



Morphtree: a polymorphic main-memory learned index for dynamic workloads

Yongping Luo¹ · Peiquan Jin¹ · Zhaole Chu¹ · Xiaoliang Wang¹ · Yigui Yuan¹ · Zhou Zhang¹ · Yun Luo² · Xufei Wu² · Peng Zou²

Received: 30 January 2023 / Revised: 17 October 2023 / Accepted: 29 October 2023 / Published online: 1 December 2023
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2023

Abstract

Modern database systems rely on indexes to accelerate data access. The recently proposed learned indexes can offer higher search performance with lower space costs than traditional indexes like B+-tree. We observe that existing main-memory learned indexes are particularly optimized for read-heavy workloads. However, such an optimization comes at the cost of model training and handling out-of-range key insertions, which will worsen the overall performance. We argue that workloads are not always read-heavy in real applications, and it is more important and practical to make learned indexes work efficiently for dynamic workloads with changing access patterns and data distributions. In this paper, we aim to improve the practicality of learned indexes by making them adaptive to dynamic workloads. Specifically, we propose a new polymorphic learned index named *Morphtree*, which can adaptively change the index structure to provide stable and high performance for dynamic workloads. The novelty of *Morphtree* lies in three aspects: (1) *a decoupled tree structure* for separating the inner search tree from the data layer consisting of leaf nodes, (2) *a read-optimized learned inner tree* for improving the performance of index search, and (3) *an evolving data layer* for automatically transforming node layouts into read friendly or write friendly according to workload changes. We evaluate these new ideas of *Morphtree* on various datasets and workloads. The comparative results with six up-to-date learned indexes, including ALEX, PGM-index, FITing-tree, LIPP, FINEdex, and XIndex, show that *Morphtree* can achieve, on average, 0.56x and 3x improvements in lookup and insertion performance, respectively. Moreover, when evaluated on dynamic workloads with changing lookup ratios and data distributions, *Morphtree* can achieve a sustained high throughput across different real-world datasets and query patterns, owing to its ability to automatically adjust the index structure according to workload changes.

Keywords Learned index · Dynamic workload · Adaptive index · Performance optimization · Polymorphic structure

1 Introduction

As the data volume grows explosively and query workloads become more and more diverse in the big data era, building a robust index structure is essential to designing high-performance storage systems [11]. Over the years, we have seen a plethora of exciting and inspirational index structures based on B+-tree [16], Trie [20], Hash table [23], LSM-tree [24], or hybrid structures [39, 42]. However, most of them are optimized for specific query workloads, such

as read-heavy or write-heavy scenarios, but are not suitable for dynamic workloads [8, 14] with changing access patterns and key distributions. This is consistent with the RUM (Read-Update-Memory) conjecture proposed by Manos et al. [3], claiming that designing access methods always makes trade-offs among read cost, write cost, and memory overhead. Designing a one-size-fits-all index structure seems impractical but worthwhile and attractive under diverse and dynamic workloads.

The trade-off of RUM is also consistent in recently-proposed main-memory learned indexes [7, 10, 18, 36, 45], which use lightweight machine-learning models to learn the data distribution and predict the position of a given record within an acceptable error. According to our benchmark of several state-of-the-art updatable learned indexes (see Fig. 1), the lookup performance of learned indexes is 3–4

✉ Peiquan Jin
jpq@ustc.edu.cn

¹ School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

² Tencent, Chengdu 610014, China

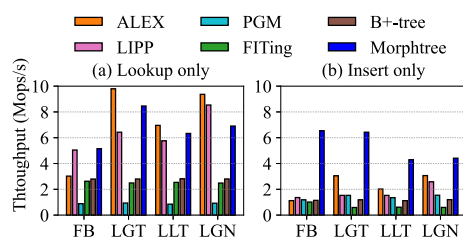


Fig. 1 The performance of state-of-the-art learned indexes, including ALEX, PGM-index, FITing-tree, and LIPP, along with B+-tree and Morphtree (proposed in this paper), on lookup-only and insertion-only workloads. The results are reported on four datasets: Facebook (FB), Longitude (LGT), Lon-Latitude (LLT), and Lognormal (LGN) [7]

times B+-tree, while the insertion performance is slightly better than B+-tree. Wongkham et al. [35] also revealed that under some real-world datasets, updatable learned indexes are worse than traditional ones when the query workload contains more than 50% of insertion queries. Technically, Learned indexes reduce lookup cost by lowering tree height and lookup cost inside tree nodes simultaneously. However, they perform poorly under insertions for several reasons: (1) insertions are likely to incur local data movements and therefore enlarge prediction errors [7]. (2) insertions will trigger costly structure modification operations like node expansion, node splitting, and model retraining. (3) insertions may invalidate the existing learned model, increase the tree height, and lower the overall performance [17]. In a word, current updatable learned indexes are read-optimized but work poorly on insertion-heavy workloads and out-of-the-bound insertions.

This paper aims to devise a polymorphic main-memory learned index that can adapt to dynamic workloads by transforming the index structure according to the change of access patterns and key distributions. As a static learned index structure can be tuned to work well on read-heavy workloads, the biggest challenge of building a polymorphic learned index is to achieve a proper trade-off between read and write costs to guide the adjustment of the index structure. To address such a challenge, we first investigated the widely used B+-tree. Figure 2 shows the access statistics of inner and leaf nodes in B+-tree under three workloads, including (a) Load (insertion-only), (b) Balanced (read-write balanced), and (c) Read Heavy. We found that the inner nodes of B+-trees always have a read tendency regardless of the workload change. On the other hand, the leaf nodes exhibit different read/write tendencies on different workloads. Motivated by the observation depicted in Fig. 2, we propose to use a read-optimized inner search tree to organize inner nodes to utilize the reading tendency of inner nodes and improve the search performance on inner nodes. Also, we propose to adopt an evolving data layer to construct leaf nodes, which can automatically adjust the node layout according to workload changes. To be more specific, when leaf nodes become read-intensive, the data

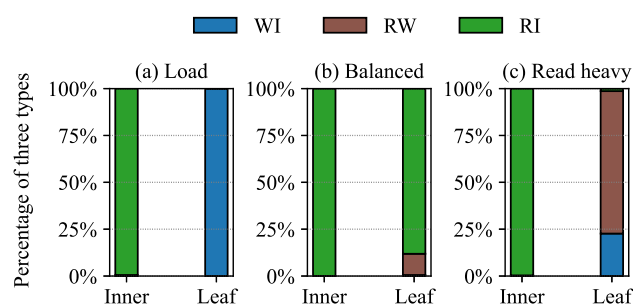


Fig. 2 The read and write tendency of B+-tree's inner and leaf nodes under different workloads: **a** Load (insertion-only), **b** Balanced (read-write balanced), and **c** Read Heavy. Each node is classified into three categories: WI (Write Intensive nodes with more than 80% writes), RI (Read Intensive nodes with more than 80% reads), and RW (Read Write balanced nodes). The Y-axis represents the proportion of the nodes within each category

layer will automatically transform into a lookup-optimized storage structure; when leaf nodes become write-intensive, the data layer will use insertion-optimized node layouts.

Following the above idea, we present a new polymorphic main-memory learned index called *Morphtree* in this paper. The word "polymorphic" means that Morphtree can adapt to dynamic workloads by automatically making the index structure morph into an optimal structure. The major difference between Morphtree and existing main-memory learned indexes [7, 10, 21, 34, 36, 45] is that Morphtree can achieve high performance on both read-intensive and write-intensive workloads while existing learned indexes are only efficient for read-intensive workloads. Moreover, Morphtree can automatically adapt to the change in query workloads.

Briefly, we make the following contributions in this paper.

- To address the issue that existing main-memory learned indexes are only read-optimized but not write-optimized, we propose a new polymorphic main-memory learned index called *Morphtree*. The novelty of Morphtree lies in three aspects. (1) *a decoupled tree structure*: motivated by the observation that the inner nodes in B+-tree are always read-intensive while the leaf nodes have different read/write tendencies on different workloads, we propose to adopt a decoupled tree structure for Morphtree, which consists of a read-optimized inner tree and a data layer for leaf nodes. (2) *a read-optimized learned inner tree*: the inner tree of Morphtree employs cache-efficient read-optimized learned nodes, which can ensure the average height of the Morphtree will not exceed three regardless of the dataset size. (3) *an evolving data layer*: the data layer can automatically transform node layouts according to workload changes, yielding an evolving storage structure.
- We propose an efficient approach to implement node morphing. First, we present a sampling approach to monitor

the access statistics for each node. Then, we present a non-blocking algorithm to automatically adjust node layouts in the background, which does not block subsequent accesses. In addition, we develop effective rules to avoid unnecessary node morphing operations.

- We evaluate Morphtree under workloads with diverse query patterns. The results show that Morphtree can achieve an average 0.56x higher lookup performance and 3x higher insertion performance than its competitors. Moreover, when evaluated on dynamic workloads with changing lookup ratios and data distribution, Morphtree achieves a sustained high throughput across different real-world datasets and query patterns among all compared indexes. Moreover, Morphtree also outperforms its competitors in concurrent experiments with multiple threads.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 presents the structure of Morphtree. Section 4 discusses the node morphing algorithm. Section 5 presents the operations and theoretical analysis of Morphtree. Section 6 reports the experimental results, and finally, we conclude the paper and discuss future work in Sect. 7.

2 Related work

In this section, we discuss existing work related to this study. Section 2.1 summarizes recent advances in learned indexes, and Sect. 2.2 introduces automatic index tuning.

2.1 Learned indexes

As proposed in 2018, learned indexes have attracted much attention in the database community [7, 9, 10, 18, 21, 28, 34, 36, 37, 43]. Learned indexes learn the distribution of a given dataset using some machine-learning models. A learned index can build a hierarchy of model nodes with a fixed-height tree structure. Each model node in the tree is associated with a function that maps a lookup key to its physical position within an error bound. Traditional B+-tree nodes use binary search or linear search to search for the corresponding child node or data record, while a learned node just needs to probe around the predicted position within a maximum error bound, which can reduce the searching overhead. In addition, learned indexes are flatter than B+-tree (flat means lower tree height) as they can construct much larger nodes through a lightweight linear model. The overall performance of recent learning indexes is dominating traditional tree indexes, as shown in Fig. 1.

2.1.1 Classification of learned indexes

According to ways of bulk-loading a learned index, learned indexes can be divided into two categories: (1) top-down index creation and (2) bottom-up index creation. In the top-down index creation, it first range-partitions the key space according to a machine-learning model and forms a root node storing all index records to each partition. Then, all keys belonging to a partition repeat the procedure and materialize the partition as a sub-tree. RMI [18], ALEX [7], and LIPP [36] are all indexes of this type. For the bottom-up index creation, it first partitions the key space according to a predefined error bound. For each partition, it forms a learned node with a learned function. The learned function guarantees that each record can be found at the predicted position within the error bound. Afterwards, it gathers all the first keys of each partition and then repeats the above procedure. PGM-index [9], FITing-tree [10], COLIN [45], FINEdex [21], XIndex [34] are of this type. Building a learned index in a bottom-up manner can bound the probing error and lead to a balanced tree in which each leaf node has the same height, but it incurs more probing cost in inner nodes. While top-down learned indexes ensure predictions in the inner nodes are accurate. However, it may result in a high and unbalanced index tree under nonlinear datasets, ending up with lower performance and higher memory consumption.

2.1.2 Implementation techniques of learned nodes

Except for index creation, the structure of learned model nodes is significant to the overall performance. We summarize the design techniques of learned nodes into three dimensions: (1) the learned model type, (2) the layout of data records in a node, and (3) how to handle insertion. As far as we know, almost all learned indexes use lightweight linear functions as the machine learning model for simplicity and efficiency, except for RMI, which uses complex functions learned from machine learning techniques. In addition, LIPP proposes to add support for kernel function or distribution conversion techniques to generate a linear distribution and then use a linear model. In terms of data layout, there are simply two ways. One is to place all data records compactly and use each key and the corresponding position to conduct model regression, which is adopted by PGM-index and FITing-tree. The other is to place all data records at the model-predicting position as much as possible. If not, shove a few records to its two sides. As for inserting new records, there are also several mechanisms, which can be concluded as *out-of-place inserting* (FITing-tree, XIndex) and *model-predicting in-place inserting*. The latter will cause some side effects, such as shoving records to two sides (ALEX), inserting them into overflow buckets (COLIN and FINEdex), and building a child node (LIPP). According to previous research

and our experiences, model-predicting in-place inserting is more attractive for overall performance.

2.1.3 Limitations of existing learned indexes

However, learned indexes come with several limitations. First, learned indexes are sensitive to data distributions. It prefers synthetic datasets, which are smooth to do piece-wise linear regression but perform poorly under real-world datasets. Second, learned indexes exhibit inferior performance under write-intensive workloads or unfamiliar data distributions. Third, learned indexes typically support integer-type and float-type keys, and they do not support string keys well.

The poor insertion performance of learned indexes can be attributed to the following reasons: (1) insertions are likely to incur local data movements and, therefore, enlarge the model prediction error [7]. (2) insertions may trigger structure modification operations like node expansion, splitting, and costly model retraining. (3) insertions with an unfamiliar data distribution will invalidate the learned model and lower the overall performance [17].

In this paper, we target improving the write performance and adaptivity of learned indexes. Technically, we propose a novel inner node structure to improve the insertion performance of learned indexes. In addition, we propose to automatically transform node layouts into read-friendly or write-friendly according to workload changes, yielding a polymorphic learned index structure that can deliver high read and write performance under dynamic workloads. Our Morphtree supports both integer and float keys that are not more than 8 bytes, which is similar to existing learned indexes.

2.2 Automatic index tuning

Automatic index tuning [32, 33] aims to help database administrators (DBAs) find an optimal index structure suitable for the workload. As there are many index candidates for DBAs to design physical database structures, it is beneficial to automatically recommend appropriate indexes for DBAs to reduce the overhead of physical database design. Automatic index tuning can be regarded as one job in self-driving database systems [27, 30]. The main challenge in automatic index tuning is to forecast the future workload [31, 38]. However, as automatic index tuning does not make morphing for the physical index structures, it is orthogonal to this study.

Early work in adaptive indexing [12, 13] focused on changing the physical index structure according to the query workload. However, they only focused on searching operations and could not adjust the index structure to fit write-intensive workloads. Jain et al. [15] proposed an on-the-fly transition from LSM-tree to B+-tree or backwards according

to workload changes. Zhou et al. [46] proposed an incremental index management system for dynamic workloads, utilizing the *Monte Carlo Tree Search* method. In addition, Anneser et al. [2] presented a hybrid index framework to adapt to skewed workload patterns.

Morphtree also aims to change the physical index structure for dynamic workloads. Unlike existing work, Morphtree is the first adaptive polymorphic index for improving learned indexes. Also, we propose to change node layouts but not the overall tree structure (e.g. changing LSM-tree to B+-tree [15]). Moreover, our Morphtree can monitor both read and write accesses, making the index fit workloads with changing read ratios and data distributions.

3 Index structure of Morphtree

In this section, we present the detailed structure of Morphtree. The key ideas of Morphtree include a *decoupled tree structure*, a *read-optimized learned inner tree*, and an *evolving data layer*.

3.1 Decoupled tree structure

The core idea of Morphtree is to decouple the index structure into a read-optimized learned inner tree and an evolving data layer, which is motivated by the observation of the accessing behaviour on B+-tree, as shown in Fig. 2. As the inner nodes on a tree index are read-heavy, even on write-intensive workloads, it is reasonable to devise a read-friendly structure for inner nodes. On the other hand, we noticed from Fig. 2 that the accessing behaviour on leaf nodes varies a lot on different kinds of workloads. To this end, the best design for leaf nodes

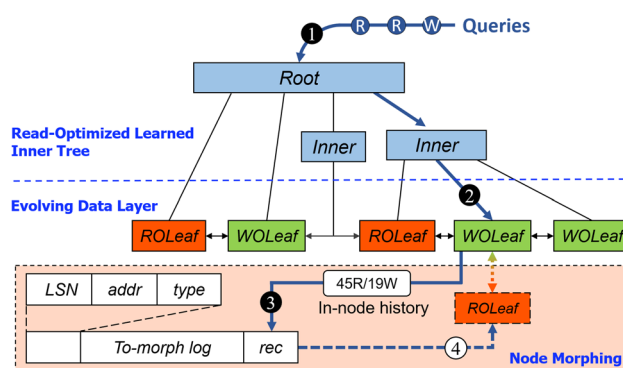


Fig. 3 The overall structure and node morphing process of Morphtree. The four numbered arrow lines show an example of node morphing: ① a query arrives at the root; ② it traverses down to a WOLeaf node; ③ according to the node's sampling history (45 reads and 19 writes), it determines to change the node to an ROLeaf node; thus, it adds a morphing record into the *To-morph log*; ④ a background morphing thread is notified to scan the log and handle node morphing, without blocking subsequent accesses

is to provide read-optimized node layouts for read-heavy workloads and write-optimized node layouts for write-heavy workloads. Following this idea, we propose to auto-tune the leaf-node structure according to workload changes, leading to an evolving data layer with leaf-node morphing support.

Figure 3 shows the overall structure of the decoupled tree structure in Morphtree. The read-optimized learned inner tree needs to be tailored for high read performance, and the data layer supports node morphing to deliver high performance for dynamic workloads.

However, it is not trivial to implement the decoupled index structure for Morphtree. The main challenges can be summarized as follows.

Challenge 1. *How to make the inner tree read-optimized for dynamic workloads?* While learned indexes can be used to implement the inner tree, their performance will be highly impacted by the change in workloads. For example, out-of-range key insertions will invalidate the model of a learned index and incur model retraining, which will worsen the search performance. Therefore, novel solutions are needed to reduce the probing cost and the average height of the inner tree. The optimization of the inner tree will be addressed in Sect. 3.2.

Challenge 2. *How to make the data layer morph and self-optimize with workload changes?* The data layer composed of leaf nodes needs to automatically change its storage structure to maintain high performance for dynamic workloads. Some existing indexes, such as B+-tree or learned indexes, are not suitable for write-heavy workloads, while other solutions like LSM-tree [15] suffer from costly compaction and poor search performance. We claim that a static node structure cannot deliver stable and high performance for dynamic workloads with changing access patterns and data distributions. Thus, new techniques have to be devised to address this challenge. In this paper, we propose an evolving data layer that can automatically adjust node layouts according to workload changes, and the details will be discussed in Sect. 3.3.

Challenge 3. *How to implement efficient node morphing without considerable extra costs?* Implementing an evolving data layer that supports node morphing will introduce unavoidable extra costs. Therefore, we need to devise an efficient approach to reduce the cost of node morphing. The most important issue is that node morphing should not block normal accesses. Otherwise, the overall performance of the index will degrade, reducing the benefits of node morphing. The detailed design of node morphing will be investigated in Sect. 4.

3.2 Read-optimized learned inner tree

According to our observation (see Fig. 2), the inner nodes of B+-trees are always read-intensive. Thus, it is reasonable to

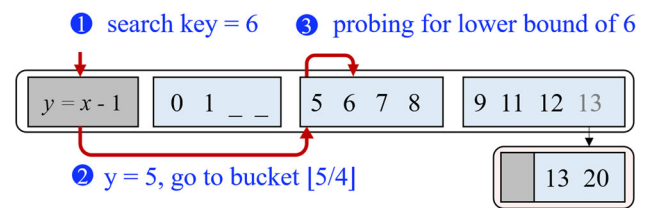


Fig. 4 The structure of a learned inner node. The linear function in the figure is $y = x - 1$, and the indexed records $\{0, 1, 5, 6, 7, 8, 9, 11, 12, 13, 20\}$ are stored in the inner node. A new inner node is built at the 3rd bucket to store overflowed records. 13 and 20 in the child inner node are overflowed records of this inner node

design large inner nodes for the index. Following this idea, we propose to construct the inner tree with learned nodes.

Specifically, we present several novel designs for the inner tree to optimize the read performance of learned nodes. First, we re-design the node structure of the inner tree to reduce the probing cost. Particularly, we present a cache-efficient learned node structure for the inner tree, which can ensure that only one cache line access is needed for probing an index record to a child node. Second, the learned nodes in the inner tree have a flat structure, and we demonstrate that the average height of the Morphtree will not exceed 3.

The structure of learned inner nodes is shown in Fig. 4. Each inner node is a learned node that contains one or several linear functions and a bucket array. The linear functions are used to calculate the precise bucket of a given search key, serving as a mapping from keys to bucket positions. The bucket array in a learned inner node is to store all index records to the child nodes, and the number of buckets is determined when building the inner node. Each bucket contains four slots, and each slot can accommodate an 8-byte key and an 8-byte pointer pointing to a child node, which can be a leaf node or an inner node. Note that all index records in a bucket are in key-ascending order. A bucket can only have one of the following three states: (a) contains empty slots, (b) full and all four slots represent leaf nodes, and (c) full and the first three slots represent leaf nodes while the last one represents an inner node. Index records stored in a child inner node are overflowed records, such as 13 and 20 in Fig. 4.

A learned inner node in the inner tree is built from an initial number of index records. We use *least squares line fitting* to fit one linear function as used in ALEX [7]. Leveraging this linear function, we can map each index record into a bucket and place records in its bucket in key-ascending order. Initially, the bucket array contains twice as many slots to store all initial index records. After building a new inner node with index records, we fill each empty bucket with a pivot record. This pivot record is a copy of the rightmost index record of this bucket.

Why are the learned inner nodes cache efficient? As all index records fall into a bucket, all accesses to them are

bound to one cache line. It avoids the high probing overhead of inner nodes as in B+-tree, FITing-tree, and PGM-index. Nonlinearity of the key distribution in an inner node will lead to a nonuniform distribution of index records in buckets. Thus, we build a new inner node for each bucket that is assigned more than four index records. Differing from the inner nodes in ALEX and nodes of LIPP where subspace lookup is precise at one position, our inner node relaxes this number to 4 as our subspace lookup is precise at one cache line. In LIPP, the collision of two records at one position is solved by building a new node to store the two records, while in ALEX, it needs to expand the collision inner node. These operations are heavy-weight and exacerbate insertion performance. Our inner node design introduces no extra cache line access but avoids collisions for free.

A data distribution might be hard to fit with only one linear function [35]. We define the fitness of a data distribution according to its linear function as the ratio of overflowed index records to total index records associated with this inner node. If more than 50% index records are overflowed, we call it hard-to-fit (this value 50% is consistent with the node rebuilding threshold later). In this situation, we should partition the index records and learn a new linear function for each partition. The partition algorithm proceeds as follows: (1) Let the inappropriate single linear function learned from all index records be LF; (2) From left to right, we add each index record into a partition greedily until its overflow rate exceeds 50% according to LF; (3) Then, we start with a new partition and continue this procedure.

How can the learned inner tree reduce the average height? As only buckets in the state (c) contain an inner child node, at least 75% of children in an inner node point to leaf nodes directly. Besides, the capacity of an inner node is not limited to enable large inner nodes for easy-to-fit data distribution. Consequently, most of the leaf nodes are directly pointed by the root node, and these leaf nodes are at the height of two. Besides, as new index records are inserted into an inner node or the key distribution is less linear, Morphtree tends to be unbalanced with local "over-height" sub-trees. Morphtree handles this by rebuilding a sub-tree if more than 50% of its leaf nodes are not directly pointed by the sub-tree's root node. The rebuild algorithm gathers all index records (excluding pivot records) of leaf nodes in the sub-tree and then builds a larger inner node with an updated linear function. Note that the new linear function reflects both the initial index records when building it and later-inserted index records. According to the theoretical analysis of Morphtree in Sect. 5.2, the average tree height will not exceed three regardless of the dataset scale.

Operations on the learned inner tree. Searching a given key k in an inner node performs as follows. We first use the corresponding linear function F to get a predicted position $F(k)$ for a lookup key k . Then, we probe the $\lfloor \frac{n \cdot F(r, key)}{4} \rfloor$ -th

Algorithm 1: Insertion to an Inner Node

```

1 Function roinner_insert(Node *n, Record r):
2    $bkt = n.buckets[\lfloor \frac{n \cdot F(r, key)}{4} \rfloor]$ ;
3   if  $bkt$  is not full then // state (a)
4     insert  $r$  into  $bkt$ ;
5   else if  $bkt[3].ptr$  is leaf then // state (b)
6      $r4, r5$  = last two records after insert  $r$  into  $bkt$ ;
7      $bkt[3].key = r4.key$ ;
8      $bkt[3].ptr = new\_roinner(\{r4, r5\})$ ;
9   else // state (c)
10     $r$  = the overflow record after insert  $r$  into  $bkt$ ;
11    roinner_insert( $bkt[3].ptr, r$ );
12  end
13  if more than 50% of index records are overflowed then
14    rebuild_roinner( $n$ );
15  end

16 Function new_roinner(Record *recs):
17    $n$  = allocate a new empty inner node;
18    $n.F$  = learn a linear model from  $recs$ ;
19   for  $r$  in  $recs$  do
20     roinner_insert( $n, r$ );
21   end
22   return  $n$ ;

23 Function rebuild_roinner(Node *n):
24    $recs$  = retrieve all index records rooted by  $n$ ;
25    $n = new\_roinner(recs)$ ;

```

bucket linearly. Until we find an index record with a lower bound key of k , we traverse to its child following the pointer within the found record. If the smallest key in the predict bucket is larger than k , we should only probe the previous bucket as it contains at least a pivot record less than k . Inserting an index record also needs to locate its corresponding bucket, as shown in Algorithm 1, which is similar to the searching operation. If the bucket contains an empty slot (as in the state (a)), the index record is inserted into its proper position. When the bucket is in state (b), inserting the index record will cause an overflow. We use the last two index records to build a new inner node and transit the bucket into the state (c).

3.3 Evolving data layer

The data layer is composed of the leaf nodes in Morphtree. As shown in Fig. 2, accesses to the leaf nodes vary significantly under read-intensive and write-intensive workloads. Thus, we propose to automatically change node layouts of the data layer according to workload changes. More specifically, if the workload becomes read-heavy, we transform the node layout in the data layer into a read-friendly structure; if the workload becomes write-intensive, we change the node layout into a write-friendly structure.

Following the above idea, we first present two types of node layouts, including a read-optimized node layout and a write-optimized layout, for the leaf nodes in the data layer.

The read-optimized node assembles the inner node layout and diverges slightly for better space efficiency. It restricts the linear probing inside a constant number of cache lines, which can reduce the probing cost of read accesses. The write-optimized node layout is log-structured and contains multiple sorted runs and an insert buffer. The insert buffer is used to absorb insertions efficiently. When reaching a threshold, the records in the buffer will be sorted in place, forming a sorted run. In a word, the read-optimized layout is friendly to read queries, and the write-optimized layout is friendly to write operations.

Consequently, we design two kinds of leaf nodes for the data layer, which are **Read-Optimized Leaf (ROLeaf)** and **Write-Optimized Leaf (WOLeaf)**. The data layer will automatically transform the node structure between ROLeaf and WOLeaf to adapt to workload changes.

3.3.1 Read-optimized leaf (ROLeaf) node

When a leaf node is tasked with read-mostly queries, adopting a read-optimized node layout is advantageous for good performance. The ROLeaf nodes are structured in Fig. 5a. It contains a learned linear function and a bucket array, similar to an inner node. The bucket array stores all data records, and the number of buckets is constant. Each bucket contains K cache lines and therefore $4K$ data record slots (we choose K to be 2 in practice). An ROLeaf node is built from a certain number of data records. Similarly, we learn a linear function to reflect the distribution of all initial keys by *least squares line fitting*, by which we map each data record to a bucket. Data records are placed in a bucket according to their key-ascending order. For buckets that are assigned with exceeding data records, we store those additional data records in an overflow node. An overflow node is a contiguous variable-length record array. Consequently, a bucket is in one of the following three states: (a) contains empty slots, (b) is just full, and (c) has an overflow node.

Why are ROLeaf nodes read-optimized? When searching a data record, we leverage the linear function to locate its exact bucket. Then, we probe the bucket linearly to locate the data record. If not found, a binary search is done to retrieve it in its overflow node. Under real-world datasets, more than 70% of data records reside in the bucket array, which means we can restrict the search overhead of most data records to 2 cache lines. For keys under hard-to-fit key distribution, we split an ROLeaf node into two to better fit all data records with two linear functions and lower the ratio of overflow data records.

Write operations on ROLeaf nodes. Inserting a data record into an ROLeaf node should first locate the bucket by the linear function. According to the state of a bucket, the following steps diverge. If the bucket contains free slots, we insert the record and keep the order. But if the bucket is in

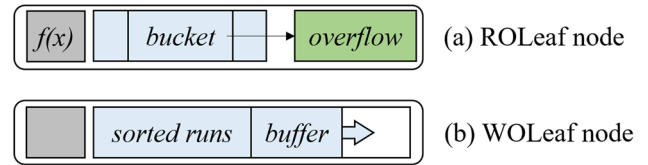


Fig. 5 The structure of ROLeaf and WOLeaf nodes

Algorithm 2: Lookup in a WOLeaf node

```

1 Function woleaf_lookup(Node *n, key_t k):
  // access monitoring information
2 if sampling this access then
3   shove "R" into n.history;
4   if n.type != preferred type of n then
5     n.pLSN = n.cLSN; n.cLSN = gLSN;
6     add record < gLSN, n.ROLeaf > into To-morph
7     log;
8     gLSN += 1;
9   end
10 end
11 // lookup process
12 for run in n.sorted_runs do
13   v = binary_search(run, k);
14   if v != NULL then
15     return v;
16   end
17 end
18 for rec in n.buffer do
19   if k == rec.key then
20     return rec.val;
21   end
22 end
23 return NULL;

```

state (b) or (c), inserting more records will cause an overflow, and an overflow node is used. Updating a data record is just a Compare-and-Swap to its payload, and deleting a data record is the inverse operation of inserting.

3.3.2 Write-optimized leaf (WOLeaf) node

When a leaf node is tasked with insertion-mostly queries, adopting a write-optimized node layout is promising for delivering good performance. The structure of WOLeaf nodes is illustrated in Fig. 5b. A WOLeaf node contains a header and a contiguous slot array. The slot array comprises several sorted runs and an unsorted insert buffer. The insert buffer absorbs new records in an append-only manner. When it reaches a predefined size (in the experiment, we set it to be 1/10 of the leaf node capacity), it is sorted in place to be a sorted run. When one writer starts to sort a run in place, other readers on the same WOLeaf node can read data records within its preceding read-only sorted runs. Only if they fail to find it there, will they wait for the sorting finishes. Besides, the sorting will not block other ongoing writers. The layout of WOLeaf nodes resembles LSM-tree's first level (L0)

but incurs no extra data copy and compaction overhead as in LSM-tree. Data records in different runs are merge-sorted when doing node splitting, and then two new WOLeaf nodes are generated, each with only one sorted run.

Why are WOLeaf nodes write-optimized? Insertions are efficient because data records are appended to the insert buffer. An insert buffer is sorted when the buffer reaches its size limit. The sorting overhead is amortized to each data record, which is trivial compared to random write cost in existing learned index candidates. Insertions in the WOLeaf node are write-optimized for the following reasons: (1) all writes to a WOLeaf node are centred at the front of the insert buffer, which enables caching as many as insert buffers into the CPU cache. (2) the hardware prefetch in a modern CPU will prefetch cache lines of the insert buffer. (3) real-world insertions tend to have space locality regarding leaf nodes.

Search operations on WOLeaf nodes. Lookup operations on WOLeaf nodes are described in Algorithm 2. A lookup operation starts with access monitoring information, which will be elaborated in Sect. 4. Then, it searches each sorted run from left to right until it finds the corresponding record. If the record is not found, the algorithm probes the buffer linearly. For scan operations, we perform a multi-way sort merge for all the sorted runs after sorting the insert buffer and retain a given number of data records on the fly.

3.3.3 Node splitting in the data layer

When the data records in a leaf node reach the capacity limit, we do a node splitting. The splitting of ROLeaf and WOLeaf nodes are similar: (1) Dump all data records of a leaf node to achieve a sorted run. (2) range partition all records into two parts, with which to build two new leaf nodes adopting the same node layout as the original one. (3) replace the old node and install the two new leaf nodes.

4 Node morphing

Morphtree can automatically change the node layouts of the data layer according to workload changes. Specifically, we transform the structure of a node into WOLeaf if the node shows write tendency or into ROLeaf if the node has read tendency. Figure 6 shows the state machine of a node in the data layer of Morphtree. It explains the layout transformation of a node. The first leaf node of an empty Morphtree is a WOLeaf node. If data are bulk-loaded into Morphtree, all the generated leaf nodes become ROLeaf nodes. In addition, all node-splitting operations will not change node layouts. Suppose the access monitoring information, i.e. the sampling history, indicates that a WOLeaf node underwent more than 70% read accesses or an ROLeaf node underwent less than 12% read accesses. In that case, the node will be transformed

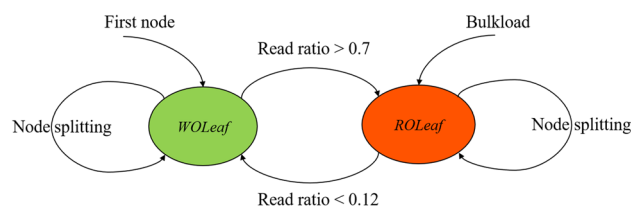


Fig. 6 State machine of a node in the data layer

to its opposite layout. Here, the morphing thresholds (70% and 12%) are empirical settings based on the experimental results. Note that the morphing threshold is used to determine the best time for morphing leaf nodes. As we found in the experiments, when the read ratio of a WOLeaf node exceeds 70%, morphing the WOLeaf node into a ROLeaf node is a better choice, and a ROLeaf node should be changed into a WOLeaf node if its read ratio is less than 12%. It is worth studying some machine-learning methods to find an optimal setting for the morphing threshold, but it does not change the main idea of Morphtree, and we leave it to future works.

The node morphing is realized by maintaining node-level access monitoring information that records the accessing history of each node. Then, we use the access monitoring information to determine the appropriate node layout for the node. The node morphing algorithm presents three new designs. First, we offer a sampling approach to maintain the access monitoring information for each node. Second, to avoid performance degradation, we devise a non-blocking node morphing algorithm to perform node morphing in the background that will not block subsequent accesses to the node. Third, we improve the node morphing performance by reducing unnecessary node morphing for the nodes with rapidly changing access patterns.

4.1 Access monitoring

We add statistics inside each leaf node to record its accessing history (read and write access times). Specifically, a search or update operation to a leaf node is regarded as a read request to the node because the two operations follow a similar data-access flow. Similarly, an insertion or deletion operation to a leaf node is treated as a write access to the node. In addition, a scan operation is treated as multiple read accesses to the node because it needs to access multiple records. Upon each access, the statistics of the accessed nodes in the data layer are updated accordingly. To reduce the overhead involved with monitoring node access under concurrent settings, we utilize an 8-byte variable to record a node's accessing history; concurrent threads on the same node update this variable blindly, without explicit synchronization. It is viable because (1) Modern CPUs guarantee 8-byte write atomicity; (2) inaccurate history leads to no correctness issue; (3) the concurrent thread number is much fewer than the number of leaf nodes,

meaning that there will be low contention on the same leaf node.

We maintain a fixed-size monitoring window W and monitor every other X access to a leaf node. The sampled oldest access will be shoved out from W by the sampled newest access. Finally, we obtain a sampled access history of the leaf node. Through this, we can determine the appropriate layout for the node. The parameters W and X are configurable knobs to control the accuracy and frequency of node morphing. They can be tuned in real applications according to the properties of workloads. For example, a moderate window length with a small X is suitable for frequently changing workloads, while a larger window with a large X should be better for less changing workloads. In our practice, we set W to 64 and X to 0, meaning that we monitor every access to the data layer.

4.2 Non-blocking node morphing

Whenever Morphtree needs to transform the layout of a node in the data layer, it adds a morph record into a memory-resident *To-morph log*. A morph record contains the node address, a log sequential number (LSN), and the expected type of node layout (as shown in Fig. 3). A background morphing thread is notified to scan the log to perform node morphing without blocking subsequent read and write requests.

The key idea of non-blocking node morphing is that we allocate a shadow node with the new layout for the original node, and a background morphing thread migrates the records in the original node to the shadow node sequentially. The procedure of node morphing consists of the following steps:

- (1) Link the shadow node to the original one and mark the original node in-morphing.
- (2) Copy all old data records in the original node to the shadow node while coordinating with ongoing accesses.
- (3) Mark the original node in-freezing.
- (4) Put the shadow node into the data layer first. If we are morphing WOLeaf nodes, check if there are remaining records in the buffer and copy them into the new ROLeaf.
- (5) Reclaim the origin node.

Figure 7 illustrates the main process of transforming ROLeaf to WOLeaf or vice versa. When a node is marked in-morphing but not in-freezing (steps 1–2), all lookup and insertion queries are unblocked, but they need to first read the node header to know whether the node is in morphing. When the node is morphing, lookup queries need to read the insert buffer in the shadow node (if not found in the original node) because new data records may exist in the buffer when transforming a node from ROLeaf to WOLeaf. However, when

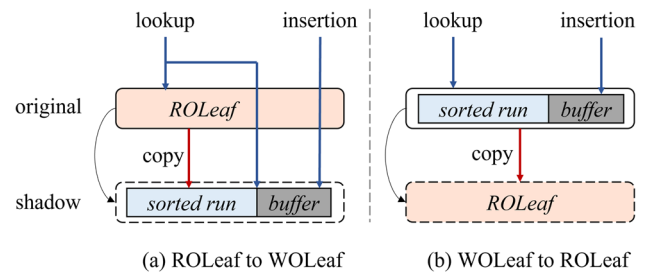


Fig. 7 Coordination of lookup and insertion operations when doing node morphing in the background. **a** ROLeaf morphs into WOLeaf; **b** WOLeaf morphs into ROLeaf

transforming a node from WOLeaf to ROLeaf, all lookups and insertions can be evaluated in the original node, and no access to the shadow node is required. For update operations in both cases, we need to update both the original and shadow nodes to ensure the correctness of lookup queries.

There is a critical section in step (4), which is guarded by a node-level spin-lock [1] in the header. In the critical section, we need to put the shadow node into the data layer and make it a normal node in the data layer. Some Compare-and-Swap updates to 8-byte pointers have to be performed during the process, and the spin-lock is efficient to realize such an operation.

4.3 Avoid unnecessary morphing

One problem of node morphing is that it incurs frequent morphing operations if the workload changes between read-intensive and write-intensive frequently and periodically. For example, a workload becomes read-intensive but only keeps this tendency for thirty seconds; then, the workload becomes write-intensive but only lasts for twenty seconds. In such cases, Morphtree will be busy in node morphing but has limited benefits because the workload will soon become read-intensive after we change it to WOLeaf and vice versa. Thus, we develop efficient ways to avoid unnecessary morphing operations in Morphtree.

Specifically, we record a global logging sequential number (LSN) for each morph record in the *To-morph log*. The LSN of a newly inserted morph record is always larger than any LSN of previous morph records. Thus, we can know that a larger LSN means that the associated morph record is more up-to-date. We maintain two LSN s for each node, namely $cLSN$ (i.e. current LSN) and $pLSN$ (i.e. previous LSN), which are the two recently LSN s associated with a node. A node's $cLSN$ should always be the maximum number among all LSN s associated with the node. Thus, we have $cLSN > pLSN$. Then, when the background morphing thread processes a morph record, it compares the LSN within the record with the node's $cLSN$. We skip the morph record if the node's LSN is smaller than its $cLSN$, which

Algorithm 3: Insert a record in Morphtree

```

1 Function Insertion(key_t k, value_t v)
2   cur = root;
3   while cur.type != ROLeaf or WOLeaf do
4     cur = roinner_lookup(cur, k);
5   end
6   if cur.type == ROLeaf then
7     splitKey, newLeaf = roleaf_insert(cur, k, v);
8   else
9     splitKey, newLeaf = woleaf_insert(cur, k, v);
10  end
11  if newLeaf != NULL then
12    // a node splitting occurred
13    roinner_insert(root, splitKey, newLeaf);
14  end

```

means that the morph record is out-of-date. We need not process node morphing for this record because the node morphing indicated by $cLSN$ will change the node layout again. Further, we skip the morph records with a small difference value of $cLSN - pLSN$, indicating that back-and-forth node morphing may be incurred.

5 Operations and analysis of Morphtree

5.1 Operations

In this section, we detail the operations of Morphtree. Morphtree provides lookup, insertion, update, delete, scan, and bulk-load operations. Below, we only describe insertion, lookup, scan, and bulk-loading due to the space limit. Note that node morphing in the data layer is an internal operation that is not open to applications.

Insertion. Insertion operations (in Algorithm 3) are costly in classical learned indexes because they may trigger node splits and expansions. In Morphtree, we improve the insertion operation by absorbing many insertions in WOLeaf nodes. We also propose a lightweight overflow mechanism in ROLeaf nodes to reduce node expansions. We first locate the target leaf node and then dispatch corresponding insertion functions (lines 7–10) according to its leaf type. If inserting a new record into the leaf node triggers a node splitting, we install the new node into the root node by Algorithm 1.

Lookup. Lookup operation uses a given lookup key to locate the data record with the matching key. To search for a key, we first start at the root node, search in the root node, and locate a child node. If the child node is another inner node, we keep searching until we reach a leaf node. The last mile is searching in the leaf node, which can be an ROLeaf or WOLeaf node. According to the node type, we can dispatch the corresponding node searching function and get the data record.

Scan. The scan operation uses the search key to locate the first data record and then retrieve the consecutive $len - 1$ records. We first locate the corresponding leaf node in the scan operation, similar to the lookup operation, and then scan the leaf node. A scan operation in an ROLeaf node probes forward to retrieve len records because the records in an ROLeaf are globally ordered. The scan operation in a WOLeaf node is a bit more complicated, as we need to merge the results from several sorted runs by the sort-merge algorithm.

Bulk-load. Existing learned indexes offer the bulk-load operation to build an index tree from a given number of data records. They learn the data distribution from the original data records and assume that the subsequent records follow a similar distribution. Morphtree also supports building up an index from scratch via a bulk-load operation. The bulk-load procedure can be divided into three stages: (1) perform a range-based partition to divide the data records into fixed-size pieces (half of the leaf node's capacity). (2) for each piece, build an ROLeaf node and output an index record pointing to the address of the ROLeaf node. The key of the index record is set to the first key of the piece. (3) use all index records to build the inner search tree in a top-down manner. It builds an inner node with learned models mapping keys to buckets and builds overflowed child nodes if necessary. When a bulk-load operation is finished, Morphtree is tentatively a read-optimized index structure because all the nodes in the data layer are with the ROLeaf node layout. On the other hand, if we build Morphtree from scratch by inserting all records consecutively, Morphtree is supposed to be a write-optimized index structure, and all the nodes in the data layer are with the WOLeaf node layout.

5.2 Cost analysis

Suppose Morphtree contains N data records, and the max capacity of a leaf node is M . The leaf nodes split like B+-tree, so the average load factor of a leaf node is 70% [41]. Therefore, there are $\frac{N}{0.7M}$ leaf nodes. According to the design criteria stated in Sect. 3, we rebuild a sub-tree to ensure that at least 50% of the index records are directly pointed by the root of the sub-tree. We define the height of a leaf node to the number of node accesses in the path from the root down to the leaf node. For example, if a leaf node is pointed directly by the root node, its height is 2.

Theorem 1 *Let the Max Height (MH) of Morphtree be the max height of all the leaf nodes, and the upper bound of MH is $O(\log_2 N)$.*

Proof The leaf node with the max height is the one whose index records always overflowed into a sub-tree in each level. Suppose that a tree contains N data records, and the max capacity of a leaf node is M , there are $\frac{N}{0.7M}$ index records in total. Among each level, at least half of the index records

are directly pointed by the inner node, which means from the root node down to this leaf, there are at most x inner nodes, in which

$$x = \log_2\left(\frac{N}{0.7M}\right).$$

Thus, the max height of Morphtree can be formulated as

$$MH \leq x + 1 = \log_2\left(\frac{N}{0.7M}\right) + 1.$$

As N is far larger than M , we can conclude that the upper bound of MH is $O(\log N)$. \square

The average height reflects the average performance of Morphtree. Next, we prove that the average height of Morphtree will not exceed three. Conceptually, the average height should be the sum of height divided by the number of leaf nodes. Since there are multiple leaf nodes at the same level, we choose to calculate the height of the leaf nodes at every level and then get the average height of the tree.

Theorem 2 *The Average Height (AH) of Morphtree will not exceed three.*

Proof Let the root node be the first level of Morphtree, and the child nodes of the root node be the second level, as the root node must have at least 50% index records stored inline, there will be $\frac{1}{2}$ of leaf nodes at the second level (height = 2). Accordingly, there are $\frac{1}{4}$ of leaf nodes at the third level (height = 3). Therefore, we can infer that $\frac{1}{2^i}$ of leaf nodes are at the $i + 1$ level (height = $i + 1$). Then, we can calculate the average height of Morphtree AH by the following formula.

$$AH \leq \sum_{i=1}^{MH-1} \frac{i+1}{2^i} \leq \sum_{i=1}^{\infty} \frac{i+1}{2^i} \leq 3$$

Thus, the average height of Morphtree will not exceed three. \square

Generally, the lookup time on a tree index mainly consists of the root-to-leaf traversing time, which is determined by the index height, and the time for reading the nodes in the accessing path [5]. Morphtree can ensure that searching an inner node only needs to fetch one cache line. Thus, the lookup time on Morphtree is dominated by the index height. To this end, reducing the index height can fasten the lookup performance. Theorem 2 indicates that the average tree height of Morphtree will not exceed three, which provides the upper bound of the lookup time on Morphtree. Note that Theorem 2 does not declare that the tree height of three layers is optimal. Instead, it only provides the upper bound of the average tree height for Morphtree. Previous work [5] has revealed that the tree height of an optimal index on persistent storage, such as

SSD and cloud storage, relies on the system environments, especially on the I/O characteristics of storage devices. While the current Morphtree is not optimized for persistent storage, in the future, we will improve Morphtree to make it adapt to different system environments, workloads, and data distributions when migrating Morphtree to persistent storage.

Cost of traversing the tree. According to Theorems 1 and 2, the lower bound of the tree traversal cost is accessing two cache lines, one for accessing the node header and another for probing in the predicted bucket. The upper bound of tree traversal cost is $2 \times MH$ cache line accesses, and the average tree traversal cost will not exceed six cache lines.

Cost of accessing ROLeaf or WOLeaf. For lookup in an ROLeaf node, most data records can be accessed within $k + 1$ cache line accesses, k accesses for probing in the predicted bucket, and one for accessing the node header. For the data records that reside in the overflow node of each bucket, an additional binary search costs $\log_2(\text{OF})$ cache line accesses, where OF is the average number of data records in overflow nodes. For insertions that cause no node splitting, the accessing cost is similar to lookup operations.

An insertion to a WOLeaf node takes two cache line accesses, one for accessing the header and another for appending to the in-node buffer. For a lookup in a WOLeaf node, we need to perform at most nine binary searches ($\log_2(\frac{M}{10})$ for each sorted run) and a linear probing in the buffer (at most $\frac{M}{10}$ data records in the buffer, a cache line contains four data records). Thus, the number of cache line accesses is $9\log_2(\frac{M}{10}) + \frac{M}{40} + 1$.

Cost of training a learned node. In our Morphtree, we use *least squares line fitting* to fit a linear model to a given set of keys. It needs to scan each index record once, do some calculations, and regress a linear function. The computation cost for each record is several float-point operations within tens of CPU cycles, which is trivial compared to memory cost that takes tens of nanoseconds. Therefore, if we trained a learned node with X index records, the training cost is $O(X)$.

Memory consumption of Morphtree. Like classical B+-tree, a leaf node of Morphtree splits when it reaches the leaf node's max capacity, so the load factor of leaf nodes is similar to B+-tree, which is proved to be $1/\sqrt{2} \approx 0.7$ [41]. For each ROLeaf node, overflowed data records (at most 30% as stated in 3.3.1) will take at most 30% extra memory. Considering the 16-byte data record, Morphtree will occupy at most $16 \times N/0.7 \times (1 + 0.3) = 29.7N$ bytes for all leaf nodes.

The inner search tree needs to store all index records to $\frac{N}{M \times 0.7}$ leaf nodes, and each index record occupies 16 bytes (8 bytes for the key and 8 bytes for the child pointer). The upper bound memory consumption for the root node is $\frac{N}{M \times 0.7} \times 16$ bytes as it should be able to accommodate all $\frac{N}{M \times 0.7}$ index records. For the second inner node level, the upper bound

memory consumption is then $\frac{N}{M*0.7} * 16/2$ as at most 50% index records fall into the second level. We can also infer that the upper bound memory consumption of the i -th level inner nodes is $\frac{N}{M*0.7} * 16/2^{i-1}$. So the upper bound of memory consumption of inner search tree M_I is

$$\begin{aligned} M_I &\leq \sum_{i=1}^{MH-1} \frac{N}{M*0.7} * 16/2^{i-1} \\ &\leq \frac{N}{M*0.7} * 16 * 2 = \frac{45.7N}{M} \end{aligned}$$

In practice, if we choose M to be 10,000, the upper bound memory consumption of the inner search tree is $0.00457 * N$ bytes, which is trivial compared to leaf nodes. To sum up, the memory consumption of Morphtree can be bound to $29.7 * N$ bytes.

6 Evaluation

In this section, we evaluate Morphtree by comparing it with several existing learned indexes under various real-world datasets and dynamic workloads.

6.1 Experimental setup

The settings of the experiments include several aspects, which are detailed below.

Datasets. We use three real-world datasets and one synthetic dataset to evaluate learned indexes. The three real-world datasets include Facebook (FB), Longitude (LGT), and Long-Latitude (LLT), which were used in previous works including ALEX [7], LIPP [36] and SOSD [29]. The LGT dataset consists of the longitudes of locations around the world from Open Street Maps, and the LLT dataset consists of compound keys that combine the longitudes and latitudes from Open Street Maps by applying a transformation to these pairs, yielding a severely nonlinear distribution of the data. The synthetic dataset, denoted as Lognormal (LGN), is generated artificially according to the log-normal distribution. For each real-world dataset, we extract 200 million data records for benchmarking after shuffling each dataset randomly, and for the LGN dataset, we just generate 200 million artificial records. Each record consists of an 8-byte double-type key and an 8-byte payload.

Competitors. We compare Morphtree with four updatable learned indexes, including ALEX, PGM-index, FITing-tree, and LIPP, with their open-source code from Github. Also, we implement two variants of Morphtree, namely ROTree and WOTree, which do not use node morphing. Note that ROTree and WOTree are introduced to verify the benefits of

node morphing in Morphtree. All the compared indexes are described as follows.

- **B+-tree**.¹ We include the traditional in-memory B+-tree in the experiments on static workloads, as B+-tree is commonly used in existing database systems.
- **ALEX** [7]. It is an in-memory learned index that utilizes a gapped array to absorb insertion queries while doing an exponential search in leaf nodes.
- **PGM-index** [9]. This is the first learned index with a bounded prediction error, which uses a differential structure to support insertion queries.
- **FITing-tree** [10]. It replaces B+-tree's leaf nodes with learned data nodes to reduce the tree height.
- **LIPP** [36]. This is the first index structure guaranteeing that searching in each node is accurate without a probing error.
- **XIndex** [34]. This is the first concurrent learned index, which will be compared in the multi-threaded experiment.
- **FINEdex** [21]. A recent concurrent learned index that offers good write performance will be compared in the multi-threaded experiment.
- **Morphtree**, which is the polymorphic main-memory learned index proposed in this paper.
- **ROtree**. This is a variant of Morphtree. The data layer of ROTree is only composed of ROLeaf nodes and does not morph with time.
- **WOTree**. This is another variant of Morphtree, whose data layer is only composed of WOLeaf nodes and does not morph with time.

Environment. We run experiments on a server equipped with Intel® Xeon® Gold 6242 CPU at 3.1 GHz. The CPU contains 40 cores, and each core owns a 32KB iCache, a 32KB dCache, and a 1 MB L2 cache. For single-threaded evaluation, we turn off the background morphing in Morphtree and run all experiments with a single thread to make the comparison fair enough. Therefore, the query and morphing operations of Morphtree are within the same benchmark thread. For multi-threaded evaluation, we turn on the background morphing for Morphtree. The operating system on the server is Ubuntu with a kernel version of 5.4.0. All codes are implemented in C++ and compiled with the -O3 option.

Workloads. Each index is warmed up by writing the initial 64 million records before every execution. In the initial 64 million records, 25% are bulk-loaded to the index, and the rest are inserted into the index one by one. Such a warm-up is to simulate the data-input scenarios in real applications, where we learned from a partial dataset and populated new

¹ STX B+-tree: <https://panthema.net/2007/stx-btree/>

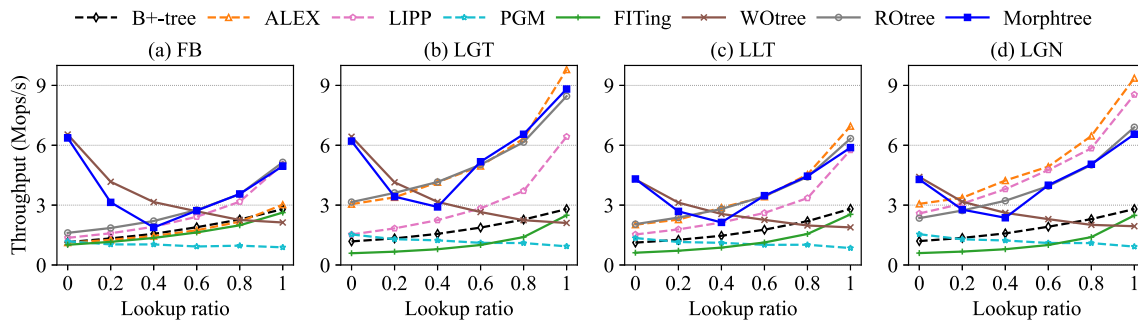


Fig. 8 Throughput of B+-tree and updatable learned indexes with various ratios of lookup queries under the four datasets

data records with insertion operations. Afterwards, we test each index with different query workloads. As Morphtree is intended for optimizing insertion queries for learned indexes, we make all workloads consist of lookups and insertions. All lookup keys are generated under the Zipfian distribution to the initial dataset with a skewness value of 0.99. The workloads in our evaluation can be categorized into two types.

(1) *Static workloads.* All static workloads consist of 128 million lookups and insertions with six different lookup ratios ranging from 1 to 0 (i.e. 1, 0.8, 0.6, 0.4, 0.2, 0), representing the ratio of lookups among the total queries.

(2) *Dynamic workloads.* The dynamic workloads consist of 288 million queries whose lookup ratio shifts from 1 to 0 step by step or vice versa. Between every two shifts, there are 48 million queries.

6.2 Performance on static workloads

The performance on static workloads is shown in Fig. 8. Each sub-figure shows the throughput of the seven indexes under different lookup ratios ranging from 0 to 1. The lookup ratio is 0 means that the workload only contains insertions.

Figure 8a–d show the results for FB, LGT, LLT, and LGN, respectively. When the lookup ratio is near zero, almost all existing learned indexes and ROTree exhibit a low throughput, roughly at 2 Mops/s. However, WOTree and Morphtree show 2–4x higher insertion throughputs than the others, owing to the data layer of WOTree/Morphtree that is optimized for insertion queries. With the increase of the lookup ratio, the performance of B+-tree, ALEX, LIPP, FITing-tree, and our ROTree is all improved. The PGM-index exhibits oppositely as its dynamic parts employ an LSM-tree-like multi-level structure, which is friendly to insertion queries but not to lookups. When the query workload becomes lookup-only, ALEX, ROTree, and Morphtree achieve up to a 7–10 Mops/s lookup throughput. We can conclude that ALEX, LIPP, FITing-tree, and ROTree are lookup-optimized, while PGM-index and WOTree are insertion-optimized. None of them are excellent at both lookup-intensive and insertion-intensive workloads.

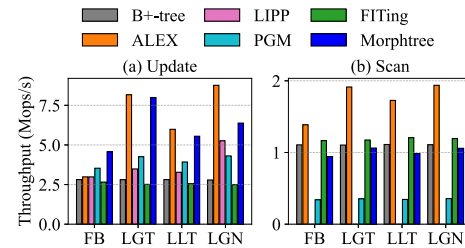


Fig. 9 Update and scan performance of B+-tree and updatable learned indexes

Morphtree combines the merits of WOTree and ROTree and achieves high performance under lookup-intensive and insertion-intensive workloads. In each sub-figure, Morphtree has a V-shaped trend of performance. On the one hand, the V-shaped trend shows that Morphtree is more versatile than other indexes. On the other hand, it reveals the complexity of devising a learned index with high performance under hybrid lookups/insertions workloads. Moreover, most existing learned indexes perform relatively poorly in the FB dataset because the data distribution in FB is not linear, i.e. not friendly to linear models that have been used in most existing learned indexes. For LGT and LLT datasets, which are also real-world datasets, existing learned indexes perform relatively better. That is because LGT is easy to fit with linear models. For the LGN dataset, which is generated synthetically, ALEX and LIPP achieve high lookup performance.

We report the performance of update and scan queries in Fig. 9. Morphtree achieves better update performance than existing indexes except for ALEX for at least 40%. All scan queries in Fig. 9b are with random length, less than 100 data records per query. Morphtree performs similarly to B+-tree and learned indexes such as PGM-index, and FITing-tree. As the open-source code of LIPP contains no scan interface, we leave it blank. ALEX delivers the best scan performance because of its compacted node layout.

To sum up, when evaluated under the four types of datasets, existing learned indexes usually incur two problems. First, they have poor insertion performance compared to lookup queries. Second, their performance is highly

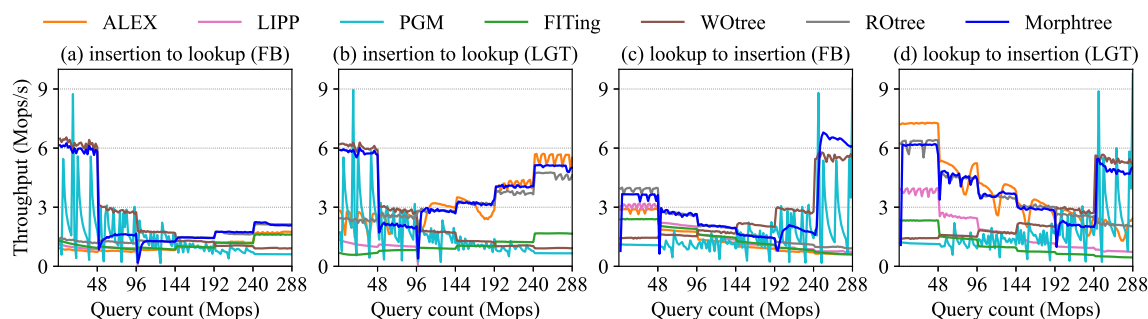


Fig. 10 Throughput under a dynamic workload that contains changing query patterns. **a** and **b** are the results of the query workloads under the FB and LGT datasets that shift from insertion-only to lookup-only

queries; **c** and **d** contain the results for the query workloads shifting from lookup-only to insertion-only queries

affected by the data distribution. On the contrary, Morphtree maintains its versatility under different types of datasets and can overcome these problems, and the experimental results in Fig. 8 support this claim.

6.3 Performance on dynamic workloads

In this experiment, we evaluate Morphtree on two types of dynamic workloads: (1) varying the ratio of lookup and insertion queries. (2) varying the dataset distributions.

Note that as shown in Fig. 8, the performance of B+-tree is much worse than that of Morphtree, and B+-tree is not a learned index. Thus, we do not include B+-tree as a competitor in the experiments on dynamic workloads.

6.3.1 Varying the ratio of lookups/insertions

This experiment reveals the throughput of indexes under dynamic workloads shifting from insertion-only to lookup-only queries (Fig. 10a, b) and shifting from lookup-only queries to insertion-only queries (Fig. 10c, d). In each test, there are six stages and 288 million queries in total. The FB and LGT datasets are used in this experiment.

As shown in Fig. 10a, b, as the query pattern shifts from insertion-only to lookup-only, the throughputs of existing read-optimized learned indexes (ALEX, FITing-tree, and ROTree) also increase. All of them can maintain a stable throughput until the lookup ratio changes. Other indexes like WOTree and PGM-index perform oppositely, and their performance decreases with the increase of the lookup ratio, which is consistent with their design choices. However, there are two exceptions for LIPP and Morphtree. LIPP encounters an assertion failure after 96 million queries are performed. Thus, we can not report its subsequent performance in the next queries. Our Morphtree performs similarly to WOTree on insertion-only workloads. Between 48 to 96 million queries, there is a performance drop of Morphtree because some leaf nodes start to morph into ROLeaf node layouts due to the

change of the query pattern. Such a drop can also be seen at the start of the 97-th million queries. Afterwards, Morphtree starts to perform similarly to ROTree because most of its leaf nodes have been transformed into ROLeaf. In a word, none of the existing learned indexes can adapt to the workload change as all of them are designed for read-heavy workloads. Our Morphtree can adjust its layout between ROTree and WOTree to achieve high performance on dynamic workloads changing from insertion-only to lookup-only.

In Fig. 10c, d, as the query pattern shifts from lookup-only to insertion-only, the throughput exhibits oppositely to that shown in Fig. 10a, b. ALEX, FITing-tree, LIPP, and ROTree encounter a performance drop each time the query pattern shifts a step to insertion-intensive queries, while the throughput of PGM-index and WOTree increase with the insertion ratio. Again, our Morphtree can adjust its data layout, achieving the best average performance at each stage.

We also noticed that some learned indexes exhibited relatively stable performance, but others did not. ALEX, LIPP, FITing-tree, ROTree, and WOTree are stable ones, as they are static in terms of layout. However, PGM-index is not so stable. Our Morphtree exhibits a small performance drop because of node morphing but maintains the highest average performance under dynamic workloads. PGM-index is unstable due to its uncertain and unpredictable internal tree merging operations when new keys are inserted into the index.

In Table 1, we list the total rebuilding frequency of inner nodes and the morphing frequency of leaf nodes, along with their time overhead normalized to the total run-time. We obtain the statistic by Linux Perf during the 288 million queries that vary the ratio of lookups and insertions. The rebuilding frequency is diverse on different datasets. The inner nodes of Morphtree under the FB dataset and the LGT dataset are rebuilt less frequently compared to the other two datasets because these two datasets are easy to fit at macroscale according to their Cumulative Distribution Function (CDF) [29]. The rebuilding overhead counts less than

Table 1 The rebuilding and morphing frequency/overhead under dynamic workloads varying the ratio of lookups/insertions

	FB	LGT	LLT	LGT
Rebuilding frequency	2	25	7858	12,068
Rebuilding overhead	0%	0.15%	0.23%	0.13%
Morphing frequency	56,185	52,112	55,058	52,438
Morphing overhead	8.01%	9.70%	9.81%	9.49%

The overhead refers to the time overhead normalized to the total run time

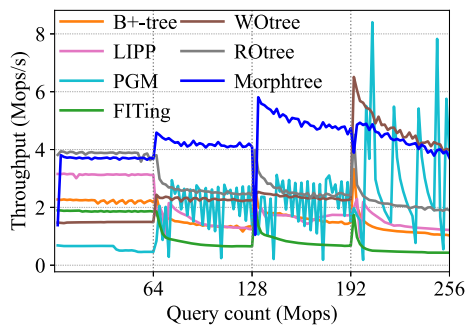


Fig. 11 Throughput of different indexes when inserting new records from different datasets. There are 128 million Zipfian lookup queries mixed up with 128 million insertions from four datasets. The experiment contains four stages, and each stage consists of 64 million queries

1% of the total run-time. The morphing frequency among different datasets is similar because it is only relevant to the query workload. According to our statistics, the morphing overhead is less than 10% of the total run-time. In general, Morphtree has to introduce additional rebuilding and morphing overhead to achieve high insertion performance and adaptability to dynamic workloads. According to the results shown in Table 1, the rebuilding overhead can be neglected, but there might be some space for reducing the morphing overhead, e.g. to make the morphing overhead below 5%.

6.3.2 Varying the dataset distribution

In this experiment, we use a synthetic dynamic workload that contains two types of dynamism: (1) data records with changing distributions are inserted, and (2) lookup queries for keys from changing domains. We populate an index with 64 million data records from the FB dataset and then initiate 128 million Zipfian lookup queries and another 128 million insertions from different datasets. The queries in this experiment are divided into four stages:

- (1) 64 million queries for the data records in the initial index.
- (2) Mixed queries containing 32 million queries for the data records in the initial index and 32 million insertions of the new data records from the LGT dataset.

- (3) Mixed queries containing 32 million queries for the data records inserted in (2) and 32 million insertions of the new data records from the LLT dataset.
- (4) 64 million insertions of the new data records from the LGN dataset.

The throughputs of all indexes are reported in Fig. 11. ALEX fails to support this workload in practice. Thus, we use B+-tree as a baseline of the traditional in-memory index, which is not sensitive to the key distribution. As shown in the figure, existing learned indexes perform similarly with B+-tree under the dynamic workload, while Morphtree exhibits a sustainable 2.9x/3x/2.1x/5x higher throughput than B+-tree/LIPP/PGM/FITing-tree. PGM-index is not stable under insertion queries because of its multi-level structure. In addition, when more data records are inserted into an existing learned index, the throughput of the index drops accordingly. This is because the data records from different datasets follow different distributions, which will invalidate the learned models trained before. However, Morphtree is robust enough to maintain 4–6 Mops/s even under hybrid lookup and insertion queries in stages (2) and (3). Owing to the polymorphic structure, Morphtree can be automatically tuned to adopt read-optimized node layouts or write-optimized ones, making it adaptable to dynamic workloads.

6.4 Performance with multiple threads

In this section, we evaluate Morphtree under multi-threaded configurations. We adopt optimistic lock coupling in Morphtree as in B+-tree, ALEX, and LIPP in [35]. In addition, each node contains an in-node read-write-lock, and each bucket in ROInner and ROLeaf nodes contains an in-bucket spin-lock. A lookup operation does not acquire any locks on a node, and it coordinates with other operations by double-checking the lock versions and deciding to retry or not; a write operation acquires the node's read-lock and the write-lock of an individual bucket it modifies; a morphing or splitting operation acquires the node's write-lock and updates its version.

We compare concurrent versions of B+-tree, ALEX, LIPP (source code obtained from [35]), FINEdex [21] and XIndex [34]. We populate each index with 100 million data records, bulk-load the first half, and then insert the rest. Afterwards, we measure the throughput of each index executing 100 million lookup or insertion operations with multiple threads varying from 2 to 40. All experiments are conducted within one socket, and we report the average value of five runs in each experiment.

Figure 12a, b show the insertion throughput of seven indexes under the FB and LGT datasets. Morphtree exhibits the best insertion performance and multi-threaded scalability. When run with 40 threads, Morphtree achieves

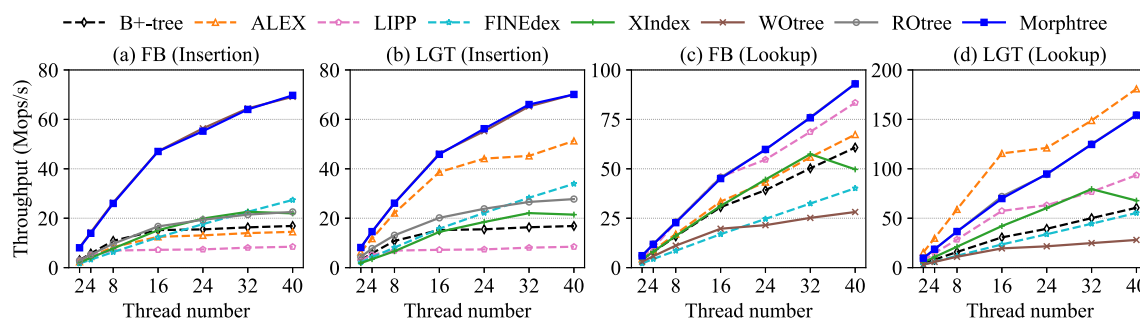


Fig. 12 Throughput under different thread numbers. We compare concurrent learned indexes, including ALEX, LIPP, FINEdex, and XIndex, with our WOTree, ROTree, and Morphtree under the FB and LGT datasets

a 3.13x/3.80x/7.23x/1.54x/2.09x/ 2.17x higher insertion throughput than B+-tree, ALEX, LIPP, FINEdex, and XIndex under the FB dataset and a 3.15x/0.36x/7.27x/1.06x /2.26x higher insertion throughput under the LGT dataset. We report the lookup performance in Fig. 12c, d. Morphtree achieves the best lookup throughput under the FB dataset and the second-best performance under the LGT dataset. Specifically, it outperforms ALEX, LIPP, FINEdex, and XIndex by 0.36x, 0.11x, 1.29x, and 0.85x under the FB dataset. Moreover, the throughput gap of Morphtree under the FB and LGT datasets is less than other learned indexes, suggesting the robustness of Morphtree under different data distributions. In a word, the experiments under multi-threaded suggest the efficiency of Morphtree for both read-intensive and write-intensive workloads.

6.5 Sensitive analysis

In this section, we discuss other aspects that may affect the performance of Morphtree.

6.5.1 On the skewness of lookup keys

The skewness of lookup keys represents the deviation of query keys to hot keys, which is a knob in the Zipfian distribution. We vary this knob from 0.4 to 0.99 and evaluate the lookup performance of the seven learned indexes. The results are shown in Fig. 13. With the increase of skewness, the lookup throughputs of all seven indexes are improved. This is mainly owing to the effect of modern CPUs, i.e. the CPU cache can help cache hot data and reduce access time. When under lookup-only queries, Morphtree exhibits similar performance as ROTree but outperforms all the compared learned indexes under the FB dataset.

6.5.2 On the bulk-load proportion of the initial dataset

Another issue that is always ignored by formal research work on learned indexes is the bulk-load proportion of the initial

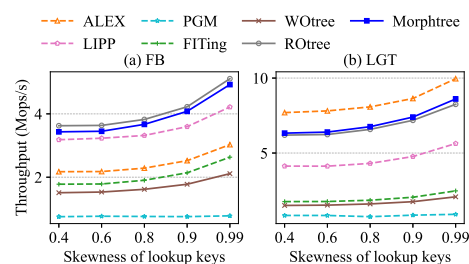


Fig. 13 Impact of the skewness of lookup keys

data. Existing learned indexes tend to bulk-load all initial data and learn the distribution of all initial data before performance evaluation. However, it is also important to see whether a learned index can still maintain high performance when not the whole initial data are bulk-loaded. Thus, we define a *bulk-load proportion* for the initial dataset to represent the proportion of the initial data used by the bulk-load operation.

Figure 14 shows the performance of all the seven indexes on different bulk-load proportions varying from 6.25% to 100%. Some learned indexes, such as ALEX and FITing-tree, are not sensitive to the bulk-load proportion, meaning that as long as the data distribution does not change, they can achieve similar performance as the whole initial data is bulk-loaded. Other indexes like PGM-index and LIPP are highly affected by the bulk-load proportion. As the PGM-index uses an LSM-tree-like structure to support insertions, if a large portion of data is inserted into the PGM-index by insertions, it will result in multiple small learned indexes, which will worsen the lookup performance. LIPP stores data records in predicted positions. When a large portion of data is inserted into LIPP, the tree structure of LIPP tends to be severely unbalanced, which will degrade the lookup performance. Our Morphtree supports building an index without bulk-load. Thus, it can maintain stable performance under various bulk-load proportions. In a word, the bulk-load proportion impacts the lookup performance of existing learned indexes, while the proposed Morphtree is not sensitive to this proportion.

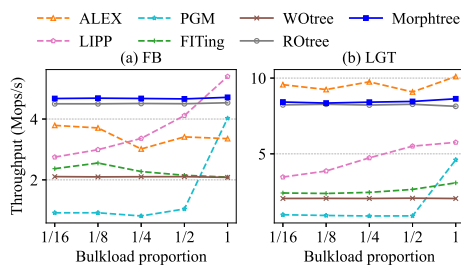


Fig. 14 Impact of the bulk-load proportion of the initial dataset

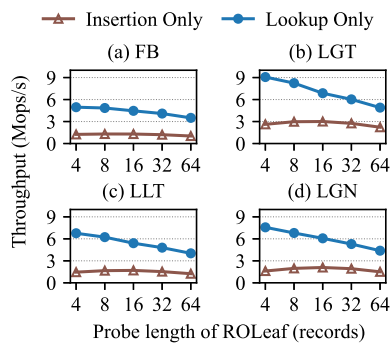


Fig. 15 Impact of the probe length on the performance of ROLeaf

6.5.3 On the probe length of ROLeaf

The probe length of ROLeaf affects both the lookup and insertion performance. According to Fig. 15, the lookup-only performance of ROTree decreases when the probe length of ROTree increases from 4 to 64 records, while the insertion-only performance of ROTree increases between 4 to 16 but drops afterwards. A small probe length benefits access performance but will increase memory consumption under less-linear data distributions. Thus, Morphtree sets the probe length to 8 to balance the lookup performance and memory consumption.

6.5.4 On the buffer size of WOLeaf

In a WOLeaf node, new records are appended to the buffer before it reaches the threshold of the buffer size. To verify how the buffer size will affect the performance of Morphtree, we conduct an experiment on the four datasets and report the throughput of Morphtree under different buffer sizes. The results are shown in Fig. 16, which indicates that the insertion-only performance of Morphtree is affected slightly by the buffer size because the buffer size only affects the in-place sort time of a WOLeaf but does not magnify the total sorting overhead. However, the lookup-only performance of Morphtree increases at first and then drops as the buffer size increases, which is mainly because a large buffer size can help to reduce the number of sorted runs in a WOLeaf node. However, a large buffer size will introduce a large unsorted

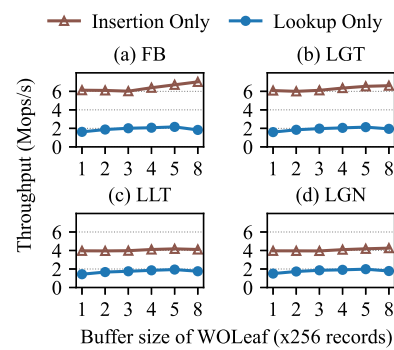


Fig. 16 Impact of the buffer size on the performance of WOLeaf

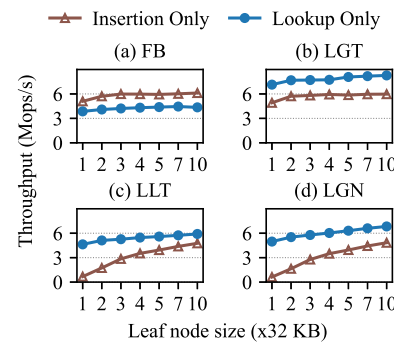


Fig. 17 Impact of the leaf-node size on the performance of Morphtree

run, which will increase the lookup time. Therefore, in our implementation, we choose 1,024 records as the buffer size of WOLeaf.

6.5.5 On the leaf-node size

The leaf-node size is fixed for ROLeaf and WOLeaf, which is beneficial for access monitoring and node morphing. We vary the node size and depict the result of Morphtree in Fig. 17. When the leaf node size increases, the performance of Morphtree shows an increasing trend. Such an increase becomes slow when the leaf node size exceeds 96 KB (one leaf node can accommodate 6,144 records). In general, a large leaf-node size might be helpful for the overall performance of Morphtree. Still, it will affect the accuracy of access monitoring and read/write tendency prediction because a large key range has to be handled. Also, large leaf node size will increase the cost of node morphing. Thus, we set the leaf node size of Morphtree to 160 KB in our implementation, which means that a leaf node can store 10,240 records.

6.5.6 On the morphing threshold of leaf nodes

There are two morphing thresholds determining when to morph a leaf node to another layout; each is in the form of the read ratio encountered by the leaf node. In our previous experiments, the two morphing thresholds are fixed to

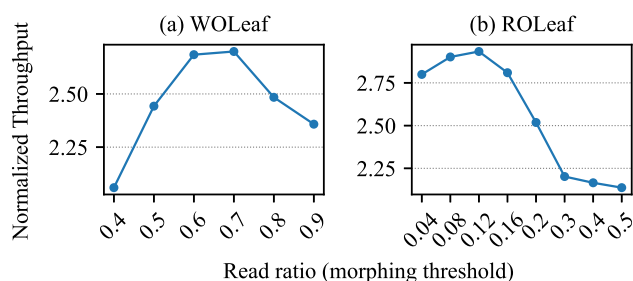


Fig. 18 Impact of the morphing threshold that triggers leaf node morphing in Morphtree

0.7 for WOLeaf nodes and 0.12 for ROLeaf nodes. Note that it is inappropriate to morph a leaf node according to common knowledge, e.g. morphing a ROLeaf node to WOLeaf when the read ratio is less than 0.5. As the morphing operations are costly, we need to carefully determine the morphing time. This subsection explores the impact of these two morphing thresholds on the overall performance of Morphtree under dynamic workloads as in Sect. 6.3.1.

We conduct two experiments that vary the morphing threshold of WOLeaf and ROLeaf nodes, respectively, while fixing the other morphing threshold. Figure 18a shows the throughputs of Morphtree with different WOLeaf morphing thresholds, and Fig. 18b reports the results when varying the morphing threshold for ROLeaf nodes. We can see that the configuration of 0.7 for WOLeaf nodes and 0.12 for ROLeaf nodes achieves the best performance. Note that the best morphing thresholds of WOLeaf are larger than ROLeaf nodes, which can reduce oscillations of node morphing.

Note that empirical settings of the morphing thresholds might not be the best choice for Morphtree, and using some machine-learning techniques might be better. However, learning techniques will cause additional periodical training costs because this study is toward dynamic workloads. To this end, currently, we prefer to use empirical settings for the morphing thresholds. In the future, we will study the feasibility of learning-based threshold determination, e.g. using reinforcement learning models.

6.6 Memory consumption

Memory consumption is also an important metric for index structures, especially in memory-constrained situations. We report the memory consumption of all the seven learned indexes in Fig. 19. PGM-index is the most memory-efficient because it uses little extra memory. LIPP consumes the most memory spaces because it has to use extra memory spaces in each tree level for storing data records in precise positions. Morphtree costs less than 2048 MB of memory, which is similar to B+-tree. In a word, Morphtree can achieve high

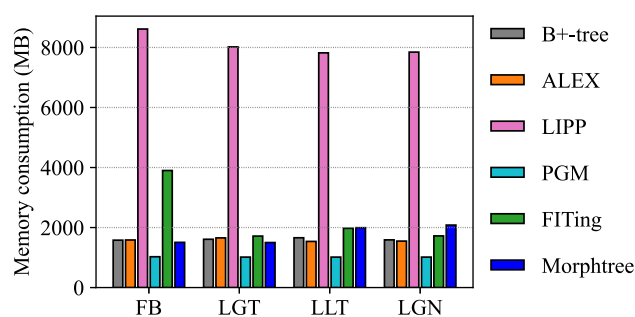


Fig. 19 Comparison of memory consumption

read/write performance with relatively low memory consumption.

7 Conclusions and future work

In this paper, we presented a new adaptive learned index called *Morphtree*, which was designed to optimize learned indexes for both lookup-intensive and insertion-intensive queries. Morphtree supports building up an index from scratch without any original data and can adapt to dynamic workloads by automatically adjusting the index structure. We developed a decoupled index structure for Morphtree, which consisted of a read-optimized learned inner tree and an evolving data layer. The inner search tree is optimized for reducing the tree height and in-node searching costs, while the flexible data layer adopts two types of node layouts that are optimized for lookup and insertion queries, respectively. Moreover, the data layer can automatically change node layouts based on read and write tendencies as the workload changes with time. We compared Morphtree with state-of-the-art learned indexes, including ALEX, PGM-index, LIPP, FITing-tree, FINEdex, and XIndex. The results showed that Morphtree achieved an average 0.56x higher lookup performance and 3x higher insertion performance than the competitors. Also, Morphtree can adaptively change the index structure to maintain high read/write performance for dynamic workloads.

The future work of Morphtree will focus on several aspects. First, we will introduce reinforcement learning to help Morphtree tune parameters on the fly. By leveraging reinforcement learning to tune parameters based on dynamic workload properties, it is promising to control morphing operations more intelligently. Second, with the development of persistent memory [40], developing persistent memory-aware indexes has received much attention [23, 25]. So far, there are few studies toward learned indexes on persistent memory [22, 44]. Thus, we will migrate Morphtree to persistent memory in the future, e.g. by putting the inner search tree into DRAM while redesigning the data layer to adapt to persistent memory. This direction will focus on devising a

crash-consistent and durable version of Morphtree. Finally, we will optimize Morphtree to make it work for persistent storage [6, 19, 26]. Previous works [4, 5] have indicated that the design choices of an optimal index structure vary vastly when considering the different I/O characteristics (e.g. latency and bandwidth) of persistent storage devices, such as SSD and cloud storage. Therefore, in addition to workload changes, we need to consider other factors like storage performance profiles and data distributions when migrating Morphtree to persistent storage.

Acknowledgements We would like to thank the editor and anonymous reviewers for their valuable comments. We also thank the KeeWiDB group in Tencent for their suggestions on improving the experiments in the paper. This work was supported by the National Science Foundation of China (No. 62072419) and Tencent. Peiquan Jin is the corresponding author of the paper.

References

1. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* **1**(1), 6–16 (1990)
2. Anneser, C., Kipf, A., Zhang, H., Neumann, T., Kemper, A.: Adaptive hybrid indexes. In: *SIGMOD*, pp. 1626–1639 (2022)
3. Athanassoulis, M., Kester, M.S., Maas, L.M., Stoica, R., Idreos, S., Ailamaki, A., Callaghan, M.: Designing access methods: the rum conjecture. In: *EDBT*, pp. 461–466 (2016)
4. Chockchowwat, S., Liu, W., Park, Y.: Automatically finding optimal index structure. In: *AIDB@VLDB*, pp. 1–5 (2022)
5. Chockchowwat, S., Liu, W., Park, Y.: Airindex: versatile index tuning through data and storage. *arXiv preprint arXiv:2306.14395* (2023)
6. Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: From WiscKey to Bourbon: a learned index for log-structured merge trees. In: *OSDI*, pp. 155–171 (2020)
7. Ding, J., Minhas, U.F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D.B., Kraska, T.: ALEX: an updatable adaptive learned index. In: *SIGMOD*, pp. 969–984 (2020)
8. Dittrich, J., Nix, J., Schön, C.: The next 50 years in database indexing or: the case for automatically generated index structures. *Proc. VLDB Endow.* **15**(3), 527–540 (2021)
9. Ferragina, P., Vinciguerra, G.: The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* **13**(8), 1162–1175 (2020)
10. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., Kraska, T.: FITing-Tree: a data-aware index structure. In: *SIGMOD*, pp. 1189–1206 (2019)
11. Idreos, S., Callaghan, M.: Key-value storage engines. In: *SIGMOD*, pp. 2667–2672 (2020)
12. Idreos, S., Kersten, M.L., Manegold, S., et al.: Database cracking. In: *CIDR*, vol. 7, pp. 68–78 (2007)
13. Idreos, S., Manegold, S., Kuno, H.A., Graefe, G.: Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.* **4**(9), 585–597 (2011)
14. Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M.S., Guo, D.: The data calculator: data structure design and cost synthesis from first principles and learned cost models. In: *SIGMOD*, pp. 535–550 (2018)
15. Jain, V., Lennon, J., Gupta, H.: LSM-trees and B-trees: the best of both worlds. In: *SIGMOD*, pp. 1829–1831 (2019)
16. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In: *SIGMOD*, pp. 339–350 (2010)
17. Kornaropoulos, E.M., Ren, S., Tamassia, R.: The price of tailoring the index to your data: Poisoning attacks on learned index structures. In: *SIGMOD*, pp. 1331–1344 (2022)
18. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: *SIGMOD*, pp. 489–504 (2018)
19. Lan, H., Bao, Z., Culpepper, J.S., Borovica-Gajic, R., Dong, Y.: A simple yet high-performing on-disk learned index: Can we have our cake and eat it too? (2023). <https://doi.org/10.48550/arXiv.2306.02604>
20. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: *ICDE*, pp. 38–49 (2013)
21. Li, P., Hua, Y., Jia, J., Zuo, P.: FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.* **15**(2), 321–334 (2021)
22. Lu, B., Ding, J., Lo, E., Minhas, U.F., Wang, T.: APEX: a high-performance learned index on persistent memory. *Proc. VLDB Endow.* **15**(3), 597–610 (2021)
23. Lu, B., Hao, X., Wang, T., Lo, E.: Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* **13**(8), 1147–1161 (2020)
24. Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: WiscKey: Separating keys from values in SSD-conscious storage. *ACM Trans. Storage* **13**(1), 1–28 (2017)
25. Luo, Y., Jin, P., Zhang, Q., Cheng, B.: TLBtree: a read/write-optimized tree index for non-volatile memory. In: *ICDE*, pp. 1889–1894 (2021)
26. Ma, C., Yu, X., Li, Y., Meng, X., Maolinyazi, A.: FILM: a fully learned index for larger-than-memory databases. *Proc. VLDB Endow.* **16**(3), 561–573 (2022)
27. Ma, L., Aken, D.V., Hefny, A., Mezerhane, G., Pavlo, A., Gordon, G.J.: Query-based workload forecasting for self-driving database management systems. In: *SIGMOD*, pp. 631–645 (2018)
28. Maltry, M., Dittrich, J.: A critical analysis of recursive model indexes. *Proc. VLDB Endow.* **15**(5), 1079–1091 (2022)
29. Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., Kraska, T.: Benchmarking learned indexes. *Proc. VLDB Endow.* **14**(1), 1–13 (2021)
30. Pavlo, A., Butrovich, M., Ma, L., Menon, P., Lim, W.S., Aken, D.V., Zhang, W.: Make your database system dream of electric sheep: towards self-driving operation. *Proc. VLDB Endow.* **14**(12), 3211–3221 (2021)
31. Perera, R.M., Oetomo, B., Rubinstein, B.I.P., Borovica-Gajic, R.: DBA bandits: self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In: *ICDE*, pp. 600–611 (2021)
32. Schnaitter, K., Polyzotis, N.: Semi-automatic index tuning: keeping dbas in the loop. *Proc. VLDB Endow.* **5**(5), 478–489 (2012)
33. Siddiqui, T., Wu, W., Narasayya, V.R., Chaudhuri, S.: DISTILL: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning. *Proc. VLDB Endow.* **15**(10), 2019–2031 (2022)
34. Tang, C., Wang, Y., Dong, Z., Hu, G., Wang, Z., Wang, M., Chen, H.: XIndex: a scalable learned index for multicore data storage. In: *PPoPP*, pp. 308–320 (2020)
35. Wongkham, C., Lu, B., Liu, C., Zhong, Z., Lo, E., Wang, T.: Are updatable learned indexes ready? *Proc. VLDB Endow.* **15**(11), 3004–3017 (2022)
36. Wu, J., Zhang, Y., Chen, S., Chen, Y., Wang, J., Xing, C.: Updatable learned index with precise positions. *Proc. VLDB Endow.* **14**(8), 1276–1288 (2021)

37. Wu, S., Cui, Y., Yu, J., Sun, X., Kuo, T.W., Xue, C.J.: NFL: robust learned index via distribution transformation. *Proc. VLDB Endow.* **15**(10), 2188–2200 (2022)
38. Wu, W., Wang, C., Siddiqui, T., Wang, J., Narasayya, V.R., Chaudhuri, S., Bernstein, P.A.: Budget-aware index tuning with reinforcement learning. In: *SIGMOD*, pp. 1528–1541 (2022)
39. Wu, X., Ni, F., Jiang, S.: Wormhole: a fast ordered index for in-memory data management. In: *EuroSys*, pp. 1–16 (2019)
40. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: An empirical guide to the behavior and use of scalable persistent memory. In: *FAST*, pp. 169–182 (2020)
41. Yao, A.C.C.: On random 2–3 trees. *Acta Inform.* **9**(2), 159–170 (1978)
42. Zeitak, A., Morrison, A.: Cuckoo Trie: exploiting memory-level parallelism for efficient DRAM indexing. In: *SOSP*, pp. 147–162 (2021)
43. Zhang, J., Gao, Y.: CARMI: a cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.* **15**(1), 2679–2691 (2022)
44. Zhang, Z., Chu, Z., Jin, P., Luo, Y., Xie, X., Wan, S., Luo, Y., Wu, X., Zou, P., Zheng, C., Wu, G., Rudoff, A.: PLIN: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proc. VLDB Endow.* **16**(2), 243–255 (2022)
45. Zhang, Z., Jin, P., Wang, X., Lv, Y., Wan, S., Xie, X.: COLIN: a cache-conscious dynamic learned index with high read/write performance. *J. Comput. Sci. Technol.* **36**(4), 721–740 (2021)
46. Zhou, X., Liu, L., Li, W., Jin, L., Li, S., Wang, T., Feng, J.: AutoIndex: an incremental index management system for dynamic workloads. In: *ICDE*, pp. 2196–2208 (2022)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.