(Node* node, uint32_t searchKey) ...
int searchPartialKeys16(Node* node, uint32_t partialKey) ...
int searchPartialKeys32(Node* node, uint32_t partialKey) ...
```

Listing 5. HOT lookup.

may otherwise suffer from a branch misprediction penalty. To ensure that the overhead in case of an actual branch mispredication is minimal, we explicitly prefetch the first four cache lines of a node while loading its data.

To get an understanding of lookup operations on HOT structures using an ALL node layout, we show the (slightly simplified) code for lookup, which traverses the tree until a leaf node containing a tuple identifier is encountered in Listing 5.

To conclude, ALL has the following benefits over FLL. Although for a given maximum fanout k and a maximum key length m, ALL has the same worst case space efficiency of $(k + log_2(m/8) + 8)$ per entry as FLL, our experiments show that in practice ALL is more space-efficient than FLL. It furthermore reduces the number of cache misses, by adaptively choosing the smallest fitting data type for the stored partial keys. Through carefully interleaving node type selection with loading the node itself, the overhead of the node type selecting becomes negligible.

5.1.4 SIMD and Linearized Node Layouts. The performance of linearized node layouts is based on the use of SIMD instructions to search a node's sparse partial keys in parallel. In our implementation, we use the AVX2 instruction set, which has a SIMD register width of 256 bits. Accordingly, we choose the maximum fanout of our ALL nodes to be 32, as a single AVX2 instruction can process at most 32 8-bit vector elements in parallel. If we wanted to optimize an ALL for another SIMD instruction set like SSE or AVX512, then it would make sense to adapt the maximum fanout accordingly. For instance, for 128 bit wide SSE registers this would lead to a maximum fanout of 16, and for AVX512 a maximum fanout of 64 accordingly. In particular, AVX512 seems to be promising, as a single AVX512 instruction can process a whole cache line at once. However, we do not only have to consider the sweet spot of 8-bit wide sparse partial keys but also the worst case of $(n-1)$-bit wide sparse partial keys for a maximum fanout of n. In the case of a fully occupied AVX512 node that stores 64-bit sparse partial keys, the structural area can span up to eight cache lines. This worst case scenario also impacts access performance, as search operation takes eight times longer than on 8-bit vector elements. In contrast, a maximum fanout to 32 limits the size of the structural area to two cache lines and accordingly impacts the access latency only by a factor of two. From these observations, we can conclude that larger vector registers can increase the throughput in the case of dense data. However, this improvement has to be weighed against the maximum node fanout and its impact on access latencies and the space consumption of the structural area in the worst case.

5.2 Hierarchical Node Layouts: Space Optimized

After discussing the performance optimized linearized node layouts, we now focus on the space optimized hierarchical node layouts.

On a logical level, each compound node of a Height Optimizing Trie is itself a binary Patricia trie structure with at most k entries. Typically, a binary Patricia trie is represented by a linked structure over its BiNodes in memory. In such a structure, each BiNode is a triple consisting of the discriminative bit position and a pointer to the left and right child, respectively. This simplicity is the major advantage of binary Patricia tries. However, as BiNodes can be scattered arbitrarily in memory and child pointers make up for a large fraction of a BiNodes memory, they suffer from bad cache locality and insufficient memory consumption. As in such a simple design we decode the internal BiNodes of the hierarchical tree structures iteratively, we cannot use SIMD optimization techniques. The reason for this is that before we traverse the actual BiNodes, we neither know all BiNode boundaries nor do we know all considered discriminative bits.

In this section, we present four different approaches for hierarchical node layouts for k-constrained tries which embrace the simplicity of a hierarchical layout, while improving cache locality and memory efficiency.

First, we introduce the direct node layout that uses node-relative pointers and places all its BiNode contiguously in memory. Next, we introduce the indirect node layout that stores BiNodes in pre-order and exploits the inherent properties of pre-order stored full binary trees to omit redundant information. Based upon the indirect node layout, we present the concept of a variable length indirect node layout that uses variable length encoding techniques to encode balanced trees more efficiently than imbalanced trees. Finally, we present the leaf optimized node layout. It extends on the concept of variable length indirect node layout by exploiting densely populated areas in a node's tree structure to reduce the overall memory footprint.

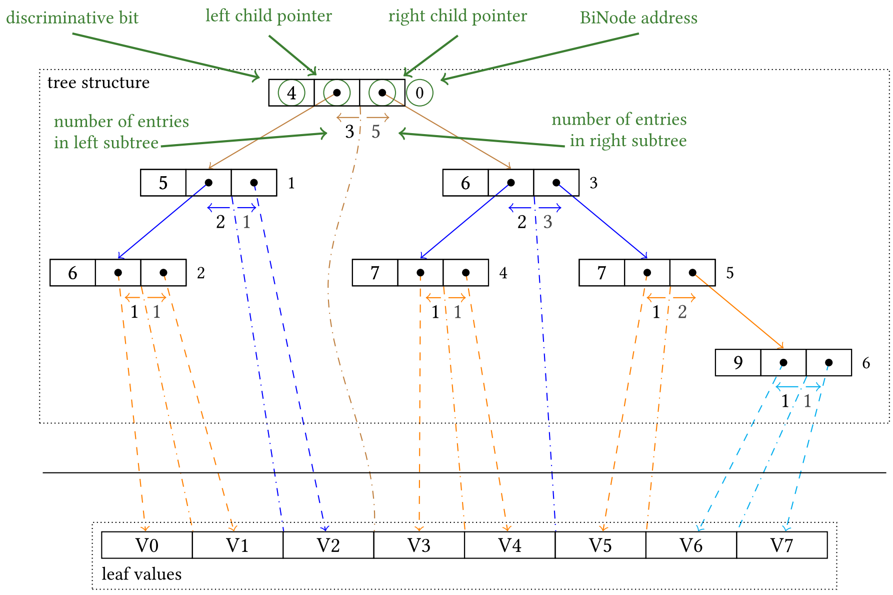


Fig. 15. A binary Patricia trie stored in pre-order. We annotate each BiNode with the number of leaf entries in its left and right subtree.

5.2.1 Direct Node Layout. The ***Direct node Layout (DL)*** is our first approach for a hierarchical node layout. It is a natural extension of a traditional binary Patricia trie, that places all its BiNodes in the structural area of the node. Hence, we can use relative addresses for node internal pointers. As a binary Patricia trie with n leaf entries contains exactly n-1 BiNodes, we use addresses 0 to n-2 to reference a node's BiNodes and addresses n-1 to 2n-2 to reference a node's leaf entries.

As the range of the child addresses used to link BiNodes and leaf entries inside a node is bound by the node's maximum fanout k, optimized data types can be used for internal pointers instead of general-purpose 64-bit pointers. For instance, 8-bit integers are sufficient to store the internal pointers of nodes with a maximum fanout of up to 128. Besides reducing the memory footprint, this also reduces cache misses as fewer cache lines are required to store the same number of BiNodes.

With relative addressing being the only difference to a traditional binary Patricia trie, it retains the simplicity of its access operations.

5.2.2 Indirect Node Layout. While already an improvement to traditional binary Patricia tries, the direct node layout presented in Section 5.2.1 requires storing the triple of discriminative bit, left- and right- child pointer for each BiNode. In this section, we introduce the ***Indirect node Layout (IL)***, which reduces the memory footprint in comparison to a DL, by using an indirect representation of the tree structure in memory. To create such an indirect representation, we exploit the following inherent properties of a full binary tree, stored in pre-order.

When storing a binary tree in pre-order, the left child of an inner node is the inner node's direct successor in memory and its right child is located at the address directly succeeding all inner nodes and leaf entries of the left subtree. Accordingly, a pre-order arranged direct node layout stores each right child leaf entry at the address in the data area, which is subsequent to all leaf entries contained in its parent's left subtree.

Figure 15 illustrates such a pre-order arranged binary Patricia trie. In this example, address 0 stores the root inner node. The subsequent address 1 stores the root's left inner node, while address 3—the next address after all inner nodes in the root's left subtree—holds the root's right inner node. Furthermore, the figure highlights another trivial property of pre-order arranged node layouts. Namely, that leaf entries contained in an inner node's right subtree are placed subsequent to all leaf entries contained in its left subtree. For instance, the left subtree of the inner node located at address 3 contains values V3 and V4 and the right subtree values V5, V6, and V7. Hence, the shown layout places the values V5 and V6 right after values V3 and V4.

Another interesting property of each full binary tree, and hence of every Patria trie, is that a full binary tree containing n leaf entries contains exactly $n-1$ inner nodes. In combination with a pre-order arranged storage layout, the correlation between the number of leaf entries and the number of inner nodes allows us to infer the address of right child inner nodes solely from its parent's address and the number of leaf entries in its parent's left subtree. More so, for an arbitrary inner node, we can infer the address of the right subtree's smallest leaf entry, from the first leaf entry of the inner node's subtree and the number of entries in the left subtree. For example, the left subtree of the binary Patricia trie shown in Figure 15 contains three leaf entries and hence, exactly two inner nodes. As the address of the root inner node is 0 in the structural area, we infer the address of the root's right child inner node to be 3 in the structural area. Further, from the address of the root's smallest leaf, which is trivially 0, and the three entries contained in the root's left subtree, we infer address 3 to be the first leaf entry's address in the root's right. Based on the fact, that the exemplary tree structure contains seven leaf entries, and the left subtree contains three leaf entries, we are able to infer the number of leaf entries in the right subtree to be four. Thus, for the subtree rooted at the inner node at address 3 in the structural area, we infer its number of leaf entries and the address of the subtree's first leaf. If for this inner node only the additional information of the number of entries contained in its left subtree is provided, then we are recursively able to infer the properties of its descending subtrees.

Thus, based on the properties that for any full binary tree stored in pre-order, the overall number of entries together with the number of entries in each inner node's left subtree is sufficient to infer the tree's structure, we introduce the IL. It always stores its inner nodes in pre-order and for each inner node it only stores the number of entries contained in its left subtree instead of direct pointers to its child pointers. Figure 16 shows such an indirect node layout for the binary Patricia trie illustrated in Figure 15. It highlights all properties of an inner node that can be inferred from an indirect layout. These are the number of entries in the right subtree, the value range of the corresponding leaf entries, and the address of the left and right subtree. By inferring these properties, only $log_2(k) + log_2(m)$ bits are required for a given maximum fanout k and given maximum key length m for each inner node in an indirect node layout. In contrast, at least $log_2(m) + 2 * log_2((2 * k) - 1)$ bits are required to represent an inner node in an equivalent direct node layout.

To search an indirect node layout, we traverse the node layout starting from the root by recursively inferring these properties for each inner node along the path until the number of entries in the currently considered subtree is 1 and we reach the leaf level.

We conclude that the indirect node layout improves over the direct node layout by retaining the simplicity of the DL's search operation, while at the same time reducing the amount of memory required to store an equivalent k-constrained binary Patria trie by more than a half.

5.2.3 Variable Length Indirect Node Layout. Even though the indirect node layout introduced in Section 5.2.2 improves over the direct node layout introduced in Section 5.2.1, it requires the same amount of memory for each key, regardless of the dataset stored. For instance, in case the tree structure resembles a complete binary tree structure, we could omit to store the size of the left subtree at each inner node, as in this specific case all inner nodes are perfectly balanced and the

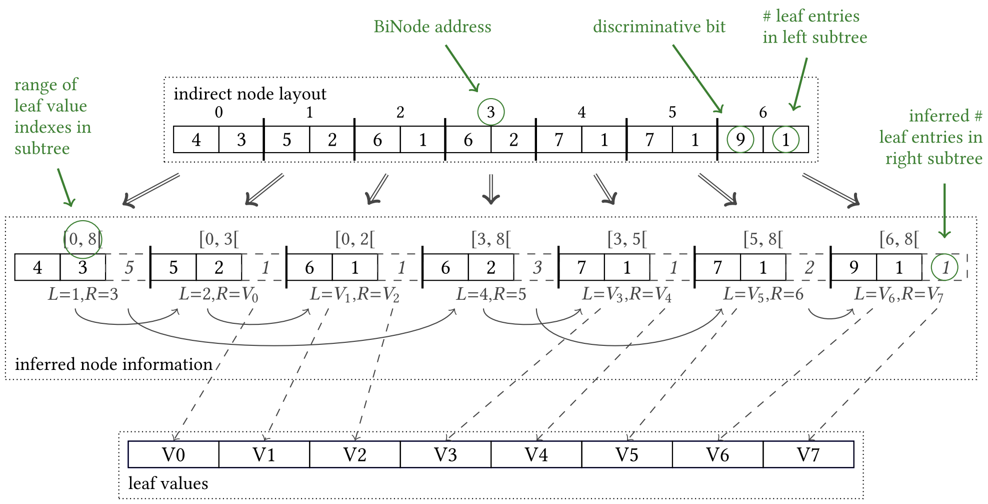


Fig. 16. An indirect node layout containing eight entries. The box in the second line illustrates how the actual tree structure can be inferred from the information stored in the indirect node layout. The letters L and R, which denote the inferred pointers to the left and right child of the respective BiNodes, represent this tree structure.

number of entries in the left subtree and right subtree is always equal. Alternatively, in the case that the parent discriminative bit b_p of any node n with discriminative bit b_n is exactly $b_p = b_n - 1$, storing the discriminative bit for all inner nodes except for the root node is also redundant. The reason is that in this case an inner node's discriminative bit can always be derived from its parent inner node. If the structure satisfies both of these special cases, then storing only the discriminative bit of the root node would be sufficient. Please note that this special case only occurs if a dense range of keys is stored (e.g., all natural numbers up to a certain value x). However, even in the case of storing uniformly distributed random keys such densely populated balanced regions may occur. Specifically, the upper-level inner nodes of a binary Patricia trie containing uniformly distributed random data tend to be (almost) balanced.

In the remainder of this section, we therefore introduce a novel node layout that exploits such regularities in tree structures. First, we present the ***Delta encoded Indirect node Layout (DIL)*** that by using an alternative encoding scheme lends itself as a basis for compressed node layouts. Second, we introduce the ***Variable length Indirect node Layout (VIL)***, which enhances the delta encoded indirect node layout with an efficient variable length encoding scheme reducing the memory consumption of nodes.

To create an indirect node layout that encodes balanced nodes more efficiently than nodes with a skewed distribution, we have to change the encoding for the number of entries contained in an inner node's left subtree. Specifically, such an encoding has to use larger numbers for the cardinality of a skewed inner node's left subtree and smaller numbers for the cardinality of a balanced inner node's left subtree. We exploit the fact that in a completely balanced inner node, half of its descending entries are stored in its left subtree and the other half in its right subtree. Hence, to create an encoding that has the desired properties, we only store the delta between the actual number of entries in the left subtree and half of the number of entries in the whole subtree.

To further optimize an inner node's storage layout, we can apply a similar delta encoding technique for storing the discriminative bit information. To optimize the encoding of discriminative bits, we leverage that the discriminative bit of a child inner node is always larger than the

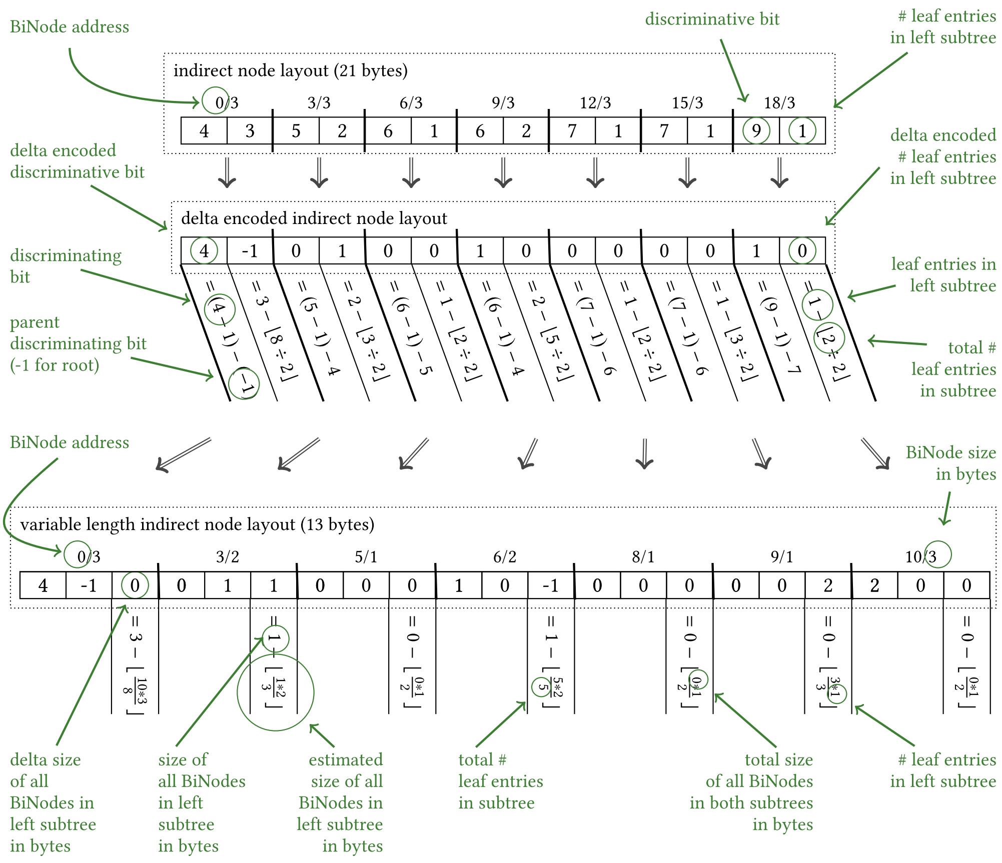

BiNode address
discriminative bit
# leaf entries in left subtree
indirect node layout (21 bytes)
delta encoded discriminative bit
delta encoded # leaf entries in left subtree
delta encoded indirect node layout
discriminating bit
parent discriminating bit (-1 for root)
leaf entries in left subtree
total # leaf entries in subtree
BiNode size in bytes
variable length indirect node layout (13 bytes)
delta size of all BiNodes in left subtree in bytes
size of all BiNodes in left subtree in bytes
estimated size of all BiNodes in left subtree in bytes
total # leaf entries in subtree
total size of all BiNodes in both subtrees in bytes
# leaf entries in left subtree

Fig. 17. The transition from an indirect node layout of the binary Patricia trie shown in Figure 15 to a variable length indirect node layout.

discriminative bit of its parent inner node. Hence, we only need to store the delta between the smallest possible discriminative bit—determined by the successor of its parent inner node's discriminative bit—and the actual discriminative bit. Thus, if a child's inner node's discriminative bit is the successor of its parent's inner node, the delta encoded discriminative bit is zero.

An example of a delta encoded indirect node layout for the tree structure depicted in Figure 15 is shown in Figure 17. It shows for each inner node how the information stored in an indirect node layout is transformed into a delta encoded indirect node layout. For each discriminative bit, it shows that the delta encoded discriminative bit can be inferred from the indirect node layout by applying Equation (1):

$$delta\_disc\_bit = (disc\_bit - 1) - (parent\_disc\_bit). \tag{1}$$

Additionally, the figure shows for each inner node that Equation (2) determines the delta between the actual number of entries in the left subtree and the expected number of entries in a balanced subtree with the same number of entries:

$$delta\_no\_left\_entries = actual\_no\_left\_entries - \left\lfloor \frac{total\_no\_entries}{2} \right\rfloor. \tag{2}$$

Although a delta encoded indirect node layout encodes balanced dense trees more efficiently, due to the fixed-sized inner node layout it still has the same memory requirement as an indirect node layout. Hence, to benefit from the delta encoding, we have to adapt the inner node layout to the content of the encoded inner nodes. However, naively using dynamically sized inner nodes prevents our addressing scheme from deriving the location of child inner nodes from the number of leaf entries in the left subtree. As we store the tree structure in pre-order, a potential solution is to store the offset of the right child inner node for each inner node. In the worst case, this offset can be as large as the total size of the structural area. This contradicts our aim to reduce the memory footprint of the overall node layout by using variable sized inner nodes. To still achieve our goal of reducing memory consumption by using variable sized inner nodes, we exploit the following assumption. When using variable sized nodes, we assume that the ratio between the memory requirement of inner nodes in the left subtree and the memory requirement of inner nodes in the right subtree correlates with the ratio between the number of leaf entries in the left subtree and the number of entries in the right subtree in the common case. We denote the memory requirement of a subtree's inner nodes as the *size* of the subtree's structure.

We leverage the relationship between the sizes of an inner node's subtrees and the number of entries contained in the corresponding subtrees to estimate the size of the left subtree by the following equation:

$$left\_subtree\_size\_estimation = \frac{total\_subtree\_size}{\#entries\_total} * \#entries\_left\_subtree. \tag{3}$$

Based on the assumption that the estimated size of a subtree's structure is close to its actual size, we propose the VIL layout. Instead of storing the actual size of its left subtree structure for each inner node, the VIL only stores the delta between the left subtree's actual structure size and its estimation.

We show an example of a variable length indirect node layout in Figure 17. The figure depicts how a VIL is derived from the IL and the DIL of the binary Patricia trie structure shown in Figure 15. It can clearly be seen that even though the structural area of the example's VIL requires 13 bytes, the delta size of the left subtree's structure of any inner node in the example is at most two and even zero for more than half of the examples' inner nodes.

Although the VIL is able to exploit regularities in the data to reduce the memory consumption in comparison to a plain IL, it only adds minimal overhead to the search operation. Please note that the actual runtime performance depends on the precise memory layout of a single inner node and the encoding techniques used to compress a single inner node. We therefore present the Leaf Optimized Node Layout next, that is an actual physical node layout that is based on the concept of the variable length indirect node layout but enhances on it by compacting so called dense regions

5.2.4 Leaf Optimized Node Layout. The memory efficient node layouts introduced in Sections 5.2.2 and 5.2.3 base on the assumption that all inner nodes have to be represented explicitly. However, we show in this section that depending on the stored dataset it is possible to completely omit certain inner nodes. Specifically, we focus on approaches exploiting so-called leaf dense regions to design node layouts with increased memory efficiency.

We define dense regions in a dataset to be a set of 2^k successive keys that share a common prefix and cover all possible 2^k combinations of the k bits following the common prefix. For instance, the four binary keys 101**00**110, 101**01**110, 101**10**110 and 101**11**110 represent a dense region that shares the common prefix 101 and covers all possible combinations of the successive bits 3 and 4. In the following, we call these successive bits the region's *distinctive bits*. In a binary Patricia trie structure, dense regions are subtrees of binary Patricia tries that have the shape of perfect binary trees, and

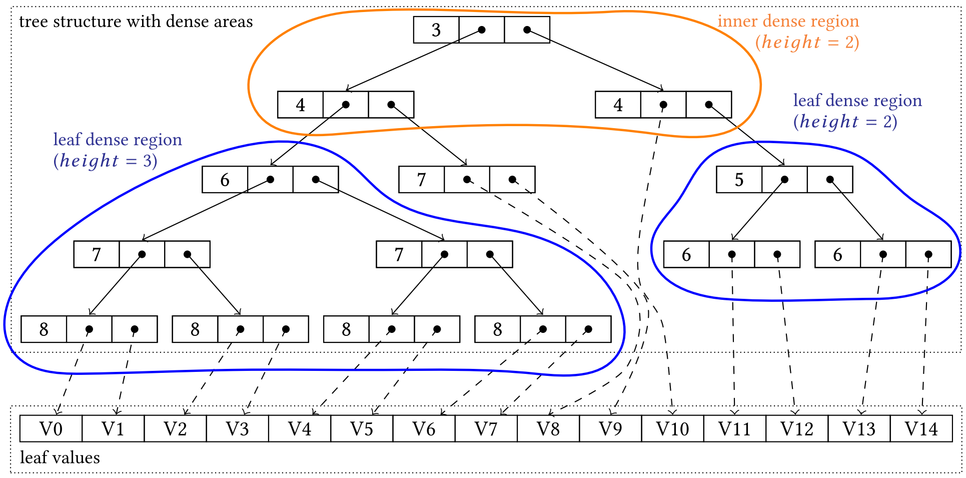


Fig. 18. Binary Patricia trie containing three dense regions. Inner dense regions are highlighted in orange, while leaf dense regions are highlighted in blue.

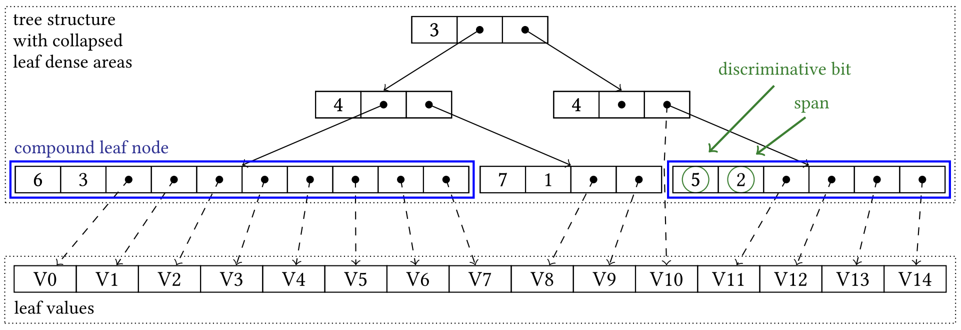


Fig. 19. A binary Patricia trie with collapsed dense regions.

therefore, all leaf entries have the same depth. Except for the region's root, the discriminative bit $b_n = b_{parent} + 1$ of each inner node n directly succeeds its parent's discriminative bit b_{parent}.

We distinguish two kinds of dense regions: *leaf dense regions* and *inner dense regions*. While nodes in inner dense regions may have descendant nodes that are not part of the same dense region, nodes in leaf dense regions only have descendant nodes that are part of the same dense region. Hence, leaf dense regions of height h always contain 2^h leaf nodes. In the Patricia trie depicted in Figure 18, we identify three dense regions, one of height three and two of height two. The inner dense regions in this figure are outlined in orange and leaf inner regions are outlined in blue.

For instance, in a dense region of height 2, the distinctive bits 00 address the first subtree, 01 the second, 10 the third, and 11 the last subtree. This addressability allows us to replace the subtree corresponding to the dense region by a single node with fanout 2^k. We denote this transformation of a region's BiNodes into a single inner node of higher arity as *collapsing* and the resulting inner nodes as a collapsed dense regions. As an example, in Figure 19, we depict the tree previously shown in Figure 18 with its leaf dense regions collapsed.

In contrast to level compressed tries [1, 31] that use dense regions to determine the number of bits considered for array-based nodes, we leverage dense regions to optimize our internal node structures.

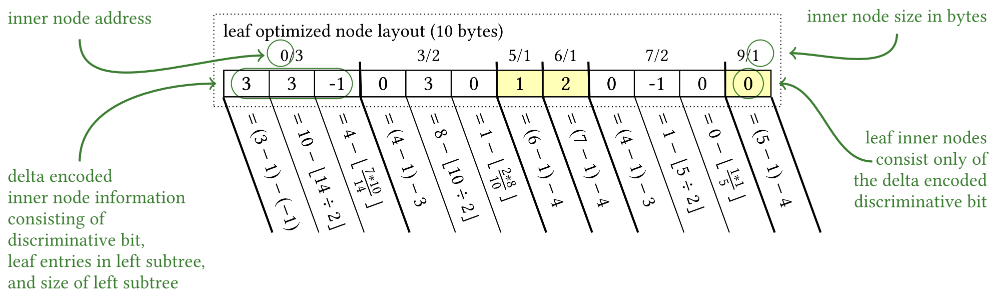


Fig. 20. A leaf optimized node layout for the binary Patricia trie shown in Figure 19. A yellow background highlights the collapsed leaf nodes.

In the remainder of this section, we present the ***Leaf Optimized node Layout (LOL)***, which extends on the VIL presented in Section 5.2.3, by first collapsing all dense regions and, second, by introducing a special inner node type that stores only the first discriminative bit position for these newly created leaf collapsed regions.

Since leaf BiNodes can also be considered as dense regions containing only two entries, storing the discriminative bit position is sufficient in these cases as well. In Figure 20, we show the resulting leaf optimized layout for the binary Patricia trie with collapsed dense regions of Figure 19. In the figure, the collapsed inner nodes are highlighted in yellow.

The search algorithm of a LOL extends on the search algorithm of a variable length indirect node layout by deriving the span of the collapsed leaf node from the number of entries in the respective dense region. Based on the derived span, the algorithm extracts all bits corresponding to the collapsed leaf region and uses these bits of the search key for addressing the potential match candidate relative to the leaf region's start. To determine whether the current BiNode is a "normal" inner BiNode or the root of a collapsed dense region, we have to check if the size of the current subtree is equal to the current BiNode's size. The complete search algorithm is shown in Listing 6.

Although LOL's search algorithm requires more operations per inner node than VIL's search operation, depending on the used dataset and the number of its collapsible dense regions, LOL compensates this overhead by omitting inner nodes in collapsed leaf dense regions.

Note that the minimum space required—which is the case for dense data—is as low as storing the actual discriminative bits. However, in the worst case, the space consumption is even higher than for an indirect node layout. For a given maximum fanout k, and a maximum key length m the worst case space consumption is $log_2(m) + log_2(k) + log_2(k * (log_2(m) + log_2(k))) + 1$ bits, which is identical to the worst case space consumption for a variable length indirect node layout. However, as we can see in Section 7 the actual space consumptions for real-world data is lower.

5.3 Discussion on Disk Optimized Node Layouts

In this article, we focus on main memory optimized node layouts for HOT. However, we want to provide a short discussion on the challenges of disk-based node layouts and how such a disk-based implementation could look like.

In general, two major aspects have to be considered when implementing a disk-based index structure in comparison to purely main memory-based structures. First, a page, which is the unit of data transfer in disk-based systems, is much larger than a cache line, which is the unit of transfer in main-memory-based systems. The typical size of a page is about 4 KiB, while the typical size of a cache line is only 64 bytes. Second, the access latencies are orders of magnitudes higher. While a random access to main memory takes roughly about 100 ns, accessing a single page on an SSD takes more than 10 μs, and executing a random seek on a spinning disk even takes around 5 ms.

```
Leaf* search(uint8_t const * key) const {
  uint currentAddr = root; // the address of the current bi node
  uint discrBitPos = -1; // the position of the previously considered BiNode discriminative bit
  uint remaining = totalNumberEntries; // entries in the current subtree
  uint indexOfFirstLeaf = 0 // index of first leaf entries in current subtree
  uint subtreeSize = sizeOfStructuralArea; // total size of biNodes in current subtree in bytes
  while(subtreeSize > 0) {
    BiNode* current = readBiNode(currentAddr); // readBiNode respects variable sized biNodes
    discrBitPos += 1 + current->deltaDiscrBitPos;
    subtreeSize -= current->biNodeSize;
    if(subtreeSize > 0) {
      uint noLeftEntries = remaining/2 + current->leftSubtreeEntriesDelta;
      uint estimatedLeftSize = (subtreeSize * noLeftEntries)/remaining;
      uint leftSize = estimatedLeftSize + current->leftSubtreeSizeDelta;
      if(extractBit(key, discrBitPos) == 0) {
        currentAddr = current->biodesSize;
        remaining = noLeftEntries;
        subtreeSize = leftSize;
      } else {
        currentAddr += leftSize;
        indexOfFirstLeaf += noLeftEntries;
        remaining -= current->entriesInLeftSubtree;
        subtreeSize -= leftSize;
      }
    } else { // a leaf entry is reached
      int span = log2(remaining);
      indexOfFirstLeaf += extractBits(key, discrBitPos, span);
    }
  }
  return leafs[indexOfFirstLeaf];
}
```

Listing 6. Lookup in leaf optimized node layout.

These differences in access granularity and disk access latency have the same impacts on design decisions for HOT node layouts that we already know from existing disk optimized index structures, like B-Trees. First, the node size needs to be adapted to the underlying storage media's page size. Second, due to the high access latencies, it is possible to use compression algorithms that would be too costly for main memory index structures. However, in the case of disk-based index structures, these compression techniques allow one to increase the node fanout, reduce the tree height, and thereby the number of random accesses. Thereby it trades CPU instructions for memory consumption and IO-operations. In the following, we will describe our conclusions of how based on these design decisions a potential HOT disk-based node layout's might look like.

The fanout of HOT nodes need to be increased to fully utilize the underlying page sizes. This by definition also increases the maximum number of bits that have to be considered per node. If in the worst case the fanout becomes close to the key length in bits of the longest key stored, then the implication is that all bits might have to be considered for a single node. This implies that SIMD will loose some of its benefits. The reasons for this is that the length of the sparse partial keys and the fanout might become too large. If the keys become too long, then—in the worst case—not even a single key could be compare by a single SIMD operation. If the fanout becomes too large, then a linear search requiring multiple SIMD instructions will be required to search a node.

For these reasons the hierarchical node layouts, especially the leaf optimized node layout that tightly packs the stored key information together presents a good starting point to develop a disk-based node layout. However, we assume that at least two improvements would be beneficial to create a competitive disk optimized index structure. First, the encoding of the inner nodes need to be adapted to support larger fanout and thereby to leverage node sizes of even multiple disk pages. Second, the memory management need to be optimized to prevent fragmentation. The reason is that currently HOT uses tightly fit variable sized nodes and a memory allocator based on size classes. While the same approach can be used for disk-based structures as well, we fear that the high number of size classes in combination with high update rates would lead to fragmentation. In addition, random accesses to separate underlying physical pages for small successive nodes, might limit the scan performance.

We consider using an indirection table and combining multiple adjacent nodes until a predefined size is reached as a potential solution to improve both the scanning performance and to limit the memory fragmentation. To implement such an approach, we would need to extend the copy on write updates and the locks used in our synchronization protocol from a single node to such a combined set of adjacent nodes.

Alternatively, due to the larger page sizes switching to in-place updates in combination with Optimistic Lock Coupling [24, 26] might be another solution to achieve high update-performance on disk-based systems.

6 SYNCHRONIZATION PROTOCOL

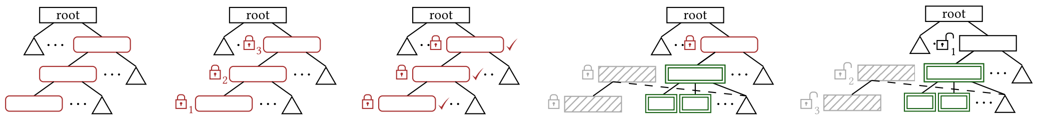
Besides performance and space efficiency, scalability is another crucial feature of any index structure. A scalable synchronization protocol is therefore necessary to provide efficient concurrent index accesses. In the following, we present such a protocol for HOT.

Traditionally, index structures have used fine-grained locking and lock coupling to provide concurrent accesses to index structures [12, 13]. 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 [26]. Therefore, different approaches based on the concept of lock-free index structures or write-only minimal locks have been proposed [21, 26–28]

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 CAS operation does not suffice to synchronize write accesses to HOT. If two insert operations are issued simultaneously, then 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′, then 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 Write EXclusion (ROWEX)** synchronization strategy [26]. 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, however, 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 5) 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 21 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



(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.

Fig. 21. 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.

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, then 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 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, then 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 [15], which frees the memory of obsolete nodes whenever no more reader or writer accesses the corresponding nodes.

General Applicability: The ROWEX synchronization strategy [26] subsumes synchronization protocols that support lock-free read access and require locks for write operations only. While the concept is generally applicable, it does not specify how to implement corresponding synchronization protocols. Hence, ad hoc implementations have been used so far to implement ROWEX synchronization protocols for different index structures. This is in contrast to other general applicable synchronization protocols, like Optimistic Lock Coupling [24, 26], which is directly applicable to several tree structures supporting in-place updates. The ROWEX-based synchronization protocol presented in this article fills this gap for *copy-on-write*-based tree structure. It is applicable not only to HOT, but to all tree structures that use *copy on write* and *modify only a single pointer* per update operation. Due to its general applicability for *copy on write*-based index structures, we call it *Copy on Write ROWEX*.

7 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.

7.1 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 HOT against the following state-of-the-art index structures:

- *ART:* The Adaptive Radix Tree (ART) [25], which is the default index structure of HyPer [40]. It features variable sized nodes and selectively uses SIMD instruction to speed up search operations.
- *Masstree:* Masstree [29] is a hybrid B-Tee/trie structure used by Silo [35].
- *BT*: The STX B$^+$-Tree[4] represents a widely used cache-optimized B$^+$ Tree (e.g., Reference [18]) 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.

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.

For all evaluated index structures, we use 64-bit pointers as tuple identifiers to address and resolve the actually stored values and keys. In case the stored values only consist of fixed-sized keys up to 8 byte length (e.g., 64-bit integers), those keys are directly embedded in their tuple identifiers. To measure space efficiency, we add custom code to the implementations of ART and the B$^+$-Tree to compute their memory consumptions without impacting their runtime behavior. For Masstree, we use its allocation counters to measure its 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.

For our experiments, we use the following datasets:

- *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 [3], 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 [17]. 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.

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

7.2 Node Layouts

In this section, we examine access performance and memory consumption of our proposed HOT node layouts. For this purpose, we implemented read-only variations of the ALL representing our most sophisticated linearized node layout and the Leaf optimized node layout representing our most sophisticated hierarchical node layout. We compare them with two modern trie variants, ART and Masstree and a standard B^+-Tree in terms of lookup throughput and memory efficiency.

We optimized both HOT node layouts in terms of performance and cache efficiency. Hence, the maximum node fanout differs between the two evaluated node layouts. In the following, we briefly describe the implementations of these two node layouts.

- *Leaf optimized node layout (LOL)*: The evaluated LOL implementation uses a maximum fanout of 256. To reduce the memory footprint, we use four different inner nodes that adapt to the actual data stored. Inner node type I requires 8 bits, inner node type II requires 16 bits, inner node type III requires 24 bits, and inner node type IV requires 32 bits. To distinguish the different inner nodes types, each inner node starts with a header encoding the node type. To reduce the overhead of storing the node types, we store the node type using a unary encoding to favor the shorter inner node types coded by smaller numbers. We distribute the remaining bits to store the delta discriminating bit, the delta encoded number of entries in the inner node's left subtree and the delta encoded size of the left subtree as follows: 3/2/2 (I), 6/6/2 (II), 9/7/5 (III), and 11/8/10 (IV). Applying our upper bounds space estimation introduced in Section 3.3 this leads to an upper bound of 24 bytes per entry.
- Adaptive Linearized Node Layout (ALL): The ALL has a maximum fanout of 32. Each ALL contains three sections: extraction information, sparse partial keys, and pointers. Depending on the stored keys, the smallest fitting node type is chosen. In contrast to the design shown in Figure 14, we support only eight different node types in this particular implementation and omit the multi-mask layout with 16 extraction masks and 32-bit partial keys. Supporting only 8 different node type allows for conveniently encoding the node type in the lower three bits of each node's address. The ALL exploits this addressing scheme to overlap branch misprediction caused by node type selection with prefetching the node's first four cache lines. For alignment reasons the ALL implementation pads the partial key section to 64-bit boundaries. Applying our upper bounds space estimation introduced in Section 3.3 and Section 5.1.3 this leads to an upper bound of 28 bytes per entry.

To evaluate the access characteristics and memory footprint of the different node types, for each of the evaluated index structures, and each of the four datasets, we insert 50 million entries and measure the resulting tree height and memory consumption. To evaluate the access characteristics, we issue 100 million randomly distributed lookup operations and measure the throughput in million operations per second. Additionally, for each lookup operation, we track the number of instructions, L1-cache misses, LLC-cache misses, branch misses, and the number of branch instructions per lookup operation. The results of this evaluation are shown in Figure 22. Additionally a dashed red line marks the average key length for the textual datasets in the mixed evaluation. This red line allows us to identify the index structures that require more space than the actual keyset.

With regards to memory consumption, for all datasets, both HOT variations achieve a lower memory consumption than the other evaluated index structures. The LOL layout is the most space-efficient HOT variation. Over all evaluated datasets, its spaced consumption lies within a 6% boundary, thereby it requires only between 33% (random) and 27% (url) of the ALL node layout's space to store the structural information. In absolute numbers, LOL never requires more than 2 bytes per entry to store its structural information. Even ALL's worst-case memory requirement for the

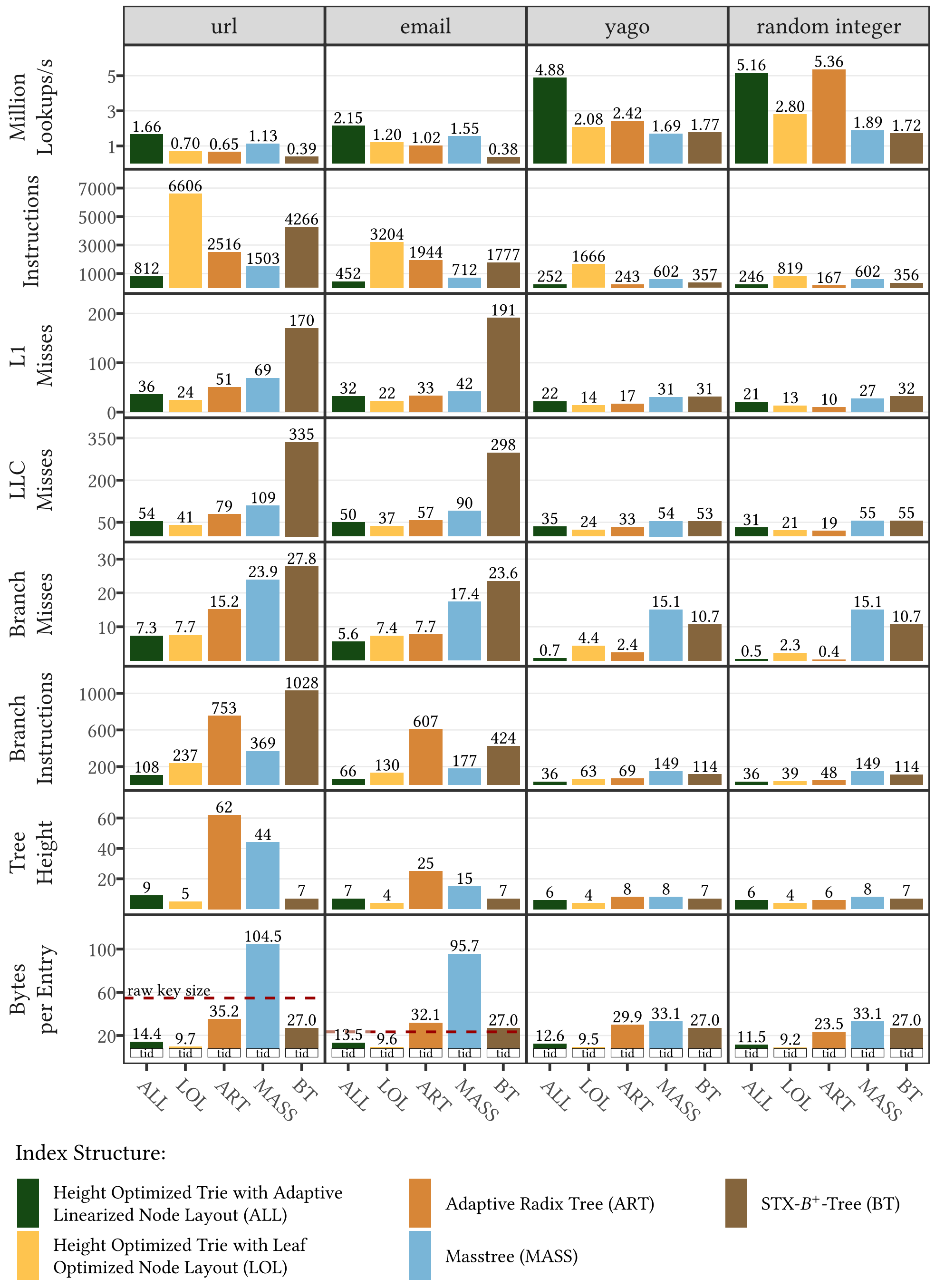


Fig. 22. Memory consumption and access performance of the fastest (Adaptive Linearized Node Layout) and the space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to two trie-based indexes (ART and Masstree) and a B$^+$-Tree. The dashed red line marks the average key length of the textual datasets (url: 55 bytes, email: 23 bytes).

structural information in the case of the URL dataset (6.45 bytes/entry) is 59% lower than the lowest measured memory requirement of the other evaluated index structures (ART:15.5 bytes/url). More remarkably, for the structural information, the memory consumption of HOT for all evaluated datasets remains within a 25% boundary, while ART's memory consumption varies up to 50% and Masstree's by as much as 215%.

Thereby, the HOT variations are the only trie-based index structures, which for all datasets have a space consumption that 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 datasets. However, for all datasets, the B^+-Tree requires at least 88% more space than the HOT variations' worst-case space consumption measured for the ALL structure on the url dataset. Moreover, the HOT variations are the only index structures that for both textual datasets require less space than the actual raw keys (email: 43%, url: 74%).

Although requiring more memory per entry, ALL achieves the best lookup performance of all evaluated datasets except for random integers, where it has a slightly lower (4%) throughput than ART in this particular case. For all other datasets, ALL executes between 93% and 161% more lookup operations than ART though. As expected, LOL's space-efficient memory layout results in the best cache efficiency of all evaluated data structures. Only in the case of the uniformly distributed random integer dataset—presenting the sweet spot for fixed span node layouts like ART—LOL has 5% more LLC misses and 23% more L1 misses. In the case of the URL dataset, LOL has 53% less L1 misses and 52% less LLC misses, though. However, the cost of LOL's memory and cache efficiency is the large number of instructions required to traverse its hierarchical variable length indirect node layout. For all evaluated datasets, LOL executes the most instructions per search operation. In particular, it executes between 35% and 160% more instructions than the index structure with the second most instructions per search operation for the respective dataset. This also affects LOL's throughput, which is between 42% and 58% lower than ALL's throughput. However, LOL has never the slowest lookup performance of the evaluated structures. In the case of the integer datasets, LOL executes between 34% (yago) and 64% (random) more lookup operations in comparison to Masstree and in the case of the textual datasets still between 22% (email) and 11% (url) more lookup operations as ART.

To get a better understanding of how our different HOT node layouts and the three state-of-the-art index structures ART, Masstree, and the STX-B^+-Tree compare to each other, we provide two faceted scatter plots that visualize the trade-offs between access performance and space consumption. In these scatter plots, the x-axis represents access latency and the y-axis represents memory consumption per entry. Each marker represents a single index structure, and the position of the marker corresponds to the access latency and memory consumption of the index structure. Figure 23 shows the memory consumption and access latency of the fastest (Adaptive Linearized Node Layout) and the most space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to ART, Masstree, and the STX-B^+-Tree, while Figure 24 focuses only on our HOT node layouts, which are presented in Section 5.

Figure 23 shows that regardless of the datasets evaluated, the two HOT variants are more space-efficient than the other evaluated index structures. Although the adaptive linearized node layout requires about half the memory of the other non-HOT index structures, the leaf-optimized node layout reduces the space required for the structural information by another 50%, regardless of the dataset being evaluated. Whereas the B-Tree has the worst latency for all evaluated datasets, ART provides the best access performance only in the case of the random integer dataset, and Masstree achieves competitive access performance only on textual data, the HOT index structures provide consistent access performance. While the adaptive linearized node layout provides the lowest latency on all but the random integer dataset (where its latency is only slightly higher than

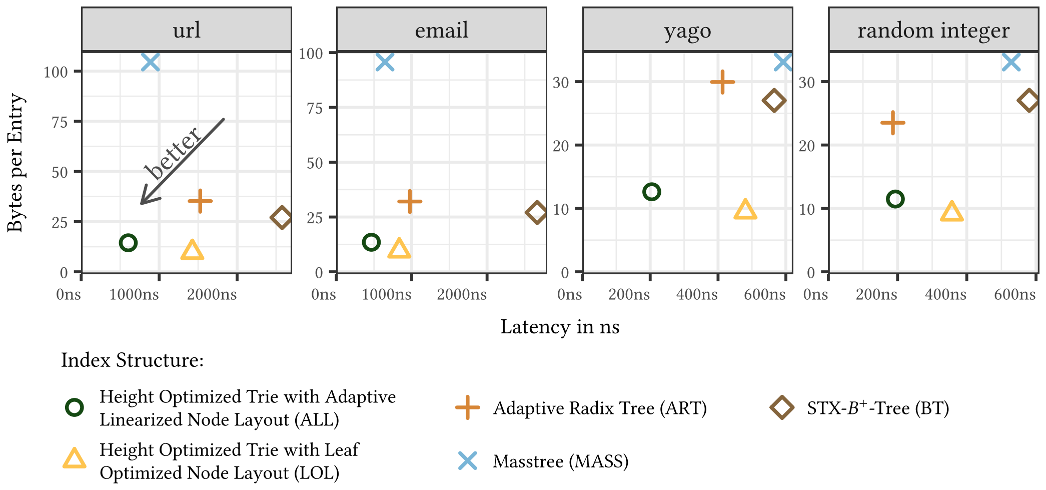


Fig. 23. Memory consumption and access latency for the fastest (Adaptive Linearized Node Layout) and the most space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to two trie-based indexes (ART and Masstree) and a B$^+$-Tree on four datasets with 50 million keys of different length (url: 55 bytes, email: 23 bytes, yago and random integers:8 bytes).

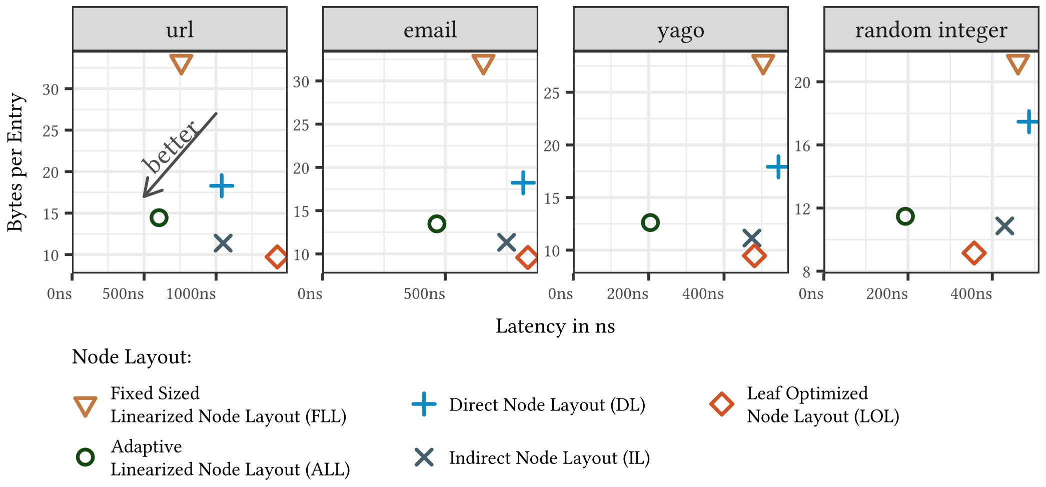


Fig. 24. Memory consumption and access latency for all HOT node layouts on four datasets with 50 million keys of different length (url: 55 bytes, email: 23 bytes, yago and random integers: 8 bytes).

ART's), also the space-optimized leaf optimized node layout offers a lower access latency than two out of three non-HOT index structures across all datasets.

Concerning the remaining HOT node layouts, Figure 24 shows that for both linearized and hierarchical node layout the more sophisticated layouts require less memory compared to their base variants. With regards to access latency, for the family of linearized node layouts the more sophisticated adaptive linearized node layout achieves lower access latency then the much simpler fixed sized linearized node layout. In contrast, for the family of hierarchical node layouts the opposite is the case. While the more sophisticated leaf optimized node layouts provide higher space efficiency,

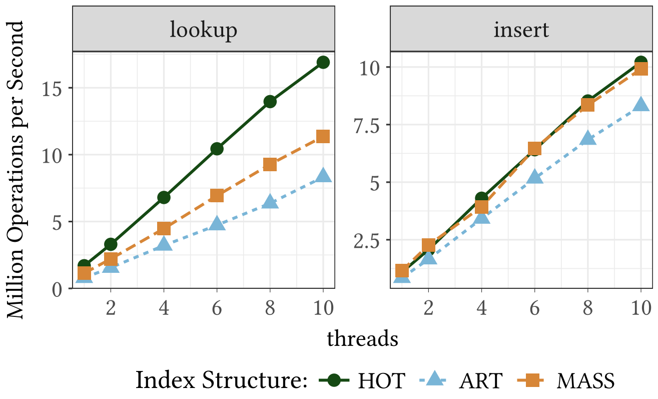


Fig. 25. Scalability on the 50M urls data set.

the simpler direct and indirect node layouts achieve lower access latency. Of these two simpler hierarchical node layouts, the indirect node layout is preferable to the direct node layout, because it requires less space and generally offers lower access latency. Only in the case of the url dataset, the indirect node layout has a negligible 1% higher latency than the direct node layout.

Overall, from this read-only micro-benchmark, we conclude that LOL is the best overall node layout in terms of memory consumption and cache efficiency, while ALL is the best overall node layout in terms of throughput. Even more, ALL is still superior to all non height optimizing trie structures in terms of memory consumption.

7.3 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 7.1, we execute a workload consisting of 50 million randomly distributed insert operations, followed by 100 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 used 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 25. 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 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.

8 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. We were able to prove that the resulting trie structures are of minimal height and have a recursively defined deterministic structure that is independent of the order of the stored keys. Furthermore, we were able to show that HOT's generic design can be geared toward different usage scenarios by implementing different physical node layouts. While the proposed adaptive linear node layout (ALL) leverages SIMD operations to provide optimized search performance, the Leaf-Optimized node Layout (LOL) embraces different encoding techniques to provide a reduced footprint.

Our experimental results shows that an ALL-based HOT is 2× more space efficient than 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 nearly linear multi-core scalability. In addition, we were able to show that the LOL-based HOT structure provides the highest memory efficiency of all evaluated index structures. For all evaluated datasets the space requirement of LOL was less than 2 bytes per key. The high access performance and excellent space efficiency makes HOT a highly promising index structure for main-memory database systems. Due to its flexible nature, we expect that potential future HOT node layouts, will also address the requirements of other storage layers like HDDs, SSDs, or persistent memory.

REFERENCES

[1] Arne Andersson and Stefan Nilsson. 1993. Improved behaviour of tries by adaptive branching. *Inform. Process. Lett.* 46, 6 (July 1993), 295–300. https://doi.org/10.1016/0020-0190(93)90068-K

[2] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: A cache-conscious trie-based data structure for strings. *Proceedings of the 30th Australasian Conference on Computer Science.* 97–105. Retrieved from http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV62Askitis.html.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference.* 722–735. https://doi.org/10.1007/978-3-540-76298-0_52

[4] Timo Bingmann. 2008. *STX B+ Tree C++ Template Classes.* Retrieved from http://panthema.net/2007/stx-btree.

[5] Robert Binna, Dominic Pacher, Thomas Meindl, and Günther Specht. 2014. The DCB-tree: A space-efficient delta coded cache conscious B-tree. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014.* 30–41. Retrieved from http://www-db.in.tum.de/hosted/imdm2014/papers/binna.pdf.

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* https://doi.org/10.1145/3183713.3196896

[7] Binna, Robert. 2020. *Fast and Space-Efficient Indexing For Main-Memory Database Systems on Modern Hardware.* Ph. D. Dissertation. University of Innsbruck, Austria. Retrieved from https://diglib.uibk.ac.at/ulbtirolhs/content/titleinfo/5124193/full.pdf.

[8] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09).* 283. https://doi.org/10.1145/1559845.1559877

[9] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient in-memory indexing with generalized prefix trees. In *Proceedings of the 14th BTW Conference on Database Systems for Business, Technology, and Web.* 227–246. Retrieved from http://subs.emis.de/LNI/Proceedings/Proceedings180/article22.html.

[10] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving index performance through prefetching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 235–246. https://doi.org/10.1145/375663.375688

[11] David E. Ferguson. 1992. Bit-tree: A data structure for fast file processing. *Commun. ACM* 35, 6 (June 1992), 114–120. https://doi.org/10.1145/129888.129896

[12] Goetz Graefe. 2010. A survey of b-tree locking techniques. *ACM Trans. Database Syst.* 35, 3, Article 16 (July 2010), 26 pages. https://doi.org/10.1145/1806907.1806908

[13] Goetz Graefe. 2011. Modern B-tree techniques. *Found. Trends Databases* 3, 4 (2011), 203–402. https://doi.org/10.1561/1900000028

[14] Goetz Graefe and P.-A. Larson. 2001. B-tree indexes and CPU caches. In *Proceedings of the 17th International Conference on Data Engineering.* 349–358. https://doi.org/10.1109/ICDE.2001.914847

[15] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

[16] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Info. Syst.* 20, 2 (Apr. 2002), 192–223. https://doi.org/10.1145/506309.506312

[17] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. 2011. 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*. 229–232. https://doi.org/10.1145/1963192.1963296

[18] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A high-performance, distributed main memory transaction processing system. In *Proceedings of the VLDB Endowment*. 1496–1499. https://doi.org/10.14778/1454159.1454211

[19] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[20] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 339–350. https://doi.org/10.1145/1807167.1807206

[21] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 691–706. https://doi.org/10.1145/2723372.2746480

[22] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In *Proceedings of the 8th International Workshop on Data Management on New Hardware*. 16–23. https://doi.org/10.1145/2236584.2236587

[23] András Kovács and Tamás Kis. 2004. Partitioning of trees for minimizing height and cardinality. *Inform. Process. Lett.* 89, 4 (2004), 181–185. https://doi.org/10.1016/j.ipl.2003.11.004

[24] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic lock coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[25] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the IEEE 29th International Conference on Data Engineering*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[26] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*. https://doi.org/10.1145/2933349.2933352

[27] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-tree: A B-tree for new hardware platforms. In *Proceedings of the IEEE 29th International Conference on Data Engineering*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[28] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.* 8, 11 (July 2015), 1298–1309. https://doi.org/10.14778/2809974.2809990

[29] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196. https://doi.org/10.1145/2168836.2168855

[30] Donald R. Morrison. 1968. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. https://doi.org/10.1145/321479.321481

[31] Stefan Nilsson and Matti Tikkanen. 1998. Implementing a dynamic compressed trie. In *Proceedings of the 2nd International Workshop on Algorithm Engineering (WAE'98)*. 25–36.

[32] Jun Rao and Kenneth A. Ross. 1999. Cache conscious indexing for decision-support in main memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*. 475–486. Retrieved from http://www.vldb.org/conf/1999/P7.pdf.

[33] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 475–486. https://doi.org/10.1145/342009.335449

[34] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*. 52–60. https://doi.org/10.1145/1565694.1565705

[35] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 18–32. https://doi.org/10.1145/2517349.2522713

[36] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. 2018. Building A Bw-tree takes more than just buzz words. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196895

[37] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. F. Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *Proceedings of the IEEE 33rd International Conference on Data Engineering*. 119–122. https://doi.org/10.1109/ICDE.2017.54

[38] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. 2014. Adapting tree structures for processing with SIMD instructions. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*. 97–108. https://doi.org/10.5441/002/edbt.2014.10

[39] Huanchen Zhang, David G. Andersen, Michael Kaminsky, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, and Kimberly Keeton. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196931

[40] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the International Conference on Management of Data*. 1567–1581. https://doi.org/10.1145/2882903.2915222

[41] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct range filters. *ACM Trans. Database Syst.* 45, 2 (2020), 5:1–5:31. https://doi.org/10.1145/3375660

[42] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 145–156. https://doi.org/10.1145/564691.564709

Received March 2021; revised November 2021; accepted December 2021