2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00031

Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes

Jiake Ge[1,2], Boyu Shi[1,2], Yanfeng Chai[1,2], Yuanhui Luo[1,2], Yunda Guo[1,2], Yinxuan He[1,2] and Yunpeng Chai*[1,2]

[1]Key Laboratory of Data Engineering and Knowledge Engineering, MOE, China
[2]School of Information, Renmin University of China, China
{gejiake, shiboyu5687, yfchai, losk, guoyunda, heyinxuan, ypchai}@ruc.edu.cn

Abstract—**Numerous high-performance updatable learned indexes have recently been designed to support the writing requirements in practical systems. Researchers have proposed various strategies to improve the availability of updatable learned indexes. However, it is unclear which strategy is more profitable. Therefore, we deconstruct the design of learned indexes into multiple dimensions and in-depth evaluate their impacts on the overall performance, respectively. Through the in-depth exploration of learned indexes, we reckon that the approximation algorithm is the most crucial design dimension for improving the performance of the learned indexes rather than the popular works that focus on the learned index structure. Moreover, this paper makes a comprehensive end-to-end evaluation based on a high-performance key-value store to answer people's concerns about which learned index is better and whether learned indexes can outperform traditional ones. Finally, according to end-to-end and in-depth evaluation results, we give some constructive suggestions on designing a better learned index in these dimensions, especially how to design an excellent approximate algorithm to improve the lookup and insertion performance of learned indexes.**

I. Introduction

Kraska et al. [1] first proposed the read-only learned index based on machine learning methods in 2018. This new technique aims to reduce the space cost and improve the query performance for indexes. After that, some more practical updatable learned indexes have been designed. e.g., FITing-tree [2], PGM-Index [3], ALEX [4], XIndex [5], APEX [6], FINEdex [7], LISA [8], and LIPP [9], which brought innovations in the dimensions of approximation CDF, supporting updates, concurrent operations, etc.

The core idea of the learned index is to use a group of models (i.e., machine learning method or piecewise linear function) to approximate the cumulative distribution function (CDF) [10] of stored data (e.g., key-value pairs) in a sorted array effectively. The group of models can efficiently calculate the position of the target key in this array rather than merely use comparison-based searching in traditional B-Tree variants. Furthermore, the group of models can be composed of a series of parameters of the neural network or a series of linear functions. Thus their space cost is also much smaller than that of traditional indexes.

The benchmarking of learned indexes can help researchers understand the behaviors of learned index in-depth and then contributes to further improving the design of learned indexes. Furthermore, if we desire to improve the availability of learned indexes further, we need to cut the existing learned indexes to some important pieces and in-depth compare them fairly, then discover more promising designs direction in these dimensions (pieces). Ryan et al. [10] evaluate the performance of RMI [1], RS [11], and PGM-Index to analyze why learned indexes acquire excellent *read-only* performance. However, Ryan et al. do not give specific design directions for those dimensions (e.g., the approximation CDF algorithms and updating strategy) to further improve the lookup and updating performance of learned indexes, which makes us ponder about **Which component inner learned indexes is the most critical to their performance? What are the development directions for the future practical learned indexes in those dimensions?**

Most likely, the learned index is expected to be the next-generation member of the index family. However, many open-source updatable learned indexes with different designs have no chance of being placed in the same environment for a fair comparison. Therefore, It is difficult for us to honestly and objectively estimate the performance and practicality of the learned index. Therefore, we want to know: **Do learned indexes still perform better than traditional indexes in modern storage systems? Which learned index outperforms others?**

Therefore, in this paper, we will address these above problems on three fronts:

(i) For the first time, we have made a comprehensive end-to-end evaluation among the representative high-performance updatable learned indexes and traditional indexes in a fair environment. Our evaluation is performed under some real-world datasets (i.e., OSM [12] and FACE [13]) and synthetic datasets (i.e., YCSB Uniform and Zipfian distribution [14]). Moreover, read-only, write-only, and read-write-mixed are selected as evaluation workloads. We summarize that the performance of the learned indexes is better than that of the traditional tree-structure indexes in most cases (see §III).

(ii) For the first time, we deconstruct the design of the existing updatable learned indexes into four dimensions (i.e., approximation algorithm, index structure, insertion strategy, and retraining strategy) and evaluate them respectively in-depth. We argue that the approximation algorithm is more important to improve the query performance of the learned index, rather than most of the previous research work on learned indexes focused on improving the index structure to improve performance. Furthermore, we are surprised to find

*Yunpeng Chai is the corresponding author.

that the design of an excellent approximation algorithm can also improve the index insertion performance (see §IV).

(iii) We give some suggestions on the design of updatable learned indexes from the four dimensions. We argue that the more crucial to improving the performance of the learned index is to first actively change the distribution of stored data before approximating the CDF of the stored data, rather than directly approximating the CDF of the stored data. The purpose is to make the approximation algorithm better fit the CDF so that it can guarantee: 1) The fewer leaf nodes; 2) The lower average error; 3) The maximum error. Apart from the approximation CDF algorithms, the index structure and the updating strategy also need concerns, making learned indexes practical for real-world usage (see §V).

The rest of this paper is organized as follows: §II introduce the background knowledge of learned indexes. §III presents our end-to-end evaluation among typical learned indexes and traditional indexes. The impact of the different dimensions of learned indexes is evaluated in §IV, and some design suggestions are given in §V. The related work are given in §VI We conclude this paper in §VII. The source code of this paper have been made available at https://github.com/YunWorkshop/viper-learned-index/tree/learned_index.

II. Learned Index

As Fig. 1 shows, the learned index structure can be divided into the internal structure and the leaf nodes. The leaf nodes are composed of models generated by the approximation CDF algorithm, which is responsible for locating the position of the target key in the sorted array. And the internal structure can be composed of different kinds of models that are responsible for getting the position of the target leaf node. However, the model generated by the approximation CDF algorithm has query errors when the leaf nodes find the target key. Therefore, learned indexes usually adopt a binary search to finally reach the exact position of the target key within a certain error range.

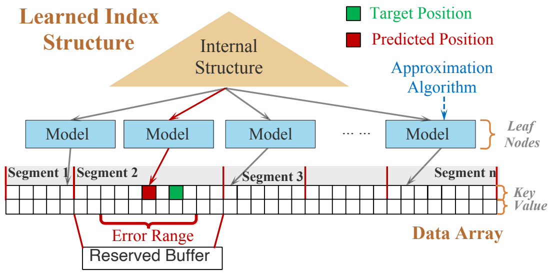


Fig. 1. The learned index structure.

Most of the learned indexes adopt piecewise linear functions to approximate the CDF, which has become a trend nowadays because of the following advantages: First, the CDF of stored data is usually not very complex, so a linear model can achieve a good enough approximation effect. Second, the computational overhead of linear model approximation CDF is much lower than that of the neural network.

Fig. 2 shows a schematic diagram of a linear function approximation for the CDF of stored data. Generally, the approximation algorithm can only approximate the shape of CDF but cannot accurately fit it. Therefore, there is an error between the target key position estimated by the linear function and the precise position. For example, Fig. 2 shows that key=4 is stored in the precise position 6, while the estimated position by linear function calculation is 3. The error between the calculated estimated position and the precise position is 3. When querying the target key, the linear function calculates the estimated position. Then, the precise position is obtained within the $error \pm estimated\ position$ by binary search.

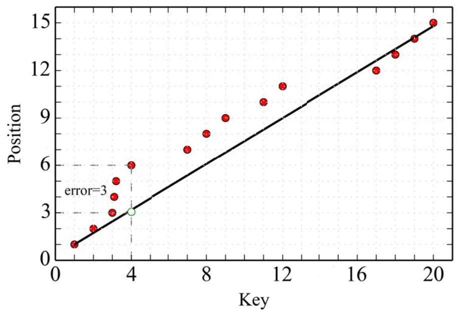


Fig. 2. Linear function approximation of a set of key.

Some learned indexes do not allow new data insertion (i.e., the read-only learned indexes), while others are updatable by inserting the new data into the sorted array or some reserved buffers. When the performance of these learned indexes deteriorates due to the insertion of too many new keys, the approximation CDF algorithm needs to be re-calculated to fit the distribution of the new sorted array; this procedure is usually called *retraining*.

A. Read-only Learned Index

1) Recursive Model Indexes (RMI): ***Approximation algorithm:*** In RMI [1], Learned Index Framework (LIF) is adopted to approximate the CDF of the storage data, which is an index composition system for generating index configurations, automatically optimizing and testing indexes. LIF can learn simple models (e.g., linear regression) and rely on Tensorflow to train complex models (e.g., neural network). However, LIF does not use Tensorflow directly when executing the modes; Instead, it automatically extracts the models' weights and generates models with efficient C++ codes. See the appendix [15] for more information on RMI.

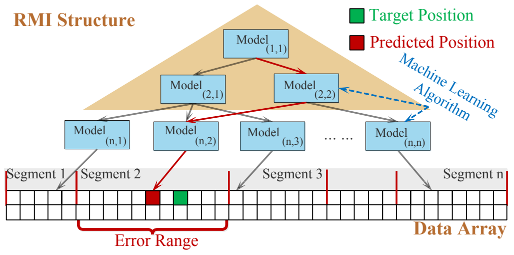


Fig. 3. The structure of Recursive Model Indexes (RMI).

Index structure: RMI is built top-down recursively; it is a layered model structure composed of multiple models. The structure of RMI is illustrated as Fig. 3. In RMI, the upper-layer model selects the most appropriate lower-layer model through the calculation to complete a query task. The output of the last layer model is the predicted position of the searched key in the sorted array. Then, the index employs the predicted position as the centre point to adopt a binary search within the error range to find the target key. In theory, each layer of RMI can adopt different models (e.g., linear models or B+ trees).

2) Radix Spline (RS): ***Approximation algorithm:*** RS [11] adopts a one-pass spline approximation algorithm, which guarantees a user-defined error range. RS can create new spline points based on user-defined error ranges. When the algorithm encounters a new r-bit prefix, RS inserts a new entry into the pre-allocated cardinality table.

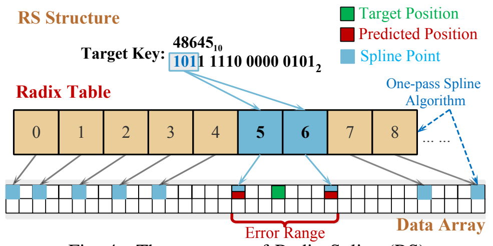


Fig. 4. The structure of Radix Spline (RS).

Index structure: Fig. 4 plots that the lowest level of RS is composed of spline curves, and its internal index structure is composed of cardinality tables. A set of spline points in a spline curve is defined based on the CDF of keys. The function of the cardinality table is to index spline points, which is to speed up the binary search of points on the spline curve.

When performing queries, RS first extracts *r* with the most significant bits *b* (e.g. *r*=3 and *b*=101). Then, after extracting the bit *b*, RS looks for offsets in the cardinality table to retrieve bits stored at bit *b* and *b+1* (e.g., bits 5 and 6). Next, RS uses a binary search on the sorted spline point set to locate the lookup key interval. Once the associated spline curve segment has been determined, RS employs linear interpolation between the two spline curve points to predict the position of the lookup key in the underlying data.

B. Updatable Learned Index

1) FITing-tree: ***Approximation algorithm:*** When constructing leaf nodes, FITing-tree [2] proposes a greedy algorithm similar to the Feasible Space Window [16], [17], which has *O(n)* time complexity and *O(1)* space cost. In essence, the approximation problem of CDF is reduced to the Piecewise Linear Approximation (PLA) problem. The algorithm divides sorted data into multiple linear segments, which guarantees the maximum query error in each segment. The maximum error guarantees the upper bound of the query time.

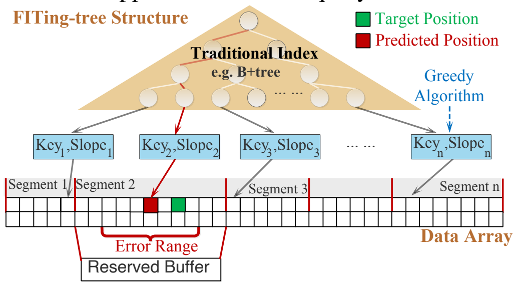


Fig. 5. The FITing-tree structure.

Index structure: Fig. 5 shows the structure of FITing-tree, whose internal structure can be replaced by different index structures (such as B+tree [18], FAST [19], etc.) according to different workloads and data distribution [2]. Its leaf nodes are composed of linear models. Each leaf node corresponds to a segment (a linear model), and the start key, slope, and intercept of each linear model are stored in the leaf node. When executing a query, the internal structure first locates the target key into a segment; then, the segment can calculate the predicted position of the target key using a linear function. Like the previous indexes, FITing-tree also uses the predicted position as the centre point to adopt a binary search within the error range to find the target key.

Insertion and retraining strategy: This index designed two insertion strategies, i.e., inplace and the buffer-based offsite insertion strategies. The inplace insertion strategy reserves a fixed-size buffer area at both ends of each leaf node. When a new key needs to be inserted, the model first predicts its insertion position in the leaf nodes. If incorrect, it is necessary to find the correct insertion position within the maximum error range by binary search. When the correct position is found, all the keys between the current position and the first or last key of the leaf node are moved one unit in the direction of the buffer, which creates a gap at the correct insertion position for the new key. The buffer-based offsite insertion strategy reserves an extra fixed-size buffer for each leaf node to store the newly inserted data temporarily and to keep them in order.

2) PGM-Index: ***Approximation algorithm:*** PGM-Index [3] also adopts the PLA model to approximate the CDF of stored data. Different from FITing-tree, the approximation algorithm of PGM-Index adopts the streaming algorithm, namely optimal-PLA, which has been widely studied in the field of lossy compression of time series and similarity search [20], [21]. This algorithm has been shown to obtain the minimum number of piecewise linear models (less than or equal to the number of segments in FITing-tree) while guaranteeing the maximum error and having an optimal time complexity of *O(n)*. In PGM-Index, this algorithm is used recursively to construct the internal linear model inside the internal structure.

Index structure: Fig. 6 gives the PGM-Index's structure composed of multiple layers of linear models. Each node is a linear model used to index the underlying nodes. Note that PGM-Index structure is similar to RMI, but its structure cannot be included in RMI, because RMI is built top-down, while the structure of the PGM-Index is built bottom-up (see §II-C).

Insertion and retraining strategy: PGM-Index adopts an offsite insertion strategy to insert a new key into an empty buffer. And motivated by the idea of LSM-tree [22], [23], PGM-Index builds a multi-level structure and maintains a learned index at each layer. PGM-Index only updates the corresponding model when the merge is triggered. In detail, this strategy establishes PGM-Index in a series of sets $S_0, ..., S_b$, where $b = \Theta(logn)$. These sets are either empty or of size S_0, S_1,...,S_b. When a key is inserted, the first empty set S_i is found and a new PGM-Index which is established according to data $S_0 \cup ... \cup S_i - 1 \cup \{key\}$ is created. In the meantime, the previous collection S_0, ..., S$_i - 1$ can be emptied. This algorithm is shown to take *O(logn)* amortized time [3].

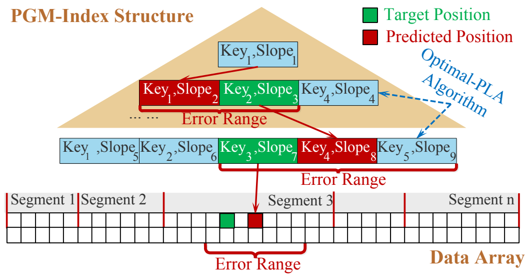


Fig. 6. The PGM-Index structure.

*3) ALEX: **Approximation algorithm:*** Unlike FITing-tree and PGM-Index, the linear approximation algorithm of ALEX [4] adopts the standard least square algorithm (LSA) [24], and the gaps are added to leaf nodes, improving the index insertion performance. To be more specific, the key contained in ALEX leaf nodes can obtain a linear model through the LSA. Then the slope and intercept of the linear function are enlarged by a particular factor to form a new linear function. Finally, the key in the leaf node recalculates the new storage location based on the new linear function. As a result, gaps emerge between adjacent keys. The purpose is that those gaps facilitate the query and insertion performance (See §IV for details). Note that the algorithm used in non-leaf nodes of ALEX is LSA.

Index structure: The structure of ALEX is illustrated in Fig. 7. Its internal structure is a variant of the RMI structure. Different from other learned index structures, the leaf node depth of ALEX is different with each other, i.e., it is not a balanced tree. In ALEX, all nodes adopt linear models. ALEX designs a fanout tree as the cost model, which can calculate the fanout of each layer to approximate CDF from top-down. The range of keys in each leaf node and the number of leaf nodes are determined by its cost model, which cannot guarantee the maximum search error.

Insertion and retraining strategy: ALEX is designed to reserve some gaps in the sorted array of leaf nodes to support insertion. Fig. 7 plots when a new key needs to be inserted, the model first predicts the insertion position. If the position is correct and empty, we can insert the key here directly. If the position predicted is incorrect, ALEX adopts exponential search to find the correct insertion position (an incorrect position predicted means that inserting a new key at this position makes the stored data unordered). Similarly, if the insertion position is empty, the key is inserted directly there; If the insertion position is not a gap, the key is moved one position in the direction of the nearest gap to form a gap at the insertion position. In ALEX, as more elements are inserted into the gap array, the number of gaps decreases and the insertion performance deteriorates. When the density of the gap array reaches the threshold, the index will determine whether the gap array is expanded or split. If the expand operation is adopted, it indicates that the linear model in the current leaf node can maintain high query performance. If the split operation is adopted, the current leaf node needs to be split into two linear models to maintain high query performance.

*4) XIndex: **Approximation algorithm:*** XIndex [5] also adopts LSA to approximate the CDF of the stored data.

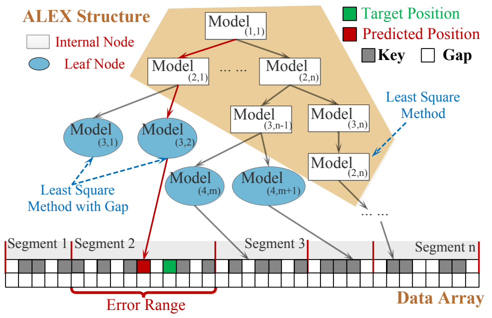


Fig. 7. The structure of ALEX.

Index structure: Fig. 8 shows that the XIndex contains the internal structure and the group nodes. The internal structure of XIndex is a double layer RMI model, and the group nodes of XIndex contain several linear models as their leaves. All nodes inside XIndex employ the linear model, and like ALEX, it is top-down constructed. Note that XIndex simply splits the data into each model for each group when building the index during the initialization phase then approximate the CDF of the data in each model. The models within each group node share a buffer and a temporary buffer. The purpose of the buffer is to support the offsite insertion strategy. The purpose of the temporary buffer is to enable the XIndex to be inserted in the retraining state.

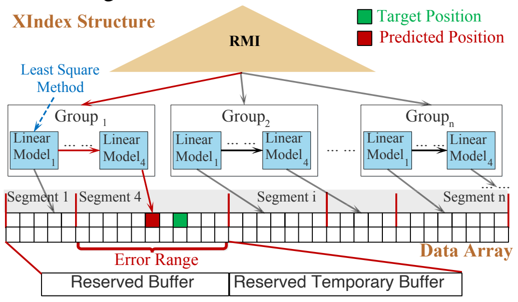


Fig. 8. The structure of XIndex.

Insertion and Retraining strategy: XIndex adopts the same offsite insertion strategy, similar to FITing-tree, and stores the inserted data in a buffer. A single group node has a reserved buffer. When the data in the buffer is merged into the sorted data during the retraining, the model is split in the group node if the error exceeds the threshold. Group node splitting is triggered if the number of models reaches the upper limit.

Other dimensions: XIndex supports concurrency insert operations. It employs existing technologies such as optimistic concurrency control [25]–[27], fine-grained locking [27]–[29], and Read-copy-Update (RCU), and carefully designs a two-phase compaction algorithm to support concurrent operations. Since the focus of the evaluation in this paper is not on the concurrency dimension of learned indexes, it is not be introduced in detail here.

C. Summary

Table I summarises the dimensions of these learned indexes.

Approximation algorithm: RMI builds indexes by gradually approximating the CDF of the stored data using machine learning. The index nodes in ALEX and XIndex use the algorithm of the LSA to approximate the CDF of stored

TABLE I
TECHNOLOGY COMPARISON OF LEARNED INDEXES.

Learned index	Structure			Approximation	Insertion	Retraining	Concurrency
name	Inner node	Leaf node	Error	algorithm	strategy	strategy	(Write)
RMI [1]	Neural net.|| Linear	Linear	Unfixed	Machine learning	-	-	✗
RS [11]	Radix tab.	Spline	Unfixed	One-pass spline	-	-	✗
FITing-tree [2]	B+tree.	Linear	Maximum	Greedy	Inplace || Offsite	Retrain one node	✗
PGM-Index [3]	Recursive	Linear	Maximum	Optimal-PLA	Offsite	LSM-Tree	✗
ALEX [4]	Asymmetric	Linear	Unfixed	LSA+gap	Inplace	Expand + retrain	✗
XIndex [5]	RMI	Linear	Unfixed	LSA	Offsit	Retrain one node	✓

data, and they do not guarantee the maximum error in leaf nodes. ALEX also employs the gaps in the leaf nodes, which improves write operation performance. FITing-tree and PGM-Index respectively adopt greedy and streaming algorithms to approximate the CDF of stored data.

Index structure: Learned indexes discussed in this paper are tree-structured except for RS. In the process of index construction, some are top-down construction (e.g., RMI, ALEX, and XIndex), and others are bottom-up (e.g., FITing-tree or PGM). The significant difference between them is that the top-down way does not guarantee the maximum error of the query. In addition, the tree structure designed by ALEX is asymmetric, which is different from other index structures.

Insertion and Retraining strategy: FITing-tree and XIndex insert data into the reserved buffer. The buffer is merged with the corresponding leaf node to trigger the retraining when the buffer is filled. Then the leaf node is split, and the index structure is updated if necessary. PGM-Index designs the insertion and retraining methods similar to LSM-tree. ALEX adopts gaps in leaf nodes so as to insert data efficiently. When the gap is about to be filled, the leaf nodes are expanded or split according to its retraining strategy.

III. END-TO-END EVALUATION

This section will make an end-to-end evaluation among the above representative learned indexes in a high-performance key-value (KV) store and compare learned indexes with typical traditional indexes comprehensively.

§III-A describes our experimental setup. §III-B $\sim$ §III-D evaluate the end-to-end performance of the learned indexes, comparing them with traditional indexes. §III-E evaluates the availability of KV store built with learned indexes.

A. Experimental Setup

The experiments were conducted on a two-socket server coupled with two Intel(R) Xeon(R) Gold 6242 CPUs @ 2.80GHz, 384GB of DRAM, and six 128GB of Intel Optane persistent memory (PMem), i.e., 768GB non-volatile memory (NVM). NVM was deployed in one socket of the server, i.e., there is no non-uniform memory accessing (NUMA) for NVM.

1) Indexes: The typical traditional indexes employed in the evaluation include STX B-Tree [18], Skiplist [30], Masstree [31], Bwtree [32], Wormhole [29], and CCEH [33]. The evaluated learned indexes include RMI, RS, FITing-tree, PGM-Index, ALEX, and XIndex. We first separately evaluate the performance of each index with different hyperparameters and choose their configurations with the best performance. When implementing its approximation algorithm, we adopt the improved optimal-PLA mentioned in PGM-Index rather than the greedy algorithm designed in FITing-tree. The main reason lies in that the approximation algorithm of PGM-Index was proved to be theoretically better than that of FITing-tree [3], and this will help us compare the other design dimensions between them. Note that the FITing-tree-inp and the FITing-tree-buf indicate the *inplace* and *offsite* insertion strategies designed by FITing-tree, respectively.

2) The Evaluation Environment: New non-volatile memory (NVM) [34], [35] has attracted wide attention in academia and industry because of its non-volatile, high density, byte-addressing, and low power consumption. Therefore, the NVM has excellent potential for the future computer system. Furthermore, using NVM for persistent KV stores and removing disk access has great potential to improve the performance of KV stores because the NVM can guarantee the persistence of byte-addressable data with close-to DRAM speed [36]. Lawrence et al. implement a hybrid NVM-oriented persistent KV store named Viper [37] that directly persists the data in NVM. They show that Viper outperforms those state-of-the-art KV stores by 4–18x. From this trend, NVM-oriented persistent KV stores are expected to be the next-generation high-performance storage technology.

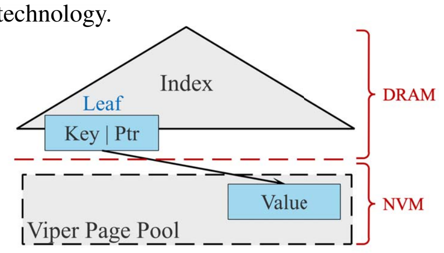
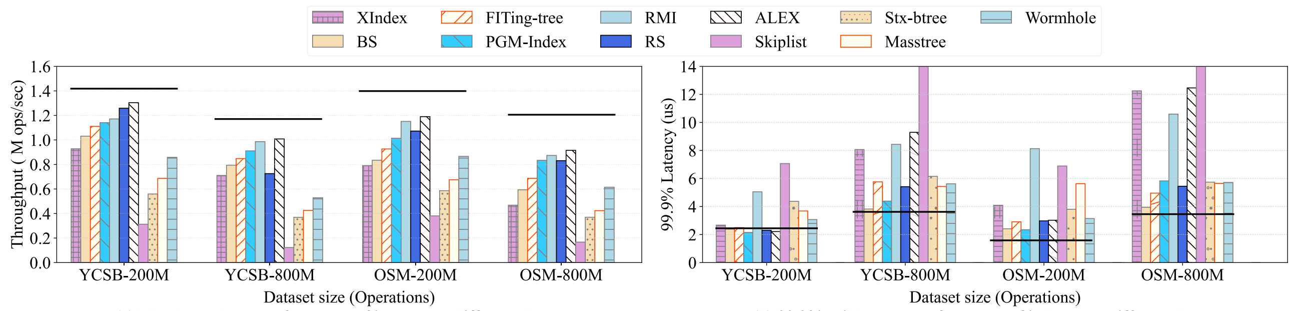


Fig. 9. Indexes and key-value pairs storage methods in Viper.

Naturally, the hypothesis to be considered is that the combination of learned indexes and NVM-oriented KV store will be the trend in future persistent storage systems, both of which have achieved outstanding performance in their respective fields. However, NVM is a cheap hardware device with slower storage performance than DRAM. Therefore, we want to explore, under the drag of slower hardware such as NVM, whether the learned index's excellent performance still has an advantage compared to traditional sorted indexes? i.e., the bottleneck of performance may be the NVM or the index, which needs to be evaluated by combining the two. Therefore, Viper is employed as a fair comparison environment to evaluate the performance of learned indexes. We not only want to fairly evaluate learned indexes with traditional ones on the same platform but also aim to observe the end-to-end speedup brought by learned indexes and the fitness of integrating learned indexes into the latest promising KV store.



(a) The throughput performance of indexes on different datasets. **(b) 99.9% tail latency performance of indexes on different datasets.**

Fig. 10. The end-to-end evaluation driven by read-only workloads. The black horizontal line represents the performance of CCEH [33]. Extended plots with all evaluations are available here: [15]

Viper uses a volatile index entirely located in DRAM to keep track of the records stored in NVM (see Fig.9). We have implemented all representative indexes listed in the previous part (§III-A1) in Viper.

3) Datasets and Workloads: Our evaluation datasets consist of the dataset generated by YCSB [14] and the real-world dataset, i.e., OSM [12] and FACE [13]. We adopted the YCSB and real-word dataset containing 200M∼800M key-value pairs with 8-byte keys and 200-byte values. In addition, §III-D uses several different datasets generated by YCSB, which were represented as YCSB-A (update mostly workload), YCSB-B (read mostly workload), YCSB-D (read latest insert workload), and YCSB-F (read-modify-update workload), respectively. YCSB follows the normal distribution in §III-A and §III-B, and it follows Zipfian distribution in §III-C. We evaluated the performance of a range query for learned indexes and included the results in the appendix [15].

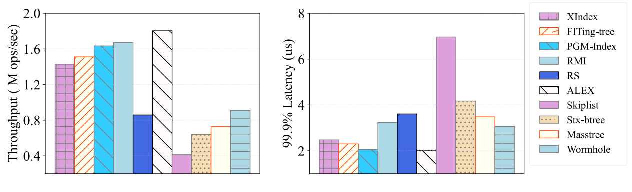
TABLE II
THE AVERAGE DEPTH OF THE LEARNED INDEXES UNDER YCSB-200M

Dataset (200M)	RMI	FITing-tree	PGM	ALEX	XIndex
YCSB	2	3	3	1.03	2
OSM	2	4	6	1.89	2

B. Read-only Case Evaluation

1) Single-thread Evaluation: Fig. 10 plots the performance of indexes under read-only workloads, which can show the ideal performance of learned indexes without updating. The throughput and 99.9% tail latency of Viper coupled with each index under YCSB and OSM datasets are illustrated in Fig. 10 (a) and (b), respectively. For both datasets, ALEX achieved the best performance, obviously superior to traditional sorted indexes. The throughput of ALEX is 4% to 30% higher than other learned indexes. There are two fundamental reasons for ALEX's higher throughput as follows: 1) Each level of the internal structure searched down causes a cache miss. The internal structure of ALEX has a lower average depth than that of the other learned indexes, which causes fewer cache misses ($\approx$ 2 on YCSB 200M, see Table II). Therefore, ALEX has excellent throughput performance. 2) ALEX's excellent approximation algorithm can better approximate the CDF of stored data, making the error of the leaf nodes minimal (see §IV-A). However, the tail latency of ALEX increases as the dataset increases from 200M to 800M because they do not guarantee the maximum error. Thus the worst query performance is not guaranteed, resulting in a sharp increase in tail latency (see §IV-A).

The performance of RS is excellent at 200M. However, the performance decreases obviously as the dataset increase from 200M to 800M. Note that the RS is segmented according to the r-bit prefixes, and the r-bit prefixes do not change when the data increases, resulting in higher query cost in the segment and lower query performance of the RS. For FACE, the performance of RS decreased significantly (see Fig. 11). The approximate algorithm of RS is to divide segments according to the r-bit prefixes of the key (we select the first 18-bits to achieve the best performance) and divide the same r-bit into a segment. However, the data in FACE possess skew characteristics, with a large number of keys falling within $(0, 2^{50})$ and the minimal number falling within $(2^{59}, 2^{64} - 1)$, which makes the first 16-bit of the RS almost useless [10].



(a) The throughput on FACE. **(b) 99.9% tail latency on FACE.**

Fig. 11. A comparison of the performance in the FACE.

RMI, PGM-Index, and FITting-tree also achieve higher performance than traditional tree-structure indexes. Among them, RMI performance is slightly better than PGM-index. The number of layers of RMI is smaller than that of PGM-index (see table II) when both sizes are similar, and the query cost of the internal structure of RMI is less than PGM-index [10]. However, since the maximum error cannot be guaranteed in RMI, the tail latency is much larger than PGM-index. The throughput of the PGM-Index is better than that of the FITing-tree. The reason lies in that the recursive linear models inside PGM-Index have a higher query performance (The target position is obtained by calculation) than the B+ tree structure inside the FITing-tree (The target position is obtained by comparison). (see §IV-B for details).

For OSM, the indexes' performance ranking is roughly the same as YCSB. However, the performance of learned indexes is degraded compared with YCSB, while traditional indexes are not affected. This is because the CDF of the OSM is more complex [10] compared with the generated data of

YCSB. In this case, the approximation functions would have to require more piecewise linear models to approximate the CDF, producing more leaf nodes in learned indexes (see §IV-A). Therefore, the learned index depth increases (see Table II), resulting in performance degradation.

Furthermore, our evaluation results driven by read-only workload prove that the end-to-end performance of learned indexes is much higher than that of the traditional tree-structure indexes in Viper, which shows that adopting a faster learned index in the NVM-based KV store will significantly improve the performance.

2) Multi-thread Evaluation: All the learned indexes support concurrent read operations, so we give the throughput and tail latency under different thread counts for 200M requests of YCSB in Fig. 12 (a) and (b). Among all the indexes, CCEH achieves the highest performance. The following are FITing-tree, PGM-Index, XIndex, and RS. Furthermore, the tail latency of FITing-tree, PGM-Index, and RS increases obviously as the number of threads increases.

However, the throughput of ALEX grows slowly as the number of threads increases. Our profiling results indicate that ALEX has already saturated the memory bandwidth with 24 threads in one socket. We found that memory bandwidth was exhausted under multi-threading, which led to the competition of NVM bandwidth, resulting in slow performance.

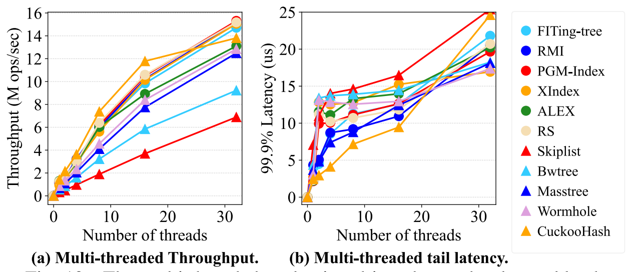


Fig. 12. The multi-threaded evaluation driven by read-only workloads

C. Write-only Case Evaluation

1) Single-thread Evaluation: Fig. 13 (a) and (b) give indexes' throughput and tail latency performance by write-only workloads. ALEX achieves the best performance compared to other learned indexes because its excellent insertion strategy is designed to take advantage of gaps in leaf nodes. (see §IV-A). In addition, except for ALEX, the performance of the other learned indexes does not show an excellent advantage compared with the traditional tree-structured index, illustrating that the design of the insertion strategy and retraining strategy is very important (see §IV-E).

On both YCSB and OSM datasets, the throughput of ALEX is the best. FITing-tree-inp has the worst throughput. To make matters worse, the tail latency of both insertion strategies of the FITing-tree exceeds 100us. The reason lies in that the insertion strategy of the FITing-tree causes much movement of stored data, leading to poor insertion performance (see §IV-D).

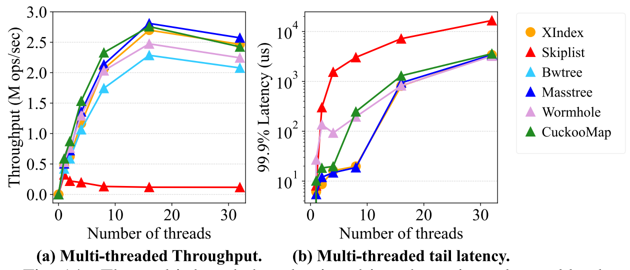


Fig. 14. The multi-threaded evaluation driven by write-only workloads

As the dataset size increases from 200M to 800M, the performance of the learned index is degraded dramatically. The throughput of XIndex and FITing-tree-buf degraded the most because they all employ an offsite insertion strategy, which can cause a large batch of retraining operations when a large number of inserted data makes most of the buffers full (e.g., the 800M case. see §IV-E).

2) Multi-thread Evaluation: Among all the learned indexes, only XIndex supports concurrent write operations. Therefore, Fig. 14 (a) and (b) respectively illustrate XIndex and traditional indexes' throughput and tail latency under different threads on 200M YCSB. Moreover, the performance of XIndex is similar to that of Masstree. Overall, XIndex's performance is close to traditional indexes.

D. Read-write-mixed Case Evaluation

Fig. 15 (a) and (b) plot each index' throughput and tail latency under various read-write-mixed YCSB workloads, which is close to real-world scenarios. ALEX has always maintained a good throughput performance, indicating that the data insertion and the retraining strategy of ALEX are excellent. Except for ALEX, the other learned indexes' performance drops a lot on YCSB-D. The main reason is that YCSB-D differs from the other read-write-mixed dataset in that its write operation is an insertion rather than an update. The insertion operation causes the learned index to be continuously inserted and retrained. Obviously, the evaluation results of YCSB-D show that the

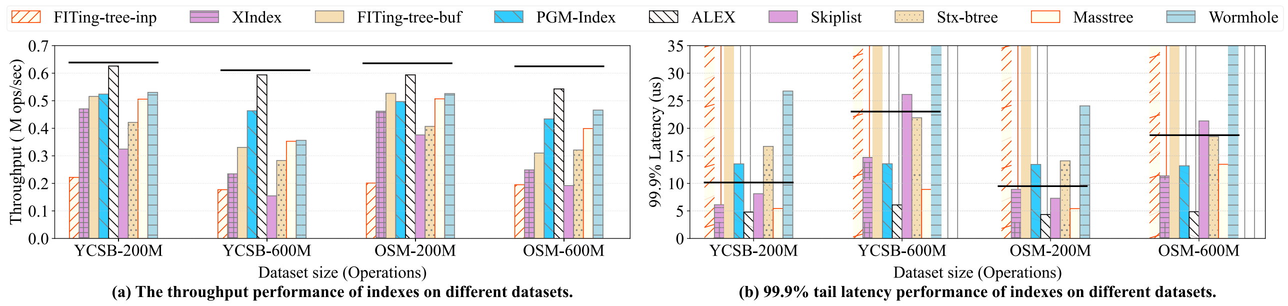
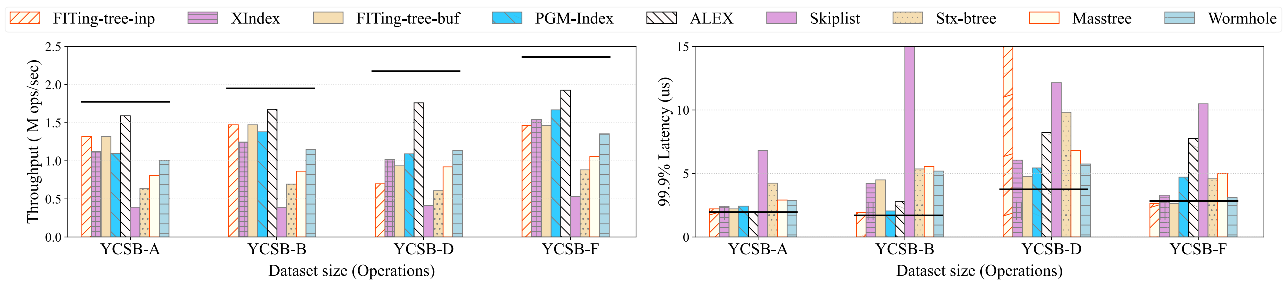


Fig. 13. The end-to-end evaluation results driven by write-only workloads. The black horizontal line represents the performance of CCEH [33].



(a) The throughput performance of indexes on different datasets. **(b) 99.9% tail latency performance of indexes on different datasets.**

Fig. 15. The end-to-end evaluation driven by read-write-mixed workloads. The black horizontal line represents the performance of CCEH [33].

robustness of many learned indexes' insertion and retraining strategies is not good enough.

E. Availability of KV Store Built with Learned Indexes.

1) Space Overhead Evaluation: Recall Fig. 1 that the top half of an index is the index structure, and the bottom half is the key-value pairs. Key-value stores can store the key and the value separately (e.g., key in DRAM and value in NVM) or store them together in DRAM or NVM. Thus there will be three possible scenarios in the storage systems.

TABLE III
THE SPACE OVERHEAD OF TYPICAL INDEXES.

Index	Index size	Index+key size	Index+KV size
RMI	2.3MB	3.2GB	41GB
PGM-Index	2.1MB	3.4GB	41GB
FITing-tree	2.96MB	3.2GB	41GB
XIndex	11.46MB	4.8GB	42GB
RS	2.09MB	3.2GB	41GB
ALEX	129KB	4.6GB	42GB
BTree	155MB	3.4GB	41GB
Wormhole	1.28MB	3.3GB	41GB
Cuckoo	10938MB	4.3GB	42GB

(i) Due to the limited memory capacity (e.g., in an embedded scenario), only the upper index structure is stored in DRAM. The size of indexes now corresponds to the leftmost column "Index Size" in Table III. Learned indexes occupy three to five orders of magnitude less space than traditional indexes. Because the learned indexes must ensure the order of keys, the system needs to maintain the order of keys in SSD or NVM whether key-value separation technology is adopted or not, which will significantly lower the system's performance when facing a write-frequent workload.

(ii) In the key-value separation scenario, the index structure and the sorted keys, which are updated frequently, are both stored in memory, and the value is stored in NVM. Maintaining the order of keys in memory is much less costly. Here, the size of indexes corresponds to the column "Index_key size" in Table III. In this case, the total consumed DRAM capacity of learned indexes is very close to traditional indexes because the sorted array of keys is much larger than the index structure. That is to say, the space advantage of learned indexes is not significant in many practical environments.

(iii) In the memory database scenario, the index structure, the keys, and the values are all maintained in DRAM. The sizes of indexes are shown as the last column in Table III. Unsurprisingly, the size of indexes is basically the same.

To sum up, although learned indexes reduce the size of the index structure itself significantly compared with traditional indexes, they cannot save obvious DRAM space overhead because the sorted key array takes much more space than the index structure. The only case that learned indexes can show a significant space advantage may happen when DRAM capacity is very small, and the workload is read-only or read-dominant.

2) Recovery Time Evaluation: The index needs to be recovered quickly when the system crashes. Therefore, the recovery time of the index also determines the system's availability.

Fig. 16 illustrates the recovery time of different indexes. The build operation is used to recover the index when restarting the KV store if the index structure is maintained in DRAM and is used to index large amounts of data at the initialization phase. The recovery time of Stx-BTree and Wormhole is the fastest. On the 200M dataset, the time to recover the learned index is similar; however, the recovery time difference between these indexes becomes more prominent as the dataset is 800M. The next ones are ALEX and XIndex, up to 112s and 113s, respectively. PGM-Index is only 40s, while RS is only 18s. The reason RS spends the least time is that it only needs Single-Pass to recover the index structure. Furthermore, RS only needs to recover the fragment according to r-bit prefixes, while other indexes require much calculation time.

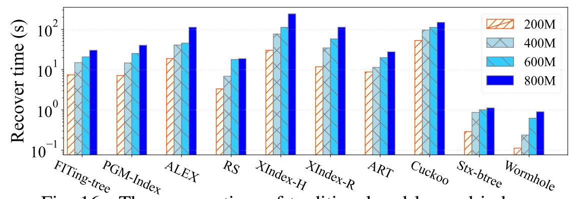


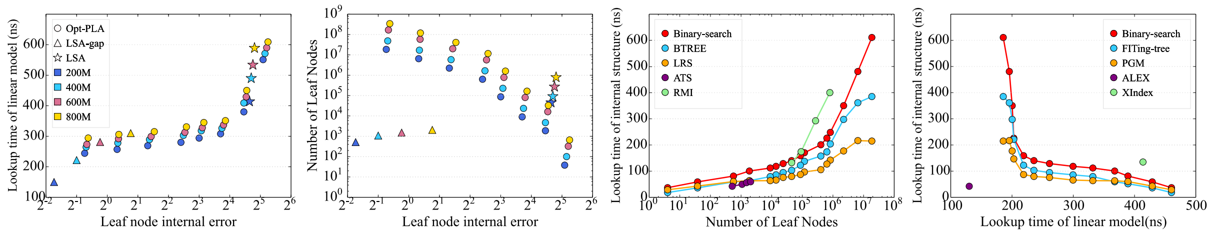
Fig. 16. The recover time of traditional and learned indexes.

IV. In-depth Inquiry into Learned Index Components

§II cuts learned indexes into four important components. Note that, in theory, the four dimensions of the existing learned indexes are orthogonal, i.e., they can be combined to form brand new indexes. Therefore, this section evaluates and analyzes those dimensions in-depth independently and attempts to give a better possible combination scheme under those dimensions. Finally, we discuss the most critical design dimension of learned indexes.

A. Approximation Algorithm Evaluation

This part will evaluate and compare the open-source approximation algorithms' performance. The query time determined by the linear model can represent the performance of the



(a) The performance of approximation algorithms. (b) The relationship between error and leaf's number. (c) The performance of Internal structure. (d) The relationship between structure and algorithm.

Fig. 17. The performance of the Learn indexes structure and the approximation algorithms

approximation algorithm. The approximation CDF algorithms of existing learned indexes are listed below:

(i) LSA: Least squares method (e.g., used in XIndex). After dividing the stored data into fixed segments, LSA is used to generate a linear model for each segment.

(ii) Opt-PLA: Streaming algorithm (e.g., used in PGM-Index). Unlike the approximation process of LSA, users first need to specify the maximum error of Opt-PLA. Then, the data is streamed divided into different segments. When the specified error is low, the CDF within each segment needs to be closer to a linear distribution. Under the condition of ensuring the maximum error, this streaming algorithm puts the maximum number of the data that can be approximated to the same straight line into the same segment, which can generate the smallest number segment.

(iii) LSA-gap: Least squares method with gaps (e.g., adopted by ALEX). LSA-gap first divides the stored data into fixed segments, and then the algorithm inserts gaps between stored keys to make room for future data insertion. Another benefit of these gaps lies in making the key distribution more easier to be approximated with linear functions (see §II-B3).

Fig. 17 (a) plots the relationship between the average error and the query performance of the leaf node linear model. When the error is lower, the linear model can locate nearer the target position during the query, and the query time will naturally be shorter. i.e., the lower the error, the higher the query performance, only for the leaf node inside.

Fig. 17 (b) illustrates the corresponding relationship between the error of leaf nodes and the number of leaf nodes. the fewer the number of leaf nodes, the better the query performance of the index's internal structure is (see §IV-B). The number of leaf nodes generated by the Opt-PLA algorithm is about two orders of magnitude less than that generated by the LSA algorithm. Therefore, the Opt-PLA algorithm is more suitable for the learned index than the LSA. Secondly, the figure shows that the number of leaf nodes generated by the Opt-PLA algorithm is less than that of the LSA-gap algorithm only when the average error is 32. At this time, the average error of the LSA-gap is much lower than that of the Opt-PLA. Obviously, the LSA-gap algorithm is more suitable for the learned index than the Opt-PLA algorithm.

Analysis of the Performance Differences: The above algorithms all adopt a linear model to approximate the CDF. Obviously, the closer the CDF is to the linear distribution, the lower the linear model error generated by the approximation algorithm. The fundamental reason for the different performance behavior of each algorithm is summarized below:

(i) LSA: LSA has two most significant issues that affect its performance. Firstly, to ensure relatively lower error, this method can only be divided into as many segments (leaves) as possible, making the CDF closer to a straight line. Therefore, the algorithm produces numerous leaf nodes, which degrades the performance of internal structure lookup. By contrast, if a small number of segments are guaranteed, the error within the segment will increase sharply, thereby affecting the query performance within the segment. Secondly, this method still cannot guarantee the maximum error. Thus LSA does not guarantee the worst query performance, which leads to a significant increase in tail latency (see Fig. 10).

(ii) Opt-PLA: Opt-PLA can guarantee to generate the smallest number of segments under the condition of ensuring the maximum error. As the guaranteed maximum error is minimal, the number of segments is still significant, which degrades the performance of internal structure lookup. By contrast, if a few segments are guaranteed, the error becomes very large, affecting the query performance of inner leaf nodes.

(iii) LSA-gap: The most significant common problem with the above two algorithms is the conflict between the linear error and the number of leaf nodes, i.e., we usually cannot get a small linear error and few leaf nodes simultaneously. The LSA-gap algorithm solves this problem by introducing a bit more consumed space. The specific implementation is that LSA-gap inserts gaps between stored keys, optimizing the CDF to get closer to a linear distribution and thus becoming more accessible to approximate as a linear model with minimal error. Therefore, there is no need to divide as many segments as possible to minimize error. LSA-gap can ensure a minor error and a smaller number of segments simultaneously.

B. Index Structure Evaluation

This part will evaluate the impact of index structures. We employ the query time from the root node to the leaf node to represent the performance of the index structure. The structures adopted by those learned indexes are as follows:

- **RMI**: Two-layer RMI structure (e.g., XIndex).
- **ATS**: Asymmetric tree structure (e.g., ALEX).
- **BTREE**: B+ tree structure (e.g., FITing-tree).
- **LRS**: Linear Recursive Structure (e.g., PGM-Index).

The performance of a learned index is highly related to its structure and the number of leaf nodes generated by the approximate algorithm. Because the index structure and the number of leaf nodes affect the overall performance, we perform all the index structures under different leaf node counts, shown in Fig. 17 (c). The figure indicates that the ATS structure has the minimum query time under the same number of leaf nodes. Since the leaf nodes' depth of the ATS structure is not fixed, this structure does not need to go through the longest internal path to query leaf nodes for every query operation. Therefore, the performance of ATS is better than other tree structures. In the case of fewer leaf nodes, the performance of LRS is similar to that of BTREE. However, when there are many leaf nodes, the performance of LRS is significantly better than that of BTREE. The reason lies in that when searching for a target key, the more accurate storage location of the target key can be calculated by LRS, the less time it takes. While BTREE requires multiple comparing operations to find the target key, taking much time. The figure also illustrates that the fewer the number of leaf nodes, the better the query performance of the index structure is.

Note that the linear models in the internal structures of learned indexes (non-leaf node) of LRS, ATS, and RMI adopt different linear approximation algorithms, which affect the performance and seem insufficient to compare the performance differences caused by the differences in index structures. Here, based on the evaluation results in §IV-A, we supplement a detailed analysis as follows:

First, in Fig. 17 (c), BTREE are not affected by different approximation algorithms. Second, although the ATS structure is coupled with the LSA algorithm and LRS is coupled with Opt-PLA that is proved to be better than LSA according to §IV-A, ATS+LSA still achieves higher performance than LRS+Opt-PLA (see Fig. 17 (c)) This indicates that ATS outforms LRS, and we may get a higher performance when combining ATS with a better approximation algorithm. Finally, the paper [10] mentions that RMI outperforms LRS. However, we do not know whether RMI will perform better than ATS after changing the approximation algorithm. This issue deserves to be further explored in the future.

C. The Relationship Between Index Structure and Approximation Algorithm

Fig. 17 (d) plots the relationship between the query cost of the leaf node and the index structure. The closer the record is to the bottom left corner of this figure, the better the index performance. Obviously, ALEX is the best. It can guarantee both low query costs inside the leaf nodes and inside the index structure, because the leaf node count of ALEX is much smaller than the others and thus its index structure is much simpler.

Summary: We emphasize that the best existing approximation algorithm (LSA-gap) reduces the number of segments (leaves) and the average error within segments (leaves) **simultaneously** by changing the CDF of original keys, which is unable to be done by the other two approximation algorithms, thus significantly improving the index performance.

Under our evaluation environment and conditions, the best design of the approximate algorithm is LSA-gap. the combination of ATS and LSA-gap (i.e., ALEX) achieves the best performance among all evaluated learned indexes.

D. Insertion Strategy

The current learned index insertion strategies can be divided into the following types:

- **Buffer**: FITting-tree-buf and PGM (i.e., Offsite insert).
- **Inplace**: FITting-tree-inp (i.e., Inplace insert).
- **ALEX-gap**: ALEX (i.e., Inplace gap insertion).

Fig. 18 (a) exhibits the performance of the three different insertion strategies. The *Inplace* and the *Buffer* strategy can adjust the size of reserved space. We evaluate the insertion time of two insertion strategies when the size of reserved space is set to 128, 256, 512, and 1024 (the number of keys), respectively. The reserved space in the ALEX-gap strategy is automatically generated, and the user cannot set the exact size.

Analysis of the Performance Differences: (i) Inplace. The insertion performance of the Inplace strategy is worse than that of the other two cases. The reason is that the Inplace strategy will cause all keys on one side of the insertion position to move to make a gap for the insertion position, significantly reducing the insertion performance. In the case of the Buffer strategy, the key does not need to be moved to start inserting into an empty buffer. With more keys are inserted, the number of stored keys moves increases.

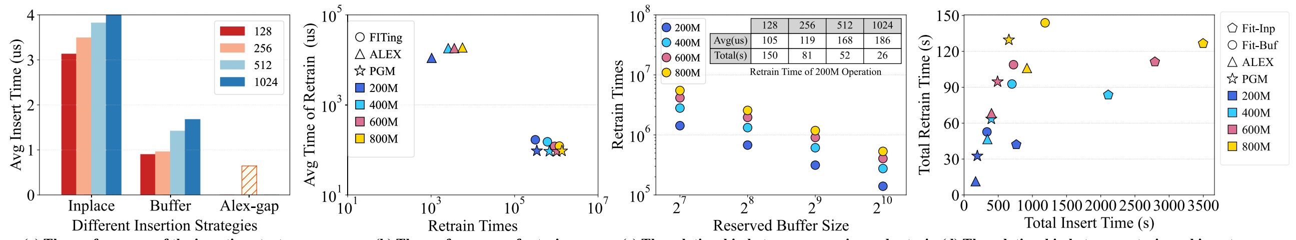
(ii) Buffer. Because the increase of reserved space leads to massive data filling, the larger the reserved space, the worse the performance of the Inplace and the Buffer strategies. Thus, the number of key moves increases when data is inserted, which results in poor insertion performance.

(iii) ALEX-gap. The ALEX-gap strategy achieves the best insertion performance. The reason lies in that this strategy reserves some gaps near the target insertion position. There is little or no key movement when inserting a new key.

E. Retraining Strategy

In this part, we will evaluate the retraining strategies of the currently learned indexes. Note that we use the names of the learned indexes to represent their retraining strategies. Fig. 18 (b) plots the retraining performance of three different strategies. First, the figure shows that as the inserted data increases, the average retraining time of the three strategies is roughly stable. Especially PGM-Index almost keeps the retraining time unchanged. Second, ALEX has the least number of retraining. On average, for every 200,000 keys inserted, ALEX retrains only once. However, the number of retraining of FITing-tree and PGM is large. On average, they retrain once for every 500 inserted keys. Third, ALEX has the longest average retraining time, and PGM-Index has the shortest time. It is important that a retraining operation could affect the tail latency of the index, which may increase the probability of system instability. Finally, the total retraining time of ALEX is shorter than the other two, while the longest is FITing-tree. When 200M KV pairs are inserted, the total time of ALEX, PGM-Index, and FITing-tree is 11.39s, 32.57s, and 52.70s, respectively.

The retraining action count is directly related to the reserved space of learned indexes. Fig. 18 (c) shows the relationship between the size of reserved space in Buffer strategy and the number of retraining times when 200M data is inserted. The



	128	256	512	1024
Avg(us)	105	119	168	186
Total(s)	150	81	52	26

(a) The performance of the insertion strategy. **(b) The performance of retrain.** **(c) The relationship between space size and retrain.** **(d) The relationship between retrain and insert.**

Fig. 18. The performance of the insertion and retraining strategy

figure shows that as reserved space increases, the number of retraining decreases. The table in Fig. 18 (c) illustrates that the average retraining time increases while the total time decreases as the reserved space increases. This is because the number of keys to be retrained increases with the increase of reserved space, resulting in a long time for single retraining. But as the reserved space increases, the number of retraining is significantly reduced, which results in a significant reduction in total retraining time.

F. The Relationship Between the Insertion and Retraining

In §IV-D, we conclude that as the size of reserved space increases, the insertion performance of Inplace and Buffer strategies decreases. §IV-E concludes that the total retraining time reduces as the size of reserved space increases. Therefore, in PGM-Index and FITing-tree, there is an irreconcilable conflict between insertion performance and retraining performance, and both performances will not be good simultaneously. However, this situation does not happen in ALEX. The reason is that the ALEX-gap strategy does not reduce the insertion performance as increased reserved space. Therefore, ALEX demonstrates excellent insert performance in §III.

Fig. 18 (d) gives the relationship between the total insertion time and the total retraining time of different indexes. We compare the total time of different update strategies (the sum of the insertion and retraining time). The figure plots that the FITing-tree-inp has the longest total time, and the next is FITing-tree-buf. The PGM has a shorter total time. The shortest total time is ALEX. To sum up, ALEX's update strategy is the best, and PGM's update strategy is also excellent.

Summary: 1) ALEX is the best in terms of the sum of the total insertion and retraining time. 2) PGM-Index has the lowest average retraining time; ALEX has the least amount of retraining. 3) Except for ALEX, larger space reserved in the index leads to worse insertion performance.

G. Summary: Which Dimension should we First Focus on?

From the above evaluation results, the design dimensions of the ALEX are the most successful compared to other evaluated learned indexes. The main reason for ALEX's excellent performance comes from resolving the two following conflicts well because of its excellent approximation algorithm design: (i) The conflict between guaranteeing the lower linear error and guaranteeing the lower number of leaf nodes (§IV-A). (ii) The conflict between the better insertion performance and the better retraining performance (§IV-F).

ALEX wisely adopts the idea of paying some additional space in its approximation algorithm design for higher performance of query processing. This strategy improves the key distribution to make CDF approximation more easier, leading to fewer leaf nodes and a smaller tree structure. At the same time, in insertion operations, the additional space (i.e., gaps) lowers the cost of insertion and the triggered count of retaining. **Therefore, designing a better approximation mechanism is the most crucial dimension for designing a high-performance learned index.**

V. How to Design a Better Learned Index?

Based on the analysis of each dimension in §IV, we discover that designing an excellent approximation algorithm is the most important to improve the performance of learned indexes. Thus, this section will discuss how to design the learned index and give some suggestions on designing next-generation learned indexes.

A. Design Suggestions for Approximation Algorithm

Firstly, if a learned index chooses a linear function to passively approximate CDF (i.e., approximates the shape of the CDF of stored data without changing the distribution of stored data), there will be an irreconcilable conflict between the number of linear segments and the maximum error (see §IV-A). Unexpectedly, through evaluations, we found that the approximation algorithm designed by ALEX breaks this constraint because it actively changes the distribution of stored data to better approximate the CDF, which makes us inspired. Therefore, to design a better approximate CDF algorithm, we should find a way to **actively optimize the CDF of stored data to make it easier to approximate before generating the linear model**. Perhaps a combination of nonlinear models and changing the CDF will help design a better approximation algorithm in the future.

Secondly, we suggest that **it is critical to consider the three dimensions simultaneously: (i) Guarantee the less leaf nodes; (ii) Guarantee the lower average error; (iii) Guarantee the maximum error.** Fewer leaf nodes can shorten the time for the internal structure to query leaf nodes significantly; lower internal average error of leaf nodes can shorten the time of querying the target key in leaf nodes; the guarantee of maximum error can reduce the tail latency of the index. However, existing algorithms only satisfy two of the three dimensions. Therefore, when designing the approximation algorithm in the future, we should consider the above three dimensions simultaneously.

B. Design Suggestions for Other Dimensions

1) Design Suggestions for Learned Index Structures: When the scan is a necessary operation, the asymmetric tree structure of ALEX performs higher performance than the symmetric tree structure (e.g., PGM-Index). However, in §IV-C, we concluded that the asymmetric tree structure of ALEX has a slight performance advantage because of the approximation CDF algorithm (LSA). If the algorithm is replaced by LSA+gap, the performance of the asymmetric tree structure will probably be better than now. Unsurprisingly, the LIPP has found this critical point and successfully implemented this method after improvement. Since it is not open source now, we cannot evaluate it. Finally, we also keenly found that the asymmetric tree structure can support the hot data to be placed closer to the root node, which can shorten the total number of queries and improve query performance, which is also our future research direction.

2) Design Suggestions for Insertion Strategy: For the insertion strategy, we found that the prerequisite for good insertion performance is to avoid moving too many other keys when inserting new keys, which should be considered when designing updated learned indexes. Furthermore, we should also consider the characteristics of the inserted data. For example, since sequential data will always be inserted at the end of the storage space, the inplace insertion strategy proposed by ALEX will waste much space. Therefore, we should design different insertion strategies according to different target data. Finally, a well-designed concurrency strategy for write operations is needed to improve the insertion performance significantly.

3) Design Suggestions for Retraining Strategy: For the retraining strategy, we suggest designing different retraining strategies for different user requirements. For instance, for users with low tolerance of tail latency, we should not employ the retraining strategy of ALEX, which retrains once too many keys. This strategy will inevitably result in a large tail latency. In this case, the retraining strategy similar to PGM-Index is more suitable. We need to reduce the total retraining time for users with high system throughput requirements as much as possible. In this case, ALEX's retraining strategy is better.

VI. Related work

A. Benchmarking Framework

SOSD is a learned index benchmark framework for queries, which can provide benchmarks for different learned indexes to compare. Moreover, search algorithms in leaf nodes of learned indexes include binary search, cardinal binary search, interpolation search, and the recently proposed three point interpolation [38].

SOSD provides only one-dimensional range query workload, and more complex workloads can consider popular testing benchmarks in the database domain. TPC-C [39] and YCSB [14] can be selected as test benchmarks in the index inserts where read-write-mixed workloads are needed. For multi-dimensional queries, public map datasets [12] or image datasets [40] and TPC-H [41] benchmarks can be selected.

B. Benchmarking for Learned Indexes

Hussam et al. [42] first evaluated the impact of learning index RMI on Bigtable [43] performance. Experimental results show that RMI improves the throughput of query and scan by 54% and 56%, respectively, and reduces the query latency by 38% compared with the default two-level index in Bigtable. In addition, using learned indexes does not affect write performance because model training is decoupled from write in Bigtable.

Ryan et al. [10] used SOSD to evaluate the read-only performance of RMI [1], RS [11], and PGM [3]. This benchmarking first focuses on analyzing the relationship between the index's size and the read-only performance of different learned indexes. They showed that the learned index structures in a worm-cache and tight-loop setting can provide better performance/size tradeoffs than several traditional index structures. Then, they offered a statistical analysis of hardware counters (i.e., the cache miss, the branch miss, instructions) to analyze why learned indexes acquire such excellent performance. Finally, they evaluated the learned indexes' performance in the memory-fenced, cold cache, and multi-threaded environments. They summarized that the learned indexes perform better than traditional indexes in those environments.

VII. Conclusion

In this paper, an end-to-end evaluation of typically learned indexes and traditional indexes has been conducted in an NVM-oriented key-value store (i.e., Viper), and we also deconstructed the learned index into four design dimensions, i.e., index structure, approximation CDF algorithm, insertion strategy, and retaining strategy, for detailed evaluations. The most important conclusions of this paper are as follows:

(i) Although the performance and space advantages of learned indexes are not as tremendous as what some previous work presented, the learned indexes show both higher performance and less space overhead compared with traditional sorted indexes supporting the scan operation.

(ii) The drawbacks of learned indexes mainly lie in that their recover time is usually slower than traditional indexes, and their performance is much easier to be affected by the key distribution of stored data.

(iii) Most previous research work on learned indexes focuses on improving the index structure, but we found that the key to design a high-performance learned index mainly lies in an excellent approximation algorithm. For example, ALEX achieves the best performance among all the evaluated learned indexes in most cases. The main reason lies in the design of inserted gaps in leaf nodes; it makes ALEX write-friendly for real-world workloads and also improves the key distribution to achieve simpler CDF approximation, fewer leaf nodes, and thus smaller querying cost.

(iv) In addition, many detailed suggestions on designing a better learned index on the four dimensions are given in this paper.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2019YFE0198600), the National Natural Science Foundation of China (No. 61972402, 61972275, and 61732014), the Outstanding Innovative Talents Cultivation Funded Programs 2021 of Renmin University of China.

References

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

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

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

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

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

[6] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: A high-performance learned index on persistent memory. *arXiv preprint arXiv:2105.00683*, 2021.

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

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

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

[10] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proceedings of the VLDB Endowment*, 14:1–13, 09 2020.

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

[12] Openstreetmap database. https://aws.amazon.com/public-datasets/osm.

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

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] Jiake Ge. Appendix, 2022. https://github.com/YunWorkshop/viper-learned-index/tree/learned_index/appendix.

[16] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. Novel online methods for time series segmentation. *IEEE Transactions on knowledge and data engineering*, 20(12):1616–1626, 2008.

[17] Zhenghua Xu, Rui Zhang, Ramamohanarao Kotagiri, and Udaya Parampalli. An adaptive algorithm for online time series segmentation with error bound guarantee. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 192–203, 2012.

[18] Timo Bingmann. Stx b+ tree, 2007. https://panthema.net/2007/stx-btree/.

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

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

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

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

[23] Mark H Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1987.

[24] Hervé Abdi et al. The method of least squares. *Encyclopedia of measurement and statistics*, 1:530–532, 2007.

[25] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.

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

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

[28] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534, 2018.

[29] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[30] Victor Costan. Skiplist, 2022. https://github.com/google/leveldb/tree/master/db.

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

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

[33] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. {Write-Optimized} dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[34] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

[35] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data structure primitives on persistent memory: an evaluation. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3, 2020.

[36] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[37] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: an efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[38] Peter Van Sandt, Yannis Chronis, and Jignesh M Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data*, pages 36–53, 2019.

[39] Tpc-c. http://www.tpc.org/tpcc/.

[40] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[41] Tpc-h. http://www.tpc.org/tpch/.

[42] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H Chi, Lyric Doshi, Tim Kraska, Andy Ly, Christopher Olston, et al. Learned indexes for a google-scale disk-based database. *arXiv preprint arXiv:2012.12501*, 2020.

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