SALI: A Scalable Adaptive Learned Index Framework based on Probability Models

JIAKE GE, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), China and School of Information, Renmin University of China, China
HUANCHEN ZHANG, Tsinghua University, China and Shanghai Qi Zhi Institute, China
BOYU SHI, Engineering Research Center of Database and Business Intelligence (MOE), China and School of Information, Renmin University of China, China
YUANHUI LUO, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), China and School of Information, Renmin University of China, China
YUNDA GUO, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), China and School of Information, Renmin University of China, China
YUNPENG CHAI*, Engineering Research Center of Database and Business Intelligence (MOE), China and School of Information, Renmin University of China, China
YUXING CHEN, Tencent Inc., China
ANQUN PAN, Tencent Inc., China

The growth in data storage capacity and the increasing demands for high performance have created several challenges for concurrent indexing structures. One promising solution is the learned index, which uses a learning-based approach to fit the distribution of stored data and predictively locate target keys, significantly improving lookup performance. Despite their advantages, prevailing learned indexes exhibit constraints and encounter issues of scalability on multi-core data storage.

This paper introduces SALI, the **S**calable **A**daptive **L**earned **I**ndex framework, which incorporates two strategies aimed at achieving high scalability, improving efficiency, and enhancing the robustness of the learned index. Firstly, a set of node-evolving strategies is defined to enable the learned index to adapt to various workload skews and enhance its concurrency performance in such scenarios. Secondly, a lightweight strategy is proposed to maintain statistical information within the learned index, with the goal of further improving the scalability of the index. Furthermore, to validate their effectiveness, SALI applied the two strategies mentioned above to the learned index structure that utilizes fine-grained write locks, known as LIPP.

*Yunpeng Chai is the corresponding author.

Authors' addresses: Jiake Ge, gejiake@ruc.edu.cn, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Beijing, China and School of Information, Renmin University of China, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China; Boyu Shi, shiboyu5687@ruc.edu.cn, Engineering Research Center of Database and Business Intelligence (MOE), Beijing, China and School of Information, Renmin University of China, Beijing, China; Yuanhui Luo, losk@ruc.edu.cn, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Beijing, China and School of Information, Renmin University of China, Beijing, China; Yunda Guo, guoyunda@ruc.edu.cn, Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Beijing, China and School of Information, Renmin University of China, Beijing, China; Yunpeng Chai, ypchai@ruc.edu.cn, Engineering Research Center of Database and Business Intelligence (MOE), Beijing, China and School of Information, Renmin University of China, Beijing, China; Yuxing Chen, axingguchen@tencent.com, Tencent Inc., , China; Anqun Pan, aaronpan@tencent.com, Tencent Inc., , China.

2836-6573/2023/12-ART258 $15.00
https://doi.org/10.1145/3626752

The experimental results have demonstrated that SALI significantly enhances the insertion throughput with 64 threads by an average of 2.04× compared to the second-best learned index. Furthermore, SALI accomplishes a lookup throughput similar to that of LIPP+.

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

Additional Key Words and Phrases: learned index, concurrency structure, probability models, adaptive structure

ACM Reference Format:
Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 258 (December 2023), 25 pages. https://doi.org/10.1145/3626752

1 INTRODUCTION

With the exponential growth of the data volume today, efficient indexing data structures are crucial for a big data system to support timely information retrieval. To improve the performance and memory efficiency of traditional tree-based indexes, Kraska et al. introduced a learned index, called the Recursive Model Indexes (RMI) that uses machine learning models to replace the internal nodes of a B+tree [15, 20]. An outstanding problem of the original RMI is that it is static: inserting or updating a key in the index requires a significant portion of the data structure to rebuild, thus limiting the use cases of the learned index.

Previous work has proposed two strategies to address the updatability issue of learned indexes. The first (i.e., the buffer-based strategy) is to accommodate new entries in separate insert buffers first to amortize the index reconstruction cost. XIndex [36] and FINEdex [21] fall into this category. The other strategy (i.e., the model-based strategy) adopted by ALEX [3] and LIPP [39] is to reserve slot gaps within nodes to handle new entries with an in-place insertion. Upon an insert collision (i.e., the mapped slot is already occupied), ALEX reorganizes the node by shifting the existing entries, while LIPP utilizes a chaining scheme, creating a new node for the corresponding slot to transform the last-mile search problem into a sub-tree traversal problem.

We found, however, that none of the above index designs scale at a high concurrency. We performed an experiment where we insert 200 million random integer keys into a learned index, with a varying number of threads each time. Figure 1 shows the results. Note that the number of threads in the grey area of the figure is larger than the number of hardware threads of the machine. This is common in practice as a database/key-value server typically handles a large number of user connections simultaneously.

As shown in Figure 1, indexes with a buffer-based strategy (i.e., XIndex and FINEdex) exhibit inferior base performance and worse scalability compared to those with a model-based strategy (i.e., ALEX+ and LIPP+[1]). This shows that **a larger margin of prediction errors prevents scaling** because the concurrent "last mile" searches saturate the memory bandwidth quickly which becomes the system's bottleneck [38].

The problem is solved in LIPP+ where each position prediction is accurate (i.e., no "last mile" search). However, LIPP+ requires maintaining statistics, such as access counts and collision counts, in each node to trigger node retraining to prevent performance degradation. These **per-node counters create high contention** among threads and cause severe cacheline ping-pong [38]. The model-based strategy in ALEX+ requires shifting existing entries upon an insert collision. Therefore, ALEX+ must acquire **coarse-grained write locks** for this operation. As the number of threads increases, more and more threads are blocked, waiting for those exclusive locks.

In this paper, we propose SALI, the **S**calable **A**daptive **L**earned **I**ndex framework based on probability models to solve the scalability issues in existing solutions. To solve the scalability

[1]ALEX+ and LIPP+ are concurrent implementations of ALEX and LIPP, respectively [38].

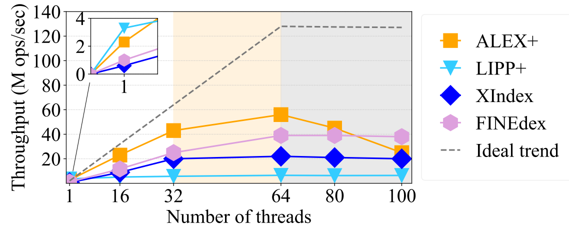


Fig. 1. Write-only performance of state-of-the-art learned indexes on the FACE dataset [11]. The evaluation is conducted on a two-socket machine with two 16-core CPUs.

bottleneck of maintaining per-node statistics, we developed lightweight probability models that can trigger node retraining and other structural evolving operations in SALI with accurate timing (as if the timing were determined by accurate statistics). In addition, we developed a set of node-evolving strategies, including expanding an insert-heavy node to contain more gaps, flattening the tree structure for frequently-accessed nodes, and compacting the rarely-touched nodes to save memory. SALI applies these node-evolving strategies adaptively according to the probability models so that it can self-adjust to changing workloads while maintaining excellent scalability. Finally, SALI adopts the learned index structure that utilizes fine-grained write locks, i.e., LIPP+, to validate the effectiveness of the aforementioned two strategies. Note that the lightweight probability models and node-evolving strategies are highly versatile and can be applied to various index scenarios, as detailed in Section 5.

Our microbenchmark with real-world data sets shows that SALI improves the insertion throughput with 64 threads by 2.04× on average compared to the second-best learned index, i.e., ALEX+, while achieving a lookup throughput comparable to LIPP+.

We make three primary contributions in this paper. Firstly, we proposed SALI, a high-concurrency learned index framework designed to improve the scalability of learned indexes. Secondly, we defined a set of node-evolving strategies in addition to model retraining to allow the learned index to self-adapt to different workload skews. Thirdly, we replaced the per-node statistics in existing learned indexes with lightweight probability models to remove the scalability bottleneck of statistics maintenance while keeping the timing accuracy of node retraining/evolving. Finally, we proved the effectiveness of the proposed approaches by showing that SALI outperforms the SOTA learned indexes under high concurrency.

The rest of this paper is organized as follows. Section 2 summarizes the basics of learned indexes and further motivates the scalability problem. Section 3 introduces the structure of SALI with an emphasis on the node-evolving strategies and the probability models. Section 4 presents our experimental results. Section 5 discusses the generalizability of the node-evolving strategies and the probability models, as well as the limitations of SALI, followed by a related work discussion in Section 6 Section 7 concludes the paper. The source code of this paper has been made available at https://github.com/YunWorkshop/SALI.

2 BACKGROUND AND MOTIVATION

2.1 The principle of learned indexes

The core concept of the learned index is to employ a set of learning models to estimate the cumulative distribution function (CDF) of the stored data [29], allowing for the prediction of the data's storage location, as depicted in the CDF diagram on Figure 2.

Figure 2 shows the scheme for the learned index structure. Each node, or only the lowest leaf nodes, stores the slope and intercept of the linear function [3, 5, 8, 15, 21, 36, 39]. Each segment

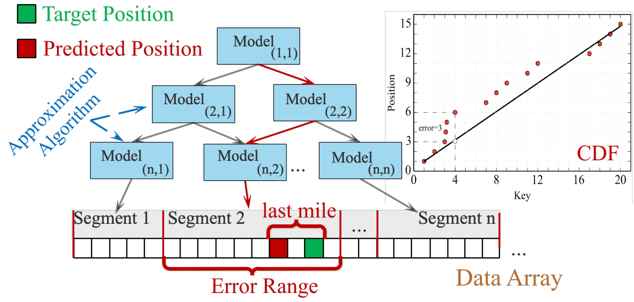


Fig. 2. The scheme of the learned index.

corresponds to a linear model, which is responsible for the approximate position of the target key. The index segments correspond to linear models that estimate the target key's position, eliminating the need for multiple indirect search operations in traditional tree-based indexes. This approach has the potential to improve indexed lookup performance significantly.

2.2 Scalable Evaluation of Learned Index Structures

Currently, learned indexes demonstrate good performance in single-threaded environments. However, their scalability remains limited [38]. In this part, our objective is to conduct a thorough investigation into the factors that contribute to the concurrent performance bottlenecks in existing learned indexes. To achieve this, we begin by introducing the insertion strategies employed in learned indexes, along with their corresponding index structures, as these design choices significantly influence the concurrent performance of the indexes. Additionally, we conduct a comprehensive evaluation of the index structures and identify their limitations in terms of scalability.

2.2.1 The insertion strategies of learned indexes. In a concurrent scenario, the blocking of index operations is primarily due to the insertion of new keys. Understanding the current insertion strategies is essential for enhancing index scalability. Thus, we present the insertion strategies as follows.

Strategy 1: Scholars try to design a buffer-based insert strategy, i.e., off-site insertion, on learned indexes to implement insert operations [5, 8, 21, 36]. As shown in Figure 3, the core idea of the buffer-based strategy is to create a buffer structure for insertion [5, 8, 21, 36]. When the buffer is full, its keys must be merged with those in the upper segment and transformed into a new linear model. Furthermore, this structure suffers from significant errors due to the intensive storage of keys (no gap) [10].

Strategy 2: The core idea of the model-based insert strategy, i.e., in-place insertion, involves reserving gaps in the nodes [3, 39]. If the linear model predicts that the target position of the inserted key is a gap, it is directly inserted. However, if the slot with the key already exists, there are two existing conflict resolution strategies:

• *Solution 1:* One potential solution for resolving conflicts is to adopt the "shift" method [3]. As shown in Figure 3, this method involves shifting the existing conflicted key and its adjacent keys to the nearest gap by one slot, allowing the target key to be inserted into the target position. However, the process of key movement can introduce errors.

• *Solution 2:* Another solution to resolve conflicts is to adopt the "chain" method [39]. As shown in Figure 3, if a key already exists at the target position of the newly inserted key, a new node is

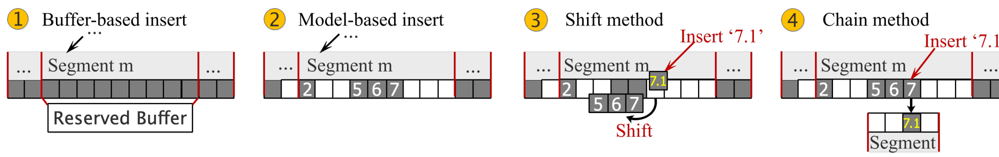


Fig. 3. The updatable strategies in learned indexes.

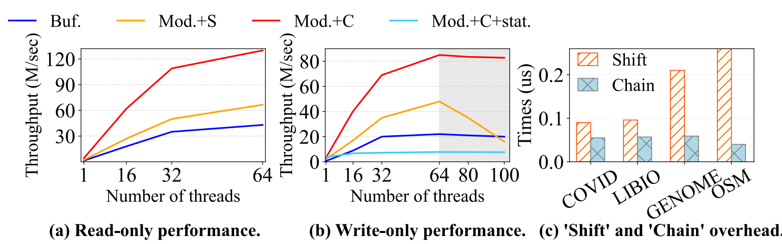


Fig. 4. The in-depth evaluation of performance on the COVID dataset [38], where the workload follows the uniform distribution. The evaluation is conducted on a two-socket machine with two 16-core CPUs.

created downward to accommodate the conflicting key. This conflict resolution approach does not involve moving any data, thus avoiding potential lookup errors (precise lookup).

2.2.2 In-depth analysis of these strategies. This part will provide an in-depth analysis of the index structures corresponding to the insertion strategies mentioned. Building upon the GRE [38], we further performed an in-depth experimental analysis of existing learned indexes and identified the scalability problems in their designs. Our objective is not to compare them with each other but to highlight the scalability bottleneck.

Figure 4 illustrates the performance for the three structures. The buffer-based structure is denoted by $buf.$ (i.e., the structure of XIndex), the ***Mod***el-based strategy with the ***S***hift method is denoted by $Mod.{+}S$ (i.e., the structure of ALEX), the ***Mod***el-based strategy with the ***C***hain method is denoted by $Mod.{+}C$ (i.e., the structure of LIPP). The notation $Mod.{+}C+stat.$ is used to represent the $Mod.{+}C$ approach along with the maintenance of ***stat***istics to track index deterioration. Note that except for $Mod.+C+stat.$, we disabled the statistics maintenance and local adjustment functions with the purpose of analyzing the impact of the structures themselves on index concurrency performance.

Figure 4(a) displays that both $buf.$ and $Mod.+S$ exhibit poor scalability due to lookup errors, which can impact both the lookup and insert performance. Note that, the average search error of $buf.$ is higher compared to $Mod.+S$ [10].

Observation 1: Improved concurrency performance can be achieved through the utilization of precise lookups. The insertion strategy employed by *Mod.+C* guarantees error-free generation even during the insertion process.

Figure 4(b) depicts the insertion performance for the three structures. The poor performance of $buf.$ is attributed to a lookup error and an off-site insertion method [10]. $Mod.{+}S$ has three reasons for its poor performance: (1) Severe lookup errors; (2) Significant write amplification and frequent "last mile" lookup lead to exhaustion of memory bandwidth and affect concurrency performance [38]. This amplification occurs because the moving key and the inserted target key need to be written into the node together; (3) During insertion, coarse-grained locks can easily cause thread blocking. The shift method leads to significant correlations between keys in the entire node, necessitating

Table 1. The limited scalability of existing schemes.

Learned index	Basic performance		Concurrency		Evolving ability
	No errors	In-place insert	Fine lock	l-w statistics	
RMI[15]	×	×	×	×	×
FITing[8]	×	×	×	×	×
PGM[5]	×	×	×	×	×
ALEX+[3]	×	√	×	√	×
LIPP+[39]	√	√	√	×	×
XIndex[36]	×	×	√	×	×
FINEdex[21]	×	×	√	×	×
SALI	**√**	**√**	**√**	**√**	**√**

the locking of the entire node during insertion to ensure accuracy. Moreover, the thread blocking issue is more pronounced in the gray area in Figure 4(b), where threads frequently access nodes locked by coarse-grained locks and release CPU time slices, resulting in invalid operations and exacerbating performance degradation. In fact, the coarse-grained locks have already caused a slowdown in concurrent performance growth under 64 threads, which may not have been apparent due to the influence of other scalability factors, e.g., lookup error.

Observation 2: *Mod.+C* facilitates fine-grained locks, i.e., one slot, no search errors, and in-place insertion, which enables its good insertion scalability.

The scalability of $Mod. + C + stat.$ is severely impacted, depicted in Figure 4(b), primarily due to the high level of contention and cache-line ping-pong that arises from maintaining statistics in a concurrent scenario, which is consistent with the findings of Wongkham et al. [38]. Figure 4(c) shows that $Mod. + C$ saves 0.8x-5x the average time compared to $Mod. + S$ with different datasets.

Observation 3: Maintaining statistics within *Mod.+C* renders the index non-scalable. Note that the scalability issue is not attributed to the structure of *Mod.+C*, but rather to the absence of a lightweight statistical approach.

Observation 4: The chain method exhibits significantly shorter operation times compared to the shift method.

2.3 The Scalable Learned Index Requirements

In consideration of scalability, we have further summarized the potential limitations of several SOTA learned indexes in Table 1, taking into account factors such as prediction accuracy (*No errors*), insert strategy (*In-placeinsert*), lock granularity (*Fine lock*), lightweight statistics support (*l-w statistics*), etc. A checkmark denotes support for the given factor, while a cross sign indicates the lack of support.

We believe that **designing a learned index requires prioritizing concurrency control and robustness as first-class considerations, adopting a holistic approach to ensure consistency in design choices.** Therefore, considering the challenges associated with learned indexes, we propose that a more scalable learned index should simultaneously address the following dimensions:

2.3.1 *Efficient concurrency.*

1) Maintaining statistics should barely impact scalability. To enable efficient insertion performance, updatable learned indexes must track statistical information that reflects the degradation of the index structure over time due to new insertions. This information is crucial for performing necessary retraining operations. However, maintaining these cumulative statistics jointly by the insertion thread can potentially lead to blocking, becoming a scalability bottleneck for some

state-of-the-art learned indexes [38]. Therefore, there is an urgent need to develop a lightweight methodology to maintain statistics.

2) Designing effective index structures for concurrent scenarios. In concurrent scenarios, insertion performance in learned indexes can be hampered by blocking that often arises when multiple insertion threads work together to uphold key consistency in a single local structure, particularly under skewed workloads. To mitigate this issue, reducing the manipulation of already-stored keys during the insertion of new ones can help minimize lock granularity and lower the risk of thread blocking [21].

2.3.2 Adaptive ability.

The learned index exhibits suboptimal performance under skewed insertion workloads compared to uniform workloads. The lack of workload-aware adaptive adjustment capability is the primary cause of this deficiency. Therefore, it is critical for a learned index to possess the adaptive capacity to guarantee its robustness in concurrent scenarios. In addition, the learned index lacks an optimization adjustment strategy for the lookup operation, which hinders its ability to maximize lookup efficiency in concurrent scenarios. Furthermore, learned indexes have yet to fully capitalize on opportunities for significantly reducing index space costs under skewed workloads [1].

2.3.3 Low overheads of basic performance.

1) Efficient lookup. Achieving high lookup performance in learned indexes typically hinges on minimizing prediction errors for lookups, as substantial errors can lead to many "last mile" operations in a concurrent scenario. These operations consume additional memory bandwidth and negatively impact concurrent performance [38].

2) Efficient insert. Adopting the model-based strategy (i.e., the in-place insertion) rather than the buffer-based strategy (i.e., the off-site insertion), can significantly enhance the insert performance of learned indexes by reserving gaps in each node (see Section 2.2).

3 SALI: A PROBABILITY-BASED EVOLVABLE LEARNED INDEX FRAMEWORK

This section introduces the SALI framework, which addresses the concerns and requirements discussed in Section 1 and 3, and facilitates the efficient scalability of learned indexes. Specifically, Section 3.1 introduces the overall architecture of SALI, including the architecture built upon $Mod.{+}C$ (i.e., the structure of LIPP) (columns 2-4 of Table 1). In Section 3.2, we proposed an adaptive evolving (adjustment) strategy to further improve the robustness of the learned index under skewed and uniform workloads (columns 6 of Table 1). Section 3.3 designs a probability-based lightweight method to maintain statistics of different roles at a meager cost. This method solves the concurrency performance bottleneck problem caused by the existing high contention statistics method (columns 5 of Table 1).

3.1 Overview

3.1.1 SALI framework. This part introduces the structure of the SALI framework, which encompasses the introduction of a probability-based lightweight methodology for statistics maintenance and the implementation of an adaptive evolving strategy. Due to the common occurrence of skewed workloads in real-world environments, it is advisable to apply different evolution strategies to nodes with varying degrees of read-write hotness. These strategies can serve as alternatives to the traditional retraining method, as they are specifically designed to enhance concurrent performance and reduce the overhead of index space (Section 3.2). Furthermore, SALI utilizes a probability-based lightweight method to maintain statistics while keeping the timing accuracy of node retraining/evolving without blocking insertion operations from multiple threads, unlike the traditional approach of globally maintaining statistics (Section 3.3).

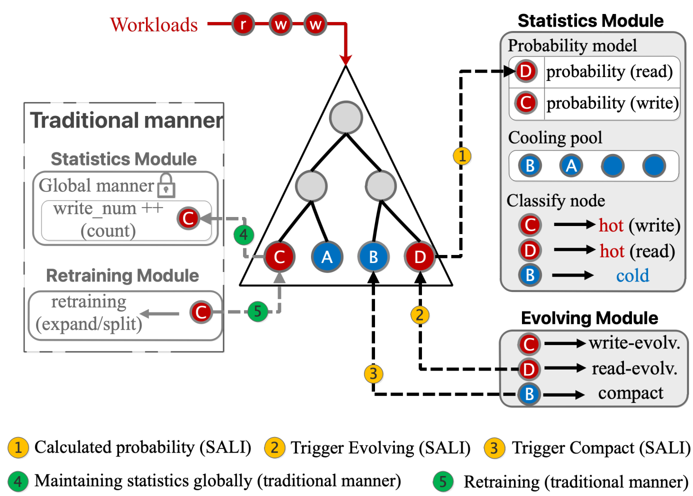


Fig. 5. The structure of SALI.

Specifically, the SALI framework consists of two phases in Figure 5. In the first phase, we calculate the probabilities 1 for each node that requires evolving during lookup/insert operations to improve performance. Calculating and maintaining probability models is a lightweight alternative to maintaining statistics, which does not cause high contention and thread blocking issues (see Section 3.3). On the contrary, the traditional manner globally maintains statistics 4, leading to high contention among threads and limiting index scalability. In the second phase, we perform evolving operations on nodes that are classified as hot 2 or cold 3. Note that our evolving strategy encompasses the functionality of retraining operations in the traditional manner 5 (refer to Section 3.2).

3.1.2 The structure of SALI builds upon the $Mod. + C$. Based on our observation in Sections 2.2, we have determined that the structure $Mod. + C$, i.e., the structure of LIPP, exhibits the highest scalability among the options considered. Consequently, we have opted to implement and evaluate our novel strategies utilizing $Mod.+C$ as the underlying index structure. Note that in the subsequent context, SALI refers to the structure built upon $Mod. + C$, as illustrated in Figure 6. Section 3.1.3 and 3.1.4 will introduce the operations of SALI and the coordination between different operations.

3.1.3 Operations of SALI.

1) Lookup operation: SALI employs a linear model to accurately predict the position of the lookup key, except for cold nodes with errors. The search is considered successful during a query if the key contained in the prediction slot is equal to the target key (Algorithm 1, line 4-6). Otherwise, it does not exist (Algorithm 1, line 7-9). However, if the predicted slot is a pointer, the search continues in the node pointed to by the pointer (Algorithm 1, line 10-22). At this stage, the type of node needs to be determined. For hot lookup nodes, SIMD [12] is utilized to locate the node containing the target key, followed by a linear model search in this node (see Figure 7(b) and Algorithm 1, line 12-15). In contrast, for cold nodes, where the linear model prediction has an error, a binary search method covers the “last mile” (see Figure 7(c) and Algorithm 1, line 16-18). The rest of the nodes are searched directly using the linear model (Algorithm 1, line 19-20).

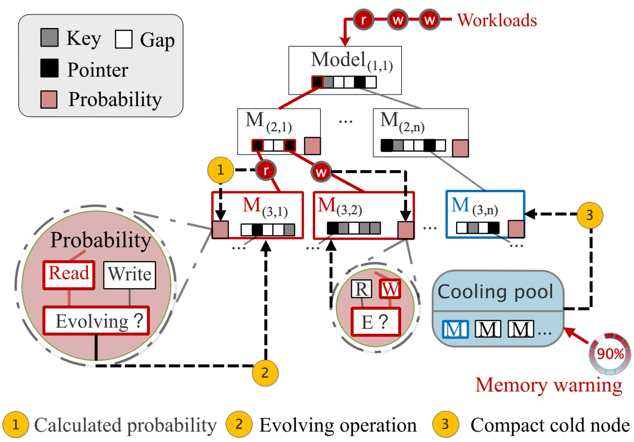


Fig. 6. The structure of SALI builds upon the *Mod.* + *C*.

2) Insert operation: Initially, the read operation algorithm is used to identify the appropriate location for inserting the key. If the key already exists at this position, a new storage space is created below, and the key is inserted in this space to handle conflicts (Algorithm 2, lines 5-9). On the other hand, if the insertion position is a gap, the key is inserted directly (Algorithm 2, lines 2-4).

3) Evolving operation: See Section 3.2 for details.

4) Building operation: SALI adopts the structure of LIPP and therefore utilizes the construction algorithm of LIPP, i.e., fastest minimum conflict degree (FMCD) [39]. In the SALI and the realm of the learned index, other linear and even non-linear approximation algorithms are crucial and intriguing avenues for future research.

3.1.4 Coordination between different operations. Reads can proceed without acquiring the lock as long as SALI verifies that the item being read (i.e., data or child pointer) has not been modified.

SALI employs an optimistic locking mechanism for the target slot during concurrent writing. Since the SALI's structure ensures that only fine-grained locks are necessary to guarantee mutually exclusive writes, write conflicts are rare under a uniform workload.

To prevent uncontrollable tail latency that may arise from prolonged evolution, we restrict the evolution process to nodes with less than one million keys. Our observations indicate that indexes with higher write rates require periodic rebuilding to maintain good performance. Consequently, during periods of relative inactivity in the storage system, the entire index structure can be rebuilt, resulting in a flatter SALI structure and improved performance.

Furthermore, when the node is evolving, we use the Read-Copy-Update (RCU) mechanism [31, 35] to prevent the blocking of read operations, i.e., reading the old version of data. Following the evolution operation, SALI utilizes the RCU to ensure all threads can access the new model. RCU barrier is a synchronization mechanism designed for concurrent systems, which enables all readers to access the new space in shared memory after evolving operation. In addition, to ensure that child nodes being read are not deleted during the evolution process, SALI utilizes the epoch-based reclamation [6] that guarantees the safety of node pointers in a concurrent scenario.

3.2 Evolving Strategies

A more comprehensive adaptation strategy than simply retraining is required to adapt the learned index structure under various workloads. This part presents the design of an evolving strategy

Algorithm 1 SALI Lookup Operation

```
Require: Target key: k.
function lookup_operation(k)
    predicted_slot ← root.linear(k)
    while (TRUE) do
        if predicted_slot.type == key then
            if predicted_slot.data == k then
                return (predicted_slot.data)
            else
                return (not found)
            end if
        else                                          ▷ predicted_slot.type == pointer
            nodes_meta ← predicted_slot.data
            if (nodes_meta.type == hot_lookup) then
                nodes_meta.linear ← nodes_meta.SIMD(k)
                predicted_slot ← nodes_meta.linear(k)
            end if
            if (nodes_meta.type == cooling) then
                pred_slot_pre ← nodes_meta.linear(k)
                predicted_slot ← bi_search(pred_slot_pre, k)
            else                                      ▷ nodes_meta.type == nomal
                predicted_slot ← nodes_meta.linear(k)
            end if
        end if
    end while
end function
```

Algorithm 2 SALI Insertion Operation

```
Require: Target key: k.
function insertion_operation(k)
    predicted_slot ← lookup_operation(k)
    if (predicted_slot = NULL) then
        insert(predicted_slot, k)
    else                                              ▷ predicted_slot.type == key
        old_k ← predicted_slot.data
        pointer ← predicted_slot.type
        new_node ← predicted_slot.data
        insert(new_node, old_k, k)
    end if
end function
```

that focuses on three aspects to enhance the concurrency performance of the learned index. Note that this part only covers the evolving strategy, while the conditions and timing for triggering the evolving process will be discussed in Section 3.3. Next, we will briefly introduce the difference between evolving and retraining:

a) Retraining is a passive adjustment strategy used in updatable learned indexes to maintain their performance. Its main features are: 1) it is driven by the deterioration of the index structure and cannot sense different workloads; 2) it is triggered only by insert operations optimized exclusively for improving insert performance; and 3) it does not change the index structure in essence.

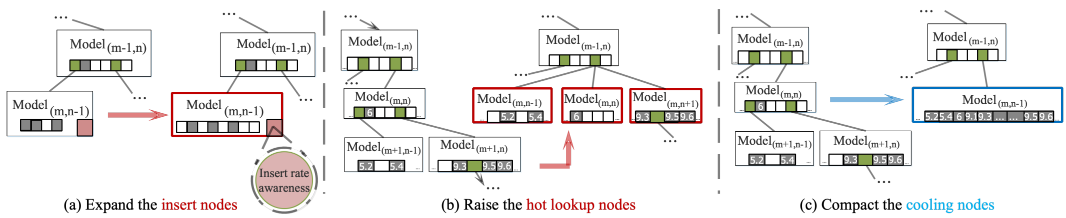


Fig. 7. The evolving strategies.

b) Evolving, proposed in this paper, is a novel concept in learned indexes that includes the retraining function and improves the index's adjustment mechanism in different dimensions. Its main features are: 1) it is an active adjustment strategy that perceives and is driven by the workload to improve index performance further; 2) it can be triggered by any operation (e.g., read); and 3) it can "evolve" into a new structure type that is suitable for the current workload for both improving the read and insert performance.

3.2.1 The insert triggers the evolving operation. Most retraining methods, including SALI, involve expanding the target node or its subtree. This expansion creates more gaps that can be used for inserting keys, thereby improving the overall insertion performance. As shown in Figure 7(a), the gap array within the data node is increased from two to five to accommodate more keys. To achieve this, the FMCD algorithm [39] is used to expand the node by inputting all of the keys in the node and the desired space size after expansion (see Algorithm 3, lines 1-4).

However, a fixed retraining expansion factor may not be sufficient to handle sudden increases in local insertions under a skewed workload, which can lead to a high number of insert conflicts in concurrent scenarios. To address this, the expansion factor should be **adaptively** adjusted based on the insertion rate to determine the optimal expansion size. Specifically, more gaps are reserved to enhance the insertion performance when the insertion rate increases. For more information, please refer to Equation (1) below:

$$n.expand\_size = \begin{cases} \gamma \times \frac{n.speed_t}{n.speed_{t-1}} \times n.build\_num, & \frac{n.speed_t}{n.speed_{t-1}} \geq 1 \\ \gamma \times n.build\_num, & \frac{n.speed_t}{n.speed_{t-1}} < 1 \end{cases} \tag{1}$$

Among them, n refers to a specific node, $n.build\_num$ represents the size of the current node. $n.speed_t$ represents the accumulation rate at time t, indicating the insertion rate of new keys into a node at that specific time. This rate is determined by a probabilistic model, as described in Section 3.3. In Equation (1), a higher speed leads to a more significant expansion rate in the current operation, meaning that more gaps will be reserved compared to the previous expansion operation. The expansion factor γ is defined as follows:

$$\gamma = \begin{cases} \theta, & n.build\_num \geq 1M \\ 2\theta, & n.build\_num \geq 100K \\ 5\theta, & n.build\_num < 100K \end{cases} \tag{2}$$

Equation (2) reveals that nodes of varying sizes should have different expansion factors. Equation (1) demonstrates that smaller nodes ($n.build\_num$), need a more significant expansion factor to achieve adequate expansion. E.g., if the number of slots in two nodes is 4 and 8, respectively, both nodes would need to expand by 32 units, requiring expansion factors of 8 and 4, respectively. The

Algorithm 3 SALI Insertion Evolving

Require: Sequence $keys\{K_1, \cdots, K_n\}$ in a $Node$ and its $Subtree$.

```
function Insert_Node_Evolving(keys)
    segs.model ← FMCD(keys, n.expand_size)
    father_slot link segs.model
end function
```

Algorithm 4 SALI Lookup Evolving

Require: Sequence $keys\{K_1, \cdots, K_n\}$ in a $Node$ and its $Subtree$.

```
1:  function Hot_Lookup_Node_Evolving(keys)
2:      gap_array ← calculate_gap(keys)
3:      gap_array ← Top − k(gap_array)
4:      for (gap_array.size > 0) do
5:          segment.key ← split(gap_array)
6:          segment.linear ← approximation_alg(segment.key)
7:          segs ← segs.append(segment.key, segment.linear)
8:      end for
9:      segs.line.slope ← segs.line.slope × n.expand_size
10:     segs.model ← insert(segment.key, segs.linear)
11:     father_slot link segs.model
12: end function
```

expansion base factor, denoted by θ, can be dynamically adjusted based on varying workloads. For our evaluation, we set θ to its default value of 1.

3.2.2 The lookup triggers the evolving operation. We have developed an evolving strategy for hot read nodes to enhance concurrent read performance under skewed workloads further. Note that if the workload is uniform, SALI can either choose to disable this evolving function or treat every node as a hot read node. As depicted in Figure 7(b), we have designed a flat structure for hot reads nodes and their subtrees. This structure flattens the nodes and promotes their levels as much as possible. Unlike the initial state where a single linear segment is linked under one slot, multiple segments can be linked under one slot after evolving. This flattening strategy reduces the tree height of the local hot structure. Furthermore, SALI can use SIMD instructions during lookup to quickly find which node the target key belongs to in the same layer.

We have developed a method to reconstruct structure, which has the effect of flattening the node and its subtrees (Algorithm 4). First, we sort all the nodes' keys, calculate the gap between two adjacent keys, and select the $top - k$ gap (Algorithm 4, line 2). Then, we split into $k - 1$ segments based on these gaps and generate a linear model using the least squares algorithm (Algorithm 4, lines 3-8). According to Equation (1), we expand the slope of the linear model by a corresponding multiple to expand the corresponding space (Algorithm 4, line 9). Reserving the gap enables the CDF of the stored data to fit more easily on a line and improve lookup performance. Finally, we calculate the positions of all keys and insert them using the linear model after the slope expands. If there are still conflicts, we handle them similarly to SALI's insertion conflict (Algorithm 4, line 10).

3.2.3 Identify the cold node and trigger evolving operation. We developed a cold node-compression evolving strategy to optimize space usage in SALI under skewed workloads. In addition to initially creating the SALI index structure, we added a cooling pool space as illustrated in Figure 8. During SALI construction or each evolving operation, each node in the index has a 10% probability of being chosen for inclusion in the cooling pool. When a node undergoes an evolving operation, that node,

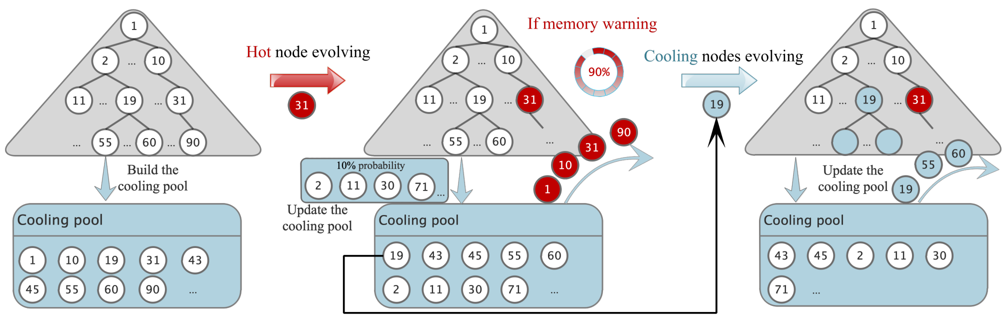


Fig. 8. The framework for identifying cold nodes.

Algorithm 5 SALI Cold Node Evolving

Require: Sequence $keys\{K_1, \cdots, K_n\}$ in a $Node$ and its $Subtree$.

```
1: function Cooling_Node_Evolving(keys)
2:     segs.model ← PLA_algorithm(keys)
3:     father_slot link segs.model
4: end function
```

its subtrees, and all nodes above it are removed from the cooling pool. At this stage, the nodes that remain in the cooling pool are considered temporarily cold. We took inspiration for cold node design from [17].

Once each evolving operation finishes, SALI checks whether the user-acceptable index size upper limit has been exceeded. If it has, SALI selects the earliest-added node in the cooling pool for the compress operation and deletes it from the cooling pool until the space is reduced to meet the user-acceptable index size.

For cold nodes, we implemented a space compression strategy. As depicted in Figure 7(c), we cancel the reserved gaps to save space for cold nodes and their subtrees. We use the PLA algorithm in PGM [5] to linearly approximate all keys in a cold node and generate the corresponding segment (Algorithm 5).

3.3 Probability Model

In order to ensure optimum performance, it is imperative that learned indexes monitor degradation statistics to initiate adjusting when necessary. Unfortunately, existing high-contention statistics techniques severely limit the scalability of learned indexes. Moreover, the implementation of the complete adjustment strategy, i.e., evolving presented in Section 3.2, demands additional statistics, resulting in intolerable overhead in a concurrent scenario.

To address this issue, we propose a probability-based strategy that employs a lightweight approach to maintain various statistics in SALI to trigger evolving operations at a minimal cost.

Note that the fundamental concept behind the probability models is to leverage probabilities in simulating the accumulation of information. For example, when simulating the cumulative number of inserted keys within a specified timeframe, we design a probability model based on the insertion rate and insertion time. Furthermore, the geometric distribution can be utilized to simulate the accumulation of information such as insertion conflicts.

3.3.1 Probability model for triggering insert evolution. Most retrains are triggered by the deterioration of learned indexes caused by insert operations. However, from an overall perspective of index performance, adjustments should be considered based on whether the local structure, following its adjustment, will continue to see the high-frequency insertion of new keys. Such consideration can make the adjustment operation more advantageous, which achieves high amortized performance benefits.

Therefore, two conditions need to be considered to trigger insertion evolution: 1) The assessment of the frequency of new key insertions in a node and its subtree is crucial in determining whether an adequate number of keys are being inserted. The node's performance gains are higher after evolving if the frequency of new key insertions is high. 2) The escalation of conflicts within a node alongside the gradual increase in the number of newly inserted keys represents a critical aspect as it can be used to indicate the deterioration of the index. Identifying deteriorating nodes is crucial, as only evolving such nodes will significantly improve performance.

Note that when a node satisfies only condition 1) and not condition 2), evolving is unnecessary because the insertion performance remains satisfactory. When a node satisfies only condition 2) and not condition 1), the amortized performance benefit is low, and the evolving operation entails overhead costs. Therefore, satisfying both conditions simultaneously is a prerequisite for triggering the insert evolving operation. In the subsequent section, we provide a comprehensive analysis of the above two doctrinal conditions.

Condition (1): the node accommodates a sufficient number of newly inserted keys. To determine if this condition is met, we need to satisfy the following equation:

$$\frac{n.current\_num}{n.build\_num} \geq \beta \tag{3}$$

$n.current\_num$ refers to the number of keys contained in the node at the end of the current insertion operation. $n.build\_num$ indicates the number of keys in the node when the last "evolving" operation was performed. The tolerance coefficient β specifies the maximum amount of data that can be inserted into the node before it needs to be adjusted. As a general guideline, we set $\beta = 2$.

Since each insertion thread must maintain the cumulative variable $current\_num$, conflicts may arise. To address this issue, we propose a lightweight probability model.

First, we multiply the insertion rate by the timestamp difference to get the total amount of inserted new keys during this period and put it into Equation (3) to get:

$$\frac{[n.speed_t \times (n.current\_time - n.build\_time)] + n.build\_num}{n.build\_num} \geq \beta \tag{4}$$

The variable $n.build\_time$ represents the timestamp corresponding to state $n.build\_num$, while $n.current\_time$ represents the current timestamp. The estimated insertion rate, denoted as $n.speed_t$[2], is calculated using Equation (5), which takes the quotient of the total number of insertions and the difference in timestamp from the previous period. Therefore, we can estimate the value of $n.current\_num$ using $n.speed_t \times (n.current\_time - n.build\_time)] + n.build\_num$ at time t.

$$\frac{n.current\_num - n.build\_num}{n.current\_time - n.build\_time} = n.speed_{t+1} \tag{5}$$

Additionally, we define the cumulative probability within a node as P_{acc} (see Equation (6)), as obtained through the transformation of Equation (4). When $P_{acc} = 1$, condition (1) is met. When $P_{acc} < 1$, we determine whether the evolving adjustment is necessary based on a Bernoulli experiment. If the experiment is successful, condition (1) is met; otherwise, it is not met.

[2]We assign a specific value to $speed_1$.

$$P_{acc} = \frac{[n.speed_t \times (n.current\_time - n.build\_time)]}{(\beta - 1) \times n.build\_num} \tag{6}$$

Note that the calculation of Equation (5) may result in *n.speed* being zero. In such cases, the cumulative probability computed by Equation (6) will always be zero, preventing any further changes in *n.speed*. To resolve this issue, we introduce a reconciling variable ϵ in the numerator of Equation (6). Expressly, we set $\epsilon = path\_size/1000$, where $path\_size$ denotes the path length from the root node to the current node. The final Equation of the cumulative probability model, Equation (7), determines whether the node accommodates a sufficient number of newly inserted keys.

$$P_{acc} = \frac{[n.speed_t \times (n.current\_time - n.build\_time)] + \epsilon}{(\beta - 1) \times n.build\_num} \tag{7}$$

Condition (2): the node accommodates an adequate number of conflicts. We calculate the ratio of insertion conflicts to the total number of insertions:

$$\frac{n.conflict\_num}{n.current\_num - n.build\_num} \geq \alpha \tag{8}$$

The variable $n.conflict\_num$ denotes the total number of conflicts in the node resulting from the insertions between the last evolving operation and the current state. Meanwhile, the conflict tolerance coefficient is denoted by α, which we typically set to 0.1 as a rule of thumb. Similar to Condition (1), in a concurrent scenario, we need to develop a probability model to estimate the number of conflicts to avoid blocking threads according to Equation (8).

For an evolving operation to occur, the node must have sufficient newly inserted keys. Therefore, we use $(\beta - 1) \times n.bulid\_num$ to estimate the number of new insertions according to Equation (3) when there are enough conflicts to cause evolution:

$$(n.current\_num - n.build\_num) \approx (\beta - 1) \times n.bulid\_num \tag{9}$$

By substituting Equation (9) into Equation (8), we obtain Equation (10):

$$n.conflict\_num \geq \alpha \times (\beta - 1) \times n.bulid\_num \tag{10}$$

When an insertion causes a conflict, we set the conflict adjustment probability to $P_{conflict}$. Using the expectation of the geometric distribution, we can estimate that the expected number of conflicts that trigger evolving after the conflict is $\frac{1}{P_{conflict}}$, i.e., $\frac{1}{P_{conflict}} \approx n.conflict\_num$. Thus, whenever a conflict occurs in a node, we trigger the probability model specified in Equation (11) to determine if the model deteriorates and requires evolving.

$$P_{conflict} = \frac{1}{\alpha \times (\beta - 1) \times n.bulid\_num} \tag{11}$$

Application in SALI: In SALI, we only compute probabilities when a conflict occurs to minimize overhead. We determine whether $P_{conflict}$ is triggered; if so, we proceed to determine whether P_{acc} is also triggered. If both conditions are met, the evolving operation is necessary to adjust the insertion structure of SALI.

3.3.2 Probability model for triggering lookup evolution. We define the probability that a target node is identified as a **h**ot read node due to a **l**ookup operation, denoted as P_{hl}, that is a hyperparameter that can be set to an appropriate value. Whenever the lookup operation encounters a node, we can check whether P_{hl} is triggered for that node. If P_{hl} is triggered, we consider the node and its subtree as a hot lookup structure.

In addition to the probability P_{hl}, the following conditions for setting the read trigger probability also need to be considered:

(1) The evolving operation has not been triggered by lookup operations on the node for a prolonged period of time.

(2) The rate at which the node accumulates data ($n.speed_t$) through insertions is not slow.

For condition (1), if the last evolve operation of a node was triggered by a hot lookup, it means that no insert operation has triggered the node to evolve since then, i.e., the node has not severely deteriorated, and the number of new insertion keys are likely to be few. In this case, we can adjust P_{hl} to a smaller value to prevent frequent evolving, i.e., $P_{hl} = P_{hl} \times \lambda$, where λ is a penalty coefficient.

For condition (2), in addition to P_{hl}, we introduce the probability P_{acc}, as defined in Equation (7). If a large number of new keys are inserted since the last evolving operation, it suggests that a new round of evolving operations may be necessary.

Application in SALI: We generate a $skip\_counter$ in each lookup thread-local, which maintains the number of lookup operations. Upon execution of a lookup operation, the $skip\_counter$ is incremented by 1. After every 10 lookup operations, a Bernoulli experiment is conducted to determine whether P_{hl} is triggered. If P_{hl} is triggered, the system verifies whether P_{acc} is also triggered. If P_{acc} is triggered, SALI proceeds with the evolving operation.

4 EVALUATION

This section conducts a comprehensive evaluation of SALI. Section 4.1 describes the experimental setup. Section 4.2 compares SALI's performance with that of several state-of-the-art concurrent learned indexes and traditional indexes using various datasets and thread counts. Section 4.3 evaluates SALI under skewed workloads. Finally, Section 4.4 conducts an ablation study on SALI.

4.1 Experimental Setup

All experiments are conducted on a two-socket server with two 16-core Intel Xeon Gold 6242 @2.80GHz CPUs (hyper-threading to 64 threads) and 384GB of DRAM. We implemented SALI with ~4k LOC of C++.

4.1.1 Baselines. We benchmarked SALI against six baselines. 1) Masstree [28], a hybrid index structure of B+Tree and Radix Tree; 2) ART-OLC [19], an exemplary concurrency implementation of the Adaptive Radix Tree (ART) [18]. 3) ALEX+ [38], an exemplary concurrency implementation of the ALEX [3]. 4) LIPP+ [38], a concurrency implementation of the LIPP [39]. 5) XIndex [36], a first attempt to design a concurrent learned index. 6) FINEdex [21], a fine-grained updated concurrent learned index.

4.1.2 Datasets. We selected several real datasets from SOSD [29] and GRE [38] benchmarks.

- COVID: Tweet ID with tag COVID-19 [24] (Uniformly sampled).
- FACE: Facebook user ID [11].
- LIBIO: Repository ID from libraries.io [38].
- OSM: OpenStreetMap locations [29] (Uniformly sampled).
- GENOME: Pairs of locations on human chromosomes [34].

Note that according to paper [38], the OSM and GENOME datasets are considered to be of "hard" difficulty for learned indexes, as fitting a Cumulative Distribution Function (CDF) on these datasets is challenging. Relatively, fitting remaining datasets with a CDF is comparably easier.

4.1.3 Workloads. We design workloads to generate requests using the aforementioned datasets. To achieve this, we randomly shuffle all 200 million keys for each dataset and issue insert and lookup requests based on the following ratios:

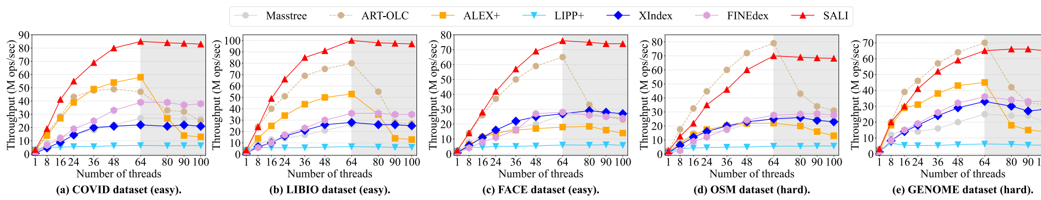


Fig. 9. The indexes scalability on write-only workloads. The grey area indicates that the threads number exceeds the maximum logical cores number. Extended plots with all evaluations are available here: [9]

• Read-Only: Load all 200M keys and randomly search 800M.

• Read-Intensive (20% insert): Load 100M random keys and perform 80% search & 20% insert, i.e., insert all the remaining keys.

• Balanced (50% insert): Load 100M random keys and perform 50% search & 50% insert, i.e., insert all the remaining keys.

• Write-Only: Load 100M keys and insert 100M keys.

• Hot-read-A (100% Read): Load 200M keys. Select 1/10 of these 200M (20M) as hot read keys and execute five rounds of read operations on these 20M keys, i.e., 100M read operations.

• Hot-read-B (16% insert): Perform an additional insert operation of 20M keys based on the keys from Hot-read-A.

• Hot-write (100% Insert): Randomly select one-eighth consecutive data in the 200M data as the hot insert, and insert it after loading the remaining data.

We repeated 10 and 5 experiments for Hot-write and other workloads, respectively, excluding the lowest and the highest measure, and reported the average of the results. Between each measurement of experiments, we wiped caches and re-loaded the data to avoid intermediate results.

4.2 Overall Results

This section evaluates the SALI's overall performance against the SOTA indexes that support concurrency. In this experiment, uniform workloads were used. To ensure fairness, SALI turns off the judgment and evolution modules temporarily of hot read nodes but reserves the evolving operations triggered by the insertion on SALI, as all indexes require maintaining statistical information and performing retraining operations when inserting data.

4.2.1 Write-only workloads. Figure 9 shows the concurrent performance evaluation of various learned indexes when executing the write-only workload on different datasets. The triggered retraining is directly executed in the foreground by the thread responsible for triggering. These threads are represented by the numbers specified on the x-axis. All indexes except LIPP+ benefit from increasing threads, but performance drops when the number of threads exceeds the logical thread count.

In easy datasets, namely COVID, LIBIO, and FACE, SALI performs better in terms of throughput overall. Compared to the learned index ALEX+ and traditional index ART-OLC, SALI exhibits the best scalability. ALEX+ and ART-OLC exhibit a sharp performance drop when the number of threads exceeds 60, whereas SALI maintains a high and stable performance. In the COVID dataset, SALI outperforms the best two baselines, i.e., ALEX+ and ART-OLC, by up to 47% and 73% at the highest solution, respectively, and the advantage continues to expand beyond 60 threads.

In hard datasets, SALI outperforms other learned indexes by up to a factor of 2.5x to 10x, as SALI can accurately fit complex CDF, so that its performance can match ART-OLC and still maintain

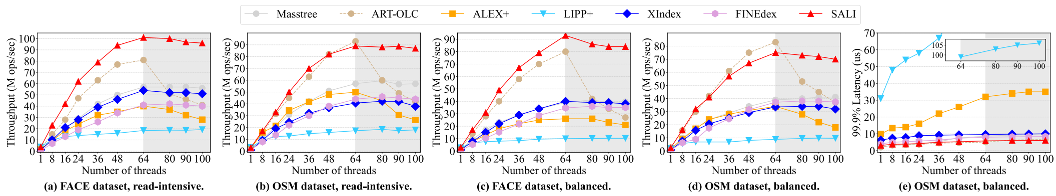


Fig. 11. The indexes scalability on read-write workloads. Extended plots with all evaluations are available here: [9]

performance even exceeding the number of logical threads, where, in contrast, the performance of ART-OLC sharply declined.

It is noteworthy that ALEX+ and ART-OLC exhibit significant performance degradation when threads exceed 60. This issue is due to the coarse-grained lock, resulting in severe thread blocking when insertion (as discussed in Section 2.2).

Additionally, the buffer-based insert strategy used by XIndex and FINEdex results in a large lookup error, and frequent “last mile” searches result in poor scalability. LIPP+ employs a high-contend method for all nodes to statistic information, leading to severe blocking of insertion threads and cache-line ping-pong, which complete loss of scalability in a concurrent scenario.

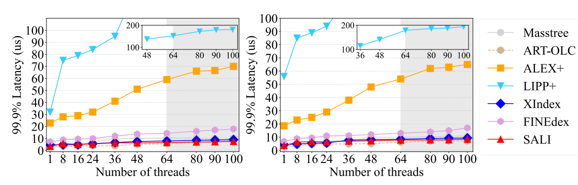


Fig. 10. The latency of indexes on write-only workloads.

Figure 10 shows the 99.9% tail latency on FACE and OSM datasets. As threads increase, ALEX+ and LIPP+ exhibit significant increases in tail latency. ALEX+ has coarse-grained locks, which cause thread blocking during insertion and retraining operations, resulting in a significant increase in tail latency. LIPP+ requires joint maintenance of statistical information by different threads, leading to an increase in thread blocking and a significant increase in tail latency. In contrast, the other indexes do not show noticeable increases, and SALI maintains the lowest in most settings.

Insight 1: Concurrent insertion often faces three challenges: a) high-contend statistics maintenance causing thread blocking; b) coarse-grained write locks leading to thread blocking; c) write amplification and lookup errors causing memory bandwidth exhaustion. SALI efficiently addresses them, leading to exceptional scalability, especially with hard datasets and a high number of threads.

4.2.2 Read-write workloads. The performance of different indexes under read-intensive and balanced workloads is shown in Figure 11.

Figure 11(a,b) exhibit that SALI outperforms other indexes under read-intensive workloads, especially in the easy dataset. Compared to ART-OLC and ALEX+, SALI improves the performance by up to 37% and 55%, respectively, under 60 threads.

Figure 11(c,d) illustrate the performance under the balanced workload. SALI maintains high scalability. Other indexes' performance also exhibits similar to the read-intensive workloads.

Nevertheless, LIPP+ remains unscalable. And when the number of threads exceeds 60, the performance of ALEX+ and ART-OLC degrades significantly. The aforementioned indicates that

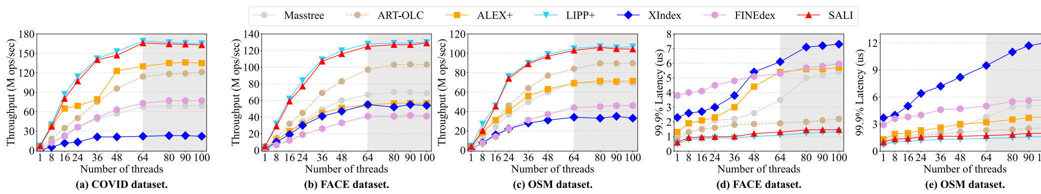


Fig. 12. The indexes scalability on read-only workloads. Extended plots with all evaluations are available here: [9]

even with a low proportion of write operations (20% insert), the indexes experience a bottleneck under ultra-high threading conditions.

In Figure 11(e), the tail latency of the index under the balanced workload is illustrated. Similar to Figure 10, ALEX+ and LIPP+ exhibit higher tail latencies compared to the other indexes. However, as the read ratio increases, the tail latency of both indexes decreases. Notably, SALI maintains consistently low tail latency throughout.

Insight 2: SALI exhibits outstanding scalability under workloads that involve insertion operations, even under hard datasets. Conversely, other indexes face scalability bottlenecks, even with a low proportion of insert operations.

4.2.3 Read-only workloads. Figure 12 presents the evaluation of the read-only workload. SALI and LIPP+ outperform other indexes in both easy and hard datasets, as they adopt a model-based insert + chain structure, which enables accurate lookups.

LIPP+ does not require high-contend maintaining statistics in read-only scenarios, while SALI needs to identify hot and cold nodes, which adds a slight overhead. Therefore, SALI's performance is slightly lower than that of LIPP+.

ALEX+ exhibits poor lookup performance due to frequent "last mile" lookups. XIndex and FINEdex perform unsatisfied in general due to serious lookup errors. ART-OLC does not have high superiority with the read-only workload due to its higher tree height in comparison to the learned index.

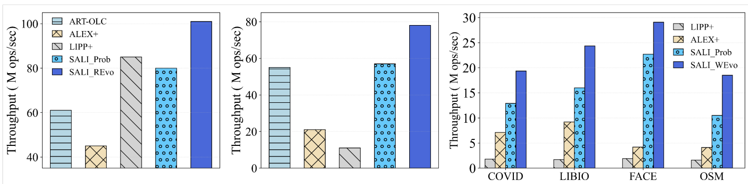
Figure 12(d,e) depict the tail latency on FACE and OSM datasets. XIndex and FINEdex perform worse due to the severe lookup errors. In contrast, SALI and LIPP+ have lower tail latency than ALEX+, as they do not suffer from any lookup errors.

Insight 3: All indexes benefit from hyperthreading under the read-only. Among them, SALI and LIPP+ deliver the best performance due to accurate lookup capability, which is essential for improving query performance.

4.3 Evolving Evaluation

4.3.1 Evolving triggered by hot read. Figure 13(a,b) compare the performance of the learned index with 48 threads in two hot-read workloads on the OSM dataset. SALI _Prob refers to SALI with only the probability model, while SALI _REvo uses an evolving strategy triggered by read. The figure exhibits that SALI _REvo outperforms SALI _Prob by 27% and 36% in the two hot-read workloads, respectively. In the Hot-read-A workload, SALI _REvo has surpassed the performance of LIPP+.

Moreover, SALI _REvo performs exceptionally well in OSM and GENOME due to the reduction in subtree height of hot read nodes, effectively increasing read performance. Table 2 (in Figure 13) shows that the SALI _Prob's depth on the two hard datasets is up to 5 and 7, respectively. Therefore, the evolution of the hot node will flatten the node to optimize read performance. However, under



(a) Hot-read-A triggers on OSM. (b) Hot-read-B triggers evolving on OSM. (c) The Hot-write triggers evolving.

Table 2: The Depth of SALI_Prob

Datasets	max_depth	avg._depth
GENOME	5	2.07
OSM	7	2.28
LIBIO	2	1.21

Table 3: Compress the Cooling Node Space

Datasets	OSM	GENOME
SALI size	52,293 MB	51,016 MB
Comp. SALI size	32,311 MB	36,166 MB
Comp. rate	37%	31%

Fig. 13. The performance of the evolving strategy.

the easy datasets, such as LIBIO, the average depth of SALI _Prob is only 1.2, so there exists a negligible improvement when using SALI _REvo as there is not enough depth to reduce.

Note that as SALI adopts the LIPP structure, the majority of keys are stored in the root node and upper levels, while the deeper subtree contains fewer keys. Therefore, during the read evolution, we found that connecting two nodes in one slot yields the best performance while connecting more nodes would increase the overhead of indexing fewer keys, rendering the evolution strategy ineffective. In this case, we directly determine which of the two nodes the target key belongs to based on the maximum value of the nodes, which is more efficient than the SIMD approach. In Section 5, we further discuss the applicability and limitations of read evolving.

Insight 4: In skewed workloads, the hot-read evolving can significantly improve read performance when the subtree of the hot-read node is deep. Flattened tree structures under easy datasets do not require evolving.

4.3.2 Evolving triggered by the hot insert. Figure 13(c) presents the evaluation of Hot-write workloads with 48 threads. SALI _WEvo includes both the probability model and evolving strategy triggered by insertion. SALI _Prob only consists of the probability model and an adjustment strategy equipped with a fixed $n.speed_t$ (as described in Equation (1)), similar to the adjustment method used by existing learned indexes, i.e., the expansion coefficient is fixed to expand the corresponding node during adjustment.

SALI _WEvo outperforms SALI _Prob by 32% to 80%, indicating that the evolving strategy significantly impacts hot write nodes. However, the performance of the index structure during the hot write workload is not as good as that of the uniform workload. This is because hot writes cause the local structure of the index to deteriorate continuously and require frequent retraining operations. Additionally, frequent local writes in ALEX+ and ART-OLC index structures with coarse-grained write locks increase thread blocking and adversely affect insertion performance.

Nonetheless, SALI _WEvo can adaptively evolve the hot node based on the insertion rate, i.e., $n.speed_t$ (see Section 3.2). When the insertion rate becomes faster, more slots are reserved to ensure excellent insertion performance through the expand operation (see Equation (1)), which significantly reduces the number of retraining operations and improves overall performance.

Insight 5: Skewed workloads with hot writes often lead to significant performance deterioration. SALI can effectively analyze and process hot insertion by dynamically reserving more gaps to maintain a high throughput.

4.3.3 Evolving triggered by cold nodes. Figure 14 illustrates the size of learned indexes on OSM. Notably, the size of learned indexes on OSM is about 1.5 times larger than that of the easy datasets. In particular, Figure 14(a) shows the internal structure size of the indexes, indicating that SALI incurs the smallest space overhead.

Note that the space overhead of the key-value pair could overshadow that of the index. Figure 14(b) presents the sum of the index's internal structure, the gap, and the key-value pair's size (no

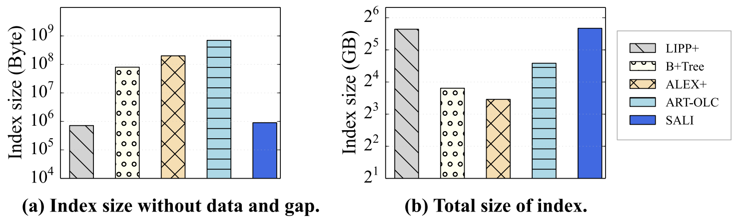


Fig. 14. The size of indexes.

compression). The figure shows that the overall space cost of SALI is higher than that of other index structures due to SALI reserving gaps for the key to be inserted.

However, when the workload is skewed, the gap utilization in cold nodes is low, and thus, it can be compressed to reduce space overhead. Table 3 in Figure 13 demonstrates the size of SALI before and after compressing cold nodes under the Hot-read-A workload. The results show that the compression scheme in SALI can reduce space cost considerably, with compression ratios of 31% and 37% for cold nodes on OSM and GENOME, respectively, evaluated using the Hot-write workload.

Insight 6: Compression scheme in SALI can save space dramatically while ensuring that all gaps are reserved at the hot node, i.e., the gaps are reserved at a more accurate and efficient location than other learned indexes.

4.4 Ablation Study

Anneser et al. [1] proposed a low-cost sampling method to identify hot and cold data, compress cold data, and expand hot data based on traditional indexes. However, due to the need to consider the linear fitting CDF problem, the compression and expansion operations proposed by Anneser et al. cannot be directly applied to the learned index. Nonetheless, this low-cost sampling method can be used in SALI for comparison with our proposed probabilistic framework and high-contend statistics maintenance.

To compare these methods, we maintain statistical information using three approaches, as shown in Figure 15: SALI _Stat. maintains statistical information with the high-contend approach that is employed in many state-of-the-art learned index structures, where every new data insertion triggers a *num*.++ operation in all nodes. SALI _Samp. maintains statistical information using the sampling method proposed by Anneser et al., where the *num*.++ operation is triggered for every ten inserted data. SALI _Prob. maintains statistical information using the probability model proposed in our paper. The results show that SALI _Samp. improves scalability compared to SALI _Stat. Furthermore, the decentralized probability-based model SALI _Prob. is more scalable than SALI _Samp., achieving up to 35% higher performance at 60 threads.

Insight 7: Probability-based models exhibit excellent performance in concurrent scenarios owing to lightweight statistical information maintenance. In contrast, any centralized maintenance of statistics can result in performance loss.

5 DISCUSSION

5.1 Generalizability and Applicability

We will describe the two main ideas in the SALI framework, the node-evolving strategies (Section 3.2) and the probability model for node statistics (Section 3.3), as general approaches that can potentially be applied to a wide range of learned indexes.

1) What we want to emphasize is that in any application scenario, including but not limited to concurrent settings, if maintaining statistics information globally becomes a performance bottleneck

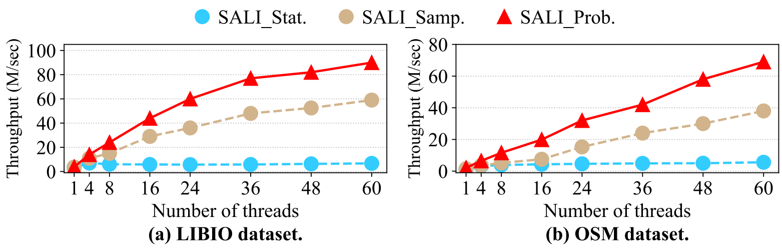


Fig. 15. Comparison of different maintaining statistics methods.

for the index, adopting a probability-based lightweight statistical maintenance approach can help enhance performance. For example, Lan et al. [16] evaluated the performance of learned index approaches on disk and mentioned that the maintenance statistics overhead in both ALEX and LIPP can hurt overall performance because fetching more blocks is required during statistics maintenance. Therefore, our lightweight probability models can address this bottleneck. Anneser et al. [1] still use a global approach to maintain hot-node information in traditional indexes. Although they designed a sampling method to reduce overhead, our proposed probability models have minimal overhead (see Figure 15). Therefore, using probability models for maintaining hot-node information has the potential to improve performance. Li et al. [23] designed a new model-based learned index framework named DILI, which combines the structures of ALEX and LIPP. The probability models can also serve this framework to improve its scalability.

2) The node-evolving strategies can be applied to different learned indexes. The read evolving can reduce the height of the hot sub-tree, improving read performance. The write evolving can allocate more space for write-hot nodes in both buffer-based and model-based learned indexes, thereby enhancing insertion performance. The cold node-compression strategy can save space overhead in model-based learned indexes. Moreover, if applied to buffer-based learned indexes, new compression algorithms need to be designed to reduce space overhead, which would be an interesting research direction. Furthermore, Numerous exceptional compression works demonstrate that data compression is a promising research direction [42, 43]. Theoretically, the SALI framework supports the implementation of any excellent compression algorithm. Improving upon current compression strategies is a key focus area for future.

3) The current experimental results show that the read-evolving strategy of SALI is effective only when a slot connects two nodes in the flattened structure. As described in Section 4.3.1, the reason is that the read performance improvement from flattening cannot cover the overhead of using SIMD to accelerate the node lookup process. This is because the flattened hot subtree contains a relatively small number of keys, and two nodes are sufficient to meet the flattening requirement. The number of keys in the subtree depends on the size of the experimental dataset and the LIPP structure used in SALI. However, we believe that in scenarios with lookup large data volumes, such as Hybrid Transaction/Analytical Processing (HTAP), the read-evolving strategy of the index requires more nodes to be flattened in the same slot and accelerated by SIMD for the lookup process, resulting in performance benefits. In future work, we plan to design an automatic mechanism to determine the optimal number of nodes to be flattened based on the current data volume and distribution.

5.2 Limitation

We have identified several limitations in this study:

1) The read-evolving strategy offers significant advantages in scenarios with complex data distributions. Insight 4 in Section 4.3.1 states that flattened tree structures under easy datasets do

not evolve. In such cases, enabling the read-evolving strategy would introduce additional overhead to determine if a target node is a hot node, which would compromise the read performance of the SALI. The overhead of node determination is detailed in additional experiments in the appendix [9]. Therefore, we have encapsulated read-evolving as a toggle switch to be enabled in scenarios where it is needed. However, we acknowledge that determining the benefit of enabling read-evolving in a specific data distribution can be challenging. This will be a focus of our future work.

2) Currently, SALI does not support duplicate keys. The reason for this is that SALI, being based on the LIPP structure, does not currently support the insertion of duplicate data. However, Wu et al. [39] have suggested that it is relatively straightforward for indexes to accommodate duplicate keys, such as by maintaining a pointer to an overflow list. As part of our future work, we plan to focus on implementing the insertion of duplicate data.

6 RELATED WORK

In 2018, Kraska et al.[7, 15, 29] introduced a learned index called RMI, which sparked a new wave of index design considerations. While the lookup performance was satisfactory, the first-generation learned indexes, such as RMI and RS[14], did not support updates. To address this limitation, Galakat et al.[8] designed FITing-tree, an updatable learned index. Ferragina et al. [5] improved the construction algorithm of FITing-tree and introduced the PGM index, which optimized the number of linear models generated while setting the maximum error and used an insertion strategy similar to LSM-Tree [33] to ensure worst-case insertion performance. However, FITing-tree and PGM suffered from significant lookup errors and had no better insertion performance than traditional indexes due to their buffer-based insertion strategy. In response, Ding et al.[3] designed ALEX, which raised the insertion performance of learned indexes to a new level by using a model-based insertion strategy. However, ALEX had prediction errors and coarse-grained write lock due to its "shift" strategy to resolve conflicts. Wu et al.[39] designed LIPP, another learned index with an error-free model-based insertion strategy. However, LIPP's high-contend statistics maintenance approach in every node hindered scalability. TONE [45] mitigates tail latency by dynamically allocating a secondary array to accommodate data, building upon the foundation of ALEX.

XIndex [36, 37] was the first to implement a concurrent update-capable learned index. However, frequent "last mile" queries made it less competitive. FINEdex [21] improved the concurrency performance by using a flattened structure to avoid coarse-grained locking. Wongkham et al. [38] implemented concurrent structures for ALEX and LIPP, named ALEX+ and LIPP+, respectively. Experimental showed that ALEX+ outperformed LIPP+.

Learned indexes have also inspired new design ideas for other application scenarios [2, 4, 13, 15, 22, 27, 30, 41, 44]. For instance, Lu et al. [25] developed APEX, a learned index based on NVM. Ma et al. [26] designed FILM, a learned index that supports larger-than-memory databases. Wu et al. [40] introduced NFL, a learned index that changes the CDF of stored data through deep learning, making it easier to approximate. Nathan et al. [32] focused on multi-dimensional in-memory learned indexes. However, these works are beyond the scope of our discussion.

7 CONCLUSION

We have developed SALI, a highly scalable learned index framework. In SALI, we have designed a probability-based framework for monitoring the "degradation signals" of the index and identifying hot/cold nodes in a decentralized manner, thereby eliminating thread blocking and improving the index's scalability in a concurrent scenario. Since the statistical overhead is negligible, the probability framework provides the necessary conditions for the index to evolve separately toward hot and cold data. Furthermore, we have devised evolution strategies that allow SALI to develop into better-performing local structures for hot and cold nodes independently. The experimental results demonstrate that SALI built upon the $Mod.{+}C$ structure offers significantly better scalability than state-of-the-art learned indexes, and the evolution strategies can increase read and write performance by at least 25% and 30%, respectively.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (No. 61972402 and 61972275). The corresponding author is Yunpeng Chai (ypchai@ruc.edu.cn).

REFERENCES

[1] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data*. 1626–1639.

[2] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 155–171.

[3] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[4] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.

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

[6] K Fraser. 2004. Practical lock-freedom (Doctoral dissertation, University of Cambridge). (2004).

[7] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. A-Tree: A Bounded Approximate Index Structure. (2018).

[8] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.

[9] Jiake Ge. 2023. Appendix. https://github.com/YunWorkshop/SALI/blob/main/SALI_appendix.pdf

[10] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 315–327.

[11] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. 2010. Walking in facebook: A case study of unbiased sampling of osns. In *2010 Proceedings IEEE Infocom*. IEEE, 1–9.

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

[13] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Learned Systems* (2019).

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[16] Hai Lan, Zhifeng Bao, J Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS-From Evaluations to Design Choices. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–22.

[17] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.

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

[19] V. Leis, F. Scheibner, Alfons Heinrich Kemper, and T. Neumann. 2016. The ART of practical synchronization. In *the 12th International Workshop*.

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

[21] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[22] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.

[23] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *arXiv preprint arXiv:2304.08817* (2023).

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

[25] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proceedings of the VLDB Endowment* 15, 3 (2021), 597–610.

[26] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A Fully Learned Index for Larger-Than-Memory Databases. *Proceedings of the VLDB Endowment* 16, 3 (2022), 561–573.

[27] Marcel Maltry and Jens Dittrich. 2022. A critical analysis of recursive model indexes. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1079–1091.

[28] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[29] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* (2020).

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[31] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-copy update. In *AUUG Conference Proceedings*. AUUG, Inc., 175.

[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 985–1000.

[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[34] Suhas SP 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, et al. 2014. A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell* 159, 7 (2014), 1665–1680.

[35] Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Efficient Concurrent Range Queries in B+-trees using RCU-HTM. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 571–573.

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

[37] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The concurrent learned indexes for multicore data storage. *ACM Transactions on Storage (TOS)* 18, 1 (2022), 1–35.

[38] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proceedings of the VLDB Endowment* (2022).

[39] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.

[40] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: robust learned index via distribution transformation. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2188–2200.

[41] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. 2023. LIFOSS: a learned index scheme for streaming scenarios. *World Wide Web* 26, 1 (2023), 501–518.

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

[43] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1601–1615.

[44] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: a cache-aware learned index with a cost-based construction algorithm. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2679–2691.

[45] Yong Zhang, Xinran Xiong, and Oana Balmau. 2022. TONE: cutting tail-latency in learned indexes. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. 16–23.

Received April 2023; revised July 2023; accepted August 2023