FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems

Pengfei Li, Yu Hua, Jingnan Jia, Pengfei Zuo
WNLO, Huazhong University of Science and Technology
{cspfli,csyhua,jingnanjia,pfzuo}@hust.edu.cn

ABSTRACT

Index structures in memory systems become important to improve the entire system performance. The promising learned indexes leverage deep-learning models to complement existing index structures and obtain significant performance improvements. Existing schemes rely on a delta-buffer to support the scalability, which however incurs high overheads when a large number of data are inserted, due to the needs of checking both learned indexes and extra delta-buffer. The practical system performance also decreases since the shared delta-buffer quickly becomes large and requires frequent retraining due to high data dependency. To address the problems of limited scalability and frequent retraining, we propose a FINE-grained learned index scheme with high scalability, called FINEdex, which constructs independent models with a flattened data structure (i.e., the data arrays with low data dependency) under the trained data array to concurrently process the requests with low overheads. By further efficiently exploring and exploiting the characteristics of the workloads, FINEdex processes the new requests in-place with the support of non-blocking retraining, hence adapting to the new distributions without blocking the systems. We evaluate FINEdex via YCSB and real-world datasets, and extensive experimental results demonstrate that FINEdex 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 FINEdex for public use in GitHub.

PVLDB Reference Format:
Pengfei Li, Yu Hua, Jingnan Jia, Pengfei Zuo. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. PVLDB, 15(2): 321-334, 2022.
doi:10.14778/3489496.3489512

1 INTRODUCTION

Data storage and access performance are important for memory systems, which however are exacerbated by the explosive growth of data. Existing index structures, such as B^+-tree [16], Hash-map [10], and Bloom filters [18], usually support in-memory systems to handle data processing tasks [22, 27, 39] in a memory-efficient manner over the past decades [21, 27, 34, 40, 46, 52].

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., Redis [48], MongoDB [32]), IBM DB2 [31], LevelDB [25] and PostgreSQL [28], construct tree-based structures to provide efficient data storage and access. Although maintaining all data and metadata completely in the main memory eliminates expensive disk I/O operations [13, 56], the high space overhead becomes exacerbated once the index structures are too large to fit into the limited-size memory. In fact, the indexes, e.g., tree-based structures, consume around 55% of the total memory in state-of-the-art memory systems [58].

In order to improve the performance and reduce the memory overhead, the powerful hardware is used to improve the performance of B^+-tree, including cache [46, 47], SIMD [34] and GPUs [33, 34, 50]. In the meantime, some compression schemes leverage the prefix/suffix truncation, dictionary compression and key normalization techniques [5, 26, 46] to save the space. The approximate structures [3, 24, 37] are also proposed to reduce the memory overhead of the B^+-tree.

However, all above schemes are designed for general-purpose data structures and mainly focus on the index structures themselves, while overlooking the patterns of data distribution in memory systems. Kraska et al. [37] argue that exact data distribution enables efficient optimization 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 100M), 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, in real-world applications and systems (e.g., processing the data of smart devices [55], the petabyte scale storage systems of Facebook [6, 8] and LMDB [1]), some patterns are extremely complex or even impossible to be represented via known patterns. Hence, we consider machine learning (ML) approaches to learn a model that exhibits the patterns of data distribution, called ***learned indexes*** [37].

The learned indexes open up a new research topic on indexing in memory systems: ***Indexes can be considered as ML models***. We use cost-efficient computations to speed up traditional comparisons, thereby increasing access speed and saving memory space. Moreover, in order to efficiently exploit the benefits of multi-core processors, we carry out concurrent operations to deliver high performance. 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 memory systems due to the following challenges.

1) Limited Scalability. The scalability requires the learned indexes to efficiently handle inserts and adjust to the new data distribution at runtime, as well as scaling to multiple threads for high concurrent performance. However, existing schemes show

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489512

limited scalability since they do not simultaneously meet all these requirements, including concurrent reading, writing and retraining. For example, FITing-tree [24], ALEX [19] and PGM-index [23] do not consider the data consistency issues to concurrently retrain the models in the multi-core systems. XIndex [53] stores the data into different data structures, hence not keeping all data sorted for efficient range query performance.

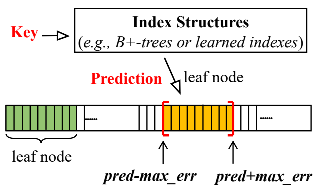
*2) **High Overheads.*** The strategies adopted by existing schemes incur high overheads due to the heavy data dependency. Specifically, XIndex [53] and FITing-tree [24] handle inserts through a *delta-buffer*, which is a tree-based structure [11, 40] (e.g., B$^+$-tree or Masstree) and has high dependency of inner nodes when traversing the tree. Moreover, XIndex [53] shows that the performance decreases about 3× when the delta-buffer becomes large, due to checking these two different structures in each index operation. ALEX [19] and PGM-index [23] preserve empty slots in the trained data arrays to handle inserts. When scaling to multiple threads, many thread collisions occur, since different threads need to put the data into the same slot during insertion, which decreases the concurrent performance.

In order to address these challenges, we present a fine-grained learned index scheme for scalable and concurrent memory systems, called FINEdex. Our proposed FINEdex achieves high scalability by appending the low-overhead *level bins* under each trained data to alleviate the data dependency, rather than building a large shared delta-buffer. By using such flattened structure, FINEdex 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. For the dynamic workloads, FINEdex adaptively assigns models according to the data distribution and adjusts to the new data distribution at runtime without blocking the system. We have integrated FINEdex into Redis [48] to evaluate the performance of real implementations. Our experimental results show that FINEdex 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.

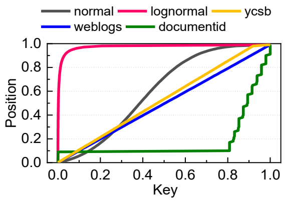
It is worth noting that the mentioned models are interpreted as linear regression ML models with bounded prediction errors, which predict the positions of the keys. To ensure that no data are lost in the system, the bounded prediction errors are determined by the data that is farthest from the central 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 [24, 37, 53].

In this paper, we have the following contributions.

- ***High scalability meeting system requirements.*** We present a fine-grained learned index scheme for concurrent memory systems, i.e., FINEdex, which efficiently meets the scalability requirements. The main insights are reducing the data dependency via the flattened data structure and concurrently retraining the models in two granularities.
- ***Low overheads for cost-efficient index operations and non-blocking retraining.*** The index operations (i.e., read and write) are cost-efficient, since FINEdex incurs a few data movements during insertion and keeps all data sorted for accessing. In the concurrent systems, FINEdex alleviates the thread collisions by reducing the data dependency, as well as concurrently retraining models without blocking the system.
- ***System Implementation and Evaluation.*** We implement and evaluate FINEdex[1] based on Redis [48]. Compared with state-of-the-art schemes, our experimental results show that FINEdex obtains high insertion performance, as well as the high search performance after inserts.



(a) Range Index Model



(b) CDF models

Figure 1: Range index and CDF models. ***The CDFs are obtained by normalizing the keys and positions to 1.***

The rest of this paper is organized as follows. Section 2 introduces the background. In Section 3, we demonstrate the different components of FINEdex. Section 4 describes the system implementations. Section 5 shows the experimental results and analysis. Section 6 discusses the related work, and Section 7 concludes our paper.

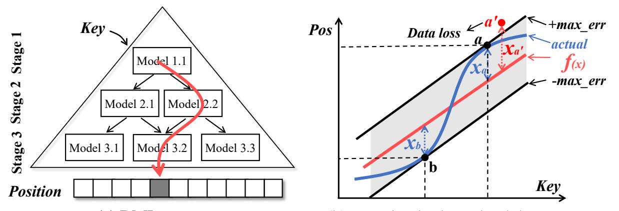
2 BACKGROUND AND MOTIVATION

2.1 New Perspectives on Indexes

From the perspective of machine learning, the range index structures are considered as regression models [37], which predict the position of a given key, as shown in Figure 1a. In the B$^+$-tree, the data are found through traversing the tree. A learned index [37] 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. We term this length as ***prediction granularity***.

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) [37, 53], as shown in Figure 1b. The datasets used in Figure 1b are the same as those used in Section 5. Based on this observation, the prediction accuracy can be improved by learning the patterns of data distribution. When the CDF between keys and positions is accurately represented via the known regression models, the lookup complexity becomes $O(1)$ since each position is calculated by the regression models. For example, a set of continuous integer keys (e.g., the keys from 1 to 100M) are stored in a piece of continuous positions (e.g., the positions from 1 to 100M). The CDF is represented as $y = x$, where x and y are keys and positions. Thus, the prediction granularity is 1 to accurately predict the positions without any errors. However, in real-world applications (e.g., managing the data

[1]The source code is available at https://github.com/iotlpf/FINEdex



(a) RMI structure (b) Insertion in the trained data array

Figure 2: The RMI structure in the learned indexes.

Table 1: The limited scalability of existing schemes.

Schemes	Insertion without data loss	keep all data sorted	concurrency	
			write	retrain
Learned index [37]	✗	✓	✗	✗
FITing-tree [24]	✓	✗	✗	✗
XIndex [53]	✓	✗	✓	✓
ALEX [19]	✓	✓	✗	✗
PGM-index [23]	✓	✓	✗	✗
FINEdex	✓	✓	✓	✓

of smart meters [55], the petabyte scale storage systems of Facebook [6]), the CDFs can't be obtained in advance and some CDFs become extremely complex or even impossible to be represented via known regression models [37]. In these situations, we don't need to reduce the prediction granularity to 1, since the length of the leaf node in B$^+$-tree has never been set to 1, which simplifies the prediction operations: regression models only need to approximately represent the CDF and reduce the prediction granularity to the same size like the leaf nodes in the B$^+$-tree.

Using a single ML model to reduce the prediction granularity (e.g., from 100M to 10) 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) [37, 51] to improve the prediction accuracy, which gradually reduces the prediction granularity from 100M to 10K, then from 10K to 100, 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 [37]. As shown in Figure 2a, 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 training 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 granularity $[pred - max\_err, pred + max\_err]$ if the picked ML model is valid, otherwise searching the B$^+$-tree.

The learned indexes [37] implement a 2-stage RMI index with a small neural network (NN) on the top and a large amount of linear regression models at the bottom. In the learned indexes, a simple (0 hidden layer) to semi-complex NN model (2 hidden layers) on the top becomes more efficient than other configured NN models (i.e., more hidden layers). It is not cost-efficient to execute complex models at the bottom since the simple linear regression models are accurate enough to learn the small subdatasets.

2.2 The Scalability Requirements

The learned indexes [37] are trained on the statically distributed data and assume that the data are uniformly accessed with a single thread, which do not consider the case that the data distribution changes along with time. It is important for the learned indexes to meet the following requirements to support high scalability.

Insertion without data loss. The foundational requirement for insertion is to guarantee that all data can be found, including the data that is farthest from the central functions (i.e., trained models). For the learned indexes, the new data should not be directly inserted into the trained data array to avoid the error that some data can not be found due to the data movements. For example, as shown in Figure 2b, the blue line represents the actual data distribution, the red line represents one of the linear regression models, and the black points are the data covered by this model. Since *max_err* (i.e., the max error of the model) is calculated via Equation 1, the error x_a of point a meets the condition:

$$abs(x_a) \leq max\_err$$

where $abs()$ returns the absolute value. Point a is found by the model since the true position meets the condition:

$$a \in [pred\_a - max\_err, pred\_a + max\_err]$$

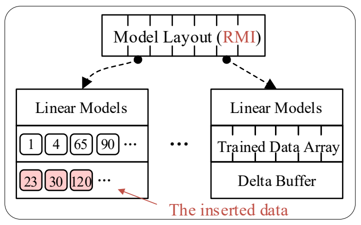
where $pred\_a$ represents the prediction result of a. Not all the covered points can be found when there are some newly inserted data. For example, when the inserted data is smaller than a, we need to move point a to a' to keep all data sorted, thus leading to an error:

$$a' \notin [pred\_a' - max\_err, pred\_a' + max\_err]$$

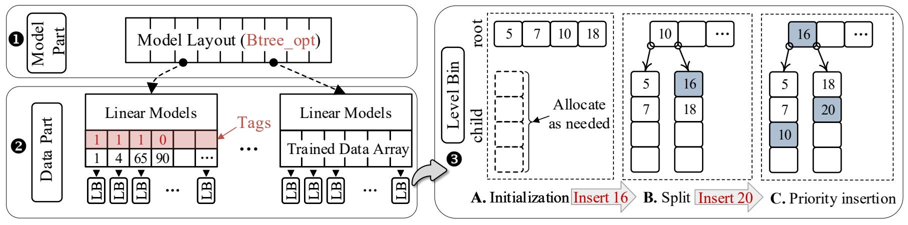
due to the new error $abs(x'_a) > max\_err$.

Keep all data sorted for efficient range query. 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, which decreases the range query 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. We also need to avoid the data inconsistency, i.e., no data are lost or redundant. However, it is non-trivial to concurrently retrain the learned indexes, since



(a) The structure of XIndex



(b) The structure of FINEdex

Figure 3: The structures of XIndex and FINEdex. ***FINEdex consists of model and data parts.***

the retraining consumes a long time to block other operations on resorting and retraining the data.

2.3 The Limited Scalability of Existing Schemes

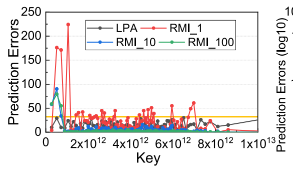
Various learned indexes leverage different strategies to support scalability, including FITing-tree [24], ALEX [19], PGM-index [23] and XIndex [53], which however show limited scalability, as shown in Table 1. Specifically, FITing-tree and XIndex handle inserts in the delta-buffers. Their differences are that XIndex uses a concurrent delta-buffer (i.e., Masstree [40]) and supports concurrent retraining, as shown in Figure 3a. Although handling inserts in the delta-buffer won't affect the trained data and guarantees the data correctness during insertion, such design is inefficient due to storing the data in two different structures. XIndex [53] 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.

ALEX and PGM-index preserve 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. In this way, ALEX and PGM-index ensure the data correctness during insertion, as well as keeping all data sorted for high range query performance. However, such design is inefficient when scaling to multiple threads, since different threads compete for the shared empty slots during insertion. Moreover, when there are insufficient empty slots, ALEX and PGM-index expand the trained data array, redistribute the data into the new trained data array and retrain new models. 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.

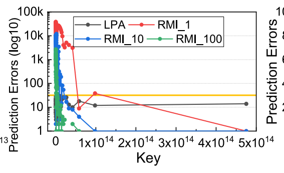
3 THE FINEDEX DESIGN

In this section, we present the design of FINE-grained scalable learned index, or FINEdex, 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, FINEdex 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 Figure 3b. 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 FINEdex are independent. Through these designs, FINEdex achieves high concurrent performance using multiple threads.

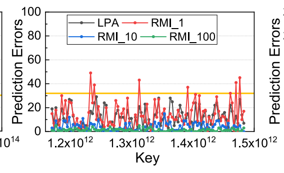
To clearly present our approach, we divide our design into the model and data parts, as shown in Figure 3b. For the model part, we train independent models and optimize the model layout (i.e., the organization structure of the models) to improve the performance (Section 3.1). The independent models are flexible to adjust to the new data distributions through concurrent retraining. For the data part, we handle inserts in the level bins (Section 3.2), which are two-level sorted arrays under each trained data and support low-overhead retraining. For the case that the data distribution dynamically changes, FINEdex efficiently adjusts the structures to fit the new data distribution at runtime, as well as ensuring the data consistency (Section 3.3). Moreover, FINEdex supports the practical operations (e.g., search, update, insert, and remove) with low overheads, as shown in Section 3.4. In concurrent systems, FINEdex provides high concurrent performance due to the few thread collisions caused by the flattened data structures (Section 3.5).



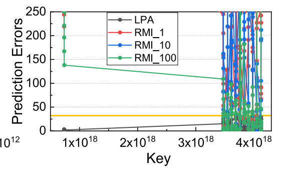
(a) Normal distribution.



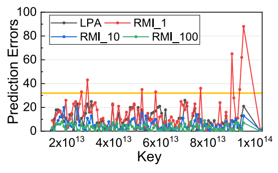
(b) Lognormal distribution.



(c) Weblogs.

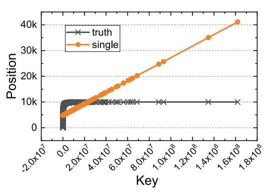


(d) DocumentID.

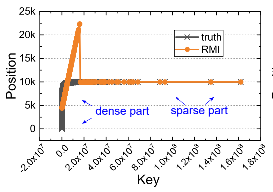


(e) YCSB zipfian.

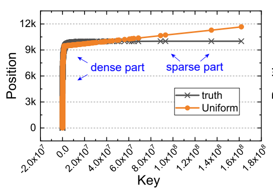
Figure 4: The prediction errors on different workloads. ***RMI_# represents that the model number in the 2nd stage of RMI is # times than that in the LPA. The horizon yellow line represents the predefined threshold.***



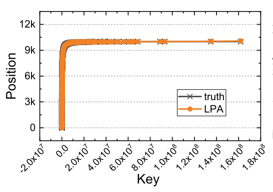
(a) Single model.



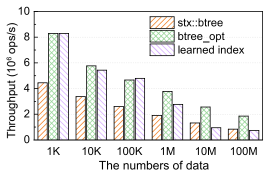
(b) RMI learning.



(c) Uniform learning.



(d) Learning probe algorithm.



(e) Search throughput.

Figure 5: ***(a-d)*** **shows the learning effects of various algorithms.** ***(e)*** **shows the search throughput of different schemes.**

3.1 Model Part

We propose Learning Probe Algorithm (LPA) to train independent models, as well as optimizing the model layout (i.e., the organization structure of the models) to access the models.

3.1.1 Improving the Model Accuracy.

A model with high accuracy incurs a low model error, i.e., the farthest distance between the prediction result and the real position, which is represented as ϵ in this paper. Existing schemes [19, 37, 53] construct the RMI structure to train models. However, the model accuracy is tightly related with the data distributions of training models. In fact, it is non-trivial to efficiently train sufficient models with small model errors [7, 12, 14, 45, 57].

Unlike them, we present Learning Probe Algorithm (LPA) to adaptively train models according to the data distribution, which ensures that all model errors are smaller than the given threshold. The idea is to find the parallelogram of 2ϵ width in the vertical direction such that no trained data are placed outside of the parallelogram, as shown of the gray block in Figure 2b. We thus obtain the linear regression model by using the line that intersects the two vertical sides of the parallelogram and bisects the parallelogram. We first show the drawbacks of RMI and then show more details of LPA in Section 3.1.2.

To show the model accuracy on different distributions, we examine RMI on various workloads, and the CDFs of these workloads are shown in Figure 1b. In this experiment, we implement a 2-stage RMI [37] and train more models for the second stage than that in LPA. We train all models on 10^6 keys and set the maximum threshold of model errors to 32, where 32 is a suitable trade-off between the prediction accuracy and the model numbers based on the evaluation results. In general, a small threshold provides high prediction accuracy but requires a large number of models, which is time/space-consuming to search/store these models. The results of the prediction errors (i.e., the distances between the prediction results and the real positions) are shown in Figure 4, and we have the following observations.

(OB#1) *RMI requires a large number of models to improve the accuracy, depending on the data distribution.* The RMI structure fails to learn the data distribution well like LPA with the same number of models, since many prediction errors exceed the predefined threshold. In some cases, in order to improve the accuracy, RMI has to use more than two-orders-of-magnitude models than LPA, as shown in Figures 4b and 4d.

(OB#2) *The model accuracies become diverse even in the same distribution.* RMI poorly performs on lognormal distribution with a small number of models, as shown in Figure 4b. To gain more insights, we further examine different schemes on learning 10K lognormal distributed data with 10 models (since 10 models are enough to show the learning effects of different schemes), and the results are shown in Figures 5(a-d).

As shown in the yellow lines in Figure 5a, we observe that it is impossible to perfectly represent the lognormal distribution by only using one regression model, since the distributions in real-world applications are more complex than the linear distribution [37]. We then examine RMI to learn the data distribution, which partitions the dataset based on the prediction results of the previous stage. Formally, each ML model is essentially treated as a mathematical function $f(x)$, in which x is the given lookup key. If we use $f_l(x)$ to denote ML models in different stages, the calculation process is described as follows:

$$f_l(x) = f_l^{(\lfloor M_l f_{l-1}(x)/N \rfloor)}(x) \qquad f_1(x) = y \tag{2}$$

where x represents the input, N represents the number of positions in stage l, $y \in (0, M_2]$ represents the prediction result of the first model, and there are M_l models in stage l. The idea to select the next model is normalization, represented as $\lfloor M_l f_{l-1}(x)/N \rfloor$. This

Algorithm 1: LPA Algorithm

```
Input: int $threshold$, int $learning\_step$, float $learning\_rate$, dataType $record[N]$
Output: trained $FINEdex$
while not reach the end of the dataset record[N] do
    add $learning\_step$ data into dataset $S$ from $record$;
    train a linear regression $model$ on $S$;
    $error$ = max($|min\_error|$, $|max\_error|$);
    while $error < threshold$ do
        add next $learning\_step$ data into dataset $S$ from $record$;
        train a new $model$ on $S$;
    end
    while $error > threshold$ do
        $step$=int($learning\_step * learning\_rate^n$);
        remove $step$ data from the end of dataset $S$;
        train a new $model$ on $S$;
    end
    $FINEdex$.append($model$);
    clean data from dataset $S$ for next probing;
end
```

formulation represents that the model $f_l(x)$ to be used in stage l is based on the results of model $f_{l-1}(x)$ in stage $l-1$.

The results of using learned indexes to learn the CDF are shown in Figure 5b. We find that the accuracy of each model varies significantly depending on the data distribution. For example, the densely distributed data are not well learned while it is much better for the sparse part. Densely distributed data are likely to be divided into the same subdataset according to Equation 2, even if these data are not linearly distributed, resulting in poor learning accuracy. Increasing the number of models allows these densely distributed data to be partitioned into multiple subdatasets, thus allowing more models to be used to improve the accuracy. However, adding more models also divides the well-learned parts into more subdatasets according to Equation 2, resulting in these added models to be redundant, since these well-learned parts have the similar linear patterns. Moreover, we have no prior knowledge of the data distribution, which increases the difficulty for configuring the number of models.

Moreover, we uniformly partition the dataset so that each subdataset has the same amount of data and the results are shown in Figure 5c. This strategy improves the learning accuracy for densely distributed data since these data are divided into multiple subdatasets and can be learned by more models. However, this strategy reduces the learning accuracy for sparsely distributed data, since we have to add some data from densely distributed data into sparse parts to achieve the same amount of data, even if these data are not linearly distributed.

The two strategies are inefficient to learn CDF well, since the two methods can't adaptively configure models according to the data distribution, which motivates us to propose the learning probe algorithm (LPA). As shown in Figure 5d, LPA learns the CDF better than the previous strategies, since LPA partitions the dataset according to the data distribution. Only the same linearly distributed data are divided into the same subdataset that is easy to learn by a linear regression model.

3.1.2 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 Equation 1. The prediction error of the obtained model determines the next operation of the probe. If $error < threshold$, the probe keeps moving forward 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:

$$small\_step = learning\_step * learning\_rate^n \tag{3}$$

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 FINEdex 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, as shown in Figure 4. The main reason is that LPA trains data according to the data distribution and only the model whose prediction error is not larger than *threshold* can be appended to FINEdex, while RMI fails. The max-error of each obtained model is controlled by the predefined parameter *threshold*.

3.1.3 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 FINEdex trains independent models to enable high scalability. However, the obtained piecewise models need to be checked via one-by-one comparisons, which delivers low search performance in a poor model layout.

To obtain an efficient model layout, we compare the search performance among the learned index [37] (2-stage RMI with $10K$ models in the second stage), stx::btree [4] (using default configuration as the original implementation) and the cache-/SIMD-optimized btree (i.e., align the btree node with cacheline and search the node with SIMD instructions, represented as *btree_opt*) on different datasets. As shown in Figure 5e, we observe that ***(OB#3)*** *the learned index is*

Table 2: The 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
	RMI	250,000	250,000	250,000	250,000	250,000

not always the best choice. For example, the learned index trains $10K$ models on $1K$ data to achieve competitive search performance with *btree_opt* when the number of data is small. However, the learned index delivers low performance when failing to train enough models(e.g., $10K$ models on $100M$ data). Moreover, the btree supports scalability while the learned indexes fail.

The total number of models trained by LPA is small, as shown in Table 2. Hence, we store the piecewise models as *btree_opt* to enable system 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 250K models in RMI achieve the best performance, we also configure 250K models to facilitate fair comparisons.

3.2 Data Part

We propose to use a flattened data structure to store the new data, which simultaneously meets the following design principles.

No data loss. The trained data array is not used for processing inserts to avoid the data-loss errors in Figure 2b.

Keep all data sorted for range query. One drawback of the shared delta-buffer is the data overlapping between the trained data array and delta-buffer, which hinders the range query performance. Unlike it, FINEdex processes inserts in the structure under each trained data without data overlapping, hence keeping all data sorted during insertion.

Alleviate the data dependency for high concurrency. FINEdex alleviates the data dependency via the flattened data structure, i.e., each trained data has its own small-sized buffer to process the modifications, rather than sharing one big buffer with tens of thousands of trained data. Moreover, we bound the buffers to small sizes at runtime via fine-grained retraining to keep low data dependency. Such designs significantly reduce the thread collisions and improve the concurrent performance as shown in our experimental evaluations.

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 Figure 3b. 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 prioritized 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, Figure 3b 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. Moreover, to insert 20, we move existing data forward to the first child bin to keep all data sorted. In this case, we move at most $(n+1)$ data, where n is the length of the child bin. When the previous child bin 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 FINEdex 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 [37], and hence are inserted evenly into all level bins. In this case, FINEdex handles nearly $(m * n)$ times more than the trained data. When the data distribution changes, FINEdex concurrently retrains the level bins to fit the new data distribution, as shown in Section 3.3.

3.3 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, FINEdex 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 [37], which blocks the systems for a long time. As shown in Figure 6, 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 Figure 6, 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, FINEdex 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.

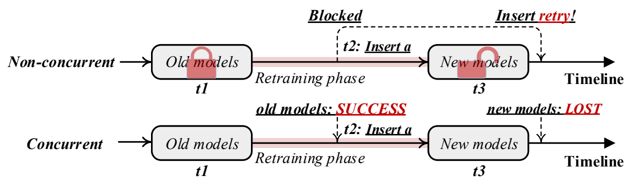


Figure 6: The challengs of different retraining strategies.

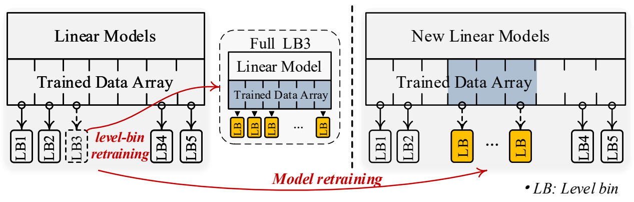


Figure 7: Concurrent retraining. *Level-bin retraining retrains full bins. Model retraining merges the small models.*

3.3.1 *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 Figure 7.

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., 27μs in our experiments), since 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.

3.3.2 *Model Retraining.*

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

As shown in Figure 7, FINEdex 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 3.2. 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 newly trained data array. After the new models are retrained, FINEdex uses the RCU-barrier [53] 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, FINEdex 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. Moreover, the overall data distribution is convergent with the growth of data according to the Law of Large Numbers [30]. Hence, FINEdex gradually adapts to the new data distribution along with time.

3.4 Practical Operations

Search: Figure 3b 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**: FINEdex completes the search if finding the given key in the prediction range, otherwise FINEdex searches the level bins or the small models.

Insert: FINEdex 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 3.2 and 3.3.

Update: If a given key exists in the structure, FINEdex 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 Figure 3b, we use the tokens (i.e., 0 and 1) to indicate whether the trained data are removed, which avoids the data-loss errors in Figure 2. The data in the level bins are directly removed, since changing the data within the level bins won't affect the model accuracy.

3.5 Concurrency

Concurrent data structures become important to existing systems that scale to a large number of cores and threads. The thread collision probability of FINEdex is rather low due to the flattened data structure. The conflicts only occur when different threads concurrently write/write or read/write the bins under the same trained data. We use the version control [40] and allocate fine-grained locks to enable FINEdex to support concurrent operations.

3.5.1 *Write/Write Conflicts.*

The write/write conflicts occur when different threads modify the same trained data or the same bin. FINEdex 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 3.4), FINEdex 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. FINEdex 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, FINEdex 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, FINEdex decreases the conflict probability, since different threads that modify the level bins under different trained data don't block each other.

3.5.2 Read/Write Conflicts.

Instead of using the locks during reading, FINEdex uses the version control [9, 40] to ensure that the obtained data is consistent and latest. FINEdex 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, FINEdex 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. FINEdex 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.

4 REAL SYSTEM IMPLEMENTATIONS

Existing systems, such as NoSQL [32, 48], IBM DB2 [31], LevelDB [25] and PostgreSQL [28], construct tree-based structures to keep the data sorted for range queries. While supporting range queries, we coalesce FINEdex with the in-memory Redis [48] for efficient data access, where Redis has been widely used in the event-driven key-value store.

Our proposed FINEdex provides a scalable learned index scheme for in-memory systems. The coalescing design with Redis only involves the basic data structure of the sorted set. Other components (e.g., transaction, cluster, client and server, etc.) are not modified. The implementation of FINEdex provides an interface with 6 easy-to-use APIs, including *TRAIN, GET, PUT, UPDATE, REMOVE* and *SCAN*. Specifically, we implement the LPA algorithm in Redis to train the models. During the runtime, we allocate 4 retraining-purpose threads in the background to concurrently execute the *model retraining*, while the *level-bin retraining* is executed by the worker threads (i.e., the threads to execute the APIs). To prevent the un-retrained models affected by the retrained one due to data movements, the trained data arrays of different models are not stored together. FINEdex leverages the following three steps to complete the index operations (e.g., read and write), including searching the models, calculating the range and operating in the level bins, as demonstrated in Section 3.4.

Instead of using the binary searching in the prediction range and level bins, we optimize the search performance with SIMD instructions, i.e., Intel AVX2, which processes 256-bit data with one instruction. FINEdex is efficient to leverage SIMD, due to searching the data in a short continuous memory (e.g., small prediction range, and small level bins), which meets the needs of Intel AVX2 [2].

5 PERFORMANCE EVALUATION

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

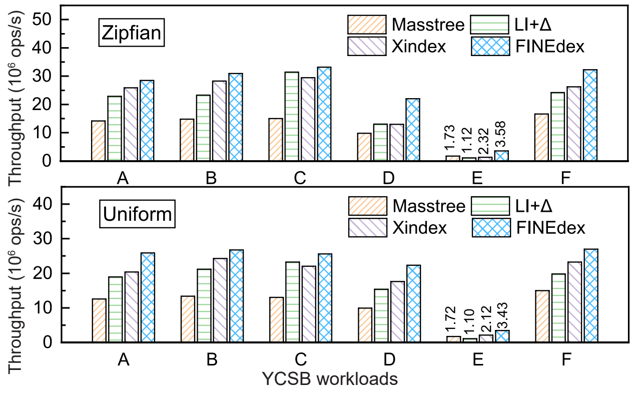
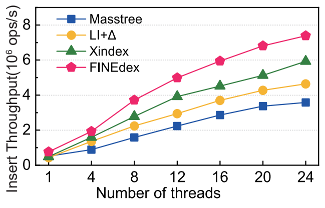


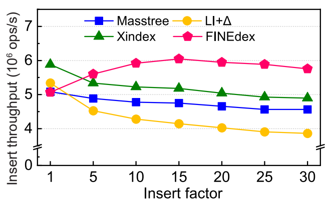
Figure 8: Throughputs on YCSB with various workloads.

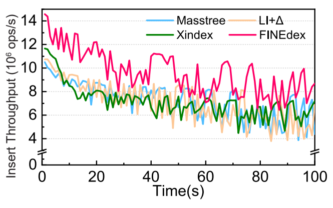
Counterparts for Comparisons. We compared our proposed FINEdex with state-of-the-art schemes. For the tree-based structures, we compare FINEdex with Masstree [40], which is a variant of scalable concurrent B^+-tree. The sorted set in Redis is not included, since it is implemented as a skip list and has the similar data access performance with tree-based structures [48]. Due to different design goals, B^ϵ-tree [44] 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 the original learned index [37] to support scalability by adding a delta-buffer (denoted as LI+Δ), where the buffer is implemented as a Masstree [40]. We compare FINEdex with XIndex [53] and LI+Δ [37], 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 [19, 23]. 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. The learned index is implemented with a 2-stage RMI following the original work [37], and the second stage configures 250K models like the setting in XIndex [53] to facilitate fair comparisons. In FINEdex, 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.

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


(a) Different numbers of threads.


(b) Different numbers of the data.


(c) The throughput timeline.

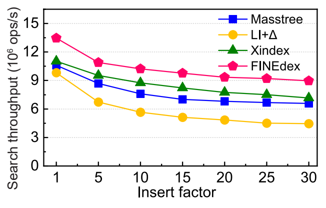

(d) Search throughput after inserts.

Figure 9: The scalability throughput in various scenarios, ***which are evaluated on the lognormal dataset.***

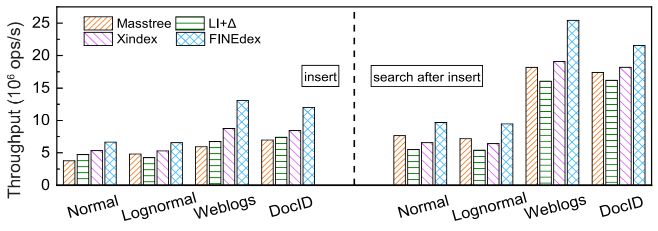


Figure 10: The throughputs on various workloads.

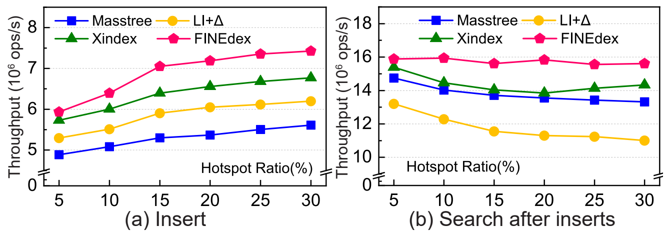


Figure 11: The throughputs on skewed workloads.

million instances in total. We also use 2 synthetic datasets with 200 million items to evaluate the behavior of FINEdex in depth: (4) *Normal* distribution with μ=4 and σ=2, and (5) *Lognormal* distribution with μ=0 and σ=2. All generated keys are scaled up to $[0, 10^{12}]$ as integers for evaluations. The CDFs of the used benchmarks are shown in Figure 1b. 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 [37, 53].

5.1 The Throughput via YCSB

Figure 8 shows the throughput of different schemes on YCSB with Uniform and Zipfian distributions. In general, FINEdex 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, FINEdex achieves comparable (even a little better) throughput than LI+Δ and XIndex, since FINEdex searches fewer models in the optimized layout and efficiently finds the data with higher model accuracy, as shown in Figures 4 and 5.

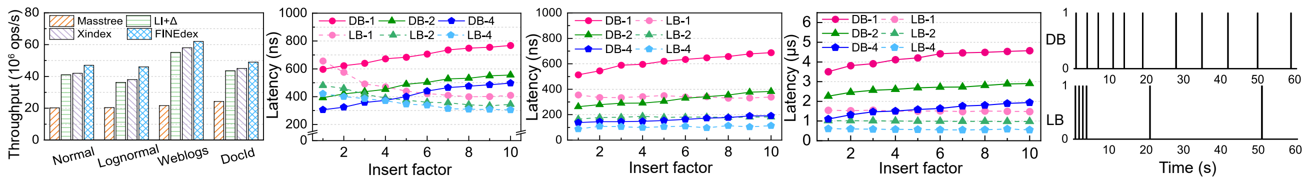
Dynamic workloads (YCSB D, E). FINEdex delivers higher throughput than other schemes on the dynamic workloads. Specifically, FINEdex outperforms LI+Δ, XIndex, and Masstree by 1.7×, 1.6×, and 2.3× on workload D. Because FINEdex incurs few data movements and has low-probability thread collisions during insertion, while LI+Δ, XIndex, and Masstree incur high overheads to traverse the trees. Moreover, FINEdex 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.

5.2 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 doesn't change during insertion. We also insert these sampled data into Masstree for fair comparisons.

The number of threads. Figure 9a shows the insert throughput with different threads. We observe that FINEdex 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. FINEdex obtains more performance improvements with more threads, since FINEdex 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. Figure 9b shows the throughput of inserting different numbers of data. We observe that the insert throughput of FINEdex is low at the beginning due to consuming time on allocating the level bins for each trained data. When inserting more data, FINEdex 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.



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

Figure 12: The 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.***

Insertion with frequent retraining. Figure 9c 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 FINEdex improves the insert throughput by about 1.8× over other schemes. Because FINEdex concurrently adapts to the new distribution by efficiently executing the level-bin retraining and model retraining.

5.3 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. Figure 9d 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 FINEdex also decreases after inserts, since the size of the level bins increases when we constantly insert data. However, FINEdex 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 Figure 10.

Different read/write ratios. Figures 13 and 14 respectively show the throughput and latency with various read/write ratios. We have the similar observations with previous evaluation results, i.e., FINEdex delivers high performance on both static and write-intensive workloads.

5.4 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. Figure 11 shows the insert and search throughputs on the skew workloads. We observe that both FINEdex and XIndex show low performance when the skewness is large, since more thread collisions occur and more retrainings are necessary. As the skewness decreases, FINEdex achieves higher performance than other schemes, due to retraining the data-intensive part and assigning a large amount of level bins. After several retrainings, FINEdex flattens the skewed data and adjusts to the new data distribution, thus decreasing the thread collisions.

5.5 In-depth Analysis for FINEdex

To examine where the performance improvements come, we leverage Control Variates [43] to evaluate different components of FINEdex, and the results are shown in Figure 12. In general, the most benefits come from the flattened data part and concurrent retraining.

Model part. Figure 12a 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 FINEdex 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. Figures 12b-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 Figure 12b, 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 Figures 12c and 12d, since the level bins keep all data sorted.

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

5.6 Overheads Analysis

5.6.1 Training Latency.

Figure 15 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 FINEdex 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 [57] only needs to traverse all data once during training. However, the learned index needs to traverse all data multiple times due to the level-by-level training strategy [37]. 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.

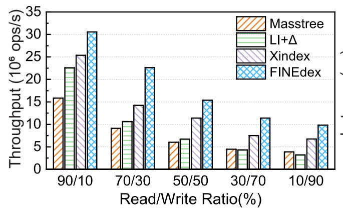


Figure 13: The throughput with various read/write ratios.

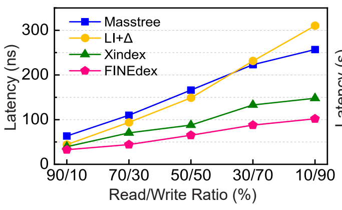


Figure 14: The latency with various read/write ratios.

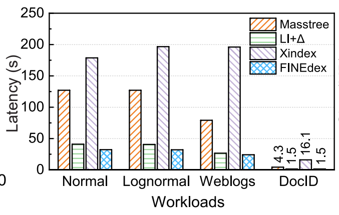


Figure 15: Training latency on various workloads.

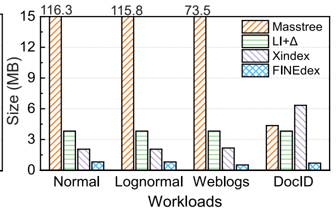


Figure 16: Memory overhead of models/inner nodes.

To dynamically adapt to the new data distribution, FINEdex performs retraining in two granularities, including level-bin retraining and model retraining. The level-bin retraining consumes 27μs to train the full level bins in our experiments. Although model retraining consumes more time (e.g., 1.5ms on 10K data), the latency doesn't affect other concurrent operations, since we perform the model retraining in background.

5.6.2 *Memory Overheads.*

The memory overheads consist of metadata and data parts. The metadata refers to the ML models in the learned structures and the inner nodes in tree-based structures, while the data refers to the real values. Different schemes require almost the same space to store the data due to the same amount of real values. Moreover, Section 3.2 shows that FINEdex allocates bins as needed and has a little higher maximum load factor than the tree-based structures to store the newly inserted data. Hence, FINEdex incurs the same (or a little lower) storage overhead for the data part.

The main difference in memory overhead of different schemes is the metadata overhead. As shown in Figure 16, we evaluate the sizes of ML models in the learned structures and the memory consumptions of inner nodes in Masstree for fair comparisons. From the results, we observe that all learned structures consume less memory than Masstree by up to two orders-of-magnitude, since one linear regression model is enough to index the same linearly distributed data. Moreover, FINEdex obtains more memory savings than the learned indexes and XIndex by up to 10× due to generating less models, as shown in Table 2.

6 RELATED WORK

The Learned Structures for Memory Systems. The learned index [37] leverages the powerful calculations to replace the traditional expensive memory consumption. To support the insertion operation, ALEX [19] reserves the slots for new inserts and synchronously allocates a new data array when there are no enough slots. PGM-index [23] obtains the temporal and spatial trade-off via an optimal number of linear models. FITing-tree [24] uses B$^+$-tree as a buffer to process the inserts. In practice, the needs for concurrency become increasingly important [49]. XIndex [53] uses the concurrent Masstree [40] as the delta-buffer and concurrently compacts the buffer with the trained model at runtime. Unlike them, RadixSpline [36] builds the index structure fast, as well as showing efficient lookup performance. SOSD [35, 41] and CDFShop [42] show the advantages of learned structures over tree-based structures. Instead of using the workload-driven approach, DeepDB [29] proposes a new data-driven approach for learned DBMS. In the KV systems, BOURBON [17] coalesces the learned index with the LSM-based key-value store to deliver high performance. XSTORE [54] leverages the learned index to improve the performance of network-attached in-memory key-value store. Moreover, Tsunami [20] achieves efficient search performance by using learned multi-dimensional indexes, while LISA [38] 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 [33, 34, 34, 46, 47, 50]. B$^\epsilon$-tree [44] improves write performance via asynchronous writes to disks with less I/Os. Masstree [40] uses fine-grained locks to provide concurrent operations. Wormhole [56] replaces the inner nodes of B$^+$-tree with a hash-table encoded Trie to process the variable lengths of keys. μTree [15] 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 [5, 26].

FINEdex achieves higher performance than other schemes in the case that there are intensive inserts with multiple threads, which however doesn't obtain significant performance improvements in the following cases. For the static workloads, i.e., the case that has no inserts, FINEdex achieves competitive search performance with other schemes, since the models of all learned structures keep high accuracy when there are no inserts. Moreover, PGM-index achieves more space-savings than FINEdex via the model compression techniques. For the disk-based storage system, LSM achieves higher performance than FINEdex due to the efficient sequential writes.

7 CONCLUSION

In this paper, we propose a fine-grained learned index scheme to exploit the concurrency benefits for scalable in-memory systems, called FINEdex. To achieve the scalability, the inserts are processed in the level bins under each trained data. Moreover, FINEdex concurrently adapts to the new data distribution with non-blocking retraining, as well as ensuring the data consistency. Our experimental results show that FINEdex 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.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202. Yu Hua is the corresponding author.

REFERENCES

[1] 2017. Symas lightning memory-mapped database. http://www.lmdb.tech/doc/

[2] 2020. Intel AVX2. https://www.intel.com/content/www/us/en/homepage.html

[3] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: approximate tree indexing. In *Proceedings of the 40th International Conference on Very Large Databases (VLDB)*.

[4] Timo Bingmann. 2007. Stx b+tree c++ template classes. http://panthema.net/2007/stx-btree/

[5] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient in-memory indexing with generalized prefix trees. *Database systems for Business, Technology and Web (BTW)* 180 (2011), 227–246.

[6] Dhruba Borthakur. 2013. Petabyte scale databases and storage systems at facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1267–1268.

[7] Chiranjeeb Buragohain, Nisheeth Shrivastava, and Subhash Suri. 2007. Space efficient streaming algorithms for the maximum error histogram. In *2007 IEEE 23rd International Conference on Data Engineering (ICDE)*. IEEE, 1026–1035.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[9] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, Vol. 1. 181–190.

[10] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX Annual Technical Conference (ATC)*. 1007–1019.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[12] Danny Z Chen and Haitao Wang. 2009. Approximating points by a piecewise linear function: I. In *International Symposium on Algorithms and Computation (ISAAC)*. Springer, 224–233.

[13] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 239–252.

[14] Qiuxia Chen, Lei Chen, Xiang Lian, Yunhao Liu, and Jeffrey Xu Yu. 2007. Indexable PLA for efficient similarity search. In *Proceedings of the 33rd international conference on Very large data bases (VLDB)*. 435–446.

[15] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. μTree: a Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 11 (2020), 2634–2648. http://www.vldb.org/pvldb/vol13/p2634-chen.pdf

[16] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

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

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

[19] 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 (SIGMOD)*. 969–984.

[20] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[21] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 371–384.

[22] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 75–88.

[23] 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 (VLDB)* 13, 8 (2020), 1162–1175.

[24] 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.

[25] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. https://github.com/google/leveldb

[26] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering (ICDE)*. IEEE, 370–379.

[27] Goetz Graefe and P-A Larson. 2001. B-tree Indexes and CPU Caches. In *Proceedings 17th International Conference on Data Engineering (ICDE)*. IEEE, 349–358.

[28] The PostgreSQL Global Development Group. 1996-2021. PostgreSQL. https://www.postgresql.org/

[29] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[30] Pao-Lu Hsu and Herbert Robbins. 1947. Complete convergence and the law of large numbers. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)* 33, 2 (1947), 25.

[31] IBM Inc. 2020. IBM DB2. https://www.ibm.com/analytics/db2

[32] MongoDB Inc. 2021. MongoDB. https://www.mongodb.com/

[33] Krzysztof Kaczmarski. 2012. B+-tree optimized for GPGPU. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (OTM)*. Springer, 843–854.

[34] 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 (SIGMOD)*. 339–350.

[35] 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 Machine Learning for Systems* (2019).

[36] 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, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, Rajesh Bordawekar, Oded Shmueli, Nesime Tatbul, and Tin Kam Ho (Eds.). ACM, 5:1–5:5. https://doi.org/10.1145/3401071.3401659

[37] 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 (SIGMOD)*. 489–504.

[38] 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2119–2133. https://doi.org/10.1145/3318464.3389703

[39] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *USENIX Annual Technical Conference (ATC)*. 739–752.

[40] 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 (EuroSys)*. 183–196.

[41] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425

[42] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2789–2792. https://doi.org/10.1145/3318464.3384706

[43] Sean Meyn. 2008. *Control techniques for complex networks*. Cambridge University Press.

[44] William Jannen Rob Johnson Bradley C Kuszmaul Donald E Porter Jun Yuan Michael A Bender, Martin Farach-Colton and Yang Zhan. 2015. And introduction to Be-trees and write-optimization. *Login; Magazine* 40, 5 (2015).

[45] Joseph O'Rourke. 1981. An on-line algorithm for fitting straight lines between data ranges. *Commun. ACM* 24, 9 (1981), 574–578.

[46] Jun Rao and Kenneth A. Ross. 1999. Cache conscious indexing for decision-support in main memory. *VLDB* 99 (1999), 78–89.

[47] Jun Rao and Kenneth A Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD)*. 475–486.

[48] redislabs. 2021. Redis. https://redis.io/

[49] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. 2018. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 339–356.

[50] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 1523–1538.

[51] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations 2017 (ICLR)*.

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

[53] 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 (PPoPP)*. 308–320.

[54] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 117–135.

[55] Tom Wilcox, Nanlin Jin, Peter Flach, and Joshua Thumim. 2019. A Big Data platform for smart meter data analytics. *Computers in Industry* 105 (2019), 250–259.

[56] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*. 1–16.

[57] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB journal (VLDB)* 23, 6 (2014), 915–937.

[58] 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 (SIGCOM)*. 1567–1581.