A Fast Learned Key-Value Store for Concurrent and Distributed Systems

Pengfei Li, Yu Hua, *Senior Member, IEEE*, Jingnan Jia, and Pengfei Zuo

***Abstract*—Efficient key-value (KV) store becomes important for concurrent and distributed systems to deliver high performance. The promising learned indexes leverage deep-learning models to complement existing KV stores and obtain significant performance improvements. However, existing schemes show limited scalability in concurrent systems due to containing high dependency among data. The practical system performance decreases when inserting a large amount of new data due to triggering frequent and inefficient retraining operations. Moreover, existing learned indexes become inefficient in distributed systems, since different machines incur high overheads to guarantee the data consistency when the index structures dynamically change. To address these problems in concurrent and distributed systems, we propose a fine-grained learned index scheme with high scalability, called FineStore, which constructs independent models with a flattened data structure under the trained data array to concurrently process the requests with low overheads. FineStore processes the new requests in-place with the support of non-blocking retraining, hence adapting to the new distributions without blocking the systems. In the distributed systems, different machines efficiently leverage the extended RCU barrier to guarantee the data consistency. We evaluate FineStore via YCSB and real-world datasets, and extensive experimental results demonstrate that FineStore improves the performance respectively by up to 1.8× and 2.5× than state-of-the-art XIndex and Masstree. We have released the open-source codes of FineStore for public use in GitHub.**

***Index Terms*—Computers and information processing, computer architecture, data structures, distributed computing.**

I. Introduction

EFFICIENT data storage and access are important to deliver high system performance, which however are exacerbated by the explosive growth of data. Existing index structures, such as B$^+$-tree [2], Hash-map [3], and Bloom filters [4], usually support in-memory systems to handle data processing tasks in a memory-efficient manner over the past decades [5], [6], [7], [8], [9].

Manuscript received 28 November 2022; revised 14 September 2023; accepted 10 October 2023. Date of publication 23 October 2023; date of current version 19 April 2024. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 62125202 and U22B2022. An earlier version of this paper was presented at the 48th International Conference on Very Large Data Bases (VLDB), 2022, pages: 321-334 [doi.: 10.14778/3489496.3489512]. This extended version represents significant improvements over the preliminary version, i.e., scaling existing learned indexes to the distributed systems via the pipeline operations and the extended RCU barrier. Recommended for acceptance by S. Salihoglu. *(Corresponding author: Yu Hua.)*

The authors are with the Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: cspfli@hust.edu.cn; csyhua@hust.edu.cn; jingnanjia@hust.edu.cn; pfzuo@hust.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3327009

In general, tree-based structures keep all data sorted for range queries, which aim to identify the items within a given range. Many systems, such as NoSQL systems (e.g., MongoDB [10]), IBM DB2 [11], LevelDB [12] and PostgreSQL [13], construct tree-based structures to provide efficient data storage and access. However, the trees deliver low search performance when storing a large amount of data, due to the expensive pointer jumping among multi-level inner nodes. Moreover, the high space overhead becomes exacerbated once the index structures are too large to fit into the limited-size memory [14].

Unlike them, Kraska et al. [15] argue that exact data distribution enables efficient optimizations for index structures. For instance, a linear regression function is sufficient to store and access a set of continuous integer keys (e.g., the keys from 1 to 100 M), which has significant advantages over traditional B$^+$-trees in terms of lookup performance and memory overhead. The patterns of data distributions become important for memory systems to deliver high performance. However, some patterns are extremely complex or even impossible to be accurately represented via known patterns. In the context of our paper, a known pattern is the data distribution that could be represented via one known mathematical function, such as the linear distribution. Instead, the extremely complex pattern has to be partitioned into multiple parts and represented via multiple mathematical functions, such as the data distributions of real-world datasets, e.g., processing the data of smart devices [16], the petabyte-scale storage systems of Facebook [17], [18] and LMDB [19]. We consider machine learning (ML) approaches to learn a model that exhibits the patterns of data distribution, called *learned indexes* [15]. The learned indexes open up a new research topic on indexing in memory systems: *Indexes can be considered as machine learning models*.

In the concurrent and distributed systems, we use cost-efficient computations to speed up traditional comparisons, thereby increasing access speed and saving memory space. The inefficiency of the traditional comparisons comes from the cache misses when accessing different data from different locations. Specifically, the traditional B$^+$-tree searches data via traversing the nodes, which adopts multiple comparisons to determine the node to be accessed next and significantly decreases the performance when sufferring from multiple cache misses. The concurrency in the context of this paper is interpreted as that the index operations (e.g., read and write data) are executed by using multiple threads. However, it is non-trivial to efficiently leverage learned indexes for concurrent and distributed systems due to the following challenges.

1041-4347 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

1) *Limited Scalability and Concurrency:* The scalability requires the learned indexes to efficiently handle inserts and adjust to the new data distributions at runtime, as well as scaling to multiple threads for high concurrent performance. However, existing schemes show limited scalability since they do not simultaneously meet all these requirements, including concurrent reading, writing and retraining. For example, FITing-tree [20], ALEX [21] and PGM-index [22] do not consider the data consistency issues to concurrently retrain the models in the multi-core systems. XIndex [23] stores the data into different data structures, hence not keeping all data sorted for efficient range query performance.
2) *Long Tail Latency under Heavy Writes:* Existing schemes incur long tail latency in concurrent and distributed systems due to the heavy data dependency. Specifically, XStore [24] maintains a B-tree on the server and relies on the B-tree to process the modifications. The servers have to traverse multi-level inner nodes of the tree, incurring long latency to obtain the required data. Moreover, deploying existing scalable learned indexes in the distributed systems fails to decrease the tail latency due to the inefficient share-based data structures. Among them, XIndex [23] and FITing-tree [20] handle inserts through a *delta-buffer*, which is a tree-based structure [9] (e.g., B^+-tree or Masstree) and has high dependency of inner nodes when traversing the tree. ALEX preserves empty slots in the trained data arrays to handle inserts, which incur many thread collisions due to the contentions for the available slots during insertions. PGM-index recursively merges multiple sets for insertions, which becomes inefficient when multiple threads concurrently write a large number of data due to competing for merging the sets.
3) *High Overheads for Consistent Guarantees:* The data consistency requires that the newly inserted or modified data is correctly identified by all machines. However, the learned indexes record the data positions during the training phase and fail to perceive the new data positions when different machines frequently modify (e.g., insert and delete) the data. Existing learned index schemes have to retrain the models to record the new data positions, which is a time-consuming operation. The challenge is how to ensure the data consistency among different machines without blocking other index operations.

In order to address these challenges, we present a fine-grained learned index scheme for concurrent and distributed systems, called FineStore. Our proposed FineStore achieves high concurrency via a fine-grained index structure, which appends the low-overhead *level bins* under each trained data to alleviate the data dependency. By using such flattened structure, FineStore mitigates the thread collisions and achieves efficient scalability. In fact, the used level bins are two-level sorted arrays, which are used to efficiently handle inserts while keeping all data sorted to support range queries. During the runtime, FineStore concurrently retrains data in two granularities to adjust to the new data distribution without blocking the system. In the distributed system, the clients efficiently identify the locations of remote data via the cached index structures, which incur low overheads of network roundtrips due to the efficient calculations. Moreover, we extend the RCU barriers in the distributed system to enable the stale cached indexes to correctly identify the modified data. Our experimental results show that FineStore improves the insertion performance respectively by about 1.8× and 2.5× than state-of-the-art XIndex and Masstree, while consuming less memory space.

It is worth noting that the mentioned models are interpreted as linear regression ML models with bounded prediction errors, which predict the positions of the keys. To ensure that no data are lost in the system, the bounded prediction errors are determined by the data that is farthest from the central function. We use multiple small models, rather than a complex model, to learn the data distribution, since multiple small models are flexible and efficient for system scalability [15], [20], [23].

In this paper, we have the following contributions.

- *High scalability meeting system requirements:* We present a fine-grained learned index scheme for concurrent and distributed systems, i.e., FineStore,[1] which efficiently meets the scalability requirements and provides high concurrent performance with low overheads. The main insights are to weaken the data dependency via the flattened data structures and concurrently retrain the models without blocking the systems.
- *High performance for cost-efficient index operations:* The index operations (i.e., read and write) are cost-efficient, since FineStore incurs a few data movements during insertion and keeps all data sorted for accessing. For write-intensive workloads, FineStore alleviates the thread collisions by weakening the data dependency. In the distributed systems, the clients efficiently access the remote data via calculating the cost-efficient learned models.
- *Low overheads for consistent guarantees:* FineStore concurrently retrains models in two granularities, which not only supports concurrent operations during retraining, but also guarantees that the new models identify the data modified by other threads. Moreover, the server leverages the extended RCU barrier to enable the clients to correctly identify all data via the stale index structures.

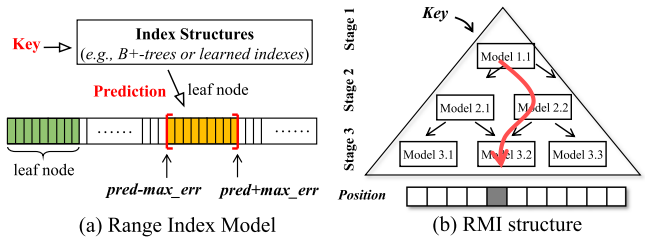
II. Background and Motivation

A. New Perspectives on Indexes

From the perspective of machine learning, the range index structures are considered as regression models [15], which predict the position of a given key, as shown in Fig. 1(a). In the B^+-tree, the data are found through traversing the tree. A learned index [15] views this process as a prediction and supports range queries, which requires the data to be sorted, thus facilitating efficient data access. The records between $[pred - max\_err, pred + max\_err]$ are the analogy with the leaf nodes in the B^+-tree. The length of $[pred - max\_err, pred + max\_err]$ is related with the lookup performance.

[1]The source codes of FineStore are available at https://github.com/iotlpf/FINEdex.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.



(a) Range Index Model (b) RMI structure

Fig. 1. Learned index model and structure.

TABLE I
LIMITED SCALABILITY OF EXISTING SCHEMES

Schemes	Keep all data sorted	concurrency		scale to distributed systems
		write	retrain	
RMI [15]	✓	✗	✗	✗
FITing-tree [20]	✗	✗	✗	✗
XIndex [23]	✗	✓	✓	✗
ALEX [21]	✓	✗	✗	✗
PGM-index [22]	✓	✗	✗	✗
FineStore	✓	✓	✓	✓

In order to provide practical and accurate prediction, the sorted keys and true positions are respectively considered as the inputs and outputs. The relationship between keys and positions is a monotonically increasing curve and similar to a cumulative distribution function (i.e., CDF, which helps to learn the data distribution) [15], [23]. Based on this observation, the prediction accuracy can be improved by learning the patterns of data distribution.

Using a single ML model to achieve high prediction accuracy is difficult, which results in complex ML models. In the meantime, it is hard to design and train this type of models due to the unacceptable training overheads. The learned indexes propose a recursive model index (RMI) [15], [25] to improve the prediction accuracy, which gradually reduces the prediction ranges via multiple small ML models. The main idea of RMI is to build a model hierarchy and predict the positions of keys via trained models [15]. As shown in Fig. 1(b), the RMI consists of 3 stages, respectively containing 1, 2 and 3 ML models. These models are trained in the order of hierarchical relationships, each of which is trained with different data. For example, Model 1.1 in the top level is trained first with the whole dataset. Based on the prediction results of Model 1.1, either Model 2.1 or 2.2 is selected and the entire dataset is also divided into two subdatasets according to the selection results. The two models in the second stage are trained with their individual subdatasets. The next stage follows the similar training process. In order to accurately find the queried key, the learned indexes store the absolute max_error for each model in the last stage, which is calculated as follows:

$$max\_err = max(abs(y_i - f_L^j(x))) \quad \forall i \in S_{L.j}, j \in M_L, \tag{1}$$

where y_i represents the true position of each key in the subdataset $S_{L.j}$, $f_L^j(x)$ represents the prediction result of j_{th} model in the last stage L and there are M_l models in stage l. If max_err is larger than the predefined threshold, the ML model becomes invalid to be replaced with a B$^+$-tree. Finally, learned indexes show the prediction accuracy $[pred - max\_err, pred + max\_err]$ if the picked ML model is valid, otherwise searching the B$^+$-tree.

B. Concurrent and Distributed Indexes Requirements

In concurrent and distributed systems, multiple threads of different machines access the index structures to process various index requests, which require the indexes to efficiently support high concurrency, guarantee the data consistency, and provide high performance.

Efficient Concurrency: Providing concurrent operations becomes important in the systems that scale to a large number of cores and threads. No or few thread collisions are generally helpful to improve concurrent performance, especially for the learned indexes to insert and retrain new data at runtime. However, it is non-trivial to concurrently retrain the learned indexes, since the retraining consumes a long time to block other operations on resorting and retraining the data.

Strong Data Consistency: Guaranteeing the data consistency becomes a foundational requirement to prevent the data loss. However, frequently modifying data decreases the model accuracy of the learned indexes, since the learned models record the data positions during the training phase and fail to perceive the new data positions unless retraining. The challenge is how to ensure the data consistency without blocking concurrent operations.

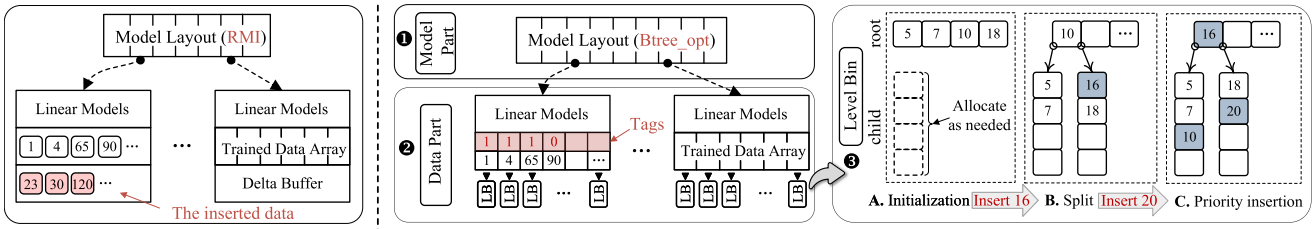
High Performance: As an ordered index structure, all data need to be kept sorted during insertion for efficient range query performance. Otherwise, we need to search the queried data multiple times, incurring long latency. Moreover, the index structure should maintain low data dependency to avoid the thread collisions for high concurrent performance.

In the distributed systems, incurring few network roundtrips among different machines becomes important to meet the high-performance requirements. The promising Remote Direct Memory Access (RDMA) network enables one machine to directly access the remote memory without involving the remote CPUs, which provides high throughput and low latency for data accessing [24], [26], [27]. However, deploying tree-based structures in the distributed system becomes inefficient, since the clients have to spend multiple network roundtrips on traversing the inner nodes and the dependency among inner nodes increases the thread collisions when modifying the data.

C. The Inefficinecy of Existing Schemes

Various learned indexes leverage different strategies to support scalability, including FITing-tree [20], ALEX [21], PGM-index [22] and XIndex [23], which however show limited scalability, as shown in Table I. Specifically, FITing-tree and XIndex handle inserts in the delta-buffers. Their differences are that XIndex uses a concurrent delta-buffer (i.e., Masstree [9]) and supports concurrent retraining, as shown in Fig. 2(a). Although handling inserts in the delta-buffer won't affect the trained data

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.



(a) The structure of XIndex (b) The structure of FineStore

Fig. 2. Structures of XIndex and FineStore. *FineStore consists of model and data parts.*

and guarantees the data correctness during insertion, such design is inefficient due to storing the data in two different structures. XIndex [23] shows that the search performance decreases about 3× when the delta-buffer becomes large, due to checking both learned indexes and delta-buffer in each index operation. Moreover, when scaling to multiple threads, the use of delta-buffer increases the thread collisions due to being shared by all the data covered by the model. The delta-buffer is a tree-based structure, which has high dependency among inner nodes during traversing the tree. To improve the performance, XIndex proposes a Two-Phase-Compact technique to enable concurrent retraining, which concurrently compacts the delta-buffer in the learned indexes without blocking the systems. However, such design still suffers from the inefficient delta-buffer, which handles inserts by constructing another delta-buffer during retraining.

In ALEX, we leverage empty slots in the trained data array to handle inserts in-place. During insertion, existing data in the trained data array are moved backward to the empty slots for the new data. At the same time, we check the trained model and expand the prediction error as needed to avoid the error that some data are moved out of the prediction range. Unlike it, PGM-index allocates multiple empty sets and merges different sets to handle inserts. However, such designs become inefficient when scaling to multiple threads, since different threads compete for the shared empty slots and the pre-allocated empty sets during insertion. Moreover, when there are insufficient empty slots, ALEX and PGM-index fail to support concurrent retraining. Before the retraining completes, we cannot concurrently insert new data, since the new model under retraining fails to perceive the error that some data are moved out of the prediction range during insertion. To guarantee the data consistency, the thread conducting retraining blocks the system for a long time, which significantly decreases the concurrent performance.

III. The FineStore Design

In this section, we present the design of FINE-grained scalable learned index, or FineStore, for concurrent memory systems. The key insight of achieving high concurrent performance is to reduce the dependency among data, as well as mitigating conflicts among threads. Based on these principles, FineStore handles inserts in the non-shared level bins and concurrently retrains models in two granularities, including the level-bin retraining and model retraining. Specifically, the level bins are 2-level sorted arrays appended behind each trained data, as shown in Fig. 2(b). Such flattened data structure significantly reduces the numbers of thread collisions, since the level bins behind different trained data have no data dependencies. The new data are inserted into the level bins according to the order to keep all data sorted. At the same time, existing trained data are not affected by the new data, which guarantees that no data are lost during insertion. When the level bins are full, we concurrently retrain the data in two granularities to adjust to the new data distribution at runtime, including the level-bin retraining and model retraining. The former retrains the full level bins to obtain a small model, while the latter merges small models to improve the performance. After retraining, the old models are easily replaced with the new ones, since all models in FineStore are independent. Through these designs, FineStore achieves high concurrent performance using multiple threads.

A. Model Part

1) The Learning Probe Algorithm: To overcome the shortcomings of previous strategies, our paper proposes the learning probe algorithm (LPA), which uses the greedy strategy to adaptively partition the data according to the data distribution. In LPA, only the same linearly distributed data are divided into the same subdataset. Therefore, each subdataset is easily learned by a linear regression model. The criterion for judging whether the data have the same distribution is to examine if the error of the obtained model exceeds a predefined threshold. If the error of obtained model is smaller than this threshold, LPA will add more data to the subdataset, otherwise remove a small amount of data in the order from back to front until the remaining data are linearly distributed. The complete process of LPA is shown in Algorithm 1.

Before using LPA, we need to configure some parameters including $threshold$, $learning\_step$ and $learning\_rate$, where $threshold$ is the max $error$ of the model we can tolerate, $learning\_step$ and $learning\_rate$ are used to determine the learning speed. As shown in Algorithm 1, the main component of LPA works like a probe, which first walks forward for a large step of length $learning\_step$, i.e., add $learning\_step$ data from the training dataset $record$ into a small dataset S (line 2). Then, we obtain a linear regression model on dataset S and calculate the prediction $error$ of the model (lines 3 and 4), where min_err and max_err are calculated by (1). The prediction error of the obtained model determines the next operation of the probe. If $error < threshold$, the probe keeps moving forward

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

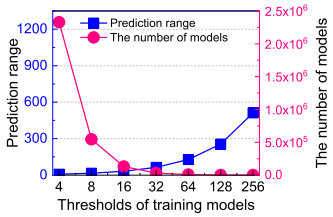


Fig. 3. Setting different thresholds to train models.

to another $learning\_step$ to obtain a new model until the error of obtained model is not smaller than $threshold$ (lines 5-8). When $error > threshold$, the probe keeps moving backward with a smaller step until the prediction error of the obtained model is not larger than $threshold$ (lines 9-13). The smaller step is determined as follows:

$$small\_step = learning\_step * learning\_rate^n, \quad (2)$$

where $learning\_rate \in (0, 1)$, and $n \in (1, 2, 3...)$ represents that the probe iteratively moves backward with much smaller steps. Finally, LPA appends the model to FineStore and cleans the dataset S for next probing (lines 14 and 15).

Unlike RMI, all the model errors in LPA are smaller than the predefined threshold, since LPA trains data according to the data distribution and only the model whose prediction error is not larger than $threshold$ can be appended to FineStore. The max-error of each obtained model is controlled by the predefined parameter $threshold$. The time complexity of LPA depends on the data distributions, which has $O(N)$ complexity in the best case and $O(N^2)$ in the worst case, where N represents the number of the trained data.

In FineStore, a small threshold ensures that the learned models predict the positions in a small range, hence providing high prediction accuracy. However, setting a small threshold needs to train a large number of linear regression models, since a few linear regression models fail to accurately represent the complicated real-world data distributions. As shown in Fig. 3, decreasing the threshold from 128 to 4 increases the number of models by more than 100 times, which results in low performance to search the models, since FineStore stores the independent models in the B-tree and a large number of models increase the height of the B-tree. From the evaluation results, we observe that 32 is an efficient trade-off between the prediction accuracy and the model numbers. Therefore, we use the predefined threshold 32 to train the models.

2) Optimized Model Layout: The model layout is interpreted as the organization structure for storing models, which affects the scalability and the performance of finding models. Unlike RMI that has heavy model dependency among different levels, our FineStore trains independent models to enable high scalability.

The total number of models trained by LPA is small, which is competitive with PGM-index, since LPA greedily trains the models according to the data distributions, as shown in Table II. Hence, we store the piecewise models as a cache-/SIMD-optimized btree (i.e., align the btree node with cacheline and search the node with SIMD instructions) to enable system

TABLE II
NUMBERS OF MODELS OF DIFFERENT SCHEMES ON VARIOUS WORKLOADS

Workloads		Normal	Lognormal	Weblogs	DocID	YCSB
Number of Data		200M	200M	127M	10M	100M
Number of Models	LPA	57,835	58,027	38,355	50,260	25,532
	PGM-index	55,226	55,352	36,256	47,542	23,125
	RMI	250,000	250,000	250,000	250,000	250,000

Algorithm 1: LPA Algorithm.

Input: int $threshold$,int $learning\_step$,float $learning\_rate$, dataType $record[N]$
Output: trained $FineStore$

```
1  while not reach the end of the dataset record[N] do
2      add learning_step data into dataset S from record;
3      train a linear regression model on S;
4      error = max(|min_error|, |max_error|);
5      while error < threshold do
6          add next learning_step data into dataset S from
             record;
7          train a new model on S;
8      end
9      while error > threshold do
10         step=int(learning_step * learning_rate^n);
11         remove step data from the end of dataset S;
12         train a new model on S;
13     end
14     FineStore.append(model);
15     clean data from dataset S for next probing;
16 end
```

scalability and deliver high performance. The models are stored as $< key, model >$ pairs, where key is the largest trained data covered by each model and $model$ is a pointer to the model. The number of models in RMI is manually set, since RMI fails to adaptively assign the number of models according to the data distribution. As XIndex shows that 250 K models in RMI achieve the best performance, we also configure 250 K models to facilitate fair comparisons.

B. Data Part

We propose the structure of level bins under trained data to process the modifications. The structure of the level bins is a modified two-level B-tree as shown in Fig. 2(b). The horizontal blocks represent the root bins and the vertical blocks represent the child bins. When the two-level bins are not full, the new data are inserted like a B-tree with the difference that the data are l to be inserted into the previous child bin. The full level bins do not grow to higher levels to avoid high data dependency in the tree-based structures. Instead, we propose fine-grained retraining with two granularities to accommodate more data.

Specifically, Fig. 2(b) shows how the level bins process inserts. At the beginning, only one bin is placed under the trained data for space savings and other bins are constructed as needed. For example, we construct two child bins to insert 16 when the root bin is full. As more data are inserted, the data are prioritized to be inserted into the previous child bin to improve the space utilization. For example, we move existing data forward to the first child bin when inserting 20 to improve the space utilization of existing bins. In this case, we move at most $(n + 1)$ data, where n is the length of the child bin. When the previous child bin

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

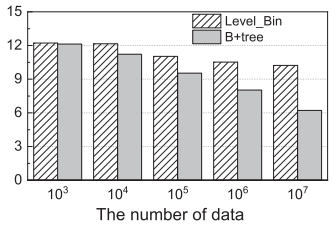


Fig. 4. Performance of adopting different schemes.

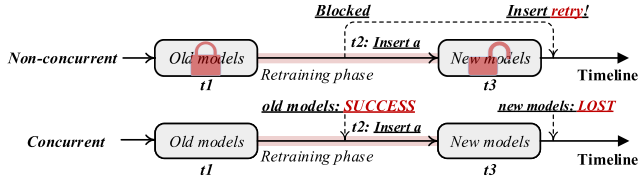


Fig. 5. Challengs of different retraining strategies.

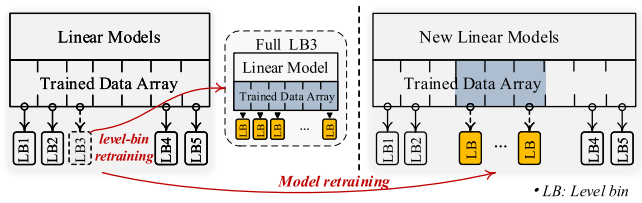


Fig. 6. Concurrent retraining. *Level-bin retraining retrains full bins. Model retraining merges the small models.*

becomes full, the data is inserted like a two-level B-tree, which moves at most $(m + n/2)$ data in the worst case, where m and n respectively represent the number of slots in the root and child bins. Each bin has 8-16 slots in our experiments to achieve an efficient tradeoff between scalability and access efficiency. We bound the level bins to two levels to alleviate the data dependency among levels, and retrain the full level bins to accommodate more data. Our experimental results show that the *maximum load factor* (i.e., the number of occupied slots divided by the total number of slots) of FineStore is about 82%, which is higher than 75% in the B-tree.

When scaling to multiple threads, the level bins behind different trained data won't block each other due to the low data dependency, which incur few thread collisions. When the learned structures learn the data distribution, the inserted data are likely to exhibit the same patterns [15], and hence are inserted evenly into all level bins. In this case, FineStore handles nearly $(m * n)$ times more than the trained data. When the data distribution changes, FineStore concurrently retrains the level bins to fit the new data distribution, as shown in Section III-C.

Compared with B$^+$-tree, level bins have lower data dependency among different nodes, since level bins are bounded to only two levels to mitigate the data dependency, while the B$^+$-trees commonly construct multi-level nodes and contains high dependency among the data nodes. Moreover, the level bins have high space utilization than B$^+$-tree due to prioritizing inserting data into existing child bins during the insertion. Compared with hash-based structures, the level bins ensure that all data are sorted for efficient range query performance, while the hash-based structure randomly stores the data and fails to keep all data sorted. Furthermore, level bins efficiently support our proposed two-granularity retraining, while other data structures fail.

Fig. 4 shows the operation throughput when adopting different schemes in FineStore. From the results, we observe that level bins deliver higher throughput to insert and access data than the B$^+$-trees, since the level bins efficiently support two-granularity retraining to reduce the dependency among data during the runtime, while the B$^+$-tree fails.

C. Concurrent Retraining

In general, some level bins are full when more data are inserted or the data distribution changes, e.g., the skewed workloads (i.e., the data are modified in certain ranges). Instead of reconstructing the indexes from scratch with high overheads, FineStore performs retraining to adjust to the new data distribution.

The challenge is how to ensure the data consistency without blocking concurrent operations. For example, retraining a model on one million data consumes up to several seconds [15], which blocks the systems for a long time. As shown in Fig. 5, if we retrain the model in a sequential manner, the data covered by the model (including the trained data array and all level bins) are blocked until the retraining is completed, which incurs high overheads in the concurrent system. On the other hand, the data inconsistency occurs during concurrent retraining. As shown in Fig. 5, the new data a is successfully inserted into the old models during retraining (i.e., t_1 to t_3). However, the new models can't find a since the new models fail to train a when the retraining begins at t_1. Moreover, it is hard to identify which data are inserted during retraining, since the newly inserted data are mixed with existing data during reordering. Processing the inserts in an extra delta-buffer separates the newly inserted data with existing data, which however fails to keep all data sorted and degrades the overall performance with the growth of buffer size.

To address these challenges, FineStore performs retraining in two granularities, including the level-bin retraining and model retraining. The former generates more space by retraining full level bins and the latter merges small models to improve the model accuracy and search performance.

1) Level-Bin Retraining: Retraining a model needs to retrain all the covered data in the trained data array and the level bins. Hence, it is expensive to retrain the whole model even if the level bins of only one trained data are full. To address this issue, we retrain a new model based on the data of the full bins, while other data in the trained data array and the level bins are not retrained. The new model is appended under the corresponding trained data, and new level bins are created under the new model to process the inserts, as shown in Fig. 6.

Level-bin retraining achieves high concurrent performance since only the full level bins are locked for the data consistency, while other data are not related. Moreover, performing level-bin retraining is cost-efficient (e.g., 27 μs in our experiments), since

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

the full level bins contain no more than $m * n$ data, where m and n respectively represent the number of slots in the root and child bins.

2) Model Retraining: The system performance decreases when a large number of small models are iteratively created via the level-bin retraining. In this case, FineStore merges these small models through the model retraining to maintain high performance.

As shown in Fig. 6, FineStore conducts model retraining by compacting the trained data arrays of different models (i.e., the large and covered small models, including the smaller ones). New models are trained on the covered trained data arrays, which are not modified by the new data according to the design principles in Section III-B. The retraining process is performed in the background to hide the latency in the concurrent system. During retraining, the level bins are not affected and concurrently process the in-place modifications without blocking the overall system. We directly append the pointers of the level bins under the new trained data array. After the new models are retrained, FineStore uses the RCU-barrier [23] to ensure that all threads access the new models. The RCU-barrier is a synchronization mechanism of concurrent systems, which enables all readers to access the new data structures, rather than the old ones, in a shared memory. Since both new and old models point to the same level bins and the modifications during retraining are processed in-place in the level bins, any concurrent modification during the model retraining is not lost.

The model retraining is triggered when the small model needs to retrain a smaller model. For skewed workloads, FineStore assigns the level bins in the data-intensive parts via retraining, which flattens the data-intensive parts after several retrains. The new data are inserted into the flattened data structure with low data dependency. For the skewed workloads, i.e., reading and writing data in a certain range, FineStore assigns the level bins in the certain range via retraining, which flattens the certain data parts after several retrains. The new data are then inserted into the flattened data structure with low data dependency. Hence, FineStore gradually adapts to the new data distribution along with time.

D. Practical Operations

Search: Fig. 2(b) shows a complete searching process for item 7 using a single thread. *Stage* ❶: Find the model that covers the item 7 in the model layout. *Stage* ❷: Search in the prediction range, which is calculated by the obtained model $f(x)$. *Stage* ❸: FineStore completes the search if finding the given key in the prediction range, otherwise FineStore searches the level bins or the small models.

Insert: FineStore searches the whole structure to identify if the given data exists, and only the unique data are inserted into the level bins as elaborated in Sections III-B and III-C.

Update: If a given key exists in the structure, FineStore updates the corresponding value via atomic writes, which is easily implemented since the value is a 64-bit pointer referring to the real data.

Remove: As shown in Fig. 2(b), we use the tokens (i.e., 0 and 1) to indicate whether the trained data are removed. The data in the level bins are directly removed, since changing the data within the level bins won't affect the model accuracy.

Range Query: FineStore processes the range query requests via two steps. First, FineStore determines the lower and upper boundaries like the point search. Second, FineStore obtains all the data between the determined boundaries, without searching the data one by one since all data are sorted during the runtime.

E. Concurrency

Concurrent data structures become important to existing systems that scale to a large number of cores and threads. The thread collision probability of FineStore is rather low due to the flattened data structure. We use the version control [9] and allocate fine-grained locks to enable FineStore to support concurrent operations.

1) Write/Write Conflicts: The write/write conflicts occur when different threads modify the same trained data or the same bin. FineStore allocates the per-record locks for the trained data and the per-bin locks for the bins to enable concurrent writes. For example, according to the principle of the modification operations (Section III-D), FineStore first updates/removes the matching record (i.e., whose key is equal to the given key) in the trained data array, and the per-record lock of the corresponding record ensures the concurrent writes. FineStore further modifies the data in the level bins when failing to match a record in the trained data array. The per-bin lock is used to enable concurrent bin to be updated and split. Specifically, FineStore locks the child bin which is determined to process the modification, while the root bin is locked as needed (i.e., when child bin splits or the largest data in the child bin changes).

Existing schemes use delta-buffers or preserve empty slots to enable scalability, which however incur high overheads due to the data dependency. For example, many locks are needed when the tree in the delta-buffer becomes large. For the schemes preserving empty slots, we need to lock all data covered by the same model to enable correct data movements for resorting. Unlike them, FineStore decreases the conflict probability, since different threads that modify the level bins under different trained data don't block each other.

2) Read/Write Conflicts: Instead of using the locks during reading, FineStore uses the version control [9], [28] to ensure that the obtained data is consistent and latest. FineStore allocates the version numbers for each trained data and bin, and increases the version count when the data are modified. During reading, if a record in the data structure matches the given key, FineStore maintains the version v in the form of snapshot before obtaining the value. The obtained value becomes valid if the version doesn't change (i.e., the version after reading the value becomes equal to v) and the data is not locked. Otherwise, the latest value is not read, since other threads are updating the value during the data locking. FineStore repeats to read the current and next child bins until obtaining the valid value, since the data are possibly moved to the next child bins if the current bin is split.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

3) Write/Retrain Conflicts: FineStore avoids the write/retrain conflicts by training the models in two granularities, including the level-bin retraining and model retraining. Specifically, the level-bin retraining locks the full level bins for retraining, which ensures the data consistency due to blocking other data modification operations. The model retraining avoids the write/retrain collisions by sharing the level bins, i.e., the write operations are conducted in the level bins via the old models and these level bins are directly referenced by the new models. Therefore, any modifications during retraining are consistent in the new models.

4) Write/Split Conflicts: FineStore avoids the write/split conflicts by leveraging the per-bin locks during the write phase. Specifically, only one write thread obtains the lock of a bin and conducts data modification operations, while other threads wait and compete for the lock of the bin to avoid data collisions. The read threads re-read the data to avoid the data inconsistency when identifying that the bin is locked.

IV. Scale to Distributed Systems

It becomes common to scale to multiple machines, i.e., the distributed systems, to enjoy the large storage capability [26], [27], [29]. We store the data on servers and access remote data via clients using the cached index structures. Caching tree-based structures consumes a large amount of memory due to the multi-level inner nodes, which becomes inefficient to cache the whole index structures [29], [30], [31]. However, caching partial index structure fails to determine the data locations since some inner nodes miss the local cache. The clients have to either spend multiple network roundtrips on traversing the tree or transfer all requests to remote servers. Both solutions become inefficient since multiple RTTs (i.e., the roundtrips time) incur long latency, while the scheme of transferring requests to servers increases the computing burden of remote servers. The servers become the system bottleneck and decreases the overall performance when failing to efficiently process the transferred data requests, since the tree-based structures contain heavy dependency of inner nodes and fail to enable high concurrency.

Compared with tree-based structures, the learned indexes save 2-4 orders-of-magnitude space consumption, which become efficient to be fully cached on clients [22], [24]. For a given key, the clients obtain the remote data via a one-sided RDMA according to the remote data location. In this case, the data searching requests are offloaded to clients. However, the challenges are to enable the servers to efficiently process data modifications and ensure the data consistency between servers and clients during the modifications. The state-of-the-art scheme, i.e., XStore [24], achieves significant performance improvements using the hybrid index structures, i.e., maintaining the tree-based structure on servers to process data modifications and caching the learned indexes on clients for remote data searching. Such design suffers from multiple thread collisions when processing intensive data requests, due to the inefficient tree-based structures on servers.

Deploying existing learned indexes in the distributed system incurs high overheads to search the data and ensure the data consistency among different machines. For example, FITing-tree [20] and XIndex [23] incur long latency to search the remote data when the buffer becomes large, since the clients have to spend multiple RTTs on determining the data locations. ALEX [21] and PGM-index [22] fail to concurrently retrain models due to the heavy data dependency, as shown in Section II-C. Although FineStore stores data in the flattened data structure and supports concurrent retraining, the index consistency among different machines during retraining needs to be guaranteed. Because in a single machine, FineStore concurrently retrains data in the background and replaces the old models using the RCU barrier, which fails to be deployed in the distributed systems due to the lack of RCU barrier among multiple machines. Moreover, the performance of FineStore decreases in the distributed systems, since the clients spend extra RTTs on determining the data locations in the level bins.

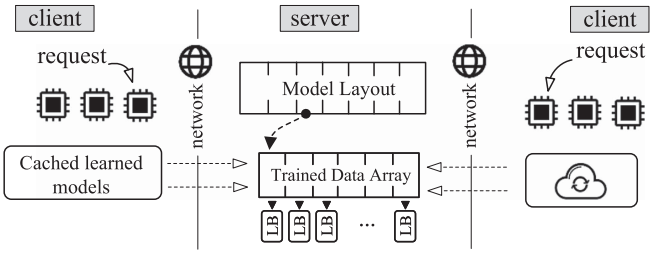


Fig. 7. Scale FineStore to distributed systems.

Fig. 7 shows the architecture of scaling FineStore to the distributed systems. To provide high performance in the distributed systems, we enable FineStore to avoid the aforementioned challenges via two techniques, including the pipeline operations and the extended RCU barrier.

A. Pipeline Operations

FineStore leverages the pipeline operations to access the remote data, i.e., the clients search the trained data array and determine the target level bin, while the servers access the determined level bin for further operations. Such designs achieve an efficient tradeoff between the network penalty and the computing resources consumptions of servers, since the clients only consume one RTT to search the trained data array and most computing tasks have been offloaded from servers to clients.

Initially, the servers train models according to the data distribution and store data in the trained data array, while the clients locally cache the whole index structures. To process different data requests, the clients calculate the data locations using the cached learned indexes, and obtain the predicted data from servers via one-sided RDMA operations. The obtained data are from trained data arrays, which are not modified during the runtime and hence are consistent with other machines. Moreover, the data modification requests are conducted in the level bins. After determining the target level bin by comparing the request data with the obtained data, clients transfer the data requests to servers for further operations. The transferring

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

request consists of $[request\_type, lb\_addr, data]$, whrere the $request\_type$ represents the type of requests, including Search, Insert, Update, Remove, and Rang requests. lb_addr is the address of the target level bin, and $data$ represents the keys or key-value pairs for various requests. The servers concurrently lock the corresponding level bins to facilitate further data operations. According to different data requests, the server returns the data in the address lb_addr and then modify the data in the level bins for Insert/Update/Remove requests. The servers deliver high concurrent performance, since the level bins are bounded to two levels and the level bins under different trained data have no data dependency.

The servers concurrently lock the corresponding level bins to facilitate further data operations according to the obtained data requests. The servers deliver high concurrent performance, since the level bins only contain low-level arrays and the level bins under different trained data have no data dependency.

B. Extended RCU Barrier

FineStore retrains a small model on the full level bins, and further merges the small models using the proposed model retraining, as shown in Section III-C. To ensure the data consistency among different machines when updating the index structures, we extend the RCU barrier for the distributed systems.

Specifically, new models are retrained on the servers in the background, i.e., constructing new trained data arrays in the newly assigned space by merging the covered trained data arrays. During retraining, FineStore processes the modifications in the level bins. We append the pointers of the level bins under the new trained data arrays to guarantee that the new models correctly identify the modified data. After retraining, we cannot remove the old models and replace them with the new ones, since other machines concurrently access remote data using the stale models before synchronizing new models from servers. Instead, we update the models in two steps. First, we change the model pointers from the old models to the new ones, and do not remove the old models and the covered trained data arrays. At the same time, we set a counter to record the number of machines identifying new models. Second, we lazily remove the stale models when all machines identify the new models according to the counter.

In this case, no data are lost since new models identify the data modifications via the pointers of the level bins. Different machines contain the consistent data, since the new models share the level bins with the old ones, which enables the machines to cache stale models and correctly identify the new data. Moreover, the servers do not reclaim the old trained data arrays until all machines update the index structures. The remaining old trained data arrays consume acceptable space, since each trained data array covers 1,400 data on average.

C. Durability

FineStore efficiently adopts existing logging-based scheme to enable durability, since all modification operations are conducted on the servers, rather than distributed systems among servers and clients. In FineStore, different threads write the modification operations into the log buffer, which contains key, value and version number to avoid collisions. One logging thread flushes the logs into the log files for persistence. For optimization, the logger merges and batches log entries to enable efficient durability.

V. Performance Evaluation

We run experiments on a Linux server (kernel version v4.19.91) that contains one 12-core Intel(R) Xeon(R) CPU @2.50 GHz (each core with 32 KB L1 instruction cache, 32 KB L1 data cache and 1024 KB L2 cache) and 48 GB DRAM. We run all schemes with 24 threads to evaluate the concurrent performance by default.

Counterparts for Comparisons: We compared our proposed FineStore with state-of-the-art schemes. For the tree-based structures, we compare FineStore with Masstree [9], which is a variant of scalable concurrent B^+-tree. Due to different design goals, B^ϵ-tree [32] is not compared since it is optimized for less disks I/Os, rather than the memory access in our scheme. Moreover, for the learned index schemes, we enable RMI [15] to support scalability by adding a delta-buffer (denoted as LI+Δ), where the buffer is implemented as a Masstree [9]. We compare FineStore with XIndex [23] and LI+Δ [15], where their difference is that LI+Δ fails to support concurrent retraining. FITing-tree is not compared due to failing to support concurrent operations, e.g., concurrent writing and retraining. We run the codes of ALEX and PGM-index with a single thread, but do not run them with multiple threads due to the thread collisions that come from their slot contentions [21], [22]. The core dump occurs when there are insufficient empty slots, since different threads construct multiple trained data arrays, respectively redistribute data and retrain new models, which incur severe data inconsistency issues.

Configurations: For the compared counterparts, we directly run their source codes with the default configurations. We implement a 2-stage RMI following the original work [15], and the second stage configures 250 K models like the setting in XIndex [23] to facilitate fair comparisons. In FineStore, we use the predefined threshold 32 (which is a suitable trade-off to obtain high prediction accuracy and small number of models), to train the models. The root and child bins respectively contain 8 and 16 keys to obtain a suitable tradeoff between the insertion capacity and search efficiency. By default, we did not enable durability for all schemes in our evaluations to achieve efficient index performance and facilitate fair comparisons.

Benchmarks: (1) *YCSB*, a benchmark with six different workloads (A-F), including update heavy (A), read mostly (B), read only (C), read latest (D), short ranges (E) and read-modify-write (F). All workloads contain 100 million data with both Uniform and Zipfian distributions. (2) *Weblogs* contains 127 million unique log entries and we use the timestamps as the indexes. (3) *DocId* contains five text collections in the form of bags-of-words, which has nearly 10 million instances in total. We also use 2 synthetic datasets with 200 million items to evaluate the behavior of FineStore in depth: (4) *Normal* distribution with $\mu = 4$ and

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

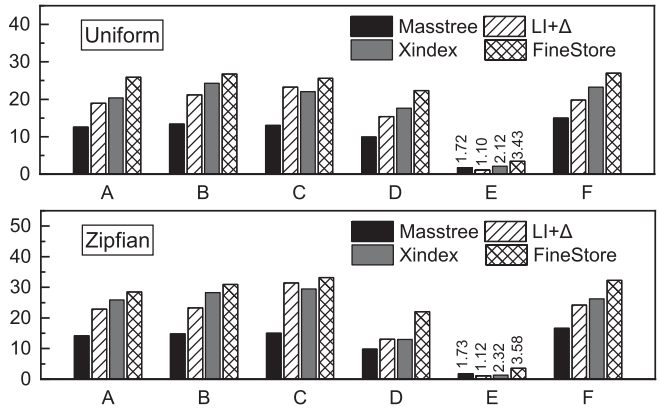


Fig. 8. Throughputs on YCSB with various workloads.

$\sigma = 2$, and (5) *Lognormal* distribution with $\mu = 0$ and $\sigma = 2$. All generated keys are scaled up to $[0, 10^{12}]$ as integers for evaluations. The CDFs of the used benchmarks are shown in Fig. 1(b). We configure all benchmarks with 8-byte keys and value-pointers (i.e., the pointers refer to the variable-length values), since existing systems support up to 8-byte computations for ML models [15], [23].

A. The Throughput via YCSB

Fig. 8 shows the throughput of different schemes on YCSB with Uniform and Zipfian distributions. In general, FineStore significantly improves the throughput on dynamic workloads over other schemes, as well as achieving higher throughput on static workloads due to the optimized model layout and high model accuracy.

Static Workloads (YCSB A, B, C, F): The data distributions of the static workloads won't change during runtime, since most requests are reading (e.g., workload C) or updating the values (e.g., workloads A, B, and F). In these cases, FineStore achieves comparable (even a little better) throughput than LI+Δ and XIndex, since FineStore searches fewer models in the optimized layout and efficiently finds the data with higher model accuracy.

Dynamic Workloads (YCSB D, E): FineStore delivers higher throughput than other schemes on the dynamic workloads. Specifically, FineStore outperforms LI+Δ, XIndex, and Masstree by 1.7×, 1.6×, and 2.3× on workload D. Because FineStore incurs few data movements and has low-probability thread collisions during insertion, while LI+Δ, XIndex, and Masstree incur high overheads to traverse the trees. Moreover, FineStore further improves the throughput by up to 3.2×, 2.7×, and 2.1× over LI+Δ, XIndex, and Masstree on workload E. The main reason is that LI+Δ and XIndex handle new inserts in the delta-buffer, which has data overlapping with the original trained data array and fails to keep all data sorted.

B. The Throughput With Heavy Writes

We evaluate the scalability throughput under heavy writes. In the experiments, we randomly sample a fraction of data to train the learned structures, and the data distribution does not change during insertion. We also insert these sampled data into Masstree for fair comparisons.

The Number of Threads: Fig. 9(a) shows the insert throughput with different threads. We observe that FineStore improves the insert throughput by up to 1.6×, 1.3×, and 2.0× over LI+Δ, XIndex, and Masstree when the number of threads increases. FineStore obtains more performance improvements with more threads, since FineStore reduces the thread collisions by inserting the data into the flattened level bins.

The Number of the Inserted Data: The number of the inserted data to the trained data is defined as *Insert Factor*, which clearly differentiates the inserted data from the trained data for the learned structures. Fig. 9(b) shows the throughput of inserting different numbers of data. We observe that the insert throughput of FineStore is low at the beginning due to consuming time on allocating the level bins for each trained data. When inserting more data, FineStore improves the throughput by up to 1.5×, 1.2×, and 1.3× over LI+Δ, XIndex, and Masstree. The main reason is that the level bins incur few data movements during insertion and handle inserts up to nearly $(m * n)$ times (m and n represent the slot numbers of root and child bins) more than the trained data without retraining. However, the delta-buffer in LI+Δ and XIndex incurs high overheads to iteratively split the nodes with massive data movements. The data dependency among nodes further hinders the concurrent performance during insertion.

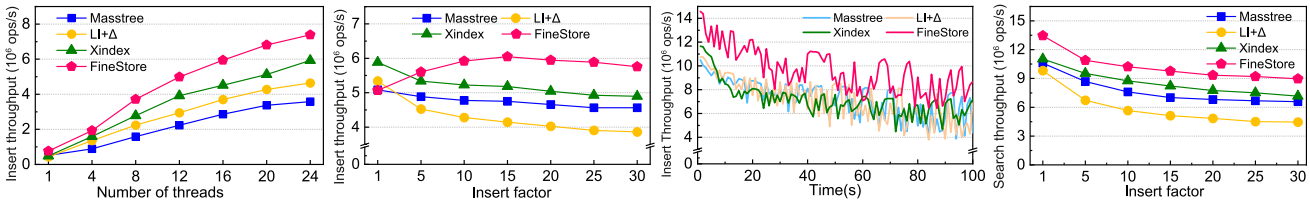
Insertion With Frequent Retraining: Fig. 9(c) shows the throughput timeline when inserting more than 1000× data than the trained data. In this case, the learned models are frequently retrained to learn the new data distribution for high accuracy. We observe that FineStore improves the insert throughput by about 1.8× over other schemes. Because FineStore concurrently adapts to the new distribution by efficiently executing the level-bin retraining and model retraining.

C. Throughput With Read-Write Workloads

The Search Performance After Inserts: The learned structures offer high search performance on the static workloads, which are important even after inserting a large number of data. Fig. 9(d) shows the search throughput after inserting different numbers of data. We observe that LI+Δ and XIndex decrease the search performance after heavy writes, since they have to spend extra time on searching the delta-buffers. The performance further decreases when the buffer becomes large. The performance of FineStore also decreases after inserts, since the size of the level bins increases when we constantly insert data. However, FineStore provides higher search performance than other schemes, since we bound the level-bins to two levels via retraining. We have the similar observations and insights on other benchmarks, as shown in Fig. 10.

Different Read/Write Ratios: Fig. 13 shows the throughput with various read/write ratios. We have the similar observations with previous evaluation results, i.e., FineStore delivers high performance on both static and write-intensive workloads.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.



(a) Different numbers of threads. (b) Different numbers of the data. (c) The throughput timeline. (d) Search throughput after inserts.

Fig. 9. Scalability throughput in various scenarios, *which are evaluated on the lognormal dataset.*

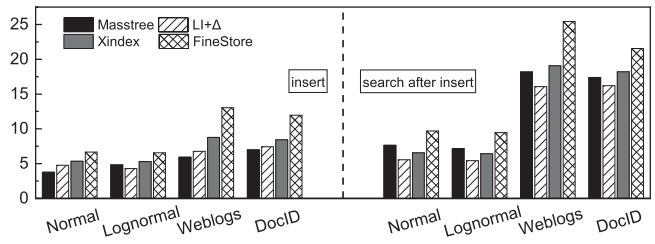


Fig. 10. Throughputs on various workloads.

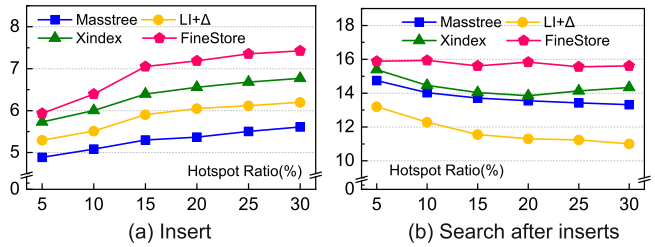


Fig. 11. Throughputs on skewed workloads.

D. Throughput With Skewed Workloads

The data distribution may change, e.g., reading/writing data in a certain range, rather than accessing the data evenly following the trained pattern. The accessed range divided by the range of trained data is defined as *Hotspot Ratio*, where the smaller hotspot ratio represents the larger skewness. Fig. 11 shows the insert and search throughputs on the skew workloads. We observe that both FineStore and XIndex show low performance when the skewness is large, since more thread collisions occur and more retrainings are necessary. As the skewness decreases, FineStore achieves higher performance than other schemes, due to retraining the data-intensive part and assigning a large amount of level bins. After several retrainings, FineStore flattens the skewed data and adjusts to the new data distribution, thus decreasing the thread collisions.

E. In-Depth Analysis for FineStore

To examine where the performance improvements come, we leverage Control Variates [33] to evaluate different components of FineStore, and the results are shown in Fig. 12. In general, the most benefits come from the flattened data part and concurrent retraining.

Model Part: Fig. 12(a) shows the performance of the model part. In this experiment, all data are stored in the trained data array and we won't insert any data. We observe that FineStore doesn't obtain significant performance improvements, compared with other learned schemes, since the models of all learned structures keep high accuracy when there are no inserts.

Data Part: Fig. 12(b)–(d) show the performance of the data part. In these experiments, we only use one model to mitigate the influence from the model layout. From Fig. 12(b), we observe that the level bins improve the insertion performance by about 1.8× than the delta-buffer with a single thread, and further improves about 2× with more threads. The reason is that the non-shared level bins have low data dependency among each other and incur few thread collisions in concurrent systems. After a large number of inserts, the level bins respectively improve about 2.1× and 3.2× point/range query performance than the delta-buffer, as shown in Fig. 12(c) and (d), since the level bins keep all data sorted.

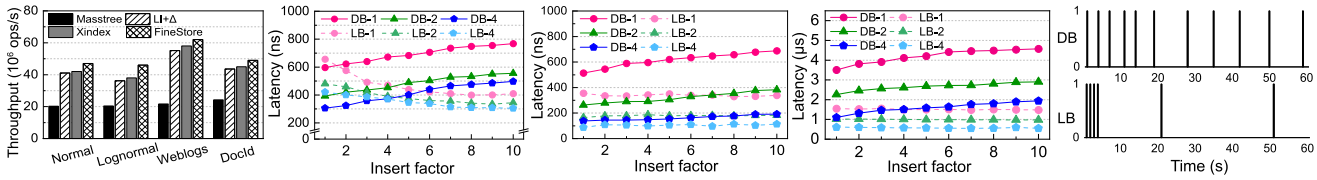
Retraining Frequency: Fig. 12(e) shows the retraining frequency when new data are constantly inserted. We observe that the scheme with a delta-buffer incurs more retrainings than FineStore, since the delta-buffer is shared by all data covered by one model and becomes large during the insertions. Unlike it, FineStore adjusts to the new data distribution after several retrainings and requires less retrainings later. Because FineStore amortizes the insertions into multiple small-sized level bins and processes more inserts with high performance.

F. Overheads Analysis

1) Training Latency: Fig. 14 shows the latency to train different structures, and the latency to train Masstree is evaluated by inserting the trained data into the tree. We observe that FineStore incurs low latency to train the model, which outperforms LI+Δ and XIndex by up to 1.3× and 8.9×. Specifically, the LPA algorithm [34] greedily trains data and obtains fewer models than other schemes during training. However, the RMI scheme needs to traverse all data multiple times due to the level-by-level training strategy [15]. The complexity to train XIndex is higher than RMI, depending on the data distributions, since XIndex needs to train RMI multiple times to improve the accuracy.

To dynamically adapt to the new data distribution, FineStore performs retraining in two granularities, including level-bin retraining and model retraining. The level-bin retraining consumes 27 μs to train the full level bins in our experiments. Although model retraining consumes more time (e.g., 1.5 ms on 10 K data),

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.



(a) Static seach throughput. (b) Insert latency. (c) Point query after inserts. (d) Range query after inserts. (e) The retraining frequency.

Fig. 12. Performance analysis. *DB represents the delta-buffer, and LB represents the level bins. The number # in $DB-\#/LB-\#$ represents the used threads. The 1/0 in figure (e) represents that the retraining is/isn't required.*

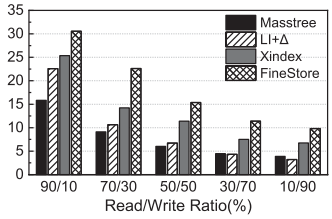


Fig. 13. Throughput with various read/write ratios.

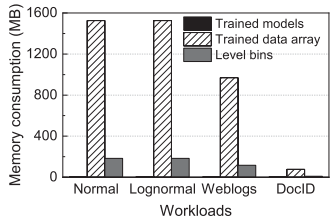


Fig. 15. Memory consumptions of FineStore.

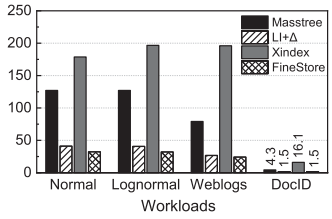


Fig. 14. Training latency on various workloads.

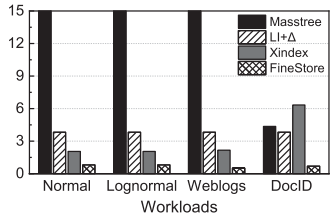


Fig. 16. Memory overhead of models/inner nodes.

the latency doesn't affect other concurrent operations, since we perform the model retraining in background.

2) Memory Overheads: Fig. 15 shows the memory usage of FineStore on various workloads. In general, the memory usage of FineStore consists of three parts, including the trained models, the trained data array and level bins. Among them, the trained models occupy about 2% of the total memory consumption, e.g., consuming no more than 10 MB space to store models when training $2 * 10^8$ lognormal data. The trained data arrays contain all the trained data, while the level bins store the new data, and these two kinds of structures are the main memory consumptions. In the distributed environment, the servers require a large amount of memory to store the models and data, while the clients only need to cache the learned models to save space. Hence, the clients have the capability to cache the full learned indexes. Moreover, although the level bins contain empty slots, FineStore allocates one bin as needed during the runtime, and fully leverages existing allocated bins to achieve high space utilization.

In the distributed systems, the clients locally cache the metadata, which are evaluated in Fig. 16, including the sizes of ML models in the learned structures and the memory consumptions of inner nodes in the tree-based schemes. From the results, we observe that all learned structures consume less memory than tree-based schemes by up to two orders-of-magnitude, since one linear regression model is enough to index the same linearly distributed data while the trees need to construct multi-level inner nodes.

G. Performance in Distributed Systems

We deploy FineStore on a cluster with 4 machines to evaluate the performance in the distributed systems, including 1 server and 3 clients. Each machine is equipped with two 26-core Intel(R) Gold 6320R CPUs @2.10Ghz, 128 GB DRAM, and one 100 Gb Mellanox ConnectX-5 IB RNIC.

We compare FineStore with 3 state-of-the-art distributed ordered KV stores, including EMT [26] (i.e., the distributed Masstree using RDMA), Cell [30], and XStore [24]. Among them, EMT maintains a Masstree on the servers, and relies on the servers to process all requests by transferring requests to servers. Cell caches partial inner nodes of B-tree on clients to speed up the index operations, which however has to spend multiple RTTs on traversing the B-tree due to not caching the whole structure. XStore leverages the hybrid index structures to process the index operations, which caches the learned indexes on clients to access the remote data while leveraging the B-tree on servers to process the modifications. Unlike them, FineStore achieves high scalability by deploying a concurrent learned index scheme on both servers and clients.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

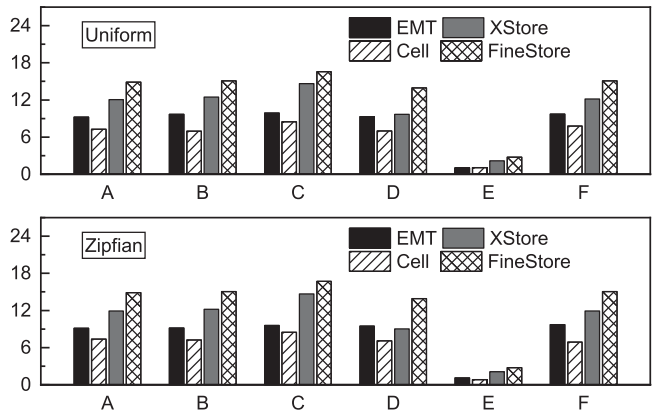


Fig. 17. Throughputs on YCSB in distributed systems.

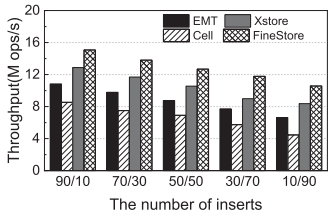


Fig. 18. Hybrid read/write throughput.

Fig. 17 shows the performance of different schemes in distributed systems using various YCSB workloads.

Read-Only Workloads (YCSB C): FineStore delivers competitive performance with XStore and achieves higher read performance than other schemes by up to 2.4x, due to enabling the efficient one-sided RDMA read operations. Unlike it, Cell has to spend multiple network roundtrips on reading remote data, while EMT has to transfer the requests to the remote server and wait for the returned results.

Read-Write Workloads (YCSB A, B, D, E, F): FineStore achieves higher write performance than other schemes by up to 1.7x on the read-write workloads. The main reason is that FineStore efficiently processes different index operations via the pipeline operations, which alleviates the computing burden of servers by offloading the computing tasks to the clients.

Performance of Using Hybrid Read-Write Workloads: Fig. 18 shows the throughput with various read/write ratios. In general, FineStore respectively improves the performance by 1.5×, 2.2×, and 1.3× over EMT, Cell, and XStore when configuring large write ratios, since FineStore incurs fewer thread collisions via the flattened data structure. The performance of EMT is limited by the Masstree due to transferring all data requests to servers, while the Masstree delivers low performance due to the dependency among inner nodes. Cell and XStore efficiently leverage the cached index structures to access the remote data, which however decrease the performance with large write ratios, due to transferring requests to servers and relying on the tree-based structures to process data modifications.

Performance With Intensive Writes: Fig. 19 shows the throughput under intensive writes. In our evaluations, we constantly insert different numbers of data, and observe that

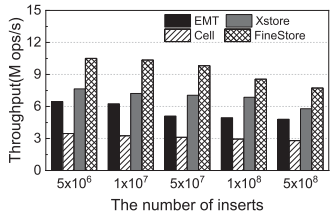


Fig. 19. Write-intensive throughput.

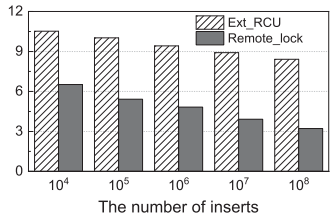


Fig. 20. Benefits of the extended RCU barrier.

FineStore improves the insert performance by up to 1.9×, 2.7×, and 1.4× than EMT, Cell, and XStore. The main reason is that FineStore efficiently process the data modifications via the concurrent learned index scheme, while other schemes rely on the inefficient tree-based structures to process various data requests. Moreover, FineStore incurs low latency since the flattened data structure enables high concurrent operations with few thread collisions.

The proposed extended RCU barrier is used to guarantee the consistency among different machines when concurrently retraining models. With the extended RCU barrier, different machines concurrently access remote data with the cached learned indexes, as well as updating the cached models when identifying new models. Without the extended RCU barrier, FineStore has to adopt the lock-based schemes to ensure the data consistency when updating the retrained models. Fig. 20 shows the results of using the extended RCU barrier and the remote locks. From the results, we observe that the extended RCU barrier significantly improves the system performance when inserting a large number of data, due to not blocking the systems when updating the models.

To show the benefits of the proposed pipeline operations, Fig. 21 shows the performance of different schemes to conduct index operations on the lognormal dataset, including the schemes that purely conduct all operations on clients and transfer data requests to servers. From the results, we observe that compared with the purely on-client scheme, the pipeline operations reduce the network roundtrip penalty due to transferring the data requests to the servers. Specifically, the purely on-client schemes require multiple network roundtrips to determine the data locations, and require multiple network roundtrips to guarantee the data consistency among different machines. Compared with the purely on-server scheme, the pipeline operations alleviate the computing bottleneck on the servers due to offloading some computing tasks onto the clients. Specifically, the purely on-server scheme transfers all operations to the servers and waits for the replies of the server. Unlike them, the pipeline operations

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

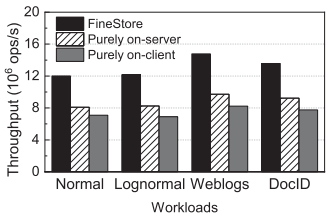


Fig. 21. Benefit analysis for the proposed FineStore.

conduct the prediction operation on the clients, while the server only need to conduct further operations in the predicted range.

VI. Related Work

The Learned Structures for Memory Systems: The learned index [15] leverages the powerful calculations to replace the traditional expensive memory consumption. To support the insertion operation, ALEX [21] reserves the slots for new inserts and synchronously allocates a new data array when there are no enough slots. PGM-index [22] obtains the temporal and spatial trade-off via an optimal number of linear models. FITing-tree [20] uses B^+-tree as a buffer to process the inserts. XIndex [23] uses the concurrent Masstree [9] as the delta-buffer and concurrently compacts the buffer with the trained model at runtime. Unlike them, RadixSpline [35] builds the index structure fast, as well as showing efficient lookup performance. SOSD [36], [37] and CDFShop [38] show the advantages of learned structures over tree-based structures. Instead of using the workload-driven approach, DeepDB [39] proposes a new data-driven approach for learned DBMS. In the KV systems, BOURBON [40] coalesces the learned index with the LSM-based key-value store to deliver high performance. XSTORE [24] leverages the learned index to improve the performance of network-attached in-memory key-value store. Moreover, Tsunami [41] achieves efficient search performance by using learned multi-dimensional indexes, while LISA [42] learns the spatial data.

Tree-Based Structures for Memory Systems: Traditional tree-based structures have been implemented with the support of hardware, including cache, SIMD and GPUs [6], [7], [7], [43], [44]. B $^\epsilon$-tree [32] improves write performance via asynchronous writes to disks with less I/Os. Masstree [9] uses fine-grained locks to provide concurrent operations. Wormhole [45] replaces the inner nodes of B^+-tree with a hash-table encoded Trie to process the variable lengths of keys. μ Tree [46] shows low tail latency than other tree-based schemes on persistent memory. Several schemes focus on compressing indexes to reduce the sizes of keys via prefix/suffix truncation, dictionary compression and key normalization [47], [48].

VII. Conclusion

In this article, we propose a fine-grained learned index scheme for concurrent and distributed systems, called FineStore. To achieve the scalability, the inserts are processed in the level bins under each trained data. Moreover, FineStore concurrently adapts to the new data distribution with non-blocking retraining, as well as ensuring the data consistency. Our experimental results show that FineStore respectively improves the performance by up to 1.8× and 2.5× over the learned-based and tree-based structures. We have released the source codes for public use in GitHub.

References

[1] P. Li, Y. Hua, J. Jia, and P. Zuo, "FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems," in *Proc. VLDB Endowment*, vol. 15, no. 2, pp. 321–334, 2022.
[2] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.
[3] H. H. Chan et al., "HashKV: Enabling efficient updates in kv storage via hashing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 1007–1019.
[4] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du, "BloomFlash: Bloom filter on flash-based storage," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 635–644.
[5] G. Graefe and P.-A. Larson, "B-tree indexes and CPU caches," in *Proc. 17th Int. Conf. Data Eng.*, 2001, pp. 349–358.
[6] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 78–89.
[7] C. Kim et al., "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 339–350.
[8] Y. Sun, Y. Hua, Z. Chen, and Y. Guo, "Mitigating asymmetric read and write costs in cuckoo hashing for storage systems," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 329–344.
[9] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.
[10] M. Inc., "MongoDB," 2021. [Online]. Available: https://www.mongodb.com/
[11] I. Inc., "IBM DB2," 2020. [Online]. Available: https://www.ibm.com/analytics/db2
[12] S. Ghemawat and J. Dean, "LevelDB," 2011. [Online]. Available: https://github.com/google/leveldb
[13] T. P. G. D. Group, "PostgreSQL," 1996–2021. [Online]. Available: https://www.postgresql.org/
[14] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1567–1581.
[15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 489–504.
[16] T. Wilcox, N. Jin, P. Flach, and J. Thumim, "A Big Data platform for smart meter data analytics," *Comput. Ind.*, vol. 105, pp. 250–259, 2019.
[17] D. Borthakur, "Petabyte scale databases and storage systems at Facebook," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1267–1268.
[18] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at Facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.
[19] Symas lightning memory-mapped database, 2017. [Online]. Available: http://www.lmdb.tech/doc/
[20] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1189–1206.
[21] J. Ding et al., "ALEX: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 969–984.
[22] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," in *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020.
[23] C. Tang et al., "XIndex: A scalable learned index for multicore data storage," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 308–320.
[24] X. Wei, R. Chen, and H. Chen, "Fast RDMA-based ordered key-value store using remote learned cache," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 117–135.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.

[25] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *Proc. Int. Conf. Learn. Representations*, 2017.
[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be general and fast," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, Boston, MA, 2019, pp. 1–16.
[27] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, "One-sided RDMA-conscious extendible hashing for disaggregated memory," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 15–29.
[28] S. K. Cha, S. Hwang, K. Kim, and K. Kwon, "Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems," in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, pp. 181–190.
[29] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b tree index on disaggregated memory," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1033–1048.
[30] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the cell distributed b-tree store," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 451–464.
[31] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast RDMA-capable networks," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 741–758.
[32] M. A. Bender et al., "And introduction to be-trees and write-optimization," *Login; Mag.*, vol. 40, no. 5, 2015.
[33] S. Meyn, *Control Techniques for Complex Networks*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[34] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng, "Maximum error-bounded piecewise linear representation for online stream approximation," *VLDB J.*, vol. 23, no. 6, pp. 915–937, 2014.
[35] A. Kipf et al., "RadixSpline: A single-pass learned index," in *Proc. 3rd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, 2020, pp. 5:1–5:5.
[36] A. Kipf et al., "SOSD: A benchmark for learned indexes," in *Proc. NeurIPS Workshop Mach. Learn. Syst.*, 2019.
[37] R. Marcus et al., "Benchmarking learned indexes," in *Proc. VLDB Endowment*, vol. 14, no. 1, pp. 1–13, 2020.
[38] R. Marcus, E. Zhang, and T. Kraska, "CDFShop: Exploring and optimizing learned index structures," in *Proc. Int. Conf. Manage. Data*, 2020, pp. 2789–2792.
[39] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: Learn from data, not from queries!," in *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 992–1005, 2020.
[40] Y. Dai et al., "From wisckey to bourbon: A learned index for log-structured merge trees," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 155–171.
[41] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," in *Proc. VLDB Endowment*, vol. 14, no. 2, pp. 74–86, 2020.
[42] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2119–2133.
[43] J. Rao and K. A. Ross, "Making b-trees cache conscious in main memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 475–486.
[44] A. Shahvarani and H.-A. Jacobsen, "A hybrid b-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1523–1538.
[45] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.
[46] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "tree: A persistent b - tree with low tail latency," in *Proc. VLDB Endowment*, vol. 13, no. 11, pp. 2634–2648, 2020.
[47] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, "Efficient in-memory indexing with generalized prefix trees," *Database Syst. Bus. Technol. Web*, vol. 180, pp. 227–246, 2011.
[48] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *Proc. 14th Int. Conf. Data Eng.*, 1998, pp. 370–379.

Pengfei Li received the BS and PhD degrees in computer science and technology from the Huazhong University of Science and Technology, in 2017 and 2023, respectively. His research interests include learned indexes for learned systems, data deduplication techniques, and network-attached disaggregated architecture.

Yu Hua (Senior Member, IEEE) received the BE and PhD degrees, respectively, in 2001 and 2005. He is a professor with the Huazhong University of Science and Technology. His research interests include cloud storage systems, file systems, non-volatile memory architectures, etc. His papers have been published in major conferences and journals, including *IEEE Transactions on Parallel and Distributed Systems*, OSDI, FAST, MICRO, ASPLOS, VLDB, USENIX ATC, SC, HPCA. He serves as PC (vice) Chairs in ICDCS 2021, ACM APSys 2019, and ICPADS 2016, and PC or ERC in OSDI, FAST, ASPLOS, ISCA, MICRO, HPCA, USENIX ATC, EuroSys, SC. He is the distinguished member of CCF, and senior member of ACM. He has been selected as the distinguished speaker of ACM and CCF.

Jingnan Jia received the BS degree from the Wuhan University of Technology, in 2018, and the master's degree from the Huazhong University of Science and Technology, in 2021. His research interests include learned indexes and systems, and non-volatile memory systems.

Pengfei Zuo received the BS and PhD degrees in computer science and technology from the Huazhong University of Science and Technology respectively, in 2014 and 2019. His papers have been published in major conferences and journals (OSDI, MICRO, ASPLOS, FAST, USENIX ATC, VLDB, DAC, etc) in the areas of computer system and architecture, with a focus on memory systems, storage systems and techniques. He served as a PC member with multiple conferences including ICDCS 2021, Cloud 2021, ICPADS 2020, etc, and a reviewer for multiple journals including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *ACM Transactions on Storage*, *IEEE Transactions on Dependable and Secure Computing*, etc.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on May 02,2024 at 02:23:48 UTC from IEEE Xplore. Restrictions apply.