HLN-Tree: A Memory-efficient B+-Tree with Huge Leaf Nodes and Locality Predictors

ANDRÉ BRINKMANN, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany
REZA SALKHORDEH, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany
FLORIAN WIEGERT, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany
PENG WANG, Huawei Technology, Hong Kong, China
YAO XIN, Huawei Technology, Hong Kong, China
RENHAI CHEN, Huawei Technology, Hong Kong, China
HUANG KEJI, Huawei Technology, Hong Kong, China
GONG ZHANG, Huawei Technologies Co Ltd, Shenzhen, China

Key-value stores in Cloud environments can contain more than 2^{45} unique elements and be larger than 100 PByte. B^+-Trees are well suited for these larger-than-memory datasets and seamlessly index data stored on thousands of secondary storage devices. Unfortunately, it is often uneconomical to even store all inner tree nodes in memory for these dataset sizes. Therefore, lookup performance is affected by the additional IOs for reading inner nodes.

This number of inner nodes can be reduced by increasing the size of leaf nodes. We propose HLN-Trees, which support huge leaf nodes without increasing the IO sizes for individual index operations. They partition leaf nodes in arrays of independent subnodes and combine ideas from BD-trees with rebalancing, learning key deviations, and storing locality predictors. HLN-Trees have been initially designed for uniform random key distributions and support arbitrary key distributions through an additional layer of hashing in leaf nodes. HLN-Trees decrease the number of inner nodes by up to 256× for uniform random key distributions and by 16× to 64× for arbitrary ones compared to B^+-Trees, while keeping their performance at the same level even at high concurrency levels. We show analytically and through real-world and synthetic benchmarks that HLN-Trees also outperform state-of-the-art learned indexes for secondary storage.

CCS Concepts: • **Information systems** → **Cloud based storage**; **Data structures;**

Authors' Contact Information: André Brinkmann (Corresponding author), Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Rheinland-Pfalz, Germany; e-mail: brinkman@uni-mainz.de; Reza Salkhordeh, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, RLP, Germany; e-mail: rsalkhor@uni-mainz.de; Florian Wiegert, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Rheinland-Pfalz, Germany; e-mail: fwiegert@students.uni-mainz.de; Peng Wang (Corresponding author), Huawei Technology, Hong Kong, Hong Kong, China; e-mail: wangpeng423@huawei.com; Yao Xin, Huawei Technology, Hong Kong, Hong Kong, China; e-mail: yao.xin1@huawei.com; Renhai Chen (Corresponding author), Huawei Technology, Hong Kong, Hong Kong, China; e-mail: chenrenhai@huawei.com; Huang Keji, Huawei Technology, Hong Kong, Hong Kong, China; e-mail: huangkeji@huawei.com; Gong Zhang, Huawei Technologies Co. Ltd., Shenzhen, Guangdong, China; e-mail: nicholas.zhang@huawei.com.

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 1553-3077/2025/01-ART00
https://doi.org/10.1145/3707641

Additional Key Words and Phrases: SSDs, index data structures, learned indexes

ACM Reference Format:
André Brinkmann, Reza Salkhordeh, Florian Wiegert, Peng Wang, Yao Xin, Renhai Chen, Huang Keji, and Gong Zhang. 2025. HLN-Tree: A Memory-efficient B+-Tree with Huge Leaf Nodes and Locality Predictors. *ACM Trans. Storage* 21, 2, Article 00 (January 2025), 27 pages. https://doi.org/10.1145/3707641

1 Introduction

Academic discoveries and innovations in industry are increasingly based on analyzing huge datasets. Accessing specific elements in these datasets requires indexes that can efficiently and unambiguously find previously stored elements. Designing indexes involves several tradeoffs, and their optimization criteria are application dependent. Following, we will optimize ultra-large indexes, where it is only possible to keep a very small fraction of the index in memory, as already storing pointers to data elements exceeds the available memory. Insert and lookup performance are then limited by the number of accesses to secondary storage devices, whereas walking the in-memory part of the index plays a minor role [2, 3, 26, 51].

B$^+$-Trees can operate as hybrid in-memory and on-disk data structures and are often used to index larger-than-memory datasets because they efficiently support point and range queries [5]. The node size b of a B$^+$-Tree is a multiple of a disk block, so nodes can easily persist. B$^+$-Trees can store either fixed-size or variable-size keys and values, and they ensure for every node except for the root node that it is at least half full. B$^+$-Trees have a height that is logarithmic in the number of elements stored in the tree. The leaf node size influences the memory requirements and performance of a B$^+$-Tree. Huge leaf nodes lead to fewer inner nodes, whereas smaller nodes can be read and updated faster.

Our work increases the leaf node size of B$^+$-Trees and was initially motivated by two scale-out Cloud applications that generate keys following a uniform random distribution[1] with limited access locality. The first use case is a Cloud backup environment that coordinates backups between clients to simplify server-side data deduplication [22]. Each client inserts new chunks during a backup run in the order of the chunks' fingerprints. The server can then reduce the number of lookup IOs by batching fingerprint lookups using range scans. Additional point queries are required when reconstructing backups and when reading data blocks that cannot be resolved by some form of locality caching [37, 52]. The second use case is an object store that assigns the most significant bits of an object id based on domains and that picks less significant bits randomly to overcome costly coordination between servers (see, e.g., Seagate Motr [38]). Range scans are required to collect information based on administrative domains. The index then stores the object IDs and tuples that define the corresponding **object storage devices (OSDs)** and locations on the OSDs.

The backend indexes of both applications can scale beyond 2^{45} elements, and the potentially huge number of tenants makes their design very cost sensitive. A B$^+$-Tree indexing an object store with 2^{45} elements and a key and pointer size of 64 bits, e.g., requires 6.3 TByte of memory to store its inner nodes when using 4 kB leaf nodes. For these scale-out scenarios, it is therefore uneconomical to store all inner nodes in memory for standard leaf node sizes, so lookup performance is typically degraded by additional IOs to inner nodes. However, memory requirements to store all inner nodes can be reduced to 395 GB for 64 kB leaf nodes and less than 24 GB for 1 MB leaf nodes, allowing all inner nodes to be stored in memory and increasing performance by requiring fewer I/O operations.

[1]The probability of a key being generated as the next key for a uniform random key distribution is the same for every possible key from the key space.

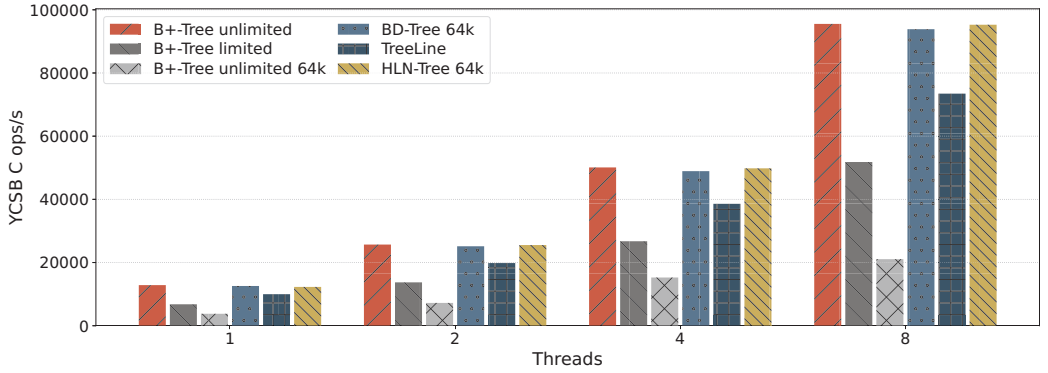


Fig. 1. YCSB C throughput for different leaf nodes.

Figure 1 shows the impact of caching inner nodes on lookup performance for a uniform at random key distribution. B$^+$-Tree unlimited has a leaf node size of 4 kB, an unlimited cache size for inner nodes, and a very small cache for leaf nodes. Lookups typically induce only a single 4 kB leaf node access, which is the minimum number of IOs possible in this setting. B$^+$-Tree limited with the same leaf node size is memory limited and can only cache 1/16th of all inner nodes, leading to two SSD accesses per lookup in most cases, and reducing throughput by 46%. Increasing the size of leaf nodes to 64 kB and then again caching all inner nodes for B$^+$-Tree unlimited 64k further decreases the throughput to one-fourth of the best possible performance, as the IO sizes for loading leaf nodes also increase by a factor of 16.

This article presents a new concurrent B$^+$-Tree design for fixed-size **key–value (kv)** pairs called ***Huge Leaf Node Tree* (HLN-Tree)**, which combines the performance of small leaf nodes with the memory efficiency of huge ones. HLN-Tree is optimized for hybrid storage, allowing the tree to cache all inner nodes in memory by significantly reducing their number, while storing most leaf nodes on SSDs. The main advantage of HLN-Trees over B$^+$-Trees is therefore that they can either reduce the IO amplification for lookups, since they do not need to load any inner nodes from backend storage, or they can significantly reduce the cost of the memory required to cache all inner nodes. In addition, HLN-Trees retain (almost) all of the other properties of B$^+$-Trees.

HLN-Trees divide large leaf nodes into equally sized arrays of smaller subnodes. This is similar to the *Bounded Disorder Access Method* (BD-Trees) [28, 30] that assigns the same key address range to each of its subnodes. The main drawback of BD-Trees is their storage utilization, which can be as low as 45% and negatively correlates with the number of subnodes and the size of kv pairs (see [4] and Section 4.2). The benefits from needing less main memory are therefore nullified by higher storage costs for BD-Trees.

HLN-Trees overcome the drawbacks of BD-Trees by rebalancing leaf nodes in case of full subnodes. They learn key range deviations between subnodes and store them as locality predictors called hints inside the last level of inner nodes. An inner node can then *guess* the correct subnode in a leaf node. Depending on the number of hint bits, the hit rate of guessing the correct subnode can be higher than 99%, whereas a miss simply involves an additional IO. HLN-Trees directly support fixed-size kv pairs and uniform random key distributions and are well suited for our deduplication and object storage workloads. HLN-Trees therefore trade some IO operations for reorganizing the tree and updating the hint mechanism to reduce the memory size of inner nodes.

Figure 1 shows that HLN-Tree using 64 kB leaf nodes achieves 99.3% of the performance of an unlimited B$^+$-Tree with small leaf nodes, while only requiring an inner node cache size that is equal to a B$^+$-Tree with 64 kB leaf nodes. HLN-Trees are slightly faster than BD-Trees at much lower

storage costs. HLN-Tree leaf nodes can even be scaled up to 1 MB without negative performance impact.

Our evaluation in Section 4 shows that HLN-Trees work very well for the uniform random key distribution of our target application areas. However, the evaluation also shows that they are sensitive to skewed key distributions that are dominant in many other real-world applications. We have therefore extended possible application domains for HLN-Trees by introducing HLN_{FNV}-Trees, which support arbitrary key distributions by an additional hashing layer inside leaf nodes. HLN_{FNV}-Trees efficiently support leaf node sizes of 64 kB to 256 kB, depending on the requirements on range-scan throughput.

HLN-Trees are related to learned indexes, as both learn properties of the key distribution and aim to decrease the index size [10, 12]. Learned indexes allow a prediction error of up to ϵ entries, whereas their size decreases by $1/\epsilon^2$ [11]. We show that HLN-Trees can be significantly smaller than learned indexes for secondary storage. The comparison is based on an analytical evaluation and a direct comparison with the performance and storage consumption of Treeline, a learned index for secondary storage [46].

The *main contribution* of this article is the design and analysis of HLN-Trees, which significantly improve BD-Trees by rebalancing and learning to provide an index structure that is memory *and* storage efficient. HLN-Trees efficiently scale *leaf node sizes up to 1 MB* for uniform random key distributions and achieve a performance very close to B^+-Trees with 4 kB leaf nodes and unlimited inner node caches. For arbitrary key distributions, HLN-Trees decrease the number of inner nodes by 16× to 64×, depending on the required range-scan performance. The article shows the *surprising result* that HLN-Trees can be even more memory efficient and faster than dedicated learned indexes for secondary storage.

The remainder of the article starts with the HLN-Tree design in Section 2 and its implementation in Section 3. The evaluation in Section 4 compares HLN-Trees with standard B^+-Trees, BD-Trees, TreeLine, RocksDB, and learned indexes in general. Related work is discussed in Section 5 and a conclusion is provided in Section 6.

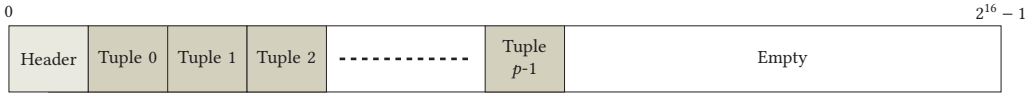
2 HLN-Tree Architecture

This section describes the HLN-Tree design. Splitting huge leaf nodes into subnodes is based on BD-Trees (Section 2.1). HLN-Trees use rebalancing to handle overflowing subnodes (Section 2.2). Hinting learns the resulting access deviations to keep the number of accesses per lookup very close to one (Section 2.3). Stash nodes additionally help to keep storage utilization high (Section 2.4). HLN-Tree was originally designed for uniform random key distributions, while arbitrary key distributions are supported by additional hashing inside the leaf nodes (Section 2.5).

We assume in the following that each leaf node occupies n disk blocks of size 4 kB and that each leaf node can store up to m fixed-size entries. We discuss huge leaf nodes w.l.o.g. for $n = 16$ and 8 byte keys and 8 byte values, so that the subnode header and 255 kv pairs fit into a subnode and up to 4,080 kv entries fit into a leaf node. We also assume that inner nodes of BD-Trees and HLN-Trees are managed identically to inner nodes of B^+-Trees. The `lookup()` function for a key in the inner nodes of BD-Trees and HLN-Trees is therefore also identical to standard B^+-Tree lookups.

2.1 BD-Trees—The Bounded Disorder Access Method

The ability to increase the leaf node size of standard B^+-Trees is limited by their node layout. A B^+-Tree leaf node stores as its first entry a header including its node id and the number of stored elements. The header is followed by a list of tuples consisting of $p \leq m$ kv pairs (see Figure 2(a)). These tuples are sorted according to their keys so that it is possible to use binary search to find



(a) Standard leaf node design

(b) BD-Tree and HLN-Tree leaf node design. The number of tuples in each subnode is limited to 3 instead of 255 in this sketch to keep the figure small.

Fig. 2. 64 kB B^+-Tree, BD-Tree, and HLN-Tree leaf node layouts when storing p elements.

a kv pair in at most $O(\log_2 p)$ accesses. The drawback of this design is that many entries must be moved for each insert, leading to an increased write amplification. Bigger leaf nodes also lead to an increased read amplification, because each leaf node lookup has to load the complete leaf node from disk (different layouts are possible with similar drawbacks).

BD-Trees increase the size of leaf nodes and partition them into equal-sized arrays of n subnodes each [28, 30], where each subnode occupies one disk block.[2] A subnode is managed mostly independently and includes its own header that stores its number of elements, the leaf node id, and the subnode id. Compared to a B^+-Tree with a leaf node size equal to the subnode size, these additional headers are not decreasing storage efficiency.

Indexing into the subnode array is an integer division of the key relative to the beginning of the array divided by the leaf node range. A BD-Tree node i therefore assigns the same key range $\frac{(key_i^{max} - key_i^{min})}{n}$ to each subnode, where key_i^{min} is the minimum key and key_i^{max} the maximum key stored in leaf node i. After locating the correct leaf node, `lookup()` computes Equation (1) to uniquely identify the subnode j for a key and then searches for the key inside this subnode:

$$j = \lfloor n \cdot \left(key - key_i^{min}\right) / \left(key_i^{max} - key_i^{min}\right) \rfloor. \tag{1}$$

In this section, we assume that keys stored in leaf nodes follow a uniform random distribution. We will lift this requirement later. Random distributions do not perfectly balance the load, and some subnodes receive more keys than others. It holds for the maximum number of keys M assigned to one subnode with high probability [41]:

$$M > \frac{m}{n} + \sqrt{2 \cdot \frac{m}{n} \log n}. \tag{2}$$

Computing Equation (2) for $n = 16$ subnodes and up to 4,080 kv pairs per leaf node results in a maximum filled subnode that must store at least

$$M > \frac{4,080}{16} + \sqrt{2 \cdot \frac{4,080}{16} \log 16} \approx 300$$

entries, which do not fit into it. BD-Trees therefore introduce overflow subnodes to store kv pairs that do not fit into their assigned subnodes, to delay the time before a node needs to be split. A kv pair is stored either in the subnode assigned by Equation (1) or in the overflow node, and BD-Tree lookups need at most two IOs to find it [30].

[2]It is in principle possible for a subnode to be larger than a single disk block. However, this would increase the read and write amplification of BD-Trees, so we do not consider such settings in the following.

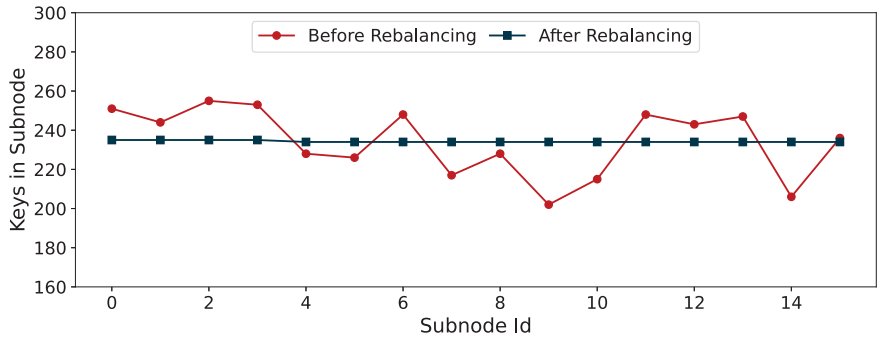


Fig. 3. Keys inside subnodes before and after rebalancing.

Perfect load balancing would ensure that $M \leq \lceil \frac{m}{n} \rceil$. In this case, each node would receive the same number of elements ±1. The absolute deviation from this optimum is given by the term $\sqrt{2 \cdot \frac{m}{n} \log n}$, and we define the relative deviation δ as

$$\delta > \frac{\sqrt{2 \cdot \frac{m}{n} \log n}}{m/n} = \frac{\sqrt{2 \cdot \log n}}{\sqrt{m/n}} = \sqrt{2 \cdot \frac{n}{m} \cdot \log n}. \tag{3}$$

The relative load deviation δ calculates the fraction of additional kv pairs relative to a perfectly balanced load that must be stored in the most heavily loaded subnode of a leaf node. Equation (3) shows that δ increases for smaller m and therefore for larger kv pairs. For fixed-size kv pairs, the pressure on the most fully loaded node also increases with more subnodes n. In this case, each subnode can store up to a constant number of elements c_{max}, and it follows that, for a fully loaded leaf node, $m = c_{max} \cdot n$. In this case, Equation (3) resolves to $\delta = \sqrt{2 \cdot \frac{\log n}{c_{max}}}$, which grows in n.

BD-Trees therefore work well for small n and huge m. However, Equation (3) reveals that the relative load deviation increases for bigger kv pairs and that the pressure on the overflow subnode also increases for more subnodes n. Finally, the number of overflow subnodes cannot be increased without introducing additional IOs. BD-Trees therefore only work in very restricted settings (see Section 4.2).

2.2 Rebalancing Subnodes and Its Impact on Lookups

HLN-Tree introduces a rebalancing mechanism that allows storing more elements inside subnodes before splitting a leaf node and that can be conceptually compared with rehashing operations in hash maps (see, e.g., [29]). Rebalancing is triggered when a subnode becomes full after inserting a new kv pair. Rebalancing first loads all subnodes into memory. It then evenly redistributes their entries over all subnodes if the number of entries is below a *rebalancing threshold*. Otherwise, the leaf node is split. Following, this rebalancing threshold is given as a percentage of the maximum leaf node fill grade.

Figure 3 shows an example distribution of the keys assigned to each subnode before and after rebalancing. In this example Subnode 2 stores 255 entries after an `insert()` and triggers rebalancing, while some subnodes still have up to 50 empty slots. After rebalancing, the difference in fill levels is bounded by one.

Rebalancing introduces new challenges. First, the operation is expensive because all the subnodes that make up a leaf node must be read from and written to storage. We will show in Section 4.6 that this number of additional IOs remains small as long as the rebalancing threshold

Table 1. Example Key Range Distribution after Rebalancing

Id	Range	Id	Range	Id	Range	Id	Range
0	0.0688	4	0.0632	8	0.0663	12	0.0610
1	0.0560	5	0.0674	9	0.0664	13	0.0586
2	0.0692	6	0.0556	10	0.0570	14	0.0616
3	0.0649	7	0.0635	11	0.0600	15	0.0605

ALGORITHM 1: Lookup inside a leaf node

```
1: procedure LOOKUP(key, key^min, key^max)
2:     subNode = guessSubNode (key, key^min, key^max)
3:     subNode = checkSubNode (key, subNode)
4:     pos = findKeyPos (key, subNode)
5:     return (subNode, pos)
```

is less than 99% for small kv pairs, while we need to adjust it and introduce additional techniques for larger kv pairs.

Second, calculating the subnode belonging to a key according to Equation (1) can now predict a subnode that does not store the sought key. Equation (1) assumes that each subnode is responsible for $1/n$ of the key range assigned to a leaf node, which is 0.0625 for the example of 16 subnodes. Table 1 shows the key ranges for different subnodes after a rebalancing step and that these key ranges may differ from the assigned fraction of the key space.

Lookups therefore can only use Equation (1) to make an educated guess about the correct subnode for a key, and the corresponding function call is denoted as `guessSubNode()` (see Algorithm 1). `checkSubNode()` then loads the guessed subnode into memory and checks whether the sought key is part of it by checking its minimum and maximum keys. If this is the case, `checkSubNode()` simply returns the subnode id. Otherwise, it loads the subnode left or right of the current subnode, depending on whether the sought key is smaller or larger than the key range of this subnode. The function continues to load additional subnodes until the correct one is found. Afterward, the call to `findKeyPos()` either returns the key position in the subnode or returns the position of the next bigger key. If the key is bigger than the biggest key stored in the subnode, `findKeyPos()` returns the number of elements currently being stored in the subnode.

The `lookup()` function needs to know the left and right key boundaries of a leaf node in order to call `guessSubNode()`. These boundaries cannot be inferred by loading the leaf node into memory, as this would immediately trigger two costly IOs to load the first and last subnode of the leaf node. However, the right boundary of the leaf node is stored in the leaf node description of its inner node,[3] while the left boundary can mostly be derived from the previous element in the inner node. Only the left boundary of the first element stored in an inner node must be derived by a recursive call to ancestor nodes, which does not trigger IOs since all inner nodes are cached.

`lookup()` is the foundation for `insert()`, `batch_insert()`, `update()`, and `scan()`. `scan()`, e.g., calls `lookup()` to find the subnode storing the first key that is greater than or equal to the searched start key and then iterates through the subnodes of a leaf (and its neighbors if necessary) until the required number of elements has been collected. Batch inserts accept a sorted batch of kv pairs. The difference between `insert()` and `batch_insert()` is that `batch_insert()` only writes

[3]Some B$^+$-Tree implementations do not store the right boundary of the rightmost element of an inner node since it is not required to select the correct key destination.

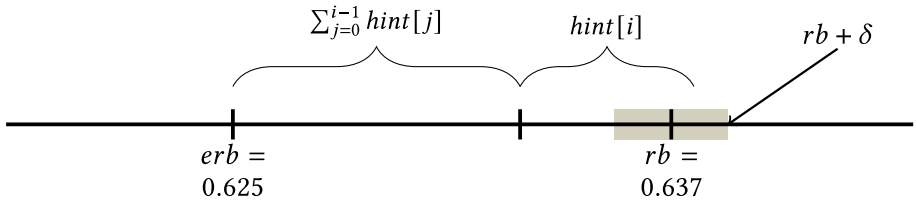


Fig. 4. Hint calculation.

ALGORITHM 2: Calculate Locality Predictors for Leaf Node

```
1:  procedure calcHint(id)
2:      range = key_id^max − key_id^min
3:      hintSum = 0; i = 0
4:      while (i < n) do
5:          rb = (key_id^max(i) − key_id^min)/range
6:          erb = (i + 1)/n
7:          Δ = ||(erb + hintSum − rb)|| + δ
8:          steps = min(Δ/stepSize, stepsMax)
9:          if (erb + hintSum < rb) then
10:             hint[i] = +steps
11:         else
12:             hint[i] = −steps
13:         hintSum = hintSum + hint[i] ∗ stepSize
14:         i = i + 1
```

back a leaf node to persistent storage if the next kv pair in the batch is for the next leaf node or if the last kv pair has been inserted. A batch can span multiple target leaf nodes.

HLN-Tree aims to reduce the number of IOs. The function `guessSubNode()` does not induce IOs, while `checkSubNode()` must load at least this guessed subnode and additional ones if the initial guess has been wrong. Section 4 will show that these additional IOs can almost always be limited to a single IO. However, an additional IO per read would already double the number of IOs for a lookup, so in the following we introduce locality predictors to help learn the key distribution and store them as hints in inner nodes.

2.3 Hinting Mechanism

Our locality predictors (or short hints) learn the deviation of the current key distribution from a perfectly even one at the start positions of subnodes to minimize additional IOs within `checkSubNode()`. Current learned indexes, on the other hand, also guarantee an error bound for keys stored in the middle of a subnode. This additional error bound of learned indexes does not improve performance in our setting, where performance is dominated by the number of IOs.

Key ranges change slightly when a key is inserted at the end of a subnode, while significant changes only occur during rebalancing or split operations. The deviation of each subnode from the even distribution can then be computed without additional IOs, since the entire leaf node must be loaded for these operations anyway. Therefore, we only recompute hints after rebalancing or split operations.

Algorithm 2 describes the hint computation for a leaf node id, and we use Figure 4 to explain its different steps based on subnode 10 out of 16. We greedily compute a $hint[i]$ for each subnode i in a while loop starting with the first subnode. The right boundary rb of subnode i is calculated by projecting its maximum key $key_{id}^{max}(i)$ onto the $[0, 1)$-interval (line 5) by dividing it by the leaf

ALGORITHM 3: Guess Subnode Id Using Hints

```
1:  procedure GUESSSUBNODE(key, key^min, key^max, hint)
2:      range = key_id^max − key_id^min
3:      target = (key − key_id^min)/range
4:      hintSum = 0
5:      i = 0
6:      while (i < (n − 1)) do
7:          hintSum = hintSum + hint[i] ∗ stepSize
8:          erb = (i + 1)/n
9:          if (erb + hintSum > target) then
10:             return i
11:         i = i + 1
12:     return i
```

node $range = key_{id}^{max} - key_{id}^{min}$ (line 2). In our example, rb is 0.637. The expected right boundary for a subnode is defined by Equation (1) as $erb = (i + 1)/n$ (line 6), which is 0.625 in Figure 4.

Each hint is a l-bit value, and these values are evenly assigned to the range between $-hint_{max}$ and $+hint_{max}$. $hint_{max}$ can be set automatically based on Equation (2). For example, for 16-byte kv pairs, $hint_{max}$ is approximately 20% of the expected subnode key range, which is 0.0125 in our example. The step size between two hint values depends on l and is set to $hint_{max}/(2^{(l-1)} - 1)$.

The difference between rb and erb is matched by the previous $hintSum = \sum_{j=0}^{i-1} hint[j] \cdot stepSize$ and $hint[i] \cdot stepSize$. We will see in the evaluation that the required for-loop to compute the difference between rb and erb slightly increases the latency for lookup requests. However, if we would simply store the difference between rb and erb for each subnode, then it would not be possible to bound $hint_{max}$ based on Equation (2), since differences can add up over multiple subnodes, and we would need more hint bits l.

We slightly overshoot the required Δ in line 7 of the algorithm by $\delta = stepSize/2$, again to account for the restricted precision of hint values. Finally, $hint[i]$ is calculated as the minimum of the number of steps required to match Δ and the maximum possible hint value in line 8 and then stored as either a positive or negative value.

The `lookup()`-function supporting hints works similarly to Algorithm 1 and only guessing the responsible subnode is replaced by Algorithm 3. `guessSubNodeId()` again requires the sought key and the minimum and maximum key values key^{min} and key^{max}. Additionally, the hint array must be passed as an argument. The subnode guess can then be calculated by checking inside a for loop whether the projection of the key to the $[0, 1)$-interval $target$ (line 3) falls into the range of the subnode (line 9). After loading this subnode into memory, the `lookup()` function continues identically to Algorithm 1.

Lookups using hinting require the hints to be stored in the last level of inner nodes, which requires an additional $n \cdot l$ bits per leaf node. Hinting considerably saves memory as long as l is significantly less than the number of bits required to store a key and a pointer to a leaf node. We will show in the evaluation that $l \leq 4$ is typically sufficient to efficiently locate the correct subnode.

2.4 Stash Nodes

Rebalancing is efficient up to a threshold that depends on the kv size (see Section 4.6), whereas higher thresholds lead to excessive rebalancing. We therefore apply the concept of stash nodes to achieve a fill grade that is equal to the fill grade of a B$^+$-Tree. A stash node has the same layout as a standard HLN-Tree leaf node. Its first subnode is used as an overflow subnode, while the remaining subnodes (called packed subnodes) work according to the HLN-Tree concepts (see Figure 5). A leaf

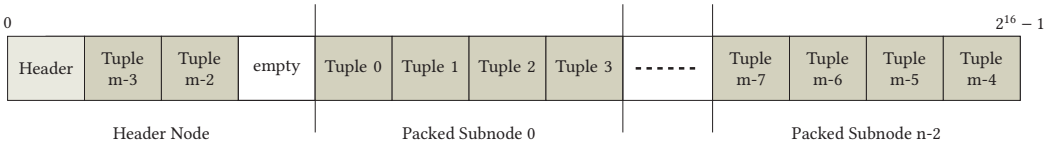


Fig. 5. Layout of stash nodes.

node is transformed into a stash node if rebalancing is triggered and its fill grade is beyond its rebalancing threshold. R

When a stash node is created, it receives a sorted array of entries, and its initialization starts by filling the $(n-1)$ packed subnodes evenly. If all packed subnodes can be completely filled, then the remaining entries are assigned to the overflow subnode. The initialization of a stash node finishes with the creation of hints for the packed subnodes. The overflow subnode can receive arbitrary keys during the following inserts and is not involved in the hint creation.

A key can be stored in a packed or in an overflow subnode. `lookup()` first calls `guessSubnode()` and `checkSubnode()` for the packed subnodes to find the responsible packed subnode for a key. It then returns this subnode and the key position if the key is stored inside it. Otherwise, it additionally checks whether the key is stored in the overflow subnode, inducing an additional IO.

`insert()` first calls `lookup()` to check if the key already exists. If this is not the case and the corresponding packed subnode is not yet full, then it stores the new kv pair in it. Otherwise, the kv pair is stored in the overflow subnode. `insert()` triggers rebalancing or splitting if the overflow subnode is full after an insert. It then loads all subnodes and merge-sorts the contents of the packed subnodes with the overflow buffer. The node is split into two standard HLN-Tree leaf nodes if the stash node is full after an `insert()`. Otherwise, its content is redistributed according to the create operation. `scan()` for stash nodes works similarly to `scan()` for standard HLN-Tree leaf nodes, but it additionally must load the overflow subnode.

Stash nodes are also used in hash maps to overcome load imbalances [24] and as overflow buffers in BD-Trees. BD-Trees nevertheless cannot redistribute in case of full overflow buckets, even if there is still space in other subnodes, leading to poor storage usage. Stash nodes instead guarantee that leaf nodes achieve the same fill grade as in B^+-Trees and that additional lookups only occur for highly loaded leaf nodes. Stash nodes also have similarities with *foster parents* in Foster B-Trees, which are overflowing nodes that temporarily act as parent nodes for a sibling node [17].

2.5 HLN-Tree for Arbitrary Key Distributions

The HLN-Tree design so far assumes a uniform random key distribution. Arbitrary key distributions could be supported by hashing keys before inserting them into the HLN-Tree. This key transformation would not impact point queries, but it would remove the opportunity to perform range queries. We have therefore designed the HLN_{FNV}-Tree, which couples the ideas of HLN-Tree with a **Fowler–Noll–Vo (FNV)** hash function `fvn()` inside leaf nodes. FNV hashing combines a magic prime number, an initial hash value, and the original key value in 8-bit chunks to generate a random, non-cryptographic hash number with a good dispersion [13].

`lookup()` inside inner nodes of the HLN_{FNV}-Tree works on the original keys k, whereas leaf nodes distribute their keys over their subnodes using the hashed keys `fvn(key)`. Leaf nodes only store the original keys k, while these keys are sorted inside the subnodes according to their hashed values. Rebalancing can then directly perform `memcpy()` operations.

The sorting order inside leaf nodes still allows point queries to use binary search on `fvn(key)`. `scan()`-operations, on the other hand, must load and sort all elements of a leaf node, as the result is spread over all subnodes. Range queries require a tradeoff between the leaf node size

and the expected number of elements returned by a range query. We will show that a good scan performance can be achieved up to 64 kB leaf nodes, whereas it decreases for bigger leaf nodes.

3 Implementation

HLN-Tree has been implemented in 6,500 lines of C++. The implementation of inner nodes follows a standard B^+-Tree with 4 kB inner nodes. Additionally, the last level of inner nodes is able to store hints. Leaf nodes support combinations of partitioned subnodes, stash nodes, and FVN hashing. B^+-Trees and BD-Trees used for comparison use the same code-base and only differ in the leaf node implementation and by not storing hints. The code includes a leaf node read cache and a read cache for last-level inner nodes. All upper levels of inner nodes are always cached. The cache sizes can be set individually, and each operation first checks if a leaf node or inner node must be evicted. Leaf node eviction uses clock page replacement to support temporal localities, and inner node eviction uses a round-robin list.

We also implemented a record cache for individual kv pairs. For skewed data distributions, the record cache is more memory efficient than the leaf node cache, because each leaf node typically also stores many cold kv pairs [31]. We have implemented the record cache using the BabyDBM in-memory database in the Tkrzw framework [18], which is based on a B^+-Tree.[4] When the record cache reaches its upper limit of cachable items, it will (by default) batch-evict 100 consecutive items. We have not provided specific protection against power outages to the in-memory record cache and use it in this article mostly to compare with the Treeline approach, which contains a similar, non-protected record cache.

We use `O_DIRECT` and `O_SYNC` to ensure that all writes persist before acknowledging them. We decided against an `mmap()`-based implementation, as it complicates the steering of cache sizes. We use vectored IO via `preadv` and `pwritev` to enable all tree implementations to perform fine-grained caching of 4 kB subnodes. Leaf and inner nodes can be stored either in individual files or on separate partitions.

Locking uses reader–writer locks, and each operation step-wise read-locks the path from the root to the leaf using lock coupling [21]. Operations changing leaf nodes additionally write-lock the leaf. Split operations can change the complete path from a leaf to the root and on-demand write-lock father nodes. To prevent deadlocks, paths from the root to leafs use try-locks and return already received locks if they cannot lock a node and try again later. Split operations and cache evictions are not allowed to fail and use standard locks [15]. Interestingly, updating hints after rebalancing is less expensive than splitting nodes, as hint updates only require, based on the inner node layout, a read-lock.

4 Evaluation

In this section we evaluate the memory and storage requirements for internal nodes of HLN-Trees, B^+-Trees, and BD-Trees. We then compare the memory overheads of HLN-Trees and learned indexes and present performance results for HLN-Trees, B^+-Trees, BD-Trees, RocksDB, and TreeLine. We also analyze the individual overheads of HLN-Trees to understand parameter tradeoffs.

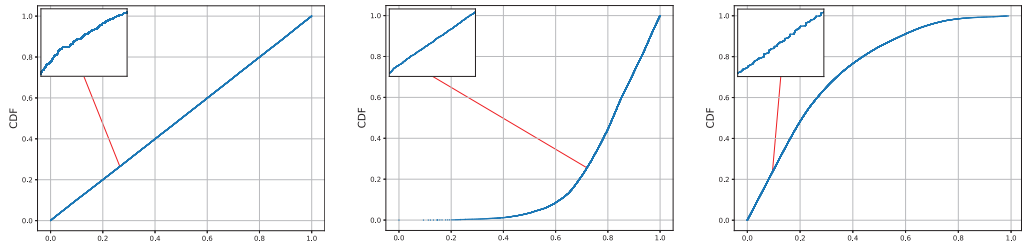
4.1 Test Environment and Data

All experiments used a server with one 10-core 2.5 GHz Xeon Gold 5212 processor, 192 GByte DRAM, and a Linux kernel on CentOS 8.2. The tests ran on a 1.92 TByte Samsung PM983 M.2 NVMe SSD with two equally sized partitions. HLN-Tree, B^+-Tree, and BD-Tree used the first partition

[4]We did not use Tkrzw as the basis for our HLN-Tree because it has scalability limitations due to the way it performs blocking structure modification.

Table 2. YCSB Configuration and Applications (from [9])

Workload	Operations	Application example
LOAD	Insert 100%	Initially loading db
YCSB A	Read 50% Update 50%	Session store recording recent user actions
YCSB B	Read 95% Update 5%	Photo tagging
YCSB C	Read 100%	User profile cache
YCSB D	Read 95% Insert 5%	User status updates
YCSB E	Scan 95% Insert 5%	Threaded conversation



(a) Facebook Dataset (b) Wikipedia Dataset (c) Amazon Dataset

Fig. 6. Cumulative distribution function of the real-world datasets.

for leaf nodes and the second for inner nodes. TreeLine and RocksDB stored all data on the first partition. All indexes were compiled with gcc 12.2 and optimization O3.

The benchmark framework was YCSB-cpp, which includes seven kv workloads and one workload loading data [9, 25]. We extended YCSB-cpp so that it can use synthetic and real-world key distributions. The evaluation assumes that the datasets are so huge that caching a huge fraction of leaf nodes is impossible. Most experiments used uniform access distributions, whereas YCSB C-Zipf uses a Zipf-distribution with a Zipf parameter of 0.99. YCSB E generates a uniform scan-length distribution with up to 1,000 kv pairs, spanning several 4 kB pages (see Table 2).

YCSB tests first loaded the elements using 8 threads and then performed 10 million operations for each benchmark and 1 to 8 threads. We used 8-byte keys and either 8-byte, 50-byte, or 100-byte values, whereas the implementation also supports bigger keys. We loaded 1 billion entries for 8-byte values and 100 million for 50- and 100-byte values to be able to compare many configurations.

We also ran three real-world traces (see Figure 6). The Facebook dataset includes a subset of Facebook user ids [14, 43]. Its overall **cumulative distribution function (CDF)** looks uniformly distributed, while each subrange contains many difficult-to-approximate jumps in the CDF. The Wikipedia dataset contains article edit timestamps with a CDF close to the CDF of a Poisson distribution, and the subranges are close to a uniform distribution. The Amazon dataset represents sales rank data for print and Kindle books on Amazon. Its overall CDF resembles a heavy-tailed Pareto distribution, while subranges of the CDF again are close to a uniform distribution. We used 200 million 8-byte keys from each dataset in a form provided by the SOSD benchmark for learned indexes and randomly generated an 8-byte value for each key [23].

A fair comparison between B$^+$-Tree, BD-Tree, and our HLN-Tree and the scaling of qualitative results is ensured by our caching, which limits the number of cached leaf nodes to 16 MB (less than 0,1% of the leaf nodes). The number of cached inner nodes has been limited in some B$^+$-Tree configurations to compare with HLN-Tree when using the same amount of memory.

The comparison with TreeLine is based on the original sources from [46]. TreeLine includes two main contributions: a fine-grained record cache instead of a page cache and page grouping to combine between 1 and 16 pages in one segment to speed up range scans. We only changed the parameter `records_per_page_goal` to 150 to accommodate the value size of 8 bytes. The record cache size has been kept at 1 million elements and has the same size of 16 MB as the HLN-Tree page cache. The comparison will show that TreeLine heavily benefits from the record cache, whereas the implementation of page grouping in TreeLine still seems experimental.

To better understand the impact of a record cache on HLN-Tree, we have also coupled HLN-Tree with a record cache. However, we want to focus on the performance of HLN-Tree when most kv pairs cannot be cached, so we have turned off the record cache in most settings. We only turned on the record cache for a direct comparison with TreeLine in Section 4.8 and then limited the size of our leaf node cache accordingly.

We also compare to RocksDB 7.9, which is based on an LSM-Tree design that distributes kv pairs over multiple levels [39]. We only changed the IO parameter to use direct IO compared to the standard configuration to overcome the effects of file system caching, which is not effective for ultra-huge databases. RocksDB supports Bloom filters for each level, which can significantly speed up lookup operations, since IO accesses are only necessary if a key is likely to be stored within a level. We did not enable Bloom filters in our tests because the amount of memory required for them becomes too large for the 2^{45} entries targeted in this article. On average, the Bloom filter requires about 10 bits for each entry in each level. Even if no Bloom filter is used in the lowest level of RocksDB, and assuming a standard growth factor of 10 between levels, the Bloom filters alone would require 2^{42} bytes = 4 TB. Our results are very similar to the results of previous studies, which have shown that RocksDB excels at inserts but is slower than trees for reads and scans [46].

The numbers of 4 kB read and write operations for HLN-Tree are based on internal counters and only account for operations to leaf nodes, which dominate the traffic to secondary storage. It has to be noted that HLN-Tree often batches 4 kB operations into bigger IOs for rebalancing and splits, so that the actual amount of operations might be smaller than the number of accounted IOs.

4.2 Memory and Storage Comparison with B$^+$-Trees and BD-Trees

HLN-Trees without support for stash nodes require the same storage capacity for leaf nodes as B$^+$-Trees using 4 kB leaf nodes, and HLN-Trees with stashed leaf nodes require the same storage capacity as B$^+$-Trees of the same leaf node size. We will now show that HLN-Trees require significantly less memory to cache all inner nodes than B$^+$-Trees by computing their memory requirements when storing 2^{45} 8-byte keys and values based on an average node fill grade of 70% [45].

We assume that nodes are uncompressed and that we must store for each child one 64-bit value that either stores the on-disk node id of the child or its memory representation, and a 64-bit key. The setting slightly deviates from the implementation of our B$^+$-Trees and HLN-Trees, as it stores a separate pointer to the on-disk location and the memory representation. However, these two 64-bit pointers can be reduced to a single one either by memory mapping the file (moving the overhead to Linux virtual memory) or by extending the tree logic without introducing additional I/Os, since the node id is redundantly stored in the child nodes.

Increasing the number of subnodes per leaf node in HLN-Trees linearly decreases the number of inner node entries. However, HLN-Trees store an additional l-bit hint for each subnode. One

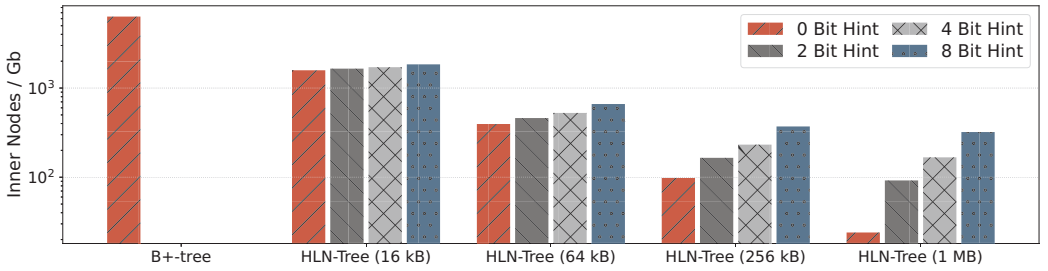


Fig. 7. Inner node memory required to store 2^{45} kv pairs.

entry for a 64 kB leaf and 2 hint bits therefore requires $196 + 16 \cdot 2 = 224$ bits, while one entry for a 256 kB leaf and 8 hint bits grows to 708 bits.

Figure 7 depicts the required **memory** to store inner nodes for 2^{45} 8-byte key and 8-byte value pairs on a logarithmic scale. It can be reduced from 6.3 TByte for a B$^+$-Tree with 4 kB leaf nodes to less than 167 GByte for an HLN-Tree with 1 MB leaf nodes and 4 hint bits, **reducing the memory footprint by nearly 40×**. Not using hinting even reduces the memory footprint by more than **256×** but has a slight impact on performance. Scaling leaf nodes beyond 1 MB has diminishing returns, as either inner nodes become dominated by the hint bits or when using no hints the number of wrong subnode guesses increases.

BD-Trees have a very low inner node memory footprint that is similar to HLN-Trees without hints. Nevertheless, indexes must be memory and storage efficient. We therefore compare the **storage efficiency** of HLN-Trees and BD-Trees with overflow subnodes [30]. Measurements at individual time steps depend on which fraction of nodes has been split recently. We therefore instead average over the fill grades of leaf nodes directly before they split. We performed measurements for 64 kB and 1 MB leaf nodes after 100 million inserts for 8-byte keys and either 8-byte or 56-byte values.

The leaf nodes of a B$^+$-Tree and of HLN-Trees with stash nodes enabled both split when they are completely full and both achieve 100% split fill grades. HLN-Trees without stash and a rebalancing threshold of 0.97 achieve the expected 97% fill grade for both node sizes and 8-byte values. It slightly decreases to 96% for 56-byte values, as the HLN-Tree leaf node without stashes stores metadata in each subnode, which reduces the number of elements per subnode by one compared to a B$^+$-Tree leaf node of the same size.

BD-Trees with 64 kB leaf nodes achieve a fill grade of 96% for 8-byte values. The fill grade drops to 92% for 1 MB leaf nodes. It further drops to 88 % for 64 kB leaf nodes and 56-byte values and 81% for 1 MB leaf nodes and 56-byte values. These measurements confirm previous work [4] and our postulations from Equation (2): the storage efficiency of BD-Trees decreases both for huge leaf node sizes and bigger value sizes, so that potential memory savings are outweighed by additional storage costs. We conclude that HLN-Trees are significantly more memory efficient than B$^+$-Trees and more storage efficient than BD-Trees.

4.3 Memory Comparison with Learned Indexes

Learned indexes apply machine learning to build a model for each tree node that approximates the CDF of the key distribution inside this node. The models approximate key positions and can reduce the number of memory accesses and key comparisons, both of which help improve performance. Additionally, the models can reduce index size and overall memory consumption.

Many learned indexes calculate the position of a key in the next level by using piece-wise **linear approximation functions (PLAs)** [10, 12]. PLAs can incorrectly predict the key location up to

Table 3. Comparison between PGM-index and HLN-Tree

Index	Additional IOs / %	Index size/bytes
PGM-index(4)	0.78	208,146,328
PGM-index(16)	3.14	16,319,144
PGM-index(32)	6.27	4,248,760
PGM-index(128)	25.10	273,840
HLNTree(8)	0.29	9,056,256
HLNTree(4)	0.87	4,698,112
HLNTree(2)	5.13	2,469,888
HLNTree(0)	19.33	548,864

a constant ϵ. In this case, the index simply searches the key within a length of $\pm\epsilon$ entries. This slack allows learned indexes to reduce the memory required for inner levels. In the following, we compare HLN-Trees with PGM-indexes, as their space complexity is optimal for learned indexes using PLAs and as the underlying PLAs are also used in TreeLine [46].

The PGM-index is a memory-based index and we therefore (1) estimate the number of additional IOs that would be induced by it for guessing wrong disk blocks based on ϵ and then (2) run its in-memory benchmark (see https://pgm.di.unipi.it/) for several ϵ to get corresponding index sizes. Assuming for (1) that each disk block can store $m = 255$ 8-byte keys and values, the percentage of incorrect computed disk blocks caused by the PGI index would be $(100 \cdot \epsilon)/(2 \cdot m)$, as the average error of the PLA function for uniform random key distributions is $\epsilon/2$ and as errors leading to additional IOs can occur on the ϵ left-most and right-most subnode positions.

Table 3 lists the index sizes and percentage of mispredictions leading to additional IOs for different configurations. The parameter for the PGM-index is ϵ and for HLN-Tree the number of hint bits. All HLN-Trees use 1 MB leaf nodes and a rebalancing threshold of 97%. All indexes store 1 billion 8-byte keys and values. HLN-Trees are much more efficient when a low misprediction rate is required. HLN-Tree(4), e.g., requires **44× less space** for its inner nodes than PGM-index(4) for a misprediction rate of less than 1%.

Learned indexes scale the required memory in $O(1/\epsilon^2)$ [11] and can decrease memory by increasing ϵ, e.g., from 4 to 32. In this case, HLNTree(2) is still **2× more space efficient** than the corresponding PGM-index(32). The PGM-index becomes more space efficient if we allow a high misprediction rate of 25%.

Section 4.7 will shown that the prediction quality of HLN-Trees decreases for larger kv pairs, leading to more additional reads. These additional reads also happen for learned indexes and the qualitative result of the comparison between HLN-Trees and learned indexes therefore also holds for larger kv pairs and smaller m. HLN-Trees are therefore more space efficient than learned indexes for uniform random key distributions if the performance bottleneck is the number of IOs, as HLN-Trees only must learn the deviation at the boundaries of leaf nodes, whereas learned indexes give guarantees over all kv pairs.

The number of IOs cannot be directly transformed into performance advantages. In the following, we will therefore compare with the TreeLine implementation, which is so far the only learned index for secondary storage [46]. Interestingly, TreeLine does not produce any additional IOs at the cost of not using the first and last ϵ entries in each subnode to ensure that mispredictions cannot occur. TreeLine therefore trades off between performance and storage efficiency. Comparing with TreeLine, we will show in Section 4.8 that the HLN-Tree design also has advantages over learned indexes for non-random key distributions.

4.4 Performance for Uniform Random Key Distributions

This section compares the performance of HLN-Tree with B$^+$-Trees, BD-Trees, TreeLine, and RocksDB for uniform random key distributions. Figure 8 shows YCSB benchmark results for 1 to 8 threads and 8-byte keys and values. We first compare B$^+$-Tree using 4 kB leafs with an unlimited inner node cache with B$^+$-Tree that can only cache as many nodes as required by an HLN-Tree with 64 kB leafs. The limited B$^+$-Tree nearly always reads one inner node and one leaf node from secondary storage for its operations, and its average throughput over all YCSB benchmarks is reduced by 45%.

Increasing leaf nodes of B$^+$-Trees decreases performance even more, as huge reads and writes are expensive. The throughput of a B$^+$-Tree with 64 kB leafs and an unlimited inner node cache drops by 70% compared to an unlimited B$^+$-Tree with 4 kB leafs. An exception is the YCSB E performance, as B$^+$-Tree with 64 kB leaf nodes benefits here from prefetching data.

Averaged over all YCSB-benchmarks and the load benchmark from Figure 9, the performance of the BD-Tree with 64 kB leaf nodes is 98% of the performance of B$^+$-Trees. BD-Trees therefore provide a competitive performance and small memory consumption for caching inner nodes at the cost of a lower storage efficiency.

Figure 8 also shows the results for HLN-Trees with leaf node sizes of 64 kByte, 256 kByte, and 1 MB with 4 hint bits per subnode and a rebalancing threshold of 97%. All HLN-Trees provide a competitive performance compared to an unlimited B$^+$-Tree with 4 kB leaf nodes. Averaged over all YCSB benchmarks for 1 to 8 threads and Load for 8 threads, all HLN-Trees are within 1% of the performance of the unlimited B$^+$-Tree. Averaging over 1 to 32 threads, HLN-Tree performance is between 95% and 101% of the B$^+$-Tree performance. YCSB E benefits from larger HLN-Tree leaf nodes for higher concurrency levels, whereas B$^+$-Tree is then faster for YCSB C-Zipf, as the current implementation of HLN-Tree still includes some non-optimized functions, leading to slowdowns for higher concurrency levels.

We were unable to run YCSB A for TreeLine, as concurrent updates and reads regularly led to deadlocks. TreeLine benefited from its record cache for YCSB C-Zipf, whereas the record cache has been too small to improve non-skewed benchmarks. TreeLine was very slow for YCSB E, indicating a not-yet-optimized implementation. Furthermore, TreeLine needed 80% more storage for its leafs than HLN-Trees, which might be based on the leaf node layout.

RocksDB excels at inserts because it first collects many new kv pairs in memory before persisting them, and because it follows the upsert-semantic and does not check whether a key already exists in the database before inserting it. It is therefore up to 7× faster than HLN-Tree when loading data using 8 threads (see Figure 9). However, RocksDB's lookup performance lags significantly behind the other indexes because the datasets have limited locality, so RocksDB has to access many levels before finding a kv pair. For example, HLN-Tree is 1.8× faster for YCSB C lookups and 2.5× faster for YCSB E range scans when using 8 threads (see Figure 8).

The lessons learned are that HLN-Tree provides the same performance as standard B$^+$-Trees at a much lower memory cost. In addition, HLN-Trees are faster than TreeLine in scenarios where TreeLine cannot benefit from its record cache, which is orthogonal to HLN-Tree and only helps for non-scale-out workloads. Compared to RocksDB, HLN-Tree is better for scenarios with many lookups and range scans, while RocksDB is better for insert-heavy workloads. The effect of using a record cache on top of HLN-Tree, also compared to RocksDB, is evaluated in Section 4.8.

4.5 Influence of the Number of Hint Bits

More hint bits allow `guessSubNode()` to better predict the correct subnode, since rounding errors based on a limited hint resolution become smaller. Also, these rounding errors have a greater impact on larger leaf nodes, as the fraction of the key space that a subnode is responsible for

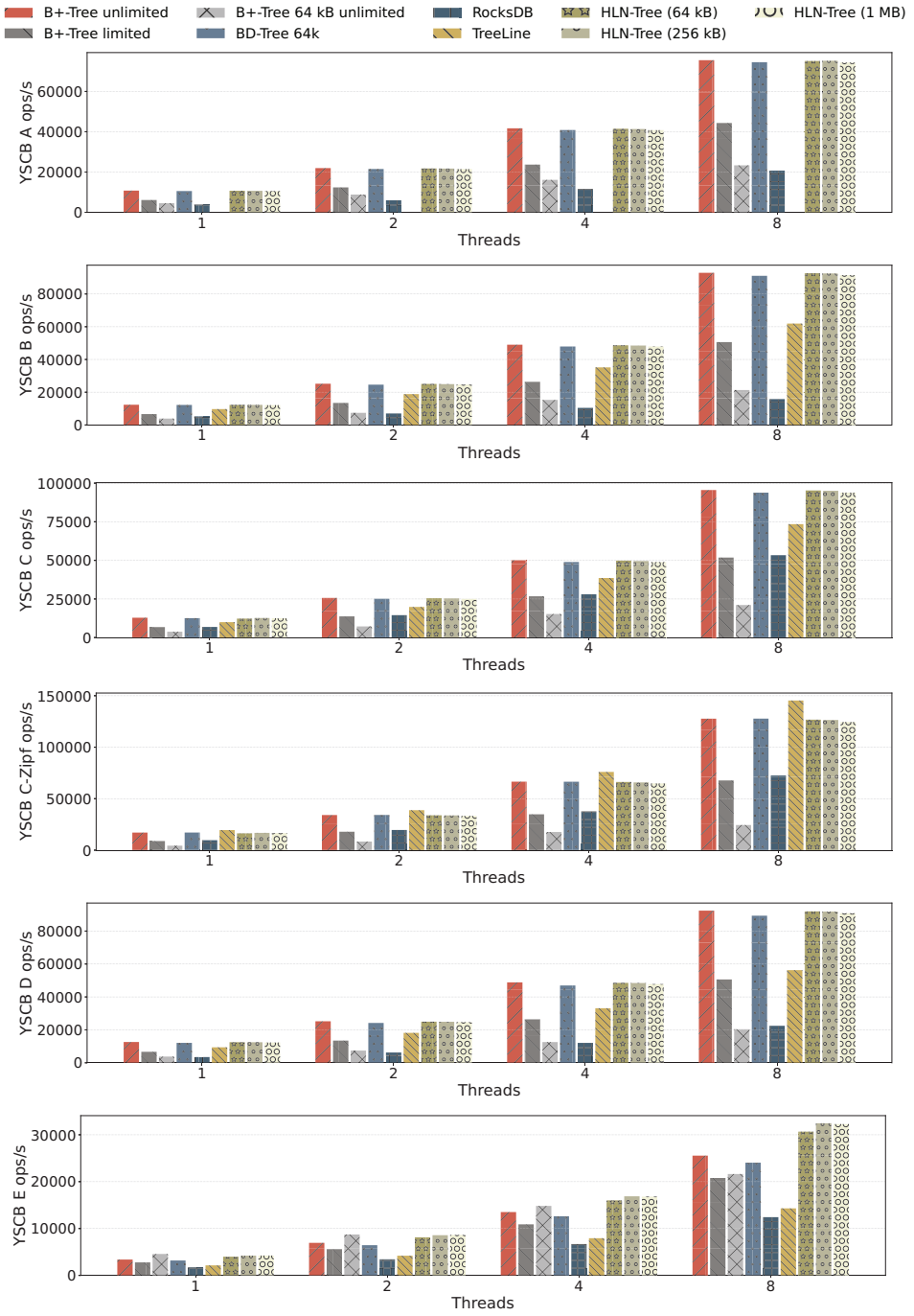


Fig. 8. YCSB throughput in operations/second for 8-byte keys and values and uniform random key distributions for 1 to 8 threads. All HLN-Trees used 4 hint bits.

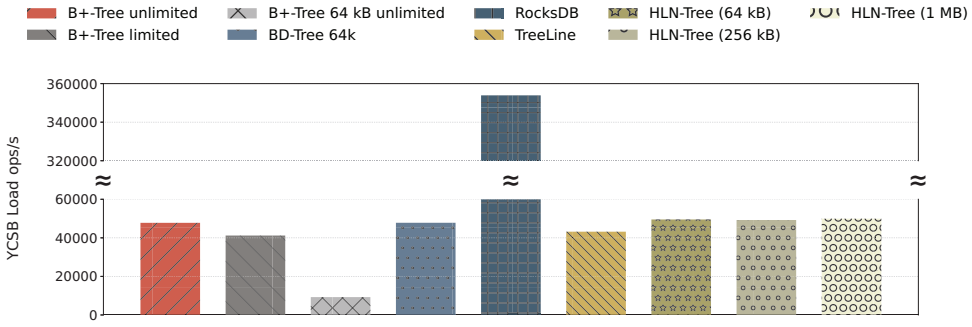


Fig. 9. Load performance using 8 threads. All HLN-Trees used 4 hint bits.

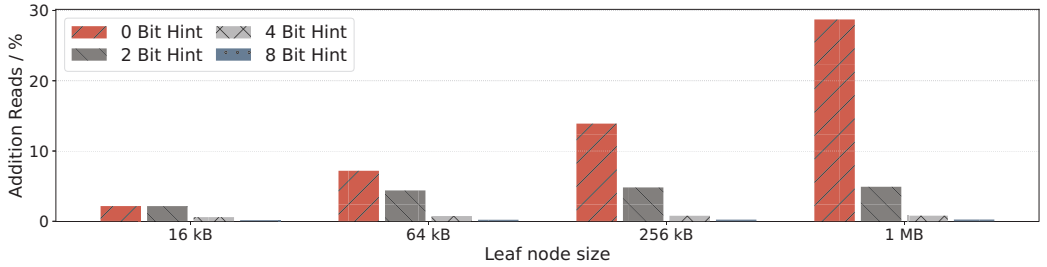


Fig. 10. Additional reads to locate correct subnode depending on the leaf node size.

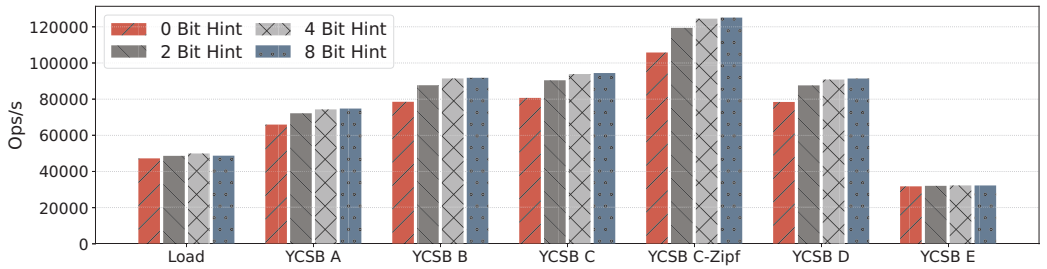


Fig. 11. HLN-Tree throughput depending on hint bits.

becomes smaller. Figure 10 shows the percentage of additional reads caused by incorrect subnode guesses for leaf node sizes from 16 kB to 1 MB, a reblancing threshold of 0.97, and 0, 2, 4, and 8 hint bits when running the YCSB C workload for 10,000,000 lookups. The figure shows that the number of wrong subnode guesses without hint bits is very low (2.1%) for 16 kB leaf nodes, while no hint bits lead to 7.2% additional reads for 64 kB leaf nodes, which increases to 28.7% for 1 MB leaf nodes. Two hint bits can reduce the percentage of additional reads to less than 5% in all cases, 4 hint bits further reduce this percentage to less than 0.8%, and 8 hint bits can reduce it to less than 0.25%.

The corresponding performance impact for 1 MB leaf nodes, a rebalancing threshold of 0.97, and 0 to 8 hint bits and 8 threads is shown in Figure 11. Eight hint bits improve performance by an average of 20% over no hints for most benchmarks except YCSB E and Load. The Load benchmark requires more computations and slightly more storage (and therefore more I/Os) when using more hint bits. For the scan-intensive YCSB E benchmark, hint bits have little impact because range scans almost always load multiple SSD pages per operation.

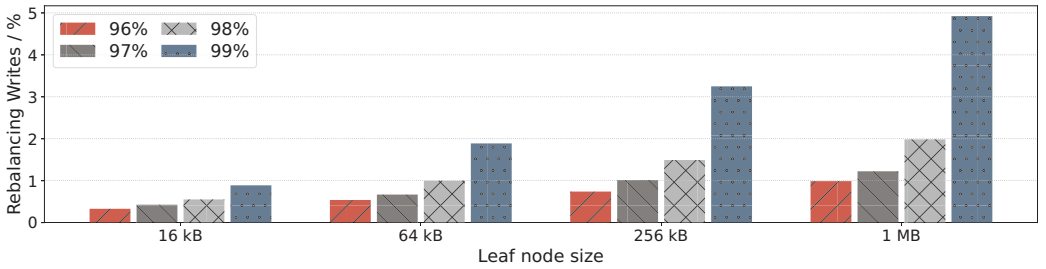


Fig. 12. Percentage of rebalancing writes compared to insert operations.

However, 4 hint bits result in only slightly lower performance than using 8 hint bits but also require significantly less memory to store inner nodes (see Figure 7). We therefore use 4 hint bits as our default setting.

4.6 Rebalancing Operations

Write operations occur for inserts, splits, and rebalancing operations. The number of inserts does not change between B$^+$-Tree and HLN-Tree, and the number of split writes is only slightly different. Rebalancing writes are the only significant source of additional writes induced by HLN-Trees. They are triggered when a leaf subnode becomes full and the overall leaf load is still below its rebalancing threshold. Rebalancing affects the write performance and write amplification of SSDs. Therefore, we evaluated rebalancing costs for rebalancing thresholds between 96% and 99%, leaf node sizes between 64 KB and 1 MB, 4 hint bits, and 8-byte keys and values.

Figure 12 shows the percentage of rebalancing write volume induced after inserting 100,000,000 kv pairs. Not surprisingly, higher maximum fill levels and larger leaf node sizes result in more rebalancing volume. This can be explained by more rebalancing operations before a split occurs for higher fill grades and more data moved per rebalancing operation for larger leaf nodes.

HLN-Trees bundle 4 kB rebalancing writes based on their leaf node size, so the number of IO operations even decreases for larger leaf nodes, e.g., from 1,045,390 for 64 kB leaves and a maximum fill level of 99% to 176,963 for 1 MB leaves and the same maximum fill level.

The total number of split writes is nearly the same for B$^+$-Trees and HLN-Trees and is also nearly independent of leaf node size and rebalancing threshold, fluctuating between 1% and 1.2% of the total write volume. The rebalancing cost is therefore of the same order as the split write cost up to a leaf node size of 1 MB and a rebalancing threshold of 0.97, which is why we chose this rebalancing threshold in most of our experiments.

4.7 Impact of Large kv Pairs and Stash Nodes

HLN-Trees efficiently achieve high leaf node fill grades at low rebalancing costs when using 8-byte keys and values. Equation (2) indicates that the deviation between subnode fill grades increases for larger values and achieving a high fill grade requires more rebalancing operations. Stash nodes reduce storage costs by transforming leaf nodes into more compact stash nodes when they reach their rebalancing threshold and by only splitting them when they are completely full. The number of leaf nodes then does not depend anymore on the rebalancing threshold, whereas this threshold still has an impact on the required number of reads and writes.[5]

Figure 13 compares standard B$^+$-Trees and HLN-Trees with 1 MB leaf nodes and 4 hint bits, with and without stash nodes, when inserting 100 million 8-byte keys and 50-byte values. The bars in

[5]The HLN-Tree performance nearly does not change when enabling stash nodes for 8-byte values. We have therefore not previously shown corresponding results.

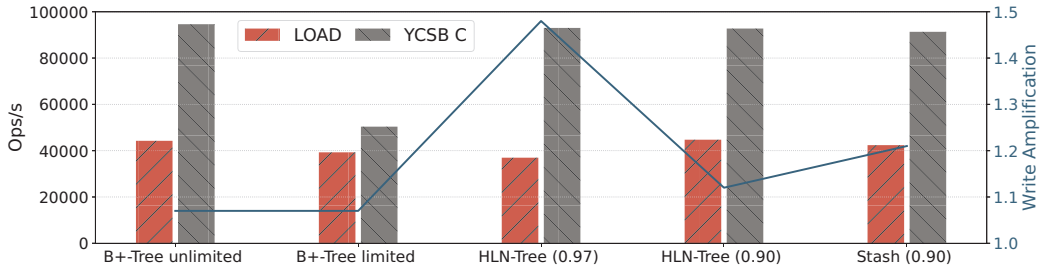


Fig. 13. B$^+$-Trees and HLN-Trees for 8-byte keys and 50-byte values.

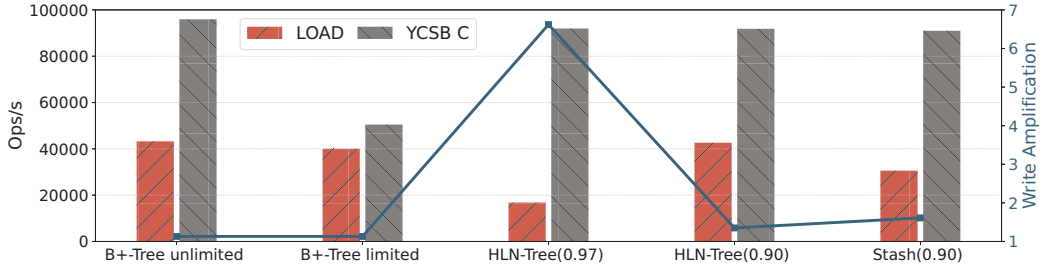


Fig. 14. B$^+$-Trees and HLN-Trees for 8-byte keys and 100-byte values.

the figure show the load and lookup performance of the indexes, and the blue line highlights the write amplification, which we define as the ratio of the number of disk IOs to the number of insert operations. A higher write amplification immediately results in a lower load throughput.

The B$^+$-Tree performance nearly does not change compared to the case of inserting 8-byte keys and values. The load throughput of HLN-Tree with a rebalancing threshold of 97%, on the other hand, significantly drops from nearly 50,000 operations per second for small kv pairs to 36,965 for large kv pairs. The reason is the significant rebalancing overhead that increases from 1.3% to nearly 50%, leading to 148 million 4 kB writes when loading 100 million kv pairs. The overhead can be significantly decreased for a rebalancing threshold of 90%, so that the load throughput of HLN-Tree is on the same level as the unlimited B$^+$-Tree. However, this reduces the average leaf node fill level by 5%, resulting in significant storage costs and diminishing benefits.

Coupling a rebalancing threshold of 90% with stash nodes balances between write overheads, storage efficiency, and lookup throughput. The load performance of the stash node design is 4% below the performance of an unlimited B$^+$-Tree and 8% higher than the performance of a B$^+$-Tree with limited inner node cache. At the same time, it improves lookup performance compared to limited B$^+$-Trees by more than 80% for the same memory footprint.

Stash nodes introduce additional lookup IOs compared to a standard HLN-Tree when a leaf node is filled between the rebalancing threshold and 100% for keys stored in the overflow subnode. The less even key distribution for larger kv pairs can also lead to additional IOs when calling `guessSubNode()`. This results in a total of 2.7% more IOs and explains the slightly lower lookup performance compared to an unlimited B$^+$-Tree.

Figure 14 shows the results for the same experiment when further increasing the value size to 100 bytes. It becomes obvious that the rebalancing overhead becomes prohibitive for a HLN-Tree with 1 MB leaf nodes and a rebalancing threshold of 0.97, as the overall write amplification increases to 6.7. Also, the stash node overhead increases to nearly 60% for insert operations, so very large values require tradeoffs between write amplification and storage cost.

The results show that stash nodes optimize storage utilization, incur only small additional costs for inserts, and provide near-optimal lookup performance for medium-sized kv pairs, while very large kv pairs require additional techniques, such as storing the values in a separate log store (see, e.g., [32]).

4.8 Skewed Key Distributions

We investigated the impact of skewed real-world key distributions using an Amazon book-sale dataset, Wikipedia timestamps, and Facebook user ids for 8-byte keys and values and 200 million sequentially inserted entries [23]. We compared HLN-Trees with 256 kB leafs, 4 hint bits, and a rebalancing threshold of 0.97 with HLN$_{FNV}$-Trees with 64 kB and 256 kB leafs, 4 hint bits, and a rebalancing threshold of 0.97; B$^+$-Trees; TreeLine; and RocksDB. We also included an HLN$_{FNV}$-Tree with 64 kB leaf nodes and a 16 MB record cache to compare its impact with the TreeLine record cache.

HLN-Trees were originally designed for uniform random key distributions. The key distributions of the studied real-world datasets do not follow this assumption, and the resulting key skewness sometimes severely affects the performance of HLN-Trees. The most challenging dataset in this respect is the Facebook dataset, for which inserts and lookups suffered the most (see Figure 15(a)). Each insert placed its element in the last subnode of a leaf node, resulting in many rebalancing operations and a write amplification that produced 2.5 IOs for each insert. Lookup performance actually dropped to one-fifth of the performance of an unlimited B$^+$-Tree, as each IO generated an average of 3.5 additional IOs due to incorrect subnode guesses.

The Wiki and Amazon datasets are much easier to handle for HLN-Tree. The Wiki performance dropped slightly by 5% to 14% for lookups and range scans compared to an unlimited B$^+$-Tree (see Figure 15(b)). However, inserts were significantly slower, again due to many rebalancing operations.[6] The dataset closest to a uniform random key distribution is the Amazon dataset. For this dataset, the performance of HLN-Tree was even equal to the performance of the unlimited B$^+$-Tree (see Figure 15(c)).

BD-trees without our FNV hashing have identical problems, so we do not show their results here. Using a record cache in front of the standard HLN-Tree helped to improve insert performance but could not overcome the many wrong lookup guesses for non-random key distributions, so we are not showing the corresponding results here.

We have introduced HLN$_{FNV}$-Trees with hashing to support arbitrary key distributions. Figure 15 shows that the load performance of HLN$_{FNV}$-Trees is within 92% to 99% of the load performance of an unlimited B$^+$-Tree, being slightly slower based on rebalancing and sorting during splits. The lookup performance of the HLN$_{FNV}$-Tree without a record cache is on the same level as for the unlimited B$^+$-Tree and 45% higher than for the limited B$^+$-Tree.

The YCSB E scan results for an HLN$_{FNV}$-Tree with medium-sized 64 kB leafs are on the same level as for an unlimited B$^+$-Tree, as it benefits from loading all required subnodes with one IO. Also, the selected range scan size of 1,000 elements on average requires about six disk blocks to be read on average, so the write amplification for the HLN$_{FNV}$-Tree remains small. Scaling leaf nodes to 256 kB decreases scan performance by 75%, because the HLN$_{FNV}$-Tree must always load all its 64 subnodes. Also, the subnodes must be sorted to find the keys that belong to a particular range query. Using 1 MB leaf nodes would further decrease scan performance.

We kept TreeLine's option `pg_bypass_cache` at `false` so that it first caches new inserts and flushes them later, combining multiple keys at the cost of a reduced crash consistency. TreeLine

[6]The load performance for the Wikipedia data was particularly high for all indexes, as this dataset contains more than 50% duplicates, which were simply dropped during inserts.

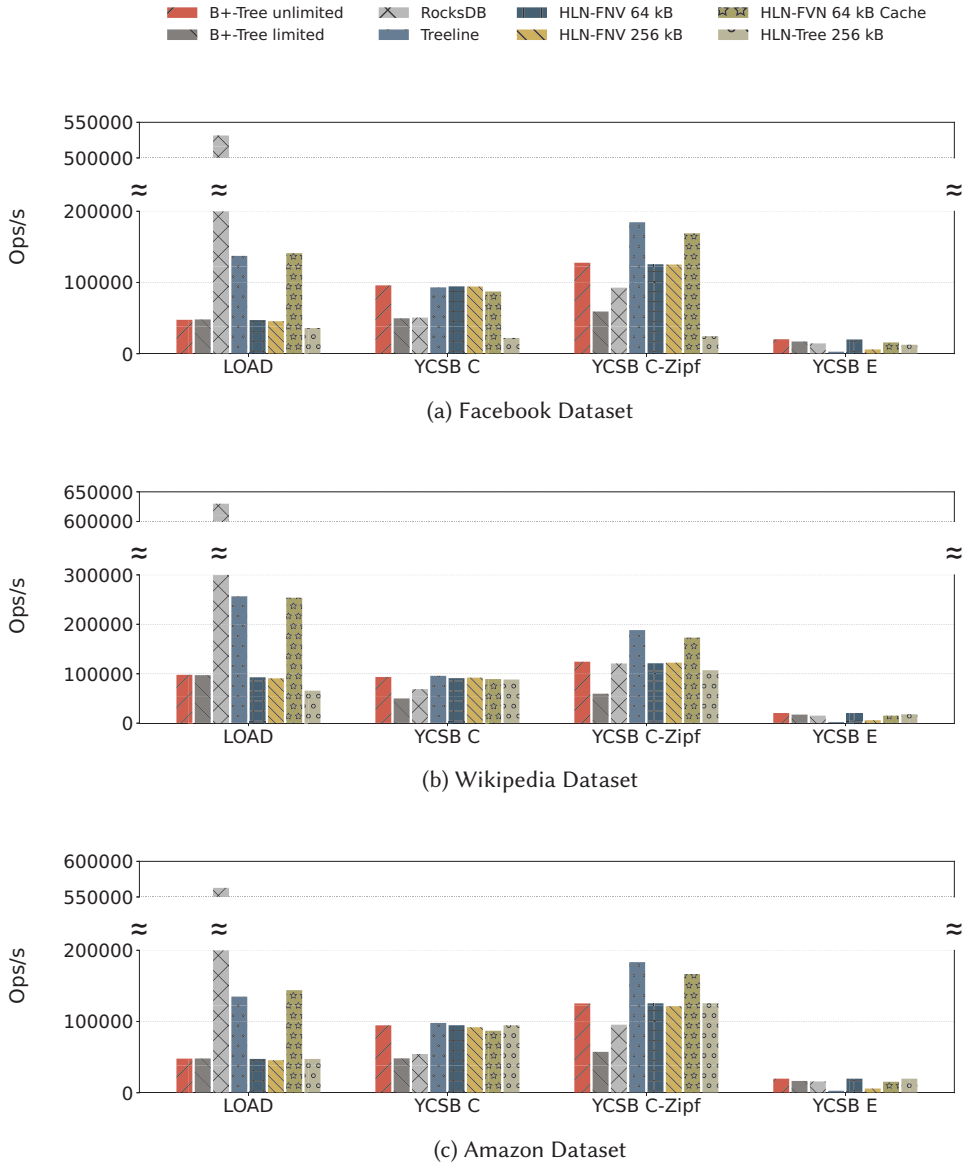


Fig. 15. Throughput for 8-byte kv pairs and 8 threads.

is therefore significantly faster for inserts than B$^+$-Trees and HLN-Trees without record cache. This positive effect of the record cache has not been visible for random inserts (Figure 8), because TreeLine has only been able to batch very few keys for each IO. Sequential inserts allow TreeLine to batch more keys per IO. Unfortunately, TreeLine always crashed when loading data concurrently, so we only report load throughput for a single thread.

The performance of TreeLine for YCSB C is on the same level as the other indexes and better than its own performance for uniform random key distributions. The underlying reason is that the number of kv pairs is 5× smaller compared to the 1 billion entries before. The fraction of keys that can be stored by the record cache, which is orthogonal to the HLN-Tree design, therefore

also increases by the same factor. This especially benefits TreeLine for the Zipf distribution. Comparing the YCSB E performance shows that TreeLine's scan performance is 7× to 8× lower than the HLN$_{FNV}$-Tree performance for 64 kB leaf nodes and 2× lower than for 256 kB leaf nodes, indicating again a not-yet-optimized implementation of TreeLine.

For comparison purposes, we have coupled HLN$_{FNV}$-Trees with a record cache, denoted as *HLN$_{FNV}$-Tree 64 kB Cache* in Figure 15, which significantly improves insert and lookup performance and reduces write amplification. Using the record cache, the insert performance of HLN$_{FNV}$-Tree is actually slightly better than the insert performance of Treeline, and its lookup performance is only 9% slower than TreeLine, for both YCSB C and YCSB C with a Zipf distribution. This small, remaining performance difference can be explained by the prototypical implementation of the record cache in HLN-Tree, which is based on Tkrzw with its known scalability limitations, while the record cache in Treeline is based on the Masstree kv store [34]. In addition, the range scan performance of HLN$_{FNV}$-Tree with a record cache is significantly higher than the range scan performance of Treeline.

Comparing the reduction of leaf nodes, we see that the HLN$_{FNV}$-Tree implementations always decrease the number of leaf nodes by 16× to 256×, only depending on the leaf node size. Page grouping in TreeLine, in contrast, only reduces the number of index entries between 1.8× for synthetic workloads and 4.8× for other benchmarks (according to [46]). This result has been unfortunately difficult to confirm when running TreeLine, as the corresponding leaf node files have been significantly blown up, requiring 12× more storage for leaf nodes than required by the HLN$_{FNV}$-Tree.

Comparing RocksDB with the other indexes for real-world key distributions shows a similar trend as for uniform random key distributions. RocksDB is again significantly faster for inserts and slower for lookups. Using the record cache before HLN-Tree reduces the insert slowdown to a factor of 2.5 for the Wikipedia workload, while HLN-Tree is 1.35× faster for lookups and on the same level for range scans. For the Amazon workload, and similarly for the Facebook workload, RocksDB is 4× faster for load but 1.6× to 1.8× slower for lookups and at a similar performance for YCSB E range scans. The differences in performance of RocksDB for the Wikipedia workload compared to the other workloads can be explained by the significantly lower number of stored kv pairs and therefore also the lower number of levels. In contrast, HLN-Tree performance is almost independent of the number of items stored.

In summary, the results show that HLN$_{FNV}$-Tree works well for arbitrary key distributions. The choice of the right HLN$_{FNV}$-Tree depends on a tradeoff between memory savings and additional cost for range queries. The comparison with TreeLine shows that the basic HLN-Tree mechanisms outperform TreeLine, while TreeLine shines when it can use its record cache. Nevertheless, we have shown that using some of the memory savings of HLN-Tree for a record cache closes the performance gap between HLN$_{FNV}$-Tree and TreeLine.

5 Related Work

Several index structures can be applied for hugedata sets. **Log-structured merge (LSM)** trees store data first in memory and later compact it and move it to persistent levels [39]. Inserts are very fast, whereas lookups can require many IOs in case of uniform random access patterns. Hash maps assign elements to buckets by applying a hash function [36]. They offer extremely fast point queries but do not support range scans. Also, techniques to increase storage efficiency for hash maps lead to additional IOs [40, 47]. Tries distribute the parts of a key over multiple nodes and reconstruct it when following a path [8]. They are very fast in-memory indexes and there has been great progress on memory efficiency of in-memory tries [6, 35], but their mapping to SSDs imposes new challenges, as node sizes do not align with storage blocks.

B$^+$-Trees well support the investigated workloads [5, 15, 16]. B$^+$-Tree optimizations for SSDs include batch updates [46], use the internal SSD parallelism [42], optimize the data layout [27], use Bloom filters to reduce unnecessary reads [19] and asynchronous IOs to saturate SSDs [44], or reduce SSD latencies by directly triggering IOs for the next path node from inside the kernel. Compression decreases the storage footprint of B$^+$-Trees and includes leaf node compression [1], storing partial keys in inner nodes [7], delta encoding [20], or dictionary-based key compression [48]. Most of these optimizations are orthogonal to the HLN-Tree design.

Our work builds on BD-Trees [28, 30] and overcomes their drawback of a low storage density [4]. BF-Trees support large leaf nodes by using Bloom filters to compress the key index [3]. They enable memory savings for large kv pairs but require many Bloom filter entries for smaller values and two writes for each insert.

Adaptive Hybrid Indexes combine B$^+$-Trees and compact indexes. They reduce the memory consumption of the inner index up to 82% by keeping 90% of the original performance [2]. The HLN$_{FNV}$-Tree with 64 kB leaf nodes is able to reduce the inner index size by 93% and keeping 99% of the B$^+$-Tree performance.

Learned indexes treat the index as a model to map a key to the position of a record. Most learned indexes work in-memory [10, 12, 50]. Treeline is a learned index for secondary storage that provides fine-grained record caching and page grouping [46]. Zhang et al. provide and evaluate guidelines for transferring knowledge from in-memory learned indexes to secondary storage indexes [49]. FILM optimizes data swapping between memory and disks but only focuses on appends [33]. HLN-Tree is more memory efficient for the investigated use case than today's learned indexes.

6 Conclusion and Future Work

We designed HLN-Trees to support uniform random key distributions for major Cloud applications by combining key distribution properties with rebalancing and a learning mechanism. HLN-Trees use these properties to increase the size of leaf nodes and decrease the memory required to store inner nodes. As a result, the size required to cache all inner nodes for uniform random key distributions can be reduced by up to 256× compared to a B$^+$-Tree. This leads to a significantly improved performance over B$^+$-Trees for the same cache size, while providing comparable performance to unlimited B$^+$-Trees.

HLN$_{FNV}$-Trees support arbitrary key distributions by an additional layer of hashing. They can, depending on the range-scan performance, reduce the number of leaf nodes by 16× to 256×, significantly improving over today's learned indexes [46] and other optimized data structures [2].

Future work will include ideas from learned indexes to speed up node traversals and optimize the integration of record caching and batch inserts.

One limitation of the HLN-Tree design is that it does not yet support variable-size kv pairs, because they can lead to a very unbalanced distribution of key ranges over subnodes. Unfortunately, these unbalanced key ranges would require a significantly higher number of hint bits to keep the number of additional IOs low, thus reducing the memory efficiency of HLN-Tree. In future work, we will therefore develop recursive hint encoding schemes that can directly support variable-size keys and non-random key distributions without the need for an additional level of hashing.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 671–682. https://doi.org/10.1145/1142473.1142548

[2] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive hybrid indexes. In *Proceedings of International Conference on Management of Data (SIGMOD)*. ACM, 1626–1639. https://doi.org/10.1145/3514221.3526121

[3] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate tree indexing. *Proc. VLDB Endow.* 7, 14 (2014), 1881–1892. https://doi.org/10.14778/2733085.2733094
[4] Ricardo A. Baeza-Yates. 1996. Bounded disorder: The effect of the index. *Theor. Comput. Sci.* 168, 1 (1996), 21–38. https://doi.org/10.1016/S0304-3975(96)00061-8
[5] Rudolf Bayer and Edward M. McCreight. 1972. Organization and maintenance of large ordered indices. *Acta Inform.* 1 (1972), 173–189. https://doi.org/10.1007/BF00288683
[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 2018 International Conference on Management of Data (SIGMOD)*. 521–534. https://doi.org/10.1145/3183713.3196896
[7] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. 2001. Main-memory index structures with fixed-size partial keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 163–174. https://doi.org/10.1145/375663.375681
[8] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers Presented at the the 1959 Western Joint Computer Conference, IRE-AIEE-ACM*. 295–298. https://doi.org/10.1145/1457838.1457895
[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 143–154. https://doi.org/10.1145/1807128.1807152
[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al.. 2020. ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. 969–984. https://doi.org/10.1145/3318464.3389711
[11] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2021. On the performance of learned data structures. *Theor. Comput. Sci.* 871 (2021), 107–120. https://doi.org/10.1016/j.tcs.2021.04.015
[12] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135
[13] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. 2019. The FNV Non-cryptographic Hash Algorithm, Version 17. Internet Draft, Network Working Group. (May 2019).
[14] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2010. Walking in Facebook: A case study of unbiased sampling of OSNs. In *29th IEEE International Conference on Computer Communications (INFOCOM)*. 2498–2506. https://doi.org/10.1109/INFCOM.2010.5462078
[15] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26. https://doi.org/10.1145/1806907.1806908
[16] Goetz Graefe. 2011. Modern B-Tree techniques. *Found. Trends Databases* 3, 4 (2011), 203–402. https://doi.org/10.1561/1900000028
[17] Goetz Graefe, Hideaki Kimura, and Harumi A. Kuno. 2012. Foster B-trees. *ACM Trans. Database Syst.* 37, 3 (2012), 17:1–17:29. https://doi.org/10.1145/2338626.2338630
[18] Mikio Hirabayashi. 2020. Tkrzw: A Set of Implementations of DBM. https://dbmx.net/tkrzw/
[19] Peiquan Jin, Chengcheng Yang, Christian S. Jensen, Puyuan Yang, and Lihua Yue. 2016. Read/write-optimized tree indexing for solid-state drives. *VLDB J.* 25, 5 (2016), 695–717. https://doi.org/10.1007/s00778-015-0406-1
[20] Rize Jin and Tae-Sun Chung. 2010. Node compression techniques based on cache-sensitive B+-Tree. In *9th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*. 133–138. https://doi.org/10.1109/ICIS.2010.9
[21] Theodore Johnson and Dennis E. Shasha. 1993. The performance of current B-Tree algorithms. *ACM Trans. Database Syst.* 18, 1 (1993), 51–101. https://doi.org/10.1145/151284.151286
[22] Jürgen Kaiser, Tim Süß, Lars Nagel, and André Brinkmann. 2016. Sorted deduplication: How to process thousands of backup streams. In *32nd Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2016.7897082
[23] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *CoRR* abs/1911.13014 (2019). http://arxiv.org/abs/1911.13014
[24] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561. https://doi.org/10.1137/080728743
[25] Youngjae Lee and Jinglei Ren. 2023. YCSB-cpp: YCSB Written in C++ for LevelDB, RocksDB and LMDB. https://github.com/ls4154/YCSB-cpp
[26] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST)*. 99–114.
[27] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1 (2010), 1195–1206. https://doi.org/10.14778/1920841.1920990

[28] Witold Litwin and David B. Lomet. 1986. The bounded disorder access method. In *Proceedings of the 2nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 38–48. https://doi.org/10.1109/ICDE.1986.7266204
[29] Zhuoxuan Liu and Shimin Chen. 2023. Pea hash: A performant extendible adaptive hashing index. *Proc. ACM Manag. Data* 1, 1 (2023), 108:1–108:25. https://doi.org/10.1145/3588962
[30] David B. Lomet. 1988. A simple bounded disorder file organization with good performance. *ACM Trans. Database Syst.* 13, 4 (1988), 525–551. https://doi.org/10.1145/49346.50067
[31] David B. Lomet. 2019. Cost/performance in modern data stores: How data caching systems succeed. In *35th IEEE International Conference on Data Engineering Workshops*. 140. https://doi.org/10.1109/ICDEW.2019.00-20
[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in SSD-Conscious storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28. https://doi.org/10.1145/3033273
[33] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A fully learned index for larger-than-memory databases. *Proc. VLDB Endow.* 16, 3 (2022), 561–573.
[34] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*. 183–196. https://doi.org/10.1145/2168836.2168855
[35] Markus Mäsker, Tim Süß, Lars Nagel, Lingfang Zeng, and André Brinkmann. 2019. Hyperion: Building the largest in-memory search tree. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1207–1222. https://doi.org/10.1145/3299869.3319870
[36] W. D. Maurer and Ted G. Lewis. 1975. Hash table methods. *ACM Comput. Surv.* 7, 1 (1975), 5–19. https://doi.org/10.1145/356643.356645
[37] Dirk Meister, Jürgen Kaiser, and André Brinkmann. 2013. Block locality caching for data deduplication. In *6th Annual International Systems and Storage Conference, (SYSTOR)*. 15:1–15:12. https://doi.org/10.1145/2485732.2485748
[38] Sai Narasimhamurthy, Nikita Danilov, Sining Wu, Ganesan Umanesan, Stefano Markidis, Sergio Rivas-Gomez, Ivy Bo Peng, Erwin Laure, Dirk Pleiter, and Shaun De Witt. 2019. SAGE: Percipient storage for exascale data centric computing. *Parallel Comput.* 83 (2019), 22–33. https://doi.org/10.1016/j.parco.2018.03.002
[39] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-Tree). *Acta Inform.* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048
[40] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *9th Annual European Symposium on Algorithms (ESA)*, Vol. 2161. 121–133. https://doi.org/10.1007/3-540-44676-1_10
[41] Martin Raab and Angelika Steger. 1998. "Balls into Bins"—A simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop (RANDOM)*. 159–170. https://doi.org/10.1007/3-540-49543-6_13
[42] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.* 5, 4 (2011), 286–297. https://doi.org/10.14778/2095686.2095688
[43] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 36–53. https://doi.org/10.1145/3299869.3300075
[44] Li Wang, Zining Zhang, and Zhenjie Zhang. 2020. PA-Tree: Polled-mode asynchronous B+ tree for NVMe. In *36th IEEE International Conference on Data Engineering (ICDE)*. IEEE, 553–564. https://doi.org/10.1109/ICDE48307.2020.00054
[45] Andrew Chi-Chih Yao. 1978. On random 2-3 trees. *Acta Inform.* 9 (1978), 159–170. https://doi.org/10.1007/BF00289075
[46] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An update-in-place key-value store for modern storage. *Proc. VLDB Endow.* 16, 1 (2022), 99–112. https://github.com/mitdbg/treeline
[47] Adar Zeitak and Adam Morrison. 2021. Cuckoo trie: Exploiting memory-level parallelism for efficient DRAM indexing. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 147–162. https://doi.org/10.1145/3477132.3483551
[48] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. 1601–1615. https://doi.org/10.1145/3318464.3380583
[49] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making in-memory learned indexes efficient on disk. *Proc. ACM Manag. Data* 2, 3 (2024), 151. https://doi.org/10.1145/3654954
[50] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A high-performance and memory-efficient learned index via hybrid construction. *Proc. ACM Manag. Data* 2, 3 (2024), 145. https://doi.org/10.1145/3654948

[51] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. XRP: In-kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 375–393.

[52] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST)*. 269–282.

Received 19 February 2024; revised 11 August 2024; accepted 31 October 2024