LOFT: A Lock-free and Adaptive Learned Index with High Scalability for Dynamic Workloads

Yuxuan Mo, Yu Hua*
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology

Abstract

In order to mitigate memory overheads and reduce data movements in the critical path, a computational and space-efficient learned index is promising to deliver high performance and complement traditional index structures, which unfortunately works well only in static workloads. In dynamic workloads, the learned indexes incur performance degradation with poor scalability due to inefficient data placement for newly inserted items and intensive lock contention. Furthermore, the model retraining is time-consuming and blocking, which hampers the performance of index operations. In order to meet the needs of dynamic workloads and efficient retraining, we propose LOFT, a highly scalable and adaptive learned index with lock-free design and self-tuning retraining technique to provide high throughput and low latency. LOFT enables all index operations to be concurrently executed in a lock-free manner by using Compare-and-Swap (CAS) primitive and an expanded learned bucket to handle the overflowed data. To minimize the impact of model retraining, LOFT leverages a shadow node to serve the clients' requests and accelerates the retraining process with the aid of index operations. To accommodate dynamic workloads, LOFT determines when and how to perform retraining based on inferred access patterns. Our extensive evaluation on YCSB and real-world workloads demonstrates that LOFT effectively improves the performance by up to 14× than state-of-the-art designs.

CCS Concepts: • **Information systems** → **Data access methods**.

*Corresponding Author: Yu Hua (csyhua@hust.edu.cn).

Keywords: learned index, dynamic workload, concurrency control

ACM Reference Format:
Yuxuan Mo, Yu Hua. 2025. LOFT: A Lock-free and Adaptive Learned Index with High Scalability for Dynamic Workloads. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717458

1 Introduction

Memory systems bridge the performance gap between high-speed CPUs and low-speed storage systems, significantly contributing to overall system performance. To enhance memory performance via efficient data management, an index plays an important role, which provides fast queries for items stored in the memory. Typical structures, e.g., B^+-trees [11, 20, 28] and Hash-maps [9, 59] are widely used in the in-memory databases and key-value stores, such as Memcached [2] and Redis [4]. However, there exists a dilemma between the rapid growth of the stored data and slowdown of DRAM scaling technology [23]. The space-inefficient index structures further worsen this dilemma, e.g., the expensive DRAM capacity occupied by the tree-like index structures is almost the same as that of the stored data [55]. Moreover, multiple I/Os are required to traverse the tree-like indexes via the limited-bandwidth I/O bus.

To address the aforementioned limitations, a learned index structure is proposed [27] to complement the traditional tree-like index structures via model computation, like "computational memory". The learned indexes replace the inner nodes in B^+-trees with learned models, significantly reducing memory overheads by only storing model parameters. Furthermore, multiple time-consuming pointer-chasing operations in tree-like indexes are replaced with model calculation, which remarkably mitigates data movements via I/O bus and accelerates the search process.

When inserting new items, the learned models become inefficient due to the changes in data distribution, which requires model retraining for performance improvements. We define such workloads that contain insert operations as *dynamic workloads*. In fact, most of the realistic key-value workloads are dynamic [7, 52, 57], some of which are write-heavy [16]. Therefore, we need to carefully design the insertion and retraining mechanisms for learned indexes to deliver high performance in real-world workloads. We evaluate the

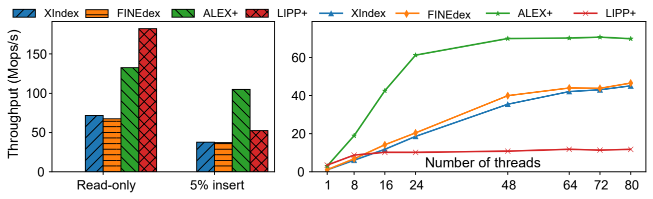

Figure 1. The performance of learned indexes. Different access patterns with 24 threads are shown in the left and different numbers of threads with 30% insert operations and 70% read operations are shown in the right. All workloads come from YCSB [12].

performance of advanced learned indexes [32, 46, 48, 49] in dynamic workloads, as illustrated in Figure 1. Unfortunately, with a few insertions, e.g., 5%, the throughputs of learned indexes decrease by about 50% on average. We analyze the root cause in §2.2. Moreover, we observe that existing learned designs fail to simultaneously deliver high throughput and achieve high scalability in the dynamic workloads. For instance, ALEX+ [48] obtains the highest throughput but only scales to 24 threads in the read-intensive workload due to the coarse-grained locks. Such suboptimal performance of learned indexes in the real-world scenarios undermines the practicality.

It is non-trivial to design efficient and effective learned indexes due to three challenges. First, **Interference introduced by Insertions.** The interference caused by insertions can be divided into two aspects. Existing insertion mechanisms weaken the effectiveness of the learned models for reads. On the other hand, existing high-performance learned indexes require coarse-grained locks for insertions [48], which bring interference to reads, i.e., contention for locks, especially in the write-intensive and highly concurrent workloads. Therefore, we need to eliminate the interference brought by insertions. Second, **Collision between Indexing and Retraining.** Prior retraining schemes inevitably harm the index performance even if retraining does not block index operations. Hence, we need to remove the retraining process from the critical path, while minimizing the performance impact of retraining on index operations. Third, **Mismatch between Fixed Parameters and Various Access Patterns.** Real-world workloads exhibit diverse access patterns, while existing retraining mechanisms just use predefined static parameters for retraining. To adapt to various workloads with different access patterns, the retraining mechanism needs to capture access patterns with minimal overheads and adjust the retraining parameters.

To address the above challenges, we present LOFT, a lock-free and adaptive learned index with scalable performance for multi-core memory systems. To the best of our knowledge, LOFT is the first lock-free learned design. To eliminate the interference introduced by insertions, LOFT enables all index operations to be concurrently executed in a lock-free manner by using the Compare-and-Swap (CAS) primitive. We design an error-bounded insertion mechanism that places the new item into the first free slot within the predicted range and does not sort existing records, i.e., without shifting. When there are no empty slots in the predicted range, LOFT uses an *expanded learned bucket* to handle the overflowed data. While the disorder of records may slightly decrease the read performance, our lock-free design fosters a significant degree of concurrency. To address the collision challenge, LOFT serves the clients' requests via the shadow node. Moreover, we allow the index operations to accelerate the retraining process. To tackle the mismatch challenge, we collect essential information at runtime to be aware of the current workload, enabling informed decisions on when and how to execute retraining.

In summary, this paper makes the following contributions.

- We present a lock-free learned index, called LOFT, which is adaptive to various thread numbers and diverse dynamic workloads with high throughputs.
- We propose an error-bounded insertion scheme with expanded learned buckets to enable lock-free index operations (§3.2).
- LOFT utilizes lock-free retraining scheme for index operations with shadow nodes to mitigate the performance impact on index operations (§3.3). During the runtime, LOFT maintains essential statistics at low costs to enable self-tuning retraining (§3.4).
- We evaluate LOFT using both YCSB and real-world workloads. Experimental results show that LOFT significantly improves the throughput with high scalability compared with state-of-the-art schemes (§5).

We have released the open-source codes of LOFT for public use in GitHub[1].

2 Background and Motivation

2.1 The Learned Index

Learned indexes [27] have been recently proposed to replace or complement the traditional index structures, e.g., B^+-trees [24, 28], hash tables [9, 45], and Bloom filters [8, 15] via the aid of machine learning models. The learned indexes exhibit significant advantages in terms of high query performance and light storage overhead.

When key-value pairs are sorted, the *Cumulative Distribution Function* (CDF) of keys maps the items from their keys to positions. Assuming that the CDF of N keys is represented as F, employing the Equation $F(K_1) * N$ can obtain the position of K_1 (K_1 is one of N keys). In contrast to the $O(\log(N))$ time complexity of B^+-tree, the time complexity of using CDF to index data is $O(1)$. The CDF and dataset size are necessary for indexing, which consumes negligible storage overhead compared with inner nodes of B^+-trees.

[1]https://github.com/yuxuanMo/LOFT.git

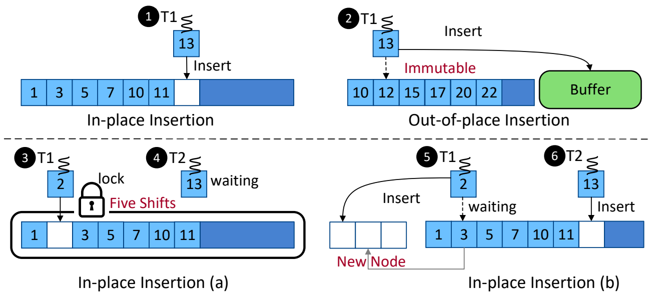

Figure 2. Two schemes for inserting new data to the data node in existing learned indexes: in-place and out-of-place insertions.

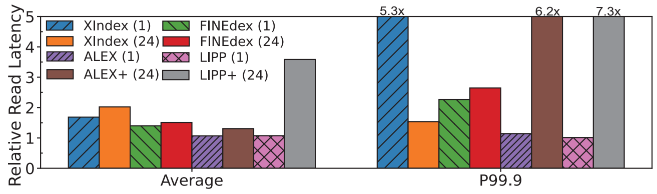

Figure 3. The relative read latencies of learned indexes with 1 thread and 24 threads ($L_{insert_{5\%}}/L_{read\_only}$, and L is the read latency). The workloads come from YCSB [12].

Hence, the key insight of the learned index is to approximate the CDF using learned models. Since a single model is hard to precisely fit the complex CDF, a staged model architecture called *Recursive Model Index* (RMI) [27] is proposed to improve the prediction accuracy. Specifically, RMI comprises multiple independent models at each stage. The model at the upper stage determines the one to be used in the next stage for a given key. The final-stage model returns the predicted position with a prediction error. Existing learned indexes [27, 32, 46] usually leverage a two-stage RMI model, which typically provides sufficient accuracy. In a learned index, the search process for a key starts from the RMI model and then reaches a data node [32, 46]. The learned index leverages the linear models in the data node to obtain the predicted position. Furthermore, it locally searches within the prediction error range ($pred\_err$) using the binary search algorithm. The time complexity for the lookup operation is $O(\log_2(pred\_err))$. The read performance of the learned index depends on the size of $pred\_err$, which enables learned indexes to achieve substantial performance enhancements compared with B$^+$-trees, especially for large datasets.

2.2 Challenges in Dynamic Workloads

While demonstrating high query performance in the read-only workloads, learned indexes show suboptimal performance in the dynamic workloads, as demonstrated in Figure 1. Due to generally achieving suboptimal performance, the learned indexes have not been widely used in real-world scenarios that often exhibit dynamic features. There are three main challenges in achieving the optimal performance for dynamic workloads.

Interference introduced by Insertions. In order to support dynamic workloads, prior schemes leverage different insertion mechanisms. XIndex [46] and FINEdex [32] use extra buffers to temporarily accommodate insertions (out-of-place insertion in Figure 2), and the records in the data array are immutable. However, in this case, the read has to execute extra searches in the B$^+$-tree-based buffer, which is inefficient due to long-latency and frequent probing operations in the buffer. Hence, learned indexes using the out-of-place insertion mechanism achieve suboptimal read performance. As shown in Figure 3, when the insertion ratio slightly increases from 0% to 5%, the average read latencies of XIndex and FINEdex increase by up to 50% (one thread without multi-thread interference).

Unlike them, ALEX [17], LIPP [49] and SALI [19] preserve some free slots in the data array for future model-based insertions. The new items can be indexed according to the existing learned models (in-place insertion in Figure 2). Hence, ALEX and LIPP maintain comparable average read latency with one thread when the insertion ratio slightly increases, as shown in Figure 3. Since SALI leverages the same data structure as LIPP, we do not show the results of SALI. However, if a collision (i.e., the predicted position is occupied) occurs, ALEX has to shift the records to make room for the new item, and LIPP and SALI need to create and chain a new data node to solve the collision.

Since neither ALEX nor LIPP supports concurrency, their concurrency versions, ALEX+ and LIPP+, are respectively proposed [48]. ALEX+ leverages node-level locks for concurrency (in-place insertion (a) in Figure 2). LIPP+ waits for the node creation and chaining to be finished (in-place insertion (b) in Figure 2). Figure 1 shows that ALEX+'s well-designed lock mechanism enables high throughput but hampers scalability. The coarse-grained locks in ALEX+ are highly contended when the thread count increases. As a result, when threads scale to 24, we observe that the average read latencies of ALEX+ also increase by 30% (Figure 3). LIPP+ utilizes fine-grained locks but suffers from more severe performance degradation. Because each node needs to maintain per-node statistics, thus leading to high contentions, and each insertion requires locking the predicted position in a node, which may store the pointer to the chaining node. Therefore, ALEX+ and LIPP+ are scalable to no more than 24 threads, as shown in Figure 1.

Collision between Indexing and Retraining. In order to guarantee prediction accuracy, learned indexes have to carry out high-overhead retraining operations. Even worse, the newly inserted data are more likely to change the existing distribution, thus causing the re-learning operation. XIndex [46] leverages background threads to periodically check each node and perform non-blocking retraining, while other indexes perform retraining in a blocking manner, once the set conditions are met. For example, ALEX+ triggers retraining when the fill ratio of the data node exceeds the threshold. The blocking retraining process decreases the performance, because the index is unable to access items in the retrained data node until the new models are ready. To reduce the overhead of retraining, FINEdex [32] proposes a fine-grained retraining technique, which only retrains the records in the level-bin (a small B^+-tree) to obtain a new small data node and blocks all reads and insertions to the retrained level-bin. ALEX+ incurs 6.2× tail latency and FINEdex has 2.3× with 24 threads due to the blocking retraining, while XIndex only shows 1.5× tail latency by using non-blocking retraining (Figure 3). To reduce the tail latency, we need to remove the retraining process from the critical path.

Non-blocking retraining in XIndex however becomes a hurdle to gain high performance, when the retrained tasks are not processed promptly. XIndex retrains data nodes with buffer sizes larger than 256 records, which is easily achievable even in read-intensive workloads. XIndex needs to retrain almost all data nodes using a single worker thread as shown in Figure 3. However, the computing resources of background threads are limited. A single background thread is unable to complete such heavy retraining tasks in a short time. Hence, the large buffer size leads to long read latency. As a result, the relative P99.9 read latency of XIndex is 5.3×, when the thread number is 1, as shown in Figure 3.

Mismatch between Fixed Parameters and Various Access Patterns. All the parameters for retraining are preset and fixed. For example, the training prediction error is a tradeoff between prediction accuracy and the model numbers based on the evaluation results. ALEX+ determines whether to retrain by comparing the fill ratio of the data node with a fixed threshold. However, different workloads have different request distributions. We need to customize the parameters for each data node under different workloads with reasonable overheads and thereby achieve higher performance. For instance, the hot data node with frequent read operations can obtain higher prediction accuracy for higher read throughput, while the cold data node with rare data accesses can preserve fewer free slots for memory saving. However, it is inefficient to pause the clients' requests and then manually modify the parameters. In order to be easy-to-use and adaptive, the learned index needs to be self-tuning depending on the access patterns. Unfortunately, existing schemes fail to achieve these design goals.

3 The LOFT Design

3.1 Overview

We propose LOFT, an adaptive and lock-free learned index designed for high scalability in dynamic workloads. Figure 4 shows the overall architecture of LOFT, which contains two layers: one root node and multiple data nodes. The root node

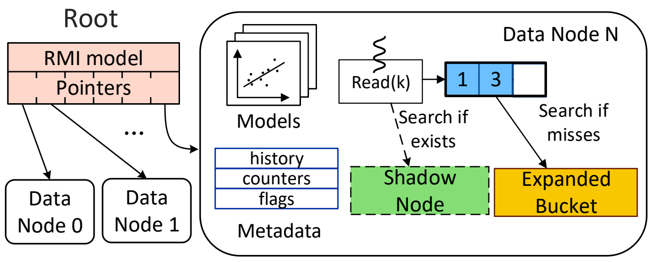

Figure 4. The overall architecture of LOFT.

consists of a two-stage RMI model and a collection of pointers to the data nodes. Each data node handles distinct key ranges without overlaps. For each client request, LOFT utilizes the RMI in the root node to locate the appropriate data node, followed by performing the index operation within the data node using the corresponding linear models. Since all index operations follow a uniform process at the root node level, our primary focus lies on the structures and techniques related to the data nodes. Specifically, to mitigate the interference brought by insertions, LOFT employs an error-bounded insertion mechanism that places new items into their predicted positions and uses expanded learned buckets to manage the overflowed items so that all index operations can be executed in a lock-free manner. We present the structure of the expanded learned bucket and demonstrate how LOFT carries out index operations concurrently and correctly without locks in §3.2. To alleviate the collision between the indexing and retraining, LOFT introduces a shadow data node to serve the clients' requests, while allowing clients to contribute to the retraining process. §3.3 outlines the retraining workflow and describes how index operations proceed during retraining. Moreover, LOFT maintains essential statistics at low costs, making it workload-aware and enabling adaptive retraining. §3.4 presents how to handle retraining tasks based on an informed decision-making strategy. Finally, we demonstrate the concurrency correctness of index operations in §3.5.

3.2 Lock-free Index Operations

LOFT supports common operations in traditional index structures, including `read`, `insert`, `update`, `delete` and `scan`. We omit the repeated details of using the RMI model in the root node to reach the data node. We present the procedures of these operations upon data nodes without performing structure modification operations (SMOs) in this subsection and with performing SMOs in §3.3.

Index operations are closely related to data node initialization since this process determines the record placement. We hence start with node initialization. Piecewise Linear Approximation (PLA) algorithm [18] is employed to obtain the linear models within the data nodes. Consider a linear model for N keys, where a represents the slope, K_1 is the smallest key, and x denotes the given lookup key. This model

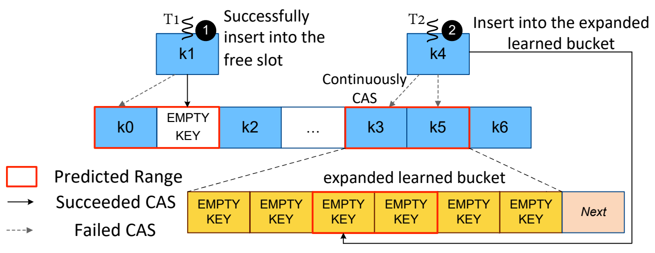

Figure 5. The process of the error-bounded insertion. Each insertion leverages CAS to swap the *EMPTY_KEY* with the inserted key atomically.

predicts a position for each key with an associated prediction error, denoted by *pred_err*. We use y to represent the actual position of a key. The relationship between the predicted position and the actual position satisfies the following condition:

$$|a * (x - K_1) - y| \leq pred\_err \quad (y \in [0, N-1]) \tag{1}$$

To facilitate in-place insertions, we multiply the linear model with the expansion factor ϵ to make room for future insertions. The ϵ is initially established at 1.5 in LOFT, which strikes a balance between memory efficiency and insertion performance. The expanded data node exhibits an average fill ratio of around 0.67 (= 1/1.5), similar to the average fill ratio of leaf nodes in B$^+$-trees [21] and ALEX [17], while maintaining moderate free slots for future insertions. Hence, the predicted position range of N keys is:

$$0 \leq \epsilon * a * (x - K_1) \leq \epsilon * (N + pred\_err) \tag{2}$$

Consequently, we set the size of the data array according to Eq.2. Leveraging the expanded model, we compute the positions of the keys and subsequently store the key-value pairs in their predicted positions. If the expected position is occupied, we look backward for an empty slot within the predefined range of *pre_ran* slots. In this way, most keys are placed in the exact predicted positions and all keys meet the condition:

$$|\epsilon * a * (x - K_1) - y| \leq pre\_ran \tag{3}$$

We define the *pre_ran* as the predicted range, distinct from *pred_err*. The *pre_ran* determines how LOFT places the records according to the expanded model. LOFT sets the key of the free slot in a data array to *EMPTY_KEY* to indicate that the slot is available. For instance, the NAN can be used as *EMPTY_KEY*, when the key type is double. After the initialization, LOFT is ready to handle clients' requests.

Lock-free Insertion. To alleviate the interference introduced by insertions, LOFT employs the in-place insertion mechanism but does not shift data. Our proposed scheme, called **error-bounded insertion**, inserts data into the first free slot within the predicted range. If the inserted key already exists, i.e., duplicated insertion, LOFT will perform an update operation with the new value.

We illustrate the process of our error-bounded insertion mechanism in Figure 5. We simplify the data array by omitting values in Figure 5. The values are usually 8-byte value-pointers to support variable-length [32, 46] and stored together with the keys in the learned indexes. For an insertion, LOFT uses the *Compare-and-Swap* (CAS) primitive [10, 31] to continuously compare keys in the data array with *EMPTY_KEY*, starting from the predicted position. CAS is an atomic operation that compares the contents from a memory location with a given value [54]. If they are equal, the given value is atomically written into the memory location. Otherwise, CAS returns the contents of the memory location. If the CAS succeeds, i.e., the inserted key successfully occupies the free slot, LOFT writes the value and completes the insertion (❶ in Figure 5). Once the CAS fails, LOFT needs to verify whether the value returned from the CAS matches the inserted key, and performs an update operation if it matches. When none of the keys within the predicted range are equal to *EMPTY_KEY* or the inserted key, LOFT inserts the data into an *expanded learned bucket* (❷).

Expanded Learned Bucket Structure. Suppose there are m keys from x_1 to x_2 and the following Equations are satisfied:

$$[x_1, x_2] \xrightarrow{\epsilon * a * (x - K_1)} [y_1, y_2] \tag{4}$$

$$y_2 - y_1 + pre\_ran < m \quad \&\& \quad \beta * (y_2 - y_1 + pre\_ran) > m \tag{5}$$

This signifies that the current model cannot accommodate m keys. However, we can expand the linear model with an expanded factor β to manage the overflowed items (Eq.5). We thus propose an expanded learned bucket structure to handle these overflow insertions. The *pre_ran* positions share an expanded learned bucket. The model in the bucket is $\beta * (\epsilon * a * (x - K_1) - L_i)$, where L_i is the start position in the data array of this bucket, and β, the expanded factor of buckets, is set to 8 in LOFT. In Figure 5, β is set to 2. The bucket contains the next pointer to the next level to handle possible overflow. If the current expanded bucket fails to accommodate new insertions, the next level bucket will double the β to make more room for future insertions. The bucket is like a small data array. Therefore, the index operations in the bucket follow the same workflow as the index operations in the data array, which are also lock-free.

When an insertion fails to CAS an empty slot in the data array, it shifts to using the expanded bucket. If the bucket does not exist, the insertion first allocates a new bucket and then uses CAS to replace the *nullptr* pointer to the expanded bucket with the allocated one. If the CAS operation fails, the allocated bucket is reclaimed, and the insertion proceeds with the existing bucket assigned by another thread. By utilizing the expanded buckets, LOFT enables all insertions to occur simultaneously without blocking each other. Our insertion scheme leverages CAS to exclusively occupy the free slot within the predicted range. Unlike existing learned indexes,

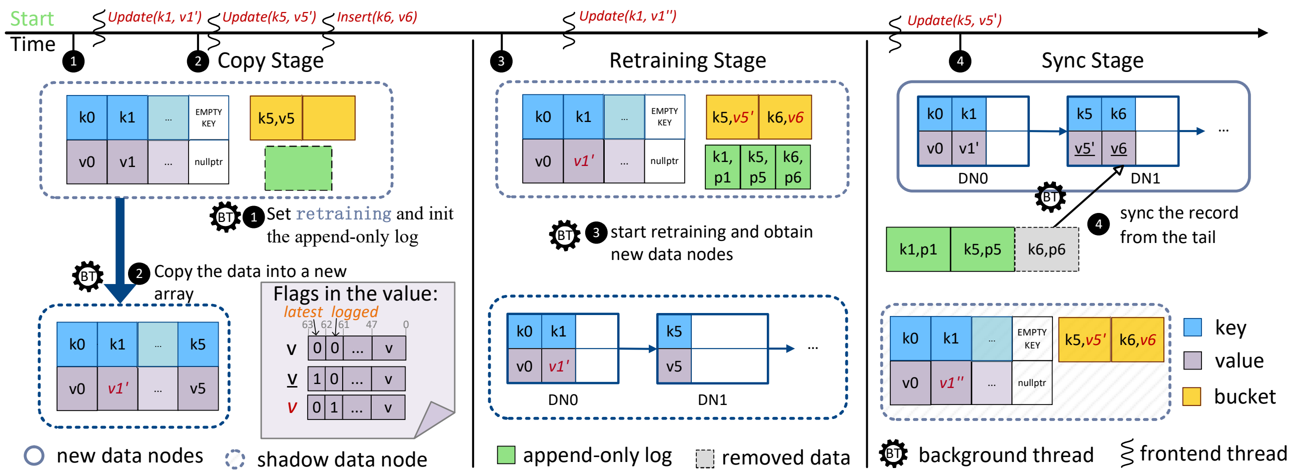

Figure 6. The process of data node retraining.

LOFT does not require the records to be sorted, but ensures that all items exist within their predicted ranges.

Lock-free Read. A read operation returns the value of a given key. If the given key does not exist, LOFT returns a *failure*. LOFT uses the linear model in the data node to obtain the predicted position. Since the records within the predicted range are unordered, we directly leverage the linear search algorithm. The search latency remains acceptable, as the predicted range is small, typically around 32 records. If LOFT fails to find the queried key within the predicted range, we then search the expanded learned bucket, if this bucket exists. The number of bucket layers usually does not exceed one, since LOFT can merge the data in the expanded learned bucket with the one in the data array and perform retraining in time. Thus, the read operation delivers reasonable tail latency.

Lock-free Update. An update operation modifies the value of the given key to a specified value. For an `update`, LOFT starts by searching the given key like the `read`. If the key exists, LOFT performs the in-place update. Moreover, LOFT can atomically update the value because the value is 8-byte. Otherwise, LOFT returns a *failure*.

Lock-free Deletion. LOFT leverages *soft deletion* to delete key-value pairs. Specifically, LOFT first locates the given key as the `read` does. LOFT then atomically updates the value in the data node to `nullptr` to indicate that the key-value pair is deleted, rather than removing the key-value pair from the data array. In this way, LOFT ensures the insertion correctness and avoids duplicated keys. Note that the deleted items will be discarded during the retraining process.

Lock-free Scan. A `scan`(k_i, n) retrieves the smallest n records whose keys are $\geqslant k_i$ from the dataset. In LOFT, although records are partially sorted, the scan operation remains straightforward. It only requires searching an additional *pre_ran* slots, as records are organized according to a linear model, allowing all items exist within their predicted ranges. Therefore, LOFT first identifies the predicted position of k_i and filters out n candidate records with keys that are $\geqslant k_i$, starting from that position. Then, it continues searching the following *pre_ran* slots to obtain the n candidates. Afterward, LOFT retrieves records from the according buckets, replacing candidates with smaller ones that still satisfy the $\geqslant k_i$ requirement. Finally, LOFT returns the n requested records.

3.3 Lock-free Retraining Process

When inserting a batch of records, LOFT becomes inefficient due to the increasing collision rate. Specifically, fewer free slots are available for insertions in the data node, and hence the insert operations need to search more slots to find a free one. The inserted item may be added to the expanded learned bucket due to overflows. Meanwhile, the read operation suffers longer latency because the search region becomes larger. Therefore, LOFT uses background threads to periodically execute retraining for performance improvement. We describe the conditions that trigger retraining and how to dynamically adjust the parameters in §3.4. In this subsection, we present how background threads perform retraining and how frontend threads execute index operations. The entire retraining process does not block index operations.

Background Threads. The background threads utilize Read-Copy-Update (RCU) [41] for efficient structure modification operations and safe garbage collection, ensuring that frontend threads executing index operations remain non-blocking. RCU is a classical synchronization technique for managing concurrent access to data structures. It enables multiple threads to safely read the shared structure by marking read-side critical sections, during which threads access the structure without blocking. Modifications are performed by creating and updating a new version of the structure,

Algorithm 1 Structure Modification Operation

```
1:  Node_Retraining(node)
2:  /*Copy Stage & Retraining Stage*/
3:  $node.log \leftarrow$ allocate a new log
4:  $node.retraining \leftarrow true$
5:  $per\_node\_rcu\_barrier()$
6:  $new\_node \leftarrow$ new data nodes      ▷ head of linked nodes
7:  $new\_node.shadow\_node \leftarrow node$
8:  $new\_node.sync \leftarrow true$
9:  /*Sync Stage*/
10: $atomic\_update\_reference(node, new\_node)$
11: $per\_node\_rcu\_barrier()$
12: while $node.log.not\_empty()$ do              ▷ sync records
13:     $new\_node.sync(node.log.pop\_out())$
14: end while
15: $new\_node.sync \leftarrow false$
16: $per\_node\_rcu\_barrier()$
17: $free(new\_node.shadow\_node)$ ▷ free the shadow node
18:
19: Root_Update(root)
20: $new\_root \leftarrow$ retrained root node
21: $atomic\_update\_reference(root, new\_root)$
22: $whole\_rcu\_barrier()$
23: $free(root)$ ▷ reclaim the memory of retrained root node
```

followed by an atomic pointer update redirecting to the modified version. The old version remains accessible to threads within their read-side critical sections and is safely reclaimed after all threads exit these sections, triggered by an RCU barrier.

Before each round of retraining, LOFT creates a double-ended work queue of the same length for each background thread (BT). The work queue stores pointers to the data nodes that require checks. Each background thread is responsible for checking various data nodes and handles one node retraining task at a time. The retraining process of the data node is divided into three stages, including copy, retraining, and sync stages. Figure 6 and Algorithm 1 illustrate how a background thread retrains models upon a data node to obtain new data nodes.

Copy Stage (CS). At the beginning of retraining, the background thread creates an append-only log and sets the *retraining* flag to true (❶ and Lines 3-4). The log records the modifications during the copy and retraining stages. The append-only log is constructed as a linked list with block-level granularity. BT initially pre-allocates a logging block with continuous space for a specified number of records, e.g., 32 records. When an index operation detects that the block is full, it dynamically allocates a new block. Because the logging size is typically very small during an SMO, the pre-allocated logging block suffices in most cases, making dynamic allocation in the critical path rare. Before copying data from the data array and buckets into a new array (❷), BT needs to wait for ongoing index operations that observed the node's *retraining* flag as false to complete. This is done by using an RCU barrier (Line 5).

Retraining Stage (RS). LOFT sorts the records in the array in ascending order by keys. BT uses the PLA algorithm to train models. After obtaining the linear models, BT initializes each data node according to the new model, linking all new data nodes together (❸ and Line 6). BT sets the *sync* flag to true and assigns the retrained node as the shadow node for the first new data node (Lines 7-8).

Sync Stage (SS). Once new data nodes are obtained, we atomically replace the pointer to the retrained node in the root with a pointer to the head of the new data nodes (Line 10). The new data nodes become visible, while the retrained node turns into a shadow node. Before beginning synchronization, we need to ensure that all modifications on the retrained node are finished and future operations are executed on the new data nodes to prevent data loss. Specifically, BT will wait for an RCU barrier (Line 11). BT then sequentially reads a record from the tail of the append-only log and inserts that record into the new data nodes, one at a time (❹ and Lines 12-14). Before updating a record in the new data nodes, we first identify if that record has been already up-to-date to avoid accidental overwriting. The reason is that during SS, all modifications are performed on the new data nodes. To ensure correctness, we leverage the highest bit of value to mark it as *latest*, since the value uses only 48 of the 64 available bits. If BT finds that the value in the new data node is marked as *latest*, it skips the synchronization. For example, BT will not synchronize the value of *k5*. Otherwise, it marks the synchronized value as *latest*. After BT synchronizes all records from the log, it sets the *sync* flag to false, marking the end of node retraining (Line 15).

After SS, BT completes node retraining and obtains the new data nodes. Note that when the retraining process produces multiple new data nodes, they are linked until the root node is updated. To achieve safe garbage collection, LOFT waits for the unfinished read threads accessing the shadow node to complete, since only read operation may access the shadow node during this stage (Line 16). BT reclaims the memory of the shadow node and the log, once all read operations are completed (Line 17).

Root Update. When all background threads finish their node retraining tasks, they fall asleep. Only the first background thread performs the root update when new data nodes are created or/and old data nodes are removed. This thread traverses all data nodes, including those in the linked list, and records the pivot keys and positions to create a new RMI. Subsequently, LOFT builds a new root with the new RMI and atomically replaces the current root pointer with the new one (Lines 20-21). This active background thread reclaims the old root node after all frontend threads exit the

Table 1. The index operations during retraining

Oper	CS & RS	SS
R	-	-new data node -shadow node -new data node
I/U/D	- -log (first time)	-label value -

old root node (Lines 22-23). So far, LOFT has finished a round of checking via the background threads.

Frontend Threads. Index operations differ from those described in §3.2 when they find that the *retraining*/*sync* flag is true. We refer to the `Insert/Update/Delete` operations as updatable since they need to modify the data nodes. Table 1 illustrates how LOFT performs index operations during the retraining process. The symbol '-' indicates that LOFT follows the same steps as described in 3.2. The CS, RS, and SS respectively refer to copy, retraining and sync stages.

CS & RS. The frontend thread follows the steps described in §3.2 to locate the data slot for the index operation. If the operation is `read`, the thread directly obtains the results. For updatable operations, LOFT needs an extra logging step if the *retraining* flag is true. To maintain data consistency between the log and the data nodes, the log stores keys and pointers to the corresponding values in the data nodes. LOFT uses the second most significant bit of the value to ensure each key is logged only once. The thread first checks whether the *logged* label of the record is set to true. If the record has not been logged, the updatable operation will attempt to atomically set the *logged* label and then add the record to the log (e.g., *Update (k1, v1')*). Otherwise, LOFT atomically updates the value without logging (e.g., *Update (k1, v1")*).

SS. When the updatable operations find that the *sync* flag is true, they label the value as *latest* and then follow the same steps outlined in 3.2 (e.g., *Update (k5, v5')*). The read operation during SS is performed in three steps. LOFT searches the new data node in the first step. If the searched key exists and the value is labeled as *latest* in the new node, LOFT returns the corresponding result (e.g., reading the value of k5/k6 during SS in Figure 6). Otherwise, LOFT continues to search the shadow data node to obtain the latest value of the queried key (e.g., reading the value of k1 during the SS). After successfully retrieving the value in the shadow node, LOFT will attempt to update the record in the new node. If observing that the queried data has been marked as *latest* by other index operations in the new data node, LOFT will return the *latest* value. If not, LOFT will synchronize the new node's value with that in the shadow node and mark it as *latest* before returning the result.

During retraining, LOFT executes index operations in a non-blocking manner with the aid of the shadow node. The usage of the shadow node incurs temporary memory overhead, but BT will release this space after finishing retraining.

Table 2. The metadata in each data node

History information:	
initialization parameters	*pred_err, ϵ, pred_ran, init_size*
previous counter values	*R1', R2', R3', W1', W2', W3'*
Read/Write counters: track amounts of different operations	
Read_small/Write_small	smaller than half of *pred_err*
Read_large/Write_large	larger than half of *pred_err*
Read_bucket/Write_bucket	hit the buckets
Flags: indicate the states and the access patterns	
states	*retraining, sync*
access patterns	*write_intensive, cold*

Moreover, the number of shadow nodes depends on that of background threads, and only a few shadow nodes exist simultaneously. By exploiting the highest two bits of the value, LOFT reduces the logging overhead by omitting redundant logging and avoids unnecessary searches upon the shadow node. The *latest* and *logged* labels are only used during retraining and will be cleared when constructing new nodes. Meanwhile, the index operations during SS can assist BT in performing synchronization, which accelerates the retraining process. As a result, the retraining has a slight performance impact on index operations in LOFT.

3.4 Self-tuning Retraining

To adapt to diverse workloads, LOFT stores metadata in the data node for informed decision-making and dynamically adjusts the retraining parameters based on the workload characteristics.

Informed Decision-making. LOFT stores metadata in each data node for performing Structure Modification Operations (SMOs), as listed in Table 2. The read counters track the number of operations that do not affect the key distribution (e.g., `read` and `update`), whereas the write counters record the numbers of `insert` and `delete` that affect the number of records. We categorize the read/write counters into three classes based on search or CAS length. For instance, if a read operation retrieves the value by searching only one slot smaller than half of *pre_ran*, LOFT increases *read_small* counter by one. If the read operation hits the buckets, we add one to *read_bucket*; otherwise, we increase *read_large* by one.

LOFT performs three kinds of SMOs upon nodes: split, expansion, and merge. The node split relearns the data distribution to update the models, while node expansion retains the current model but increases the expansion factor. Node merge combines a data node with its adjacent one to reduce the cost of locating nodes. We preserve the static triggering mechanism similar to other learned indexes [17, 32, 46] as the baseline, when considering two factors: the fill ratio and bucket size. A lower fill ratio represents lower space efficiency, while a higher fill ratio hampers performance since insertions successfully CAS an empty slot with a low probability. Additionally, a large bucket size indicates that newly inserted items exceed the training prediction error. Therefore, the baseline triggers retraining if the fill ratio of the data node reaches a preset threshold (e.g., 0.8/0.4) or if the bucket size exceeds a threshold (e.g., 256).

Unlike the static triggering mechanism, LOFT determines when and how to perform retraining based on specific access patterns. To minimize the overhead of collecting statistics for the data node at runtime, LOFT maintains only a few counters and does not precisely track each operation. In our experience, this information is enough to capture the index performance. According to the counters and history information, we estimate the average performance and identify the access pattern of the node at the runtime. ❶ If the data node is hot, where the read/write counters have significantly increased from the previous check, LOFT estimates the average performance and tail latency from the last BT check until this point. If the estimated performance is lower than the threshold, LOFT performs retraining for performance enhancement. The type of the triggered retraining is based on the effectiveness of the existing model in the retrained node. If there are many empty slots in the data array but index operations still fail to execute within the predicted range, the model of the retrained node is ineffective. Therefore, LOFT triggers a node split to relearn the data distribution. Conversely, LOFT performs node expansion to create more free slots. ❷ When the data node is *write-intensive*, LOFT leverages the increment of write counters to predict the fill ratio in the next round of checking. If LOFT predicts that future insertions will overflow the data array, BT will perform retraining for the node in advance to prevent significant performance degradation. ❸ For a *cold* data node, characterized by infrequent index operations, if it remains *cold* in two consecutive checks and has a low fill ratio (< 0.5), we can compact the data node to decrease memory overhead by using a smaller *expansion factor* (0.6).

Dynamically Adjusted Parameters. For retraining, we need to determine the **training prediction error (*pred_err*)**, **expansion factor (ϵ)**, and **predicted range (*pre_ran*)**. We fix the training prediction error to 32, the same as existing learned schemes [32, 46], to obtain high prediction accuracy and a moderate number of data nodes, since a larger number of data nodes decreases the efficiency of upper RMI models. **The key insight is that LOFT can adjust the expansion factor and predicted range after obtaining models during the node initialization**, as we mentioned in §3.2. After acquiring the linear model in a data node, LOFT repositions the records and resets the prediction accuracy to *pre_ran*. For a read operation, a smaller *pre_ran* can decrease the search region for reads. For an insertion, a larger *pre_ran* increases the probability of finding a free slot and improves the memory utilization ratio of the data array. A larger ϵ improves performance by reducing the probability of collisions but sacrifices the memory efficiency. Based on these observations, we propose the following tuning principles. For the common cases, the *expansion factor* (ϵ) and *predicted range* (*pre_ran*) are set to 1.5 and 32 respectively.

❶ If the retrained node is *write-intensive*, LOFT increases the ϵ by 0.5 until it reaches 3 to avoid using excessive memory space. This allows the new node to preserve more free slots for future insertions, thus reducing the insertion collision rate and the frequency of retraining. When the node is no longer write-intensive, LOFT will reset the ϵ to 1.5. When the retrained node is a cold one, a smaller ϵ (= 0.6) is used to compact the space of this node. ❷ When the node is *write-intensive*, we utilize a larger *pre_ran* (= 64) to improve the write performance. If the retrained node is *read-intensive*, we decrease the *pre_ran* to 24, thus reducing read latency. When the retrained node becomes cold, LOFT sets the *pre_ran* to 72 to increase the memory utilization of the data array. Note that these parameters only adjusted during the retraining process and have an effect on the newly retrained node.

Our proposed self-tuning retraining mechanism is easy to use and efficient to improve index performance. LOFT triggers retraining based on the estimated performance and performs retraining in advance for write-intensive workloads to prevent a significant drop in performance. According to the access pattern of each node, LOFT automatically customizes the parameters to adapt to dynamic workloads.

3.5 Concurrency Correctness

LOFT is a scalable and high-performance index scheme using read-copy-update (RCU) [5, 46] and CAS [10, 54] to implement lock-free concurrency control for multi-core memory systems. There are no concurrency issues among background threads, as each thread handles separate, non-overlapping data nodes. In general, we need to address the write/write and read/write conflicts in multi-thread workloads while ensuring concurrency correctness despite background threads.

Write-Write Conflicts. LOFT ensures serialized execution of updates to the same key since the value is 8-byte and can be read and written atomically. When there are multiple threads inserting records with the same key, only one writer can successfully occupy a free slot by CAS, and the position of the inserted key is determined. All insertions will atomically write the value into the same slot. Thus, the value of a key is modified one by one if the write-write conflicts exist.

Write-Read Conflicts. LOFT needs to return the latest version of the value for reads. For each insertion, LOFT checks the key when the CAS fails. If the key is the same as the inserted one, LOFT performs the update operation. Hence, there are no duplicated keys in LOFT except during the retraining. For 8-byte values, LOFT can atomically read and write the value, thus guaranteeing the correctness of concurrency control.

Interleaving with Retraining. The states of a node are divided into three types: normal, *retraining*, and *sync*, while index operations follow different steps to interact with data nodes in different states. When the node state switches, LOFT needs to guarantee that 1) no updates, insertions, or deletions are lost. When the node state turns to *retraining*, the unfinished index operations will not be logged because they did not observe the node state as *retraining*. Hence, LOFT waits for these unfinished threads to be implemented to ensure that these modifications can be copied into the new nodes during CS and observed by future readers. When the state of the node changes to *sync*, BT will wait for all modifications during CS and RS to finish and be logged before synchronizing these modifications to the new data nodes from the log. In this way, these modifications can be synchronized to the new nodes, allowing readers to obtain the latest values. Furthermore, LOFT ensures that 2) readers can obtain a consistent value despite the existence of concurrent writers when the two versions of a value exist during SS. As described in 3.2, LOFT leverages a flag to help the reader distinguish the latest version of a value during SS. In summary, LOFT guarantees the linearizability [22] of read and write operations by affirming that BTs operate upon the stale data structure until all index operations in the stale state are finished and the latest version of values can be identified regardless of the existence of multiple versions.

4 Implementation Details

We employ a lightweight implementation of RCU, userspace RCU(URCU) [5] and self-implemented per-node RCU to implement lock-free designs and garbage collection in LOFT. Each index operation starts with a call to `rcu_read_lock()` and ends with `rcu_read_unlock()`, keeping URCU informed about this operation. `rcu_read_lock()` function serves as a notification mechanism rather than a traditional lock, which does not block any other threads. When modifying the root node during retraining, we utilize `rcu_xchg_pointer()` to atomically exchange the old structure with the new one, followed by `synchronize_rcu()` to ensure that all threads accessing the index structure have exited. The implementation of `synchronize_rcu()` in URCU contains locks. However, these locks can be removed in LOFT because only the active background thread needs to acquire them. When performing SMOs upon data nodes, we achieve a per-node RCU barrier by using the atomic counter to track the number of threads accessing the data node. When switching the state of the data node, the background thread will wait for the thread counter of the stale state to reach 0 as an RCU barrier. Regarding garbage collection, background threads use an RCU barrier to wait until all threads have exited the old structure. Afterward, they promptly and safely reclaim the memory space occupied by the old structures, including logs, shadow nodes, and the root node. For the search process, we leverage the SIMD, i.e., Intel AVX2 [6], to accelerate the search process since keys are continuously stored.

5 Performance Evaluation

5.1 Experimental Setup

We have implemented LOFT in C++. We run all experiments on a two-socket Linux server (5.4.0-132-generic) with two 26-core Intel(R) Xeon(R) Gold 6230R CPUs @2.10Ghz and 188GB DRAM.

Counterparts for Comparisons. We compare LOFT with eight state-of-the-art designs of the B^+-tree, hash table, adaptive radix tree (ART) and learned index. Masstree [40] is a trie-like concatenation of B^+-tree and is optimized for high concurrency. DyTIS [51] is a dynamic index structure based on extendible hashing. (Note that the open-source code of DyTIS is single-threaded version, so we only examine its read performance.) ART-OLC [30] is a concurrency version of the ART [38]. Both XIndex [46] and FINEdex [32] use extra space to handle new insertions. XIndex leverages a non-blocking retraining scheme, whereas other learned indexes do not. ALEX+ [48], LIPP+ [48], and SALI [19] leverage the in-place insertion mechanism with locks. SALI is the latest learned index that can be adaptive to different workloads.

Benchmarks. (1) We use YCSB benchmarks [12] with various read/insertion ratios following Zipfian, Uniform and Read-latest request distributions. The Zipfian coefficient is 0.99 by default. Besides, we leverage YCSB-C [34, 44], a C++ version of YCSB to generate workloads with a Zipfian factor of 0.5. (2) We employ five typical real-world datasets with 200 million unique 8-byte keys to generate workloads. Ref. [48] introduces a method to quantify the "**hardness**" of datasets for learned indexes, which is widely used in existing schemes [19, 51]. Dataset hardness determines how well a learned index captures data distribution, impacting performance gains. The datasets are listed below in terms of their *hardness*, from the easiest to the hardest.

- *Longitudes*: The longitudes of locations around the world from OpenStreet Maps [3].
- *Covid*: The sampled Tweet IDs with the tag COVID-19 [36].
- *Genome*: The loci pairs in human chromosomes [43].
- *Fb*: The upsampled Facebook user IDs [26].
- *OSM*: The uniformly sampled OpenStreetMap locations [26].

All benchmarks contain 8-byte keys and value pointers with only read and insert operations, which are identical to [19, 54]. For all tests, we use 100 million records to train the learned structure or initialize Masstree, ART-OLC and DyTIS, and then allow each worker thread to constantly execute operations from the workloads.

Configurations. For all compared indexes, we use the default configuration from their original papers respectively. The predefined error bound threshold is 32 for training all learned indexes. In LOFT, we follow the same setup as XIndex [46], assigning a background thread to every twelve

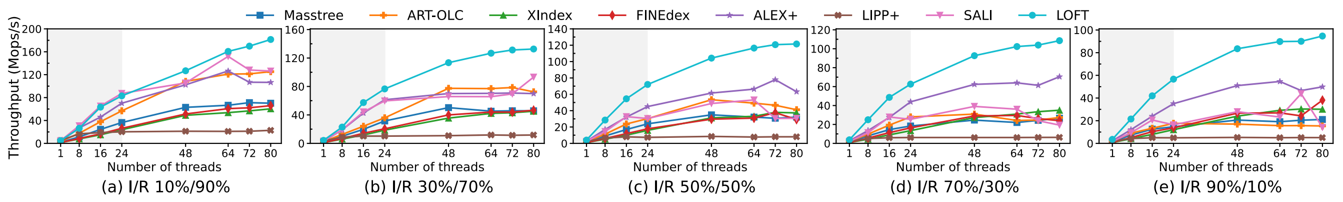

Figure 7. The throughputs on different read/write ratios. (I/R: Insertion/Read ratio, the gray region: within an NUMA node)

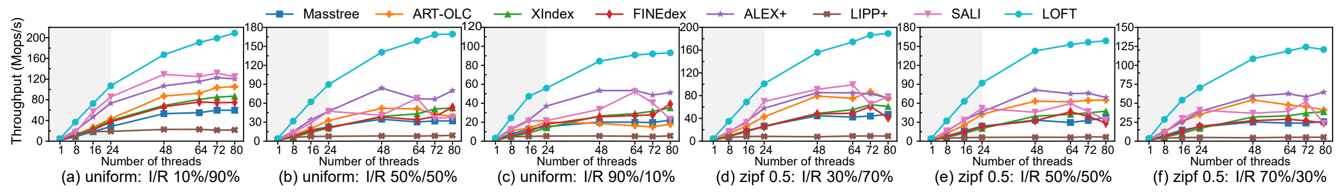

Figure 8. The throughputs on different read/write ratios with different Zipfian factors. (I/R: Insertion/Read ratio, the gray region: within an NUMA node)

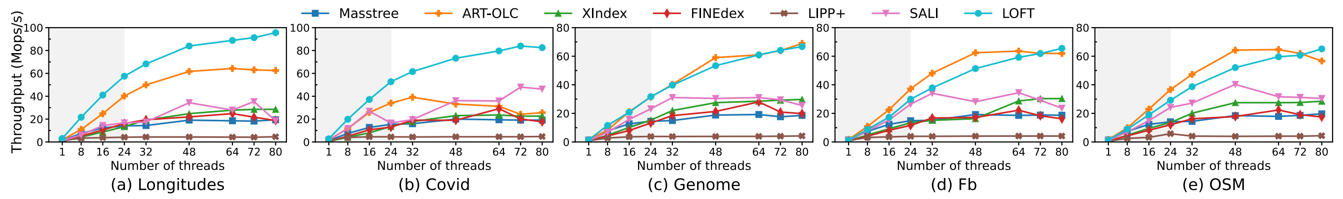

Figure 9. The write performance under different datasets. (The gray region indicates the scenario within an NUMA node)

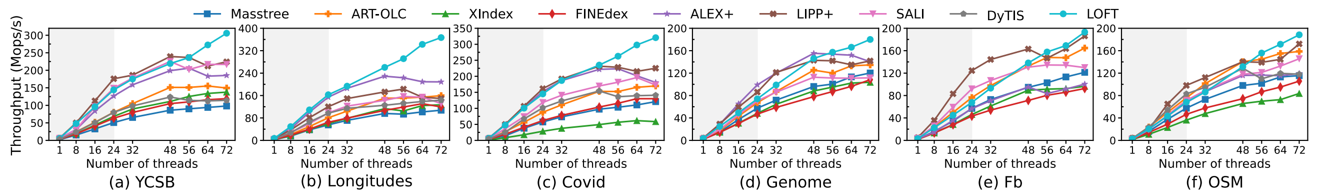

Figure 10. The read performance under different datasets. (The gray region indicates the scenario within an NUMA node)

worker threads, which slightly increases the CPU consumption by about 8%. Compared with the performance gains, the extra CPU overhead is negligible. Our default thread number refers to worker threads without counting the background threads. We run all indexes with 24 worker threads by default, since most indexes scale well within 24 threads [48] and this is a moderate thread number to evaluate the performance within the NUMA node and without using hyperthreads in our multi-core platform. When the thread number exceeds 24, we leave the task that maps threads to cores to OS.

5.2 Evaluation on Scalability

Scalability with Read/Write Workloads. Figure 7 displays the scalability of indexes in workloads with various read/insertion ratios. All workloads are generated from YCSB with a Zipfian factor of 0.99. We calculate the average throughput after running 200 million operations. In general, LOFT achieves the best performance under all workloads. As the insertion ratio increases, the throughputs of all indexes decrease and the scalability is limited by more frequent structure modification operations (SMOs). LOFT can scale to 80 threads in read-intensive workloads and 48 threads in write-intensive workloads. On average, the performance of LOFT is enhanced respectively by a factor of 3.6×, 3.3×, 3.1×, 3.4×, 1.7×, 14×, and 3.8× compared with Masstree, ART-OLC, XIndex, FINEdex, ALEX+, LIPP+ and SALI with 80 threads.

In read-intensive workloads, ALEX+, SALI, and LOFT benefit from the in-place insertion mechanism, thus achieving higher throughputs. Meanwhile, ART-OLC also shows high throughput due to its cache friendliness [29]. With the growth of insertion amount, the conflicts where the predicted position of an insertion has been already occupied become more frequent. This results in the increased node creation and chaining in SALI and LIPP+, which significantly reduces throughput and hinders scalability. Due to the coarse-grained locks, ART-OLC and ALEX+ scale only up to 24 threads. In contrast, LOFT supports lock-free index operations and non-blocking retraining, enabling it to scale to 48 threads with higher throughput. XIndex and FINEdex, limited by buffer efficiency, achieve similar throughputs to Masstree.

We also evaluate LOFT's performance across different Zipfian factors, specifically selecting factors of 0 (uniform)

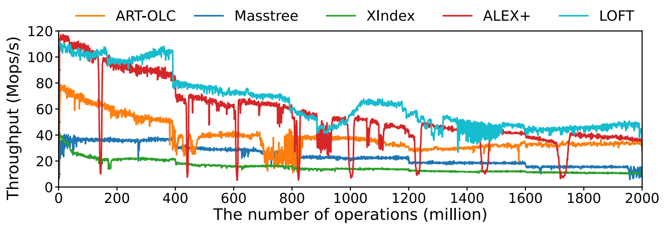

Figure 11. Throughput per million operations under dynamic workloads.

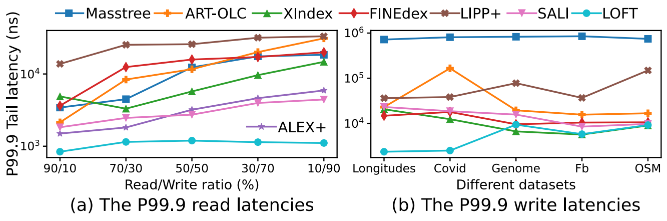

Figure 12. P99.9 latencies on different read/write ratios and datasets.

and 0.5. Figure 8 shows the scalability of all indexes under less-skewed workloads. In these workloads, we observe similar performance trends as those discussed for highly skewed workloads (Zipfian factor 0.99). LOFT consistently outperforms other indexes, demonstrating high throughputs and superior scalability across all workloads with varying read/write ratios and Zipfian factors.

Scalability with Write-only Workloads. We evaluate the write performance using diverse datasets with different *hardnesses*. We allow these indexes to continuously run 100 million insert operations. Note that ALEX+ suffers from deadlocks in real-world workloads, so we do not show the performance of ALEX+ under these workloads. Overall, LOFT outperforms all other learned schemes and scales to 48 threads in the write-only workloads.

As shown in Figure 9, LOFT is sensitive to the data distribution. In the easy datasets, i.e., *Longitudes* and *Covid*, LOFT shows high throughput and scalability because our error-bounded insertion design is efficient and the key distributions are easy to approximate. On the other hand, in the hard datasets, the traditional index structure ART-OLC performs as well as or better than LOFT. However, *genome*, *fb*, and *OSM* are the three hardest datasets from [48], while most real-world workloads are easy for the learned index to approximate the CDF [48]. Moreover, LOFT can still maintain high scalability in the hard workloads due to our lock-free design. Therefore, our scheme demonstrates higher performance than the traditional index structure in most real-world scenarios. Apart from this, we have similar observations with previous evaluations.

Scalability with Read-only Workloads. Figure 10 demonstrates the read scalability with YCSB C workload and real-world datasets. We run 200 million read operations with Zipfian distribution. When the hardness of the dataset increases,

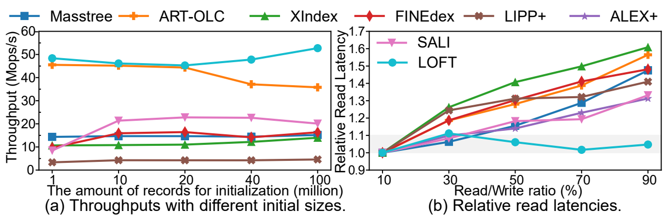

Figure 13. (a) The impact of initial sizes. (b) The impact of read/write ratios.

the read performance of learned schemes decreases. Compared with top-performing indexes, i.e., LIPP+ and ALEX+, LOFT obtains lower throughput because LOFT needs to sequentially search the data instead of using binary search. Nevertheless, LOFT delivers higher scalability, and its read performance can catch up with the advanced designs when the number of threads increases.

5.3 Evaluation on Adaptiveness

To verify the effectiveness of our retraining mechanism and evaluate the performance stability, we run index operations under a dynamic workload that shifts access patterns every 400 million operations. We track throughput changes by measuring the average throughput over every one million operations. The access patterns are 90% read/10% insertion, 70% read/30% insertion, 50% read/50% insertion, 30% read/70% insertion, and 10% read/90% insertion, respectively. Since this process involves a large amount of data, we leverage YCSB [12] to generate and combine these five workloads into a single benchmark that contains 2 billion operations. Due to the DRAM capacity limitation of our used machine, FINEdex, SALI, and LIPP+ are unable to handle such a large volume of data and consequently crash. As a result, we have excluded their performance results from this evaluation.

As shown in Figure 11, the average throughputs of all indexes decline as the proportion of insertions increases. This drop mainly comes from the higher latency of insert operations and the increased frequency of retraining and SMOs. ALEX+ delivers high throughputs but suffers from periodic performance jitters since many data nodes in ALEX+ need to perform blocking retraining almost at the same time. With the growth of insertion ratios, such sharp jitters occur more frequently, since the retraining process is also triggered in a higher frequency. In contrast, LOFT maintains stable and high throughputs even in the write-intensive workloads, thanks to our non-blocking and self-tuning retraining schemes. Compared with the second best-performing scheme, i.e., ALEX+, LOFT improves the average throughput by 16% while ensuring long-term stability.

5.4 Evaluation on Other Factors

P99.9 Latencies. To explore the performance impact of retraining and concurrency control techniques on index operations, we record the tail read latency in the mixed workloads

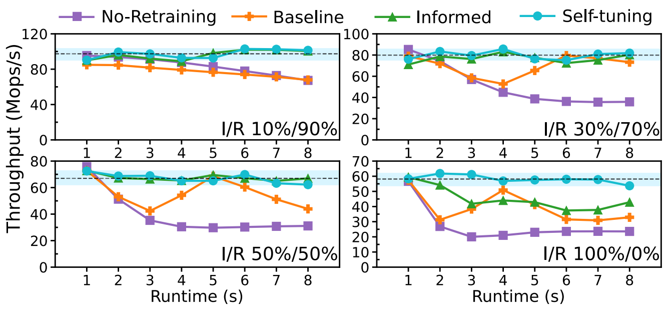

Figure 14. The throughputs with different retraining schemes.

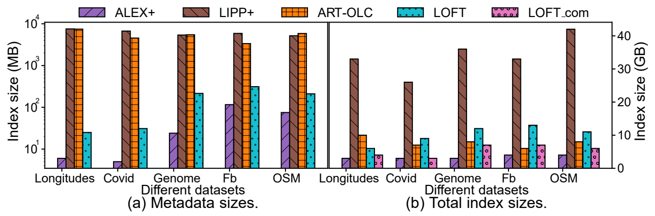

Figure 15. The index sizes under different workloads.

consisting of reads and insertions, and the tail write latency in the write-only workloads with 24 threads. The mixed workloads are derived from YCSB. When the insertion ratio increases, the read operation of indexes using locks has a higher probability of being blocked by locks for retraining and insertions and thus obtains higher tail latency, as shown in Figure 12 (a). Compared with other designs, LOFT reduces tail read latency by up to 90%. Figure 12 (b) shows the tail latency in write-only workloads. LIPP+ and LOFT are sensitive to data hardness and experience long tail latencies in hard datasets due to the increased number of retraining operations. Overall, LOFT achieves the lowest tail latency, i.e., around 1,000ns per read operation and 5,000ns per insert operation. The main reason is that all operations are executed in a non-blocking manner and LOFT performs in-time retraining to improve the performance.

Initial Sizes. To evaluate the impact of the initial size on performance, we vary the amounts of records to initialize indexes and continuously insert the records from *Covid* dataset to reach a size of 200 million. We observe that the throughputs of LOFT and ART-OLC are not deterministically related to the initial size, whereas the throughputs of the other indexes increase with larger initial sizes, since the SMO frequency decreases, as shown in Figure 13 (a).

Insertion Interference on Read Latency. Figure 13 (b) presents the relative P50 read latency, normalized by dividing it by the P50 latency observed in the workload with a 10% insertion ratio. All workloads come from YCSB. To efficiently investigate the performance impact of insertions on reads, we set the access pattern of reads as read-latest, and hence the newly inserted items have higher probabilities of being read. When the insertion ratio increases, all counterparts experience longer read latency caused by insertion and SMO blocking, while LOFT maintains stable read latency. This demonstrates our error-bounded insertion mechanism effectively eliminates the interferences introduced by insertion, and our retraining scheme has a negligible performance impact on read operations.

5.5 In-depth Analysis on LOFT

The Effectiveness of Self-tuning Retraining. LOFT's self-tuning retraining scheme incorporates informed decision-making and dynamically adjusted parameters. Figure 14 illustrates LOFT's throughputs under four retraining schemes: no retraining, static retraining (Baseline), informed decision-making (Informed) and self-tuning (Self-tuning). For all four schemes, the expansion factor is initially set to 1.5, and the predicted range is set to 32. Only Self-tuning scheme adjusts these parameters during runtime. All workloads are generated by YCSB with the read-latest access pattern. Informed decision-making triggers retraining more effectively than Baseline, leading to improved and stable performance. In read-intensive workloads, the prediction range decreases from 32 to 24, with a slight increase in performance. In the write-intensive workloads, the dynamically adjusted parameters significantly boost the performance by evolving the expansion factor to 2.5 and the prediction range to 64, thereby effectively reducing the retraining frequency. The performance jitter does not exceed 10% of the average throughput across all workloads with the self-tuning scheme.

Memory Overheads. Figure 15 illustrates the metadata sizes of indexes and the total index sizes, including the data and leaf nodes, after running the write-only workloads. Here, *LOFT_com* represents the LOFT with compaction enabled. In particular, the learned index size, excluding the data nodes, is much smaller than that of ART-OLC, except for LIPP+, due to many node pointers caused by write collisions. As the datasets become more complex, the models become more complicated, thus leading to larger metadata sizes and more data nodes. Since our self-tuning retraining increases the expansion factor (e.g., to 2) to create more free slots in write-only workloads, the overall index size of LOFT is larger than that of ALEX+, which consistently maintains a fill ratio of 0.7 for the data array. However, when the workloads become idle, LOFT performs compaction on cold data nodes with a low fill ratio, effectively saving space.

Other Characteristics of LOFT. We collect runtime statistics of LOFT to examine the performance impact due to the expanded learned buckets and logging. For easy-to-approximate datasets, the expanded buckets are rarely allocated, since most records fall into the predicted positions within the data array. The space overhead of expanded buckets is near 0. When the dataset becomes harder and the workload becomes write-intensive, more records are spilled into the expanded buckets. However, our retraining scheme efficiently merges data from buckets into the data array, conserving space and enhancing performance. As a result, the overall bucket levels remain low, typically not exceeding one, with 6-22% of records utilizing expanded buckets in write-intensive scenarios. Thus, setting the expanded factor to 8 incurs reasonable overheads. During retraining, insertions need to be logged. However, the average logging size is no more than the pre-allocated log size, i.e., 32, because our retraining scheme is fast, e.g., about 30ns per record, and the number of retrained records in a data node is negligible compared with the whole data size. Hence, our retraining scheme performs retraining promptly to improve the system performance with reasonable time and space overhead.

6 Discussions

8-byte Keys. Similar to existing lock-free designs like Bw-Tree [31], and Clevel [10], our proposed scheme also leverages the 8-byte primitive CAS for executing lock-free operations. We align with previous designs of learned indexes that operate with 8-byte keys. For longer keys, we can generate an 8-byte tag to represent the key, e.g., an 8-byte fingerprint, and store the tag along with the pointer to the key-value pairs in the data node. Therefore, we can CAS with the tag to determine the existence of a given key within the prediction range. While performing index operations, LOFT requires two additional steps: computing the tags with the hash function in the data node and dereferencing the pointer to ensure the key matches the queried key. LOFT resolves hash collisions by labeling the conflicted record as *collision* and temporarily storing the new record in the expanded bucket. During retraining, LOFT will change the hash function for this node.

Duplicated Entries. When inserting a duplicated entry, LOFT will find the existence of the record and update the value with the newly inserted one. However, LOFT can support duplicated entry insertions via minor modifications to the existing insertion design. When inserting a redundant key into the index, LOFT appends the new value with the existing one and atomically replaces the pointer to the value region with the new one, since we store the value pointer, instead of the value, in the data array.

7 Related Work

Existing schemes explore the implementations of learned indexes for various index structures, such as spatial indexes [33], secondary indexes [25, 53], Bloom filters [35, 42] and LSM-Trees [14]. These schemes leverage machine learning schemes to obtain performance optimization and accelerate the search process, or delicately design the learned index structures to address the limitations of traditional indexes.

However, most of the current learned designs still focus on the learned range indexes to optimize the write performance. ALEX [17], and LIPP [49] reserve some free slots in advance to hold the insertions. The concurrent learned index, e.g., XIndex [47], uses delta buffers to temporarily store the newly inserted records. The two-stage compaction design compacts the buffer with the data array for retraining. PGM-index [18] generates the models according to the data distribution and provides error-bounded searches. In order to alleviate the data dependency in the learned indexes, FINEdex [32] proposes a learning probe algorithm to train independent models and leverages fine-grained retraining to improve the write performance. CARMI [56] designs a cache-aware RMI framework to improve the performance under real-world workloads. SALI [19] leverages node-evolving strategies to adapt to dynamic workloads. Besides, DiffLex [13] aims to eliminate the impact of NUMA nodes on the learned index performance.

Learned indexes leverage the non-volatile memory, e.g., persistent memory (PM) [50], to improve system performance, which mainly focus on reducing random PM accesses. APEX [37] based on ALEX uses hash-based data nodes to avoid data shifts caused by insertions and thus improves system performance in PM [1]. PLIN [58] puts the entire index into PM to achieve instant recovery. Moreover, FILM [39] utilizes learned models for indexing data that span different storage devices and uses an adaptive Least Recent Used (LRU) structure to reduce disk I/Os.

None of the existing schemes can simultaneously achieve high throughput and high scalability in dynamic workloads. Compared with current in-memory learned indexes, LOFT mitigates the performance variation over different workloads and delivers high performance in dynamic workloads with various key distributions and access patterns.

8 Conclusion

In this paper, we propose LOFT, a scalable and concurrent learned index to provide low-latency and high-throughput services. LOFT adopts an error-bounded insertion to settle writes in the predicted positions, thus benefiting read performance. LOFT also performs retraining with reasonable costs and adjusts the parameters for retraining to adapt to dynamic workloads. Extensive evaluation on YCSB and real-world workloads demonstrates that our design effectively improves the performance by up to 14× than state-of-the-art schemes. We have released the open-source codes for public use on GitHub.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd, Shimin Chen, and anonymous reviewers for their feedbacks and suggestions.

References

[1] Intel optane persistent memory (pmem). https://www.intel.ca/content/www/ca/en/architecture-and-technology/optanedc-persistent-memory.html. last accessed on 7/4/2023.

[2] Memcached: A distributed memory object caching system. https://memcached.org/. last accessed on 5/10/2024.

[3] Openstreetmap on aws. https://registry.opendata.aws/osm. last accessed on 7/4/2023.

[4] Redis. https://redis.io/. last accessed on 5/10/2024.

[5] Userspace rcu. https://liburcu.org. last accessed on 7/4/2023.

[6] Intel advanced vector extensions 2 (avx2) instruction set. https://software.intel.com/content/www/us/en/homepage.html, 2013. last accessed on 5/10/2024.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Christian A. Lang, and Kenneth A. Ross. Buffered bloom filters on solid state storage. In Rajesh Bordawekar and Christian A. Lang, editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–8, 2010.

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, Boston, MA, July 2018. USENIX Association.

[10] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[11] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. Difflex: A high-performance, memory-efficient and numa-aware learned index using differentiated management. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 62–71, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–171. USENIX Association, November 2020.

[15] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja, and David H.C. Du. Bloomflash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, 2011.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220.

[17] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, April 2020.

[19] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. Sali: A scalable adaptive learned index framework based on probability models. *Proc. ACM Manag. Data*, 1(4), dec 2023.

[20] G. Graefe and P.-A. Larson. B-tree indexes and cpu caches. In *Proceedings 17th International Conference on Data Engineering (ICDE)*, pages 349–358, 2001.

[21] Goetz Graefe and Harumi Kuno. Modern b-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1370–1373, 2011.

[22] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. 12(3):463–492, July 1990.

[23] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. STYX: exploiting smartnic capability to reduce datacenter memory tax. In Julia Lawall and Dan Williams, editors, *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 619–633. USENIX Association, 2023.

[24] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, page 424–439, 2021.

[25] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. Lsi: a learned secondary index structure. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '22, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[27] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, Houston TX USA, May 2018. ACM.

[28] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, dec 1981.

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[30] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313, 2013.

[32] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.*, 15(2):321–334, oct 2021.

[33] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2119–2133, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. {ElasticBF}: Elastic bloom filter with hotness awareness for boosting read performance in large {Key-Value} stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[35] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. Stable learned bloom filters for data streams. *Proc. VLDB Endow.*, 13(11):2355–2367, 2020.

[36] Christian E Lopez and Caleb Gallemore. An augmented multilingual twitter dataset for studying the covid-19 infodemic. *Social Network Analysis and Mining*, 11(1):102, 2021.

[37] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. APEX: A High-Performance Learned Index on Persistent Memory. *Proceedings of the VLDB Endowment*, 15(3):597–610, November 2021.

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A High-Performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 553–571, Boston, MA, July 2023. USENIX Association.

[39] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. Film: A fully learned index for larger-than-memory databases. *Proc. VLDB Endow.*, 16(3):561–573, nov 2022.

[40] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[41] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. Rcu usage in the linux kernel: One decade later. *Technical report*, 2013.

[42] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems (NIPS)*, 31, 2018.

[43] Suhas S. P. Rao, Miriam H. Huntley, Neva C. Durand, Elena K. Stamenova, Ivan D. Bochkov, James T. Robinson, Adrian L. Sanborn, Ido Machol, Arina D. Omer, Eric S. Lander, and Erez Lieberman Aiden. A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell*, 159:1665–1680, 2014.

[44] Jinglei Ren, Long Deng, and Chris Kjellqvist. Ycsb-c. https://github.com/basicthinker/YCSB-C.

[45] Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo. Mitigating asymmetric read and write costs in cuckoo hashing for storage systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 329–344, Renton, WA, July 2019. USENIX Association.

[46] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 308–320, San Diego California, February 2020. ACM.

[47] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. The concurrent learned indexes for multicore data storage. *ACM Trans. Storage*, 18(1), jan 2022.

[48] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are updatable learned indexes ready? *Proc. VLDB Endow.*, 15(11):3004–3017, jul 2022.

[49] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, apr 2021.

[50] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, Santa Clara, CA, 2020. USENIX Association.

[51] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. Dytis: A dynamic dataset targeted index structure simultaneously efficient for search, insert, and scan. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 800–816, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[53] Jianan Yuan, Huan Liu, Shangyu Wu, Yiquan Lin, Tiantian Wang, Chenlin Ma, Rui Mao, and Yi Wang. Work-in-progress: Lark: A learned secondary index toward lsm-tree for resource-constrained embedded storage systems. In *2022 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 11–12, 2022.

[54] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems. *Proc. VLDB Endow.*, 15(6):1187–1200, feb 2022.

[55] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1567–1581, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Jiaoyi Zhang and Yihan Gao. Carmi: a cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.*, 15(11):2679–2691, jul 2022.

[57] Wei Zhang, Jinho Hwang, Timothy Wood, K.K. Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, June 2014. USENIX Association.

[58] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. Plin: A persistent learned index for non-volatile memory with high performance and instant recovery. *Proc. VLDB Endow.*, 16(2):243–255, oct 2022.

[59] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 461–476, 2018.

A Artifact Appendix

A.1 Abstract

This appendix provides instructions on obtaining the source code, setting up the environment, and compiling the project.

A.2 Description & Requirements

All requirements for running the source code are listed in the LOFT GitHub repository. The README file also includes an overview of LOFT's organization.

A.2.1 How to access. LOFT is open-source and available on GitHub at https://github.com/yuxuanMo/LOFT.git. The artifact repository can also be accessed from Zenodo (https://doi.org/10.5281/zenodo.14931074)

A.2.2 Hardware Dependencies. None.

A.2.3 Software Dependencies. The project requires the following dependencies:

- **CMake (3.2+)** for building and testing.
- **Intel MKL** for optimized mathematical operations.
- **jemalloc** for efficient memory management.
- **Userspace RCU (URCU)** for synchronization mechanisms.

A.2.4 Benchmarks. We provide a simple benchmark, named `microbench`, located in the bench folder. This benchmark generates synthetic workloads, including uniform, normal, and lognormal distributions. It does not require external input datasets.

A.3 Setup

All installation and configuration steps required to prepare the environment for LOFT evaluation are provided in the README.

A.4 Evaluation Workflow

The README includes instructions for installing dependencies such as Intel MKL, jemalloc, and URCU. After installation, the `CMakeLists.txt` file needs to be modified accordingly, as described in the README.

To build the project:

1. Create a build folder.
2. Use `cmake` to configure the build.
3. Compile using `make`.

A.5 Major Claims

The provided benchmark evaluates the functionality of LOFT. Users can select from three workload distributions—normal, lognormal, and uniform—to measure throughput after executing a specified number of operations.

A.6 Experiments

The `microbench` benchmark supports several configurable parameters:

- `bg_n`: Number of background threads.
- `fg_n`: Number of worker threads.
- `data_num`: Total number of generated keys.
- `init_num`: Number of keys used to initialize the index.
- `oper_num`: Total number of generated operations.
- `benchmark`: Workload type (0: normal, 1: lognormal, 2: uniform).
- `insert`: Insert operation ratio.
- `read`: Read operation ratio (the sum of `insert` and `read` should be 1).