ALT-index: A Hybrid Learned Index for Concurrent Memory Database Systems

Yuxin Yang[1], Fang Wang[1,3*], Mengya Lei[2], Peng Zhang[1] and Dan Feng[1]
[1] *Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of data storage systems and Technology, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[2]*Hubei University of Technology, Wuhan, China*
[3]*Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, China*
{yuxinyang, wangfang, zhangpeng19, dfeng}@hust.edu.cn, lmy_up@hbut.edu.cn

***Abstract*—The learned index technique has been widely explored as a strong competitor to traditional indexes. It adopts static learning-based models to fit the distribution of sorted data and locate keys through predictions, which shows outstanding query speed. However, frequent retraining is required when it comes to concurrent insertion scenarios. Despite existing studies introducing sparse slots and delta buffers to mitigate this effect, the read-write performance of the learned index still falls short of expectations, especially in concurrent conditions.**

In this paper, we first propose a novel hybrid index scheme that combines a read-efficient learned index with an insert-efficient Adaptive Radix Tree (ART) to realize high performance for read-write scenarios. However, it is not trivial due to expensive model prediction errors, complicated model hierarchy, and redundant node traversals. Therefore, we then introduce ALT-index, an efficient hybrid learned index with high concurrency for memory database systems. ALT-index highlights a delicate two-tier architecture where linear data are stored in the learned index without prediction errors and conflict data are hosted in the lower layer as an optimized ART. Besides, we develop a Greedy Pessimistic Linear (GPL) algorithm to support flattened data structures for concurrency. In the optimized ART layer, we introduce a fast and compact pointer buffer to further improve the overall performance. Experimental results conducted on various real-world datasets with 32 threads illustrate that ALT-index improves performance by up to 1.9x, 2.1x, and 2.3x compared with ALEX+, FINEdex, and XIndex in read-write-balanced scenarios, respectively.

***Index Terms*—Memory database, Index structure, Learned index**

I. Introduction

Index structures are the fundamental components that support fast data access for memory databases. Recently, there has been a surge of interest in Learned Index [1], which aims to supplant traditional indexes (such as B-tree) with machine learning models to improve index efficiency. The core idea of the learned index is using learning-based models to fit the distribution of sorted data and locate keys through predictions, which significantly minimizes query and space overhead. To train a learned index, the dataset will be stored and partitioned into several segments. These segments will be the input to train multiple models that approximate the Cumulative Distribution Function (CDF) curve of the dataset. Once a learned index is trained, each model can predict the position of a given key with O(1) complexity. Typically, the average read performance of a learned index is 1.5x–3x faster than that of a B-tree [1].

However, when dealing with insertion and concurrent scenarios, the learned index has limited performance. To be specific, the static learned models require a blocked retraining process to handle insertions. The retraining process is expensive especially when the insertions and read-retrain conflicts increase in the concurrent conditions. Our experiments find that existing learned indexes' performance decrease 68.2%-93.4% caused by insertions with 32 threads under read-write-balanced workloads compared to read-only workloads.

Existing studies explore techniques such as using delta buffers [2]–[4] and reserving sparse slots [5]–[7] to improve insertion performance of the learned index. Nevertheless, delta buffers require merging overflowed buffers with the learned models through the working or background threads, which becomes a significant bottleneck when the concurrency scales out. Reserving sparse slots in a model is another way to accommodate insertions. However, when there is no empty slot available for insertions, existing studies cannot gain high read-write performance resulting from read-write blocking [5] or cache invalidation [6]. Until now, none of the existing designs can fully solve the insertion issue of the learned index.

Different from the learned index, the Adaptive Radix Tree (ART) [8], one of the traditional indexes, is renowned for its outstanding insert performance [9]–[11]. ART is an optimized trie tree structure that employs a dynamic node size to minimize the tree height and gain efficient insertions. Nevertheless, ART exhibits inferior performance under read-only workloads compared with the learned index [2]–[6]. Therefore, an initial two-tier idea arises: we can place ART under the learned index to handle insertions, which prevents the loss of concurrent read-write performance caused by insertions in the learned index. In addition, the previous model located in the learned index can accelerate queries of ART, and it is not straightforward to utilize the hybrid learned index effectively due to the following challenges.

* Corresponding author

1) Expensive model prediction errors in concurrent scenarios: For model prediction errors when searching or inserting a key in the learned index, existing researchers suggest using secondary binary search or exponential search to correct them [12], [13]. When the number of threads is small, the impact of the secondary search is acceptable. However, in high-concurrency scenarios where inaccurate predictions occur frequently, the secondary search will quickly saturate the memory bandwidth, making the prediction error a significant system bottleneck [14].

2) Complicated model hierarchy restricts model location and concurrent access: Existing learned indexes use a tree-based structure to manage models containing the keys layer by layer. However, when the CDF curve of a dataset is hard to fit, current segmentation algorithms in learned indexes will create an excessive number of leaf models, which increases the traversal levels and performance overhead to locate a key from root to leaf. Moreover, the complicated model hierarchy will severely block structural modification operations due to lock contention in concurrent scenarios [15].

3) Redundant node traversals in ART exacerbates performance: If a key is not found in the learned index layer, we need to search for it from the root node of ART. However, we find that keys requested from the same learned index leaf model have the common prefix, suggesting they have multiple common parent nodes in ART. Traversals for these common nodes are redundant and costly.

To address the challenges above, we present **ALT-index**, **A** hybrid learned index scheme that combines **L**earned index with adaptive radix **T**ree. ALT-index enables the utilization of the remarkable read performance of the learned index and takes advantage of ART's outstanding write performance. In detail, this paper makes the following contributions:

- **A novel hybrid learned index architecture.** We introduce a novel concurrent hybrid learned index architecture of read-write efficiency that takes advantage of the read performance of the learned index and the write performance of ART. ALT-index keeps the linear data in the learned index layer and evicts the data with prediction errors to an optimized ART layer, which ensures accurate predictions in the learned index layer and replaces the secondary search with direct queries to the ART layer.
- **An efficient segmentation algorithm supports flattened data structure.** We design a Greedy Pessimistic Linear (GPL) algorithm to partition data efficiently for a flattened data structure. This flattened structure can reduce the overhead of model locating and improve the concurrency of the learned index. Moreover, ALT-index incorporates a retraining scheme for dynamic data distribution.
- **A fast and compact pointer buffer to eliminate redundant traversals.** We introduce a fast pointer buffer to link each leaf model to an intermediate ART node, thus eliminating redundant node traversals in ART. Furthermore, we also propose a pointer merging scheme, which significantly improves the space efficiency of the pointer buffer and the scalability of ALT-index.
- **Implementation with vigorous experiments.** We implement ALT-index in C++ and compare it against state-of-the-art learned indexes, including ALEX+, LIPP+, FINEdex, and XIndex. We use four real-world datasets with varying threads from 1 to 32 under read-write workloads. Experimental results show that ALT-index improves up to 1.9x–2.3x compared with the other solutions.

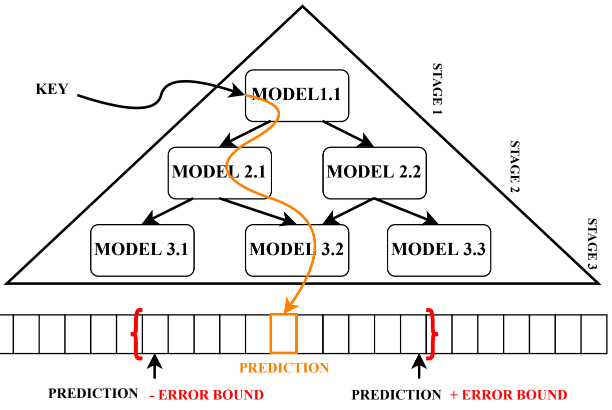

Fig. 1. The structure of RMI

In the remainder of this paper, we give background with motivation (Section II) and present the design of ALT-index (Section III). Afterward, we show the experiments and analysis of ALT-index (Section IV). We briefly discuss the inspiration and limitations of ALT-index (Section V). Finally, we review related work (Section VI) and conclude the paper (Section VII).

II. Background and Motivation

A. Static Learned Index

In recent years, the learned index [1] has been proposed to replace B-tree with machine learning models, which can predict the position of a key to finish index queries. For a sorted data array, modeling its CDF curve for indexing is a logical approach. To shorten the model training process, researchers use linear functions to approximate the CDF. A linear function can be represented as $prediction(key) = a * (key) + b$, where a and b are the parameters stored in the data structure. In other words, a linear function is similar to the hash function in a hashmap [16], because both require one single calculation to locate a key. Due to the uncertainty of the model, the learned index may have prediction errors at runtime. To restrict the prediction error and enable accurate search, the learned index defines an $error\_bound$ metric that the correct position must fall in an error range $[predict - error\_bound, predict + error\_bound]$ around the prediction position. Thus, the learned index needs a binary search or an exponential search within the error range to locate the key when the prediction is inaccurate.

However, a single linear model fails to approximate complex CDF curves efficiently. To address this issue, Kraska et al. [1] introduce Recursive Model Index (RMI). As shown in Fig. 1, RMI builds a directed acyclic graph with multiple upper stages of models. These upper stages can predict picking a sub-model in the next layer. When it reaches the leaf model at the bottom, the leaf model finishes the last mile search to find the key position. A secondary search around the predicted position is required once a wrong prediction occurs. Instead of searching the entire data array, the leaf models only need to approximate a small interval in a subset of the whole dataset, which reduces the model error bound. Despite the previous works such as RMI and RadixSpline [17] have greatly improved the read performance of databases, the static learned index still has the crucial drawback that it does not support any structure modifications or insertions due to expensive model retraining.

B. Updatable Learned Index

Inspired by RMI, recent studies aim to build an updatable learned index to replace the traditional indexes through sparse slots and delta buffers. ALEX [5] and LIPP [6] preserve sparse slots in their data nodes to handle insertions in place. Inserted data will be put into the slots which are not occupied. For insertions to an occupied slot, ALEX designs a data-shifting scheme to keep the data sorted, which consumes 25.2% overhead of the entire insertion. When a data node's fullness reaches a predefined level, ALEX will make node splits and reserve extra sparse slots in the new data nodes. Additionally, ALEX employs a cost model to decide when to do structural modification operations. LIPP pays much more attention to prediction conflicts caused by insertions. It mitigates the overhead of secondary queries by segregating conflict data into separate child nodes. Furthermore, the Fastest Minimum Conflict Degree (FMCD) algorithm is introduced to enable rapid reconstruction and adjustment of subtrees, thereby minimizing the overall height of LIPP. LIPP creates new child nodes if the predicted position is occupied, which consumes 40.7% overhead of the entire insertion. Based on ALEX and LIPP, ALEX+ and LIPP+ further adopt an optimistic scheme [14] to implement concurrency.

Another way to address the insertion issue is using delta buffers. Data inserted into the learned index will be put into delta buffers. Delta buffers will be merged periodically with the learned models. PGM-index [7] handles the new insertions with the idea of LSM trees [18]. It builds multiple sub-indexes to cover subsets of keys in the next layer. After that, to guarantee the keys stay sorted, insertions are handled by establishing new sub-indexes by merging smaller ones with the new key. Besides, there are other solutions based on the idea of delta buffers. The difference between other related work, e.g. FINEdex [2] and XIndex [3], is the granularity of delta buffers. XIndex uses a dynamic RMI model on the top, and data are located in the bottom leaf node of RMI. XIndex allocates a delta buffer for each leaf node and the retraining procedure will be done by background threads. FINEdex has a smaller granularity of delta buffers. It allocates a level bin structure as a delta buffer for each data slot, which can prevent write conflicts within a node's delta buffer.

TABLE I
STATISTICS OF THE CONCURRENT UPDATABLE LEARNED INDEXES AND ART ON TWO 200 MILLION ITEMS DATASETS, THE LIBIO [19] AND THE OSM [20] DATASETS. THE THROUGHPUT AND THE LATENCY ARE PRESENTED IN MILLION OPS/SEC AND μs

Index	Dataset	Throughput	P99.9 latency	Limitation
ALEX+	libio	50.69	3.51	data shifting
	osm	18.18	43.76	
LIPP+	libio	7.69	30.88	statistic info
	osm	5.54	46.85	
FINEdex	libio	28.76	9.06	predict error
	osm	24.64	7.21	
XIndex	libio	27.56	6.59	predict error
	osm	24.19	3.59	
ART	libio	48.81	5.37	node traversal
	osm	37.20	9.59	

To evaluate the effectiveness of the sparse slots and delta buffers solutions, we conduct an experiment with two real-world datasets under read-write-balanced workloads. The results are shown in Table I. Overall, none of the existing works can simultaneously maintain high throughput and low tail latency on various datasets. ALEX+ has a high tail latency on the osm dataset because the insertion conflicts trigger data-shifting scheme. LIPP+ gains terrible performance in concurrent scenarios where every node in the insert path needs to update its statistic information, which will introduce cacheline invalidation, especially on the root node [14]. XIndex uses a masstree as delta buffers, and FINEdex introduces a level bin data structure similar to B-tree. FINEdex and XIndex have similar performance due to their delta buffer solutions. Moreover, the data are intensively sorted in the leaf nodes of XIndex and FINEdex with a given error bound. The average prediction error in a linear model is still high [21], and it will severely damage the lookup performance when the number of threads increases.

C. Motivation

The above designs focus on generating an updatable concurrent learned index to replace the traditional indexes. Sadly, neither the sparse slots nor the delta buffers can effectively address the insertion and concurrency issues. On the other hand, unlike the learned index, which is known for its good performance under read-only workloads [14], ART is widely used in memory database systems due to its excellent insert performance [8]. Our tests show that the average read-write performance of ART is faster than learned indexes in Table I. This observation gives us a different way from previous solutions to build an efficient index for memory databases. In this work, we attempt to take advantage of the learned index's read performance and ART's write performance through a combination. Therefore, a new hybrid index idea that combines the learned index with ART is proposed. To implement this idea, we need to meet the following goals.

1) Efficient hybrid learned index construction: We need to draw a blueprint on how to construct the learned index and ART. The insertions and prediction errors are two key factors that restrict the concurrent read-write performance of the learned index. Existing schemes indicate that insertions are difficult for the learned models to handle, and prediction errors will introduce a secondary search to correct them. Moreover, prediction errors occur more frequently in concurrent conditions than in single-thread conditions, and the secondary search will quickly saturate the memory bandwidth [14]. Therefore, we must design the architecture of a hybrid learned index that can efficiently handle insertions and prevent prediction errors in the learned index.

2) High concurrency: It is imperative to enhance the concurrency of indexes to fully utilize the computing power of multi-core processors. However, the concurrency of existing learned indexes is often limited by high read-write conflicts and low cache hit rates. Such as ALEX+ [14], which shows low throughput on the osm dataset, attributed to thread collisions caused by frequent structure modifications and retraining. LIPP+ [14] uses counters for each node to get information about insertions. But these counters introduce frequent updates, which cause cache invalidation, especially on the root node. Therefore, we should ensure that our hybrid learned index can mitigate thread collisions and minimize the cache invalidation to obtain high concurrency.

3) High performance with low space consumption: An efficient in-memory database index structure must have both high performance and low space consumption. However, it is difficult for existing works to achieve both requirements at the same time. For example, LIPP allocates a great number of sparse slots, which sacrifice space efficiency to keep high performance. FINEdex [2] and XIndex [3] use multiple delta buffers with different granularities to improve insertion performance. However, when dealing with workloads with frequent insertions, the number of pre-allocated delta buffers increases, which consumes too much space for high performance. Therefore, we should improve the performance of our design while minimizing space consumption.

III. ALT-index Design

A. ALT-index Overview

In this section, we present ALT-index, a high-performance hybrid learned index tightly integrated with ART. ALT-index is an in-memory, updatable index that can take advantage of the learned index and ART for outstanding performance and high concurrency. Our design goals are: (1) ensuring competitive lookup times; (2) supporting efficient insert operations; (3) keeping scalability and high concurrency; and (4) minimizing memory consumption.

First, we introduce our novel hybrid learned index architecture. The overall structure of ALT-index is shown in Fig. 2. We put the data that can be easily predicted with a linear function model into the learned index layer, and peel out the data points that incur prediction errors. These data points and the newly inserted conflict data will be put into ART-OPT layer at the bottom. Further, this approach ensures the accuracy of predictions, thereby preventing prediction errors

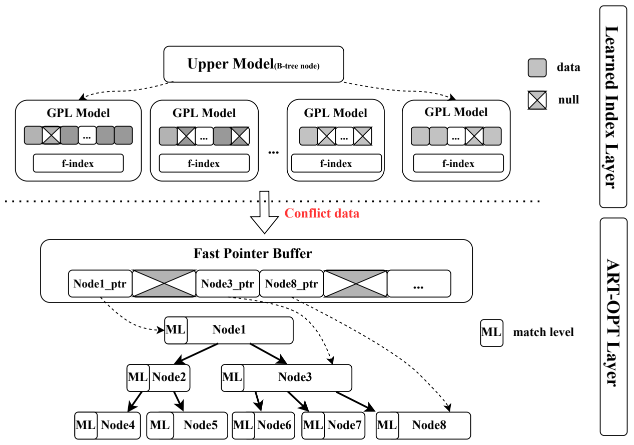

Fig. 2. The hybrid data structure of ALT-index

in the learned index layer. Moreover, the secondary search step in the learned index will be replaced with the query in ART. Significantly, this two-layer hybrid index leverages the superior read performance of the learned index and capitalizes on ART's read-write performance simultaneously.

Second, in the learned index layer (Section III-B), we introduce a GPL algorithm to partition data efficiently. This scheme swiftly divides the dataset into linear segments through a suggested error bound and initializes model variables. The learned index layer maintains GPL models sorted by first keys at its bottom to enhance concurrency. As depicted in Fig. 2, a sequential GPL model layer for linear data storage is deployed in ALT-index. Thread collisions are minimized through this flattened data architecture, while the upper model of the learned index functions as a sorted array, facilitating efficient binary search operations.

Third, in ART-OPT layer (Section III-C), we manage the data that cannot be absorbed by GPL models, referring to conflict data during bulk loading and runtime insertions. ALT-index incorporates a novel shortcut mechanism: a fast pointer buffer with a merging scheme that reduces latency during a search miss in the learned index layer. This structure improves search latency and effectively manages concurrency between the learned index and ART with minimal space consumption. As illustrated in Fig. 2, each GPL model holds a fast pointer index that swiftly locates the corresponding entry in the buffer.

In the remainder of this section, we show our concurrency control design (Section III-E) in different conditions. Also, ALT-index supports a dynamic retraining process when necessary. We design a simple retraining temporal buffer to expand the GPL model (Section III-F). We implement ALT-index with practical operations with low overheads (Section III-G).

B. Learned Index Layer

We propose a GPL algorithm to make partitions. It can split a dataset into segments efficiently with O(n) complexity in the learned index layer.

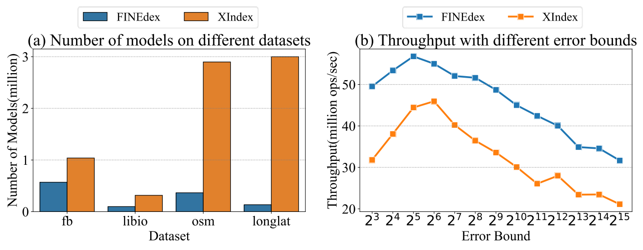

Fig. 3. Model number and error bound factors on four datasets with 200M keys under read-only workload

1) Problems in existing learned index works: A single linear model can result in significant prediction errors when approximating complex CDF curves. Consequently, researchers have employed multiple models to improve the approximation. Various existing split algorithms are available for the learned index to make segments quickly and fit into models. However, existing partition algorithms result in many models, severely damaging the performance of learned indexes.

We test XIndex and FINEdex with 200 million keys on four real-world datasets, setting the error bounds suggested in their papers. The average model number reflects the lookup time required for the model location. The outcomes are depicted in Fig. 3(a), illustrating that although the Learning Probe Algorithm (LPA) in FINEdex decreases the number of models compared to the dynamic RMI algorithm in XIndex, the number of models still maintains at the million level, while our ALT-index controls the number of models to thousand level shown in Fig. 6(a) to keep high performance. Besides, it is possible to facilitate a bigger error bound to reduce the number of models. Nevertheless, the increase in error bound adversely affects throughput due to the larger range necessitating secondary searches. We evaluated the throughput of FINEdex and XIndex under read-only workloads across various error bounds. As shown in Fig. 3(b), indexes reach peak throughput with error bounds around 32 and 64. However, they suffer a severe decline as the error bound further expands. The results indicate that current learned indexes cannot deal with the number of models and prediction errors at the same time. However, ALT-index decouples these issues using a two-tier structure, which puts the prediction errors into ART layer. In this condition, ALT-index only needs to handle the number of models and preserve linear data as much as possible in the learned index layer.

2) Greedy Pessimistic Linear Algorithm: We introduce the GPL algorithm to create linear partitions efficiently throughout a dataset scan. First, we manually set a predefined error bound, which triggers the segmentation until the prediction error of newly inserted data exceeds it. The prediction error of a data point will influence the following data points' error by updating the upper/lower slopes. This scheme is pessimistic because it assumes that once a prediction error happens, we need to make partitions as quickly as we can. Moreover, each linear function of a GPL model is assumed to go through the first point of each segment. The GPL procedure is detailed in Algorithm 1.

Algorithm 1 Greedy Pessimistic Linear Algorithm

```
1:  Init $cur\_position \leftarrow 0, N \leftarrow remaining\_keys$
2:  while $cur\_position < N$
3:    Calculate $new\_slope$ of the current and the first point
4:    if $new\_slope > upper\_slope$
5:      $upper\_slope \leftarrow new\_slope$
6:    if $new\_slope < lower\_slope$
7:      $lower\_slope \leftarrow new\_slope$
8:    Calculate $upper\_error$ and $lower\_error$;
9:    if $MAX(upper\_error, lower\_error) > \epsilon$
10:     return $cur\_position$;
11:   $cur\_position$++;
12: return $cur\_position$;
```

To begin with, we set up a few parameters to better illustrate the idea behind our design. We present $upper\_slope$ and $lower\_slope$ as the maximum and minimum line slopes that go through the first point and an intermediate point. The $upper\_slope$ and $lower\_slope$ are used to calculate the prediction error. The calculated prediction error will be compared with the parameter ϵ, which is a GPL model's prediction error bound. Every time a new point is added to a model, as shown in Algorithm 1, we check whether the $upper\_slope$ and $lower\_slope$ need to be updated with the added point. The next phase involves adding a new point to the GPL model if $max(upper\_error, lower\_error) \leq \epsilon$. Once $max(upper\_error, lower\_error) > \epsilon$, the GPL algorithm will make a split and a new GPL model is generated. Then, the following data point will be used to initialize the next fresh GPL model.

As shown in Fig. 4(a), points 1,2,3 initialize $upper\_slope$ and $lower\_slope$ with blue lines. When point 4 is added, GPL model updates $lower\_slope$ with the line colored in red. After point 5 is added, GPL model checks the $upper\_error$ and $lower\_error$ is smaller than ϵ. Once the error of a new point is bigger than ϵ, we repeat the GPL partition process with this new and following points.

To expand the performance of a GPL model, we present a series of optimizations for model training after the partition processing. First, to decrease the prediction error of a GPL model, we take an array gaps scheme to handle some coming insertions into a model. Second, a key lookup process will first search the upper model to locate a GPL model. The number of GPL models in our experiments is small, so we use an optimized binary search rather than a radix table. Third, for searching in a GPL model, we use a bitmap to reduce the unnecessary slot checks in the search procedure.

3) Algorithm comparison: The GPL algorithm gains advantages compared to other algorithms. When a data point is located outside the linear function and leads to a prediction error, this error will scale out during the following training processing until it exceeds the threshold ϵ. The Shrinking Cone algorithm mentioned in FITing-tree [4] uses two lines to go through the same point to make segments. As shown in

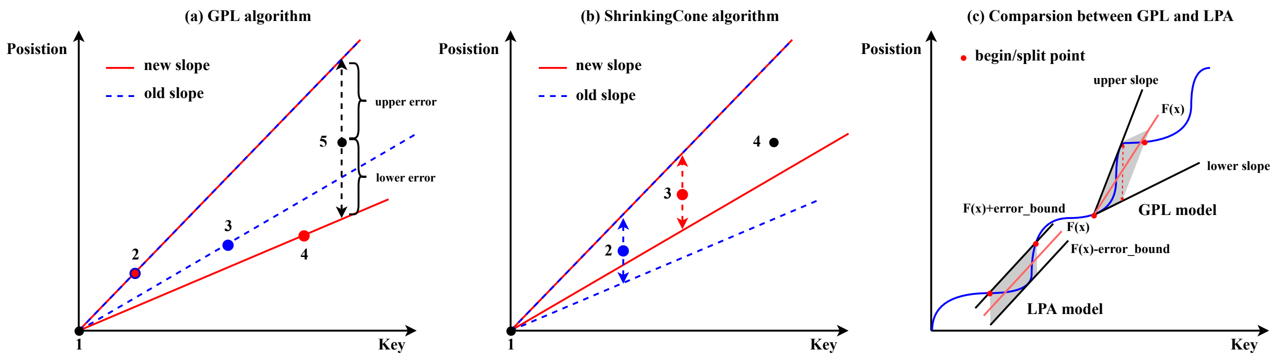

Fig. 4. Comparison between GPL algorithm in ALT-index, ShrinkingCone algorithm in FITing-tree, and LPA algorithm in FINEdex.

Fig. 4(b), the Shrinking Cone model checks whether a new point is located within the area between two slopes. On the contrary, when a new point (x,y) is added to the segment, the model's upper and lower slopes are updated to two lines that pass through (x,y+error_bound) and (x,y-error_bound). This will introduce more frequent updates of two slopes than GPL, severely damaging the segment performance.

Moreover, in Fig. 4(c), we show how the LPA algorithm in FINEdex and the GPL algorithm in ALT-index approximate the CDF of a dataset. The ϵ of a GPL is the diagonal line of the parallelogram, which is vertical to the x-axis. Based on the similarity theorem of triangles, the prediction error of each data point in this parallelogram is smaller than ϵ. However, the LPA cannot make segments efficiently when it comes to too many data points with small prediction errors. Besides, our GPL algorithm takes into account these errors to help make segments with O(n) complexity.

C. *Optimized ART Layer*

ART [8] is an optimized trie for main memory indexing and adapting to data distribution. In ALT-index, we leverage ART to accommodate the conflict data from the learned index layer, ensuring accurate model predictions and faster queries. Nevertheless, if the origin ART is used directly in the ALT-index, it will incur expensive secondary search overhead. Specifically, a conflict data query will find incorrect keys at the predicted locations in the learned index layer. Then, an expensive secondary search starting from the root node in the ART will be needed to locate the data, leading to high latency for the entire query operation. To solve this problem, we propose an optimized ART (ART-OPT) scheme, which can meet the following design goals.

Low latency. The model location has already restricted the key to a small range in the learned index layer, so searching from the root node of ART is a resource-wasting behavior. ALT-index deploys a fast pointer buffer structure to link each GPL model to the subtree of the ART, which can largely cut down the overheads of ART layer.

Minimal space consumption. Introducing an extra fast pointer buffer will consume space consumption in ART layer. Thus, we improve our fast pointer buffer with a merge scheme to minimize space consumption. Moreover, we only add a variable to each ART node to identify the prefixes that have already matched. These optimizations above can further eliminate space consumption.

1) Fast pointer buffer construction: We propose a fast pointer buffer between the learned index layer and ART to improve ALT-index. The fast pointer buffer stores the pointers from GPL models to the tree nodes in ART. The process of building the fast pointer will begin after the entire hybrid index has been initialized. ① We obtain the first keys from two adjacent GPL models. ② We look up the maximum corresponding prefix node in ART. As shown in Fig. 5, if the query footprint of two keys is in different nodes, we create a pointer to the parent node and add it to the fast pointer buffer. ③ We return the index of this fast pointer in the fast pointer buffer array to the upper GPL model. Within the buffer, duplicate fast pointers are merged. Consequently, the number of fast pointers becomes less than that of GPL models.

2) Space efficiency optimization: First, although queries for conflict data can easily jump to an ART node through a fast pointer, we cannot get the prefix that has been matched in the parent nodes. To configure the matched prefixes, we add a variable called $match\_level$ in each node to record the length of the matched prefix to assist the fast pointer lookup. Second, we find fast pointers pointing to the same ART node, which causes redundant space. So we merge the pointers pointing to the same node to ensure space efficiency. Moreover, this approach can keep data consistent and simplify the management of the fast pointers across different GPL models during conflict insertions, especially when ART has structure modifications.

3) Workflow of ART-OPT layer: The lookup workflow is as follows. When a query in the learned index layer finds the data in the expected slot is occupied, it retrieves the fast pointer index saved in the current GPL model to look up the fast pointer buffer. After locating the corresponding fast pointer through this index, the query can continue directly from the intermediate ART node pointed by the fast pointer. Thus, ALT-index can finish the secondary query by traversing

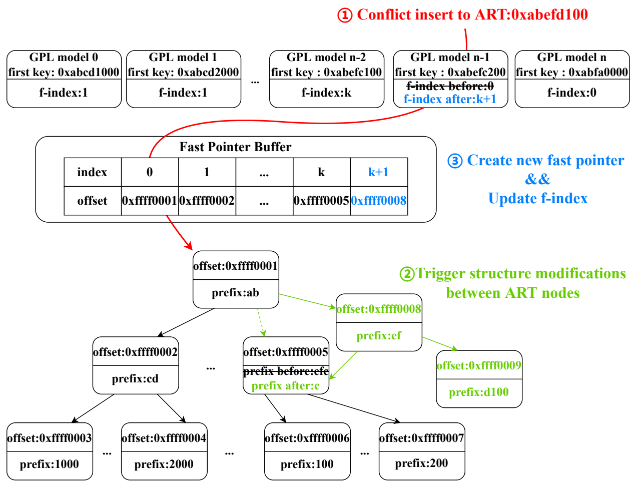

Fig. 5. Fast pointer construction and modification

this subtree of ART that can greatly alleviate the performance bottleneck of accessing the root node in concurrency situations and effectively cut down the latency.

Insertion is similar to the lookup in most conditions. However, if an insertion in ART-OPT layer causes the structure modifications, the fast pointer may become invalid. These invalid pointers can lead to illegal or wrong pointer accesses and cause segment faults. We conclude these issues with two specific scenarios.

① Prefix extraction. After inserting a conflict data item, the node corresponding to a fast pointer undergoes a prefix extraction operation. The newly extracted prefix will be put into a newly created node and set as the parent node of the former node. In this case, this GPL model's fast pointer needs to be updated to this newly created node, which ensures data consistency in the fast pointer buffer. By the way, if the prefix extraction operation affects a node without being pointed by any fast pointers, then this structure modification will not cause invalidation. These steps are shown clearly in Fig. 5.

② Node expansion. After inserting a conflict data item, ART undergoes a node expansion operation during insertion using the fast pointer (e.g., a node expands from 16-span to 48-span). ALT-index expands a node by creating a new one, which indicates that the old offset stored in the fast pointer buffer is invalid. In that case, the fast pointer buffer will find that invalid pointer and update its value to prevent illegal visits.

D. Error bound and performance analysis

It is evident that the error bound has a substantial impact on both the learned index and ART-OPT layers, which collectively determine the overall performance of the ALT index. Initially, the error bound influences the number of models in the learned index layer. To find the relation between the error bound and the number of GPL models, as shown in Fig. 6(a), we first test the number of GPL models with different error bounds. We observe an inversely proportional relationship between the error bound and the total number of models when initializing ALT-index with the same number of data. Based on our test results, we briefly summarize an equation as follows:

$$N_{total} = \delta_h * \epsilon * N_{model} \tag{1}$$

In (1), ϵ is the error bound, and δ_h represents the difficulty of fitting the CDF curve of a dataset with linear functions. To reflect the number of data in the lower layer, we set α representing the proportion of data in ART-OPT layer, shown in (2).

$$N_{ART} = N_{total} * \alpha \tag{2}$$

Moreover, based on the algorithm analysis in Fig. 4(c), ϵ determines the size of the parallelogram of a GPL model. If we replace an ϵ GPL model with n small GPL models equipped with ϵ/n, the total size of these GPL parallelograms is n times smaller than the old one. The bigger the total size of these parallelograms, the more conflict data will be put into ART-OPT layer. We use S_ϵ to represent the total size of the GPL parallelograms with ϵ error bound. Based on our analysis, we can use the total size of GPL parallelograms to reflect the number of conflict data in ART-OPT layer. Finally, the relation between the error bound and conflict data in ART-OPT layer is shown in (3).

$$\frac{N_{ARTm}}{N_{ARTn}} \approx \frac{S_{\epsilon_m}}{S_{\epsilon_n}} = \frac{\epsilon_m}{\epsilon_n} \tag{3}$$

The average latency of ALT-index includes the latency in two layers. In the learned index layer, the model locating procedure consumes $log_2 N_{model}$ latency, and the calculation latency in a GPL model is constant. Lookup in ART-OPT layer has O(k) complexity. Based on our analysis and the equations above, the total average lookup latency for ϵ can be modeled by the following expression as shown in (4), where k_{cal} is the calculation expense of GPL model, ϵ_0 is the error bound that can host the whole dataset in one GPL model, α_0 is the proportion of data in ART-OPT layer when there is only one GPL model in the learned index layer, k_{ART} represents the time complexity of the optimized ART, and c is a constant representing the latency of a cache miss on the hardware.

$$T_{avg}(\epsilon) = c\,[\underbrace{\log_2\left(\frac{N_{total}}{\delta_h * \epsilon}\right) + k_{cal}}_{T_{learned\_index\_layer}} + \underbrace{\alpha_0 * \frac{\epsilon}{\epsilon_0} * k_{ART}}_{T_{ART-OPT\_layer}}] \tag{4}$$

Through (4) we can find that the throughput largely depends on the learned index layer when the error bound is small. However, when the error bound increases, ART-OPT layer will consume more time because of the increasing conflict data. The constant value varies across different datasets. So the optimal error bound depends on the data distribution of each dataset.

$$T'_{avg}(\epsilon) = c\,[-\frac{1}{\ln 2 * \epsilon} + \frac{\alpha_0 * k_{ART}}{\epsilon_0}] \tag{5}$$

In (5), the throughput will peak when the derivative of the average latency function equals 0. However, we can only get

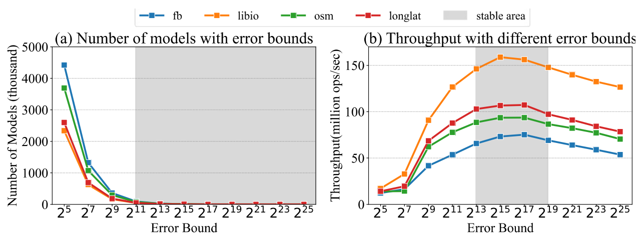

Fig. 6. The relationship between error bound and throughput of ALT-index on four datasets.

the number of items in ALT-index N_{total} when we initialize ALT-index, which strongly correlates with ϵ_0. Thus, we make a few assumptions about these constant values referring to various real-world datasets and replace ϵ_0 with N_{total}. Finally, we suggest users set $N_{total}/1000$ as the error bound when initializing ALT-index.

We test different error bounds on various datasets to verify our conclusion. As shown in Fig. 6(b), the throughput will quickly increase as the error bound increases when it is small. As the error bound increases, the throughput increases slower and finally begins to drop after it exceeds the peak point, which corresponds to our latency formula. Even though the throughput decreases after reaching the extreme point, ALT-index still performs well. We called the area with slow increasing or decreasing rates a “stable area”. This area ensures the high performance of ALT-index even if our suggested error bound based on our assumptions is not optimal across different datasets.

E. Concurrency

ALT-index shows outstanding performance in concurrent scenarios. First, the learned index layer can handle concurrency through an optimistic scheme with slot granularity. Second, we use spin locks in the fast pointer buffer. Third, we use an existing optimistic lock coupling implementation [22] in ART-OPT layer. The following part discusses the different concurrent issues of ALT-index.

1) Write-write conflicts: Write-write conflicts occur when multiple threads simultaneously write data into two layers. First, in the learned index layer, ALT-index assigns an atomic variable to each data slot in the GPL model to record the current data version. A writer will read the current version number of the slot when a write operation begins. If no thread is writing this slot (the version number is even), the writer will increase the version number to odd. When the write operation is completed, the version number will increase to even. When other threads begin a write operation on the same slot, they check the version number to prevent conflicts. If the version number is odd, they will retry until the previous thread’s write operation is completed and the version number is updated to even before the current write operation is performed. Second, new fast pointers are appended to the fast pointer buffer using spin locks to address conflicts, and the concurrency control of ART prevents the update conflicts of fast pointers. We improve the write function of ART to fit our fast pointer design mentioned in Section III-C in concurrent scenarios.

2) Read-write conflicts: For read-write conflicts, ALT-index employs an optimistic scheme in the learned index layer to ensure the correctness of data. Read operations will get the version number of the data slot and retry if the version number is odd. After retrieving the data from a slot, it will read the version number to confirm the data is the latest. If the version number is changed, the read operation will retry to get the latest data. In addition, the existing optimistic lock coupling implementation of ART with node granularity [22] to handle read-write conflicts.

F. Dynamic Retraining

The cost associated with model reconstruction significantly affects performance. We need to develop a dynamic adjustment of the entire index to accommodate changes in data distribution. To address this issue, we adopt a partial refactoring strategy. The model is expanded only when the insertions of a specific GPL model exceed its build size, suggesting that the GPL model is crowded and the following insertions will be put into ART layer. The dynamic retraining process consists of three steps:

- **Expansion preparation.** We create a temporal empty data buffer twice larger than the old GPL model. In addition, we double the new train slope to the original train slope.
- **Data eviction.** For an expansion GPL model, every insertion will evict old data to the temporal buffer once the predicted position of this key is occupied in the GPL model. New data will be directly inserted into the temporal buffer.
- **Expansion finishing.** When the number of insertions in the temporal buffer equals that of the old GPL model, we evict the remaining data in the old GPL model to the temporal buffer and update the model pointer.

In addition, if we find a key in ART layer but the predicted position in the GPL model is empty after the retraining, we will write the data back to the GPL model and delete it from ART. If the retraining GPL model is the last one, we will create a new GPL model behind it to handle the insertions that fall out of the range. Overall, a significant portion of the data can be effectively stored in the learned index layer through dynamic retraining, thereby maintaining the high performance of ALT-index.

G. Practical Operations

ALT-index supports common operations. Every operation may require two processes due to the hybrid construction. All operations first find the corresponding GPL model through an upper model using a binary search and look up the predicted position of the requested key by a calculation. The procedures of different operations became diverse after that.

Search and Update: If the predicted position is empty, it indicates that this key does not exist, for even a conflict data must be predicted to a position first. Otherwise, ALT-index

Algorithm 2 Search and Insert

```
Search $(key)$
    $gpl \leftarrow$ get_gpl$(upper\_model, key)$
    $pred\_pos \leftarrow$ make_prediction$(gpl, key)$
    $val \leftarrow$ empty
    if $!\, pred\_pos.occupied$
        return $val$
    if $gpl[pred\_pos].key == key$
        $val \leftarrow gpl[pred\_pos].val$
    else
        $val \leftarrow$ get_from_art$(key, gpl.fast\_index)$
        if $gpl[pred\_pos].key == 0$
            $gpl[pred\_pos] \leftarrow$ write_slot$(key, val)$
            remove_from_art$(key)$
    return $val$

Insert $(key, val)$
    $gpl \leftarrow$ get_gpl$(upper\_model, key)$
    $pred\_pos \leftarrow$ make_prediction$(gpl, key)$
    if $pred\_pos.occupied$ && $gpl[pred\_pos].key! = 0$
        $ret \leftarrow$ write_to_art$(key, val, gpl.fast\_index)$
    else
        $gpl[pred\_pos] \leftarrow$ write_slot$(key, val)$
        $ret \leftarrow true$
    return $ret$
```

will check whether the key stored in the predicted position is correct. If the check fails, it uses the fast pointer index in the GPL model to search in ART-OPT layer shown in Algorithm 2. A write-back scheme will move the data from ART to GPL if necessary. An update can locate the relevant data item through a search operation and update data in place.

Insert and Remove: For insertion, ALT-index writes the data into the empty slot in place if the predicted position is empty. If not, it inserts the data into ART-OPT layer and updates the fast pointer buffer if necessary. For remove, ALT-index first locates the key through a search operation and sets the key to zero for remove operations if the deletion happens in GPL model. If not, it removes the key from ART-OPT layer and updates the fast pointer buffer if needed.

Range Query: ALT-index finishes a range query in two steps: a range query in the learned index layer and a range query in ART-OPT layer. The final scan result is merged from these two steps.

IV. Evaluation

A. Experiments Setup

1) Datasets: The Search on Sorted Data (SOSD) benchmark is a toolset designed to test learned indexes. We select two datasets from the SOSD [23] in our experiments. In addition to these datasets, we utilize several real-world and synthetic datasets to enhance the diversity of our experimental data. Each of the datasets above contains 200 million 8-byte records. Duplicated datasets are excluded because our work and competitors don't support them. (1) The "fb" dataset consists of user IDs from Facebook. (2) The "libio" dataset [19] represents the repository ID from the libraries.io website. (3) The "osm" dataset consists of uniformly sampled locations from OpenStreetMap [20]. (4) The "longlat" dataset is extracted from OpenStreetMap which combines longitudes and latitudes using a transformation.

2) Workloads: We bulkload 50% of the datasets to initialize the indexes. The workload configurations are as follows: (1) Read-Only workload: 100% reads. (2) Read-Heavy workload: 80% reads and 20% insertions. (3) Read-Write-Balanced workload: 50% reads and 50% insertions. (4) Write-Heavy workload: 20% reads and 80% insertions. (5) Write-Only workload: 100% insertions. (6) Hot write workload: We reserve 20M consecutive keys in each dataset for insertions to trigger the retraining process frequently after initializing indexes with 100M keys.(7) Scan workload: Uniformed 10 million scan queries containing 100 keys each. Read operations follow a zipfian distribution with 0.99 theta value, and insertions are distributed uniformly in each dataset.

3) Competitors: We conduct thorough experiments between ALT-index and state-of-the-art learned indexes. XIndex is an updatable learned index based on the RMI model, supporting update operations through delta buffers associated with each learned index model. FINEdex is based on the LPA algorithm, enabling insertion operations via level bins within each data slot. ALEX+ [24] and LIPP+ [14] are the concurrent implementations of ALEX and LIPP using an optimistic locking scheme. Besides, updatable learned indexes such as FITing-tree and PGM-index do not support concurrent scenarios, so we exclude them from our experiments. We add ART [22] with optimistic lock scheme as a competitor.

4) Configurations: We adopt the settings recommended by the authors for ALEX+, LIPP+, XIndex, and FINEdex. For ALT-index, we set ϵ to $bulkload\_number/1000$ in the learned index layer. The fast pointer and dynamic retraining scheme are enabled by default.

5) Hardware and platform: The experiments are implemented on a Linux server running Ubuntu 18.04, equipped with two 18-core Intel(R) Xeon(R) Gold 6240 CPU clocked at 2.60GHz, and 186GB of DRAM. We adopt the C++17 standard, and hyperthreading is disabled during our experiments to ensure the stability of the results. Our code is compiled using GCC 9.4.0 under the O3 optimization level.

B. Read-Only Workload

In read-only workload, ALT-index outperforms FINEdex, XIndex, with throughput improvements of 104.8%, and 131.5%, respectively, as shown in Fig. 7(a). FINEdex and XIndex suffer from prediction errors leading to considerable secondary queries. ALT-index has competitive throughput with ALEX+ and LIPP+, attributed to the flattened learned index layer. In addition, the two-tier data structure in ALT-index has low tail latency. This reflects that our fast pointer buffer scheme has largely cut down the search overhead in ART.

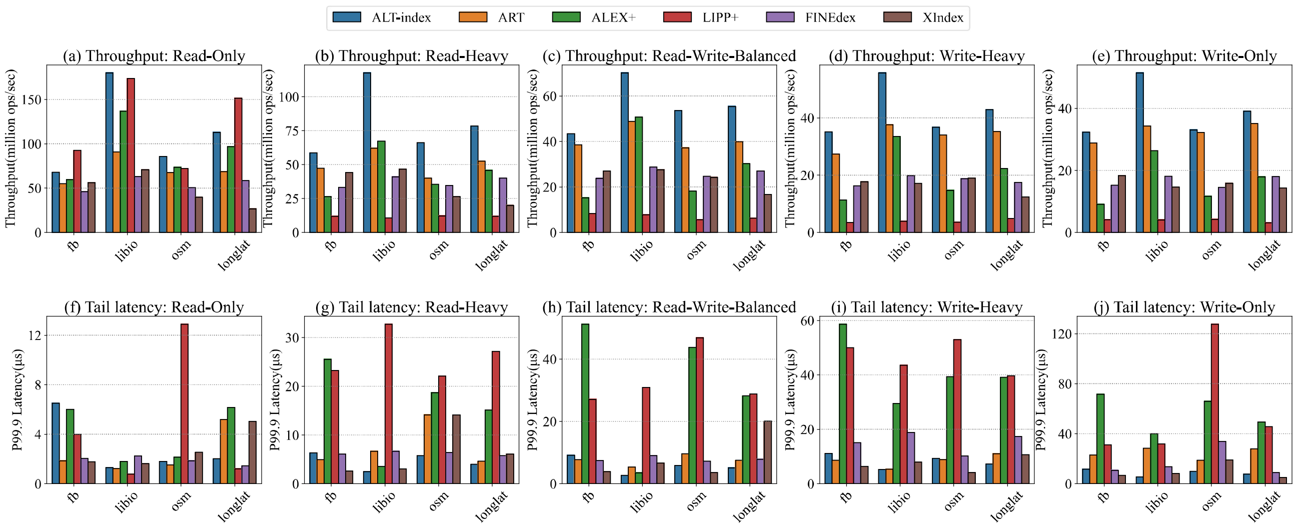
Fig. 7. Throughput and P99.9 tail latency under various workloads using 32 threads on four datasets.

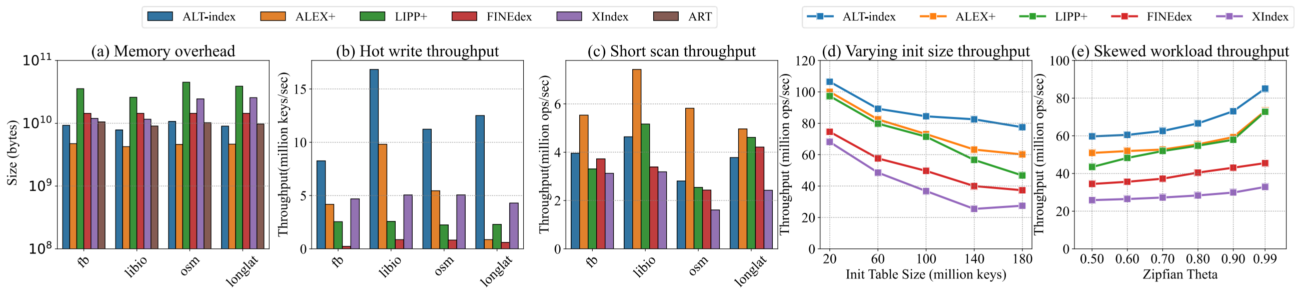
Fig. 8. Memory overhead on different datasets is shown in (a). Hot write and short scan workload are evaluated in (b-c). The throughput with different init table sizes and skewed workload on the osm dataset are concluded in (d-e). All experiments use 32 threads.

C. Read-Write Workloads

ALT-index continues to exhibit excellent performance compared with competitors in read-write workloads. The result is shown in Fig. 7(b-e). As the proportion of write operations increases, both ALT-index and other indexes suffer a decline in throughput to varying degrees. FINEdex and XIndex store newly inserted data in delta buffers, which exhibit degraded performance as the data increases. Because of the small granularity of delta buffers in FINEdex, the tail latency and throughput perform better than XIndex.

LIPP+ has a bottleneck in concurrent write scenarios caused by its statistic updates. Notably, ALEX+ gains a high tail latency due to its data-shifting scheme when the insert ratio increases. Origin ART design always has a lower throughput than ALT-index due to node traverses. Overall, ALT-index keeps both a high throughput and cuts down the tail latency.

D. Space Overhead

We initialize the indexes with 100M keys and insert the remaining keys. As shown in Fig. 8(a), the test results reveal that ALT-index takes less memory space than others except ALEX+. LIPP+ reserves too many empty slots in its nodes to improve write performance, but only a few are used during the insertion. Both XIndex and FINEdex cost more space because of their delta buffer scheme.

E. Hot Write and Scan Workloads

We use the hot write experiments to reflect the performance of indexes when the data distribution has changed. In Fig. 8(b), ALT-index shows better throughput than others due to our dynamic retraining scheme, in which the subsequent insertions and queries average the retraining expense. XIndex has a stable performance because it introduces several background threads to handle the retraining process while others are not.

ALT-index separates the linear and conflict data, leading to dual scans in GPL models and ART-OPT. We use a scan workload to reflect this factor. As shown in Fig. 8(c), ALEX+ has the best scan performance, but ALT-index has a competitive performance with the rest of the learned indexes.

F. Init Size and Skewed Workload

To explore the influence of init size factor, we present the read throughput performance of 32 threads with different initialization data ratios in Fig. 8(d). As the initial data ratio

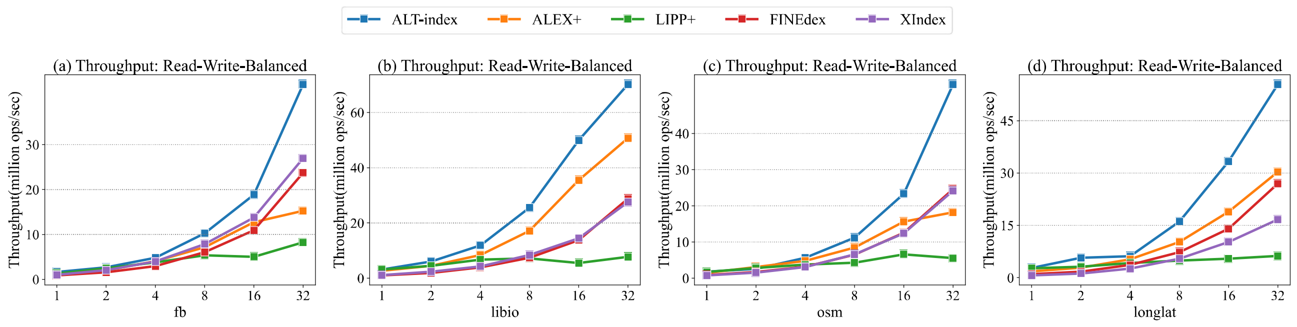

Fig. 9. The indexes scalability under read-write-balanced workloads. Evaluated on four datasets.

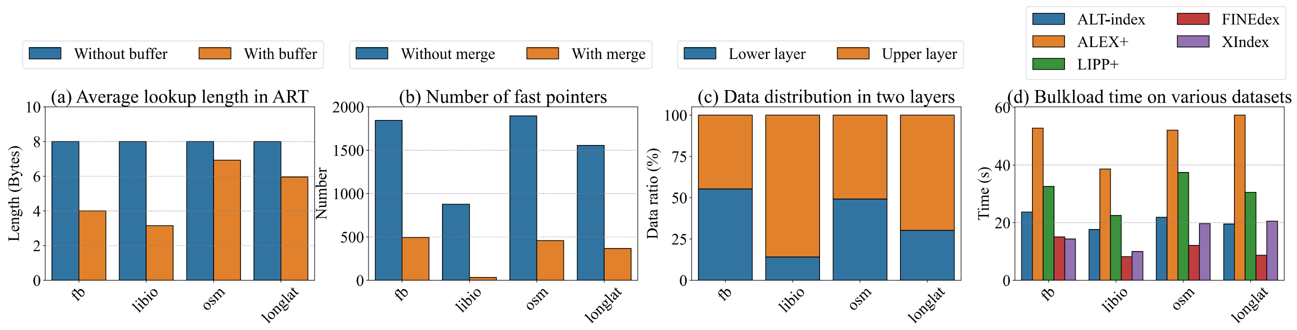

Fig. 10. The average lookup length in ART related to fast pointer buffer is shown in (a). The fast pointer buffer with and without the merge scheme are compared in (b). Data distribution in different ALT-index layers is shown in (c). Bulkload time is shown in (d).

increases, the indexes' performance gradually declines. For other competitors, a larger initialization data quantity leads to a higher number of models, which results in extra overhead in model locating. ALT-index effectively ensures the number of models within an expected range for different init sizes through the GPL algorithm. Overall, ALT-index shows better performance than competitors as the init size increases.

In our experiments, we adjust the θ of the zipfian distribution for lookups. As shown in Fig. 8(e), the improved throughput is attributed to a higher cache hit ratio when the skewness increases. Still, ALT-index keeps leading when the skewness increases, which indicates that our hybrid design is friendly for hotspot queries.

G. *Scalability*

The scalability test is under read-write-balanced workloads, as shown in Fig. 9. ALT-index outperforms other indexes when the thread number increases. LIPP+ shows limited scalability as we expected. FINEdex and XIndex perform good scalability, but the prediction error problem restricts its increasing speed when the thread number scales out. Notably, when the thread number increases from 16 to 32, ALEX+ gains less increment. This attributes to high write-amplification and high prediction overheads, which will introduce memory bandwidth exhaustion. Overall, ALT-index shows the best scalability out of other learned indexes.

H. *Inside Analysis of ALT-index*

1) Factor of fast pointer and merge scheme: The fast pointer buffer with merge scheme can merge the redundant fast pointers to improve the read performance with minimal memory consumption and avoid inconsistencies in ART-OPT layer caused by node structure adjustments. To explore their effectiveness, we produce a deep analysis using four datasets under the read-only workload. First, we measure the average prefix matches of the conflict data in ART-OPT layer. The result is shown in Fig. 10(a), we find that the fast pointer buffer reduces the average lookup length, which accelerates the lookup performance in ART. Then, we count the number of fast pointers in the fast pointer buffer with and without the merge scheme. As shown in Fig. 10(b), the merge scheme significantly cut down the number of fast pointers, further minimizing the memory consumption of ALT-index.

2) Data distribution and bulkload time: We measure the number of data in the learned index and ART-OPT layers for each of the four datasets. As shown in Fig. 10(c), the test results indicate that more than 50% of the real-world datasets can be absorbed by the learned index layer. In addition, the remaining data are compressed by ART-OPT, thereby improving space efficiency. Specifically, over 80% of the whole dataset is accommodated by the learned index layer on the libio dataset. Moreover, as shown in Fig. 10(d), ALT-index only spends a short time to bulkload compared with ALEX+ and LIPP+. Overall, ALT-index performs well as expected.

V. Discussion

Replacement or Accelerator. None of the current learned index structures can replace the traditional index completely in modern database systems. ALT-index takes a different approach using the learned index similar to the high-speed cache layer in computer architectures. It places a portion of the data in the high-speed learned index layer to accelerate accesses, whereas the remaining data are stored in the underlying ART-OPT. By leveraging this hybrid design, ALT-index maximizes the efficiency of learned indexes in terms of read performance while minimizing space overhead [1]. It significantly improves performance compared to the existing learned indexes and traditional tree-based indexes.

Limitations & Strengths. ALT-index destructs the data into two layers, which harms the range query performance of the index. Moreover, ALT-index uses ART-OPT as the second layer, but when it comes to disk-intensive scenarios, frequent random read/write operations caused by the node pointers will destroy the performance. In our future work, we will delve deeper into these directions and conduct more thorough investigations to improve our design. Besides, delta buffers cut down the overhead of insertions, but users must pre-allocate extra space for delta buffers. ALT-index's design can replace these delta buffers into a single ART to save space, and the fast pointer buffer can efficiently help each GPL model get the intermediate node in ART-OPT. In this way, ALT-index draws on the insertion advantages of delta buffers through the fast pointer buffer but cuts down the space cost by merging these buffers into a single ART.

VI. Related Work

Hybrid Index. The concept of hybrid index data structures has received significant attention over time [25]. Masstree employs a hybrid indexing structure that combines B+ trees and trie trees. Specifically, Masstree [10] utilizes B+ trees as the internal nodes of the traditional trie tree, which expands the span of the trie tree while reducing its height, thereby achieving improved read and write performance. Christoph et al. [26] proposed a strategy to identify hot and cold nodes in the index structure. By analyzing the access patterns of the nodes, the encoding scheme of nodes is dynamically adjusted to strike a balance between performance and space efficiency. The wormhole [27], on the other hand, explores the replacement of the internal nodes of the B+ tree with trie trees. It accelerates the lookup operation by employing a hash table as the internal nodes of the B+ tree, thereby ensuring performance gains while conserving space. Furthermore, hybrid indexing methodologies, as referenced in the work of Shahvarani et al. [28] and Su [29], have shown a marked improvement in performance optimization for specific heterogeneous platforms, which is also a potential area for the learned index. ALT-index is the first work that attempts to introduce the learned index and ART to the hybrid index area. It addresses insertion and concurrency issues efficiently with careful construction and optimization.

Learned structures in database systems. Numerous design approaches have emerged for learned indexes, with a significant focus on addressing the challenges related to write operations. ALEX [5], for instance, tackles data insertion by reserving sparse slots within the data array. LIPP [6] effectively mitigates the overhead of secondary queries by segregating conflicting predicted data into separate child nodes. FINEdex and XIndex partially address concurrency concerns, their performance advantages over traditional indexing methods are limited. PGM-index [7] achieves a high space utilization rate by utilizing a multi-layer learned index model. Through a progressive narrowing down process based on layered predictions of a given key, the query range interval is gradually reduced until a smaller model can efficiently handle the query operation. NFL [30] enhances the fitting performance of linear models by transforming the original CDF curve into a linear approximation using a simple neural network model. Notably, the training task for the NFL model is offloaded to specialized GPUs, thereby reducing the computational burden on CPUs and memory resources. Furthermore, other research efforts in the field of learned indexes are concentrated on multidimensional data [31]–[36], string data [37], and applications in diverse areas [38]–[40]. ALT-index presents a novel solution that addresses shortcomings in insertions, improves concurrency, and cuts down the secondary search costs caused by prediction errors.

VII. Conclusion

In this paper, we introduce ALT-index, a novel hybrid index scheme that combines a read-efficient learned index with an insert-efficient ART. ALT-index stores linear data in the learned index layer and puts the data causing prediction errors in ART-OPT layer. Then in the upper learned index layer, we develop a new GPL segmentation algorithm to support flattened data structure for concurrency. In ART-OPT layer, we design a fast pointer buffer with a merge scheme to improve the access speed in ART. Moreover, a dynamic training process can redistribute the data if necessary. Eventually, our experimental results demonstrate that ALT-index exhibits excellent performance and concurrency, and outperforms existing updatable learned indexes by a factor of 1.9x–2.3x in real-world datasets under read-write-balance workloads, respectively.

Acknowledgments

This work is funded by the National Key Research and Development Program (No. 2022YFB4501300), the National Natural Science Foundation of China (No. U22A2027 and 61821003), the Project of Shenzhen Technology Scheme (JCYJ20210324141601005), the Open Project Program of WNLO (No. 2023WNLOKF006), and the Natural Science Foundation of Hubei Province (No. 2024AFB263 and Q20231409). We also thank the anonymous reviewers for providing insightful comments to improve the paper.

References

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, 2018, pp. 489–504.

[2] P. Li, Y. Hua, J. Jia, and P. Zuo, "FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems," *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 321–334, 2021.

[3] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. 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*. Association for Computing Machinery, 2020, pp. 308–320.

[4] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "FITing-Tree: A Data-aware Index Structure," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2019, pp. 1189–1206.

[5] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[6] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *Proceedings of the VLDB Endowment*, vol. 14, no. 8, pp. 1276–1288, 2021.

[7] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020.

[8] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 38–49.

[9] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A Height Optimized Trie Index for Main-Memory Database Systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 521–534.

[10] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 183–196.

[11] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang *et al.*, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[12] M. Maltry and J. Dittrich, "A critical analysis of recursive model indexes," *Proceedings of the VLDB Endowment*, vol. 15, no. 5, pp. 1079–1091, 2022.

[13] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *Proceedings of the VLDB Endowment*, vol. 14, pp. 1–13, 2020.

[14] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 3004–3017, 2022.

[15] G. Graefe, *Encyclopedia of Database Systems*, 2009.

[16] I. Sabek, K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, and T. Kraska, "Can learned models replace hash functions?" *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 532–545, 2022.

[17] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: a single-pass learned index," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2020, pp. 1–5.

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[19] C. E. Lopez and C. Gallemore, "An augmented multilingual twitter dataset for studying the covid-19 infodemic," *Social Network Analysis and Mining*, vol. 11, no. 1, p. 102, 2021.

[20] "OpenStreetMap," 2017. [Online]. Available: http://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap

[21] J. Ge, B. Shi, Y. Chai, Y. Luo, Y. Guo, Y. He, and Y. Chai, "Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 315–327.

[22] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The art of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–8.

[23] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Sosd: A benchmark for learned indexes," *arXiv preprint arXiv:1911.13014*, 2019.

[24] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "APEX: a high-performance learned index on persistent memory," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, pp. 597–610, 2021.

[25] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1567–1581.

[26] C. Anneser, A. Kipf, H. Zhang, T. Neumann, and A. Kemper, "Adaptive Hybrid Indexes," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1626–1639.

[27] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[28] A. Shahvarani and H.-A. Jacobsen, "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*, 2016, pp. 1523–1538.

[29] X. Su, J. Qi, and E. Tanin, "A fast hybrid spatial index with external memory support," in *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, 2023, pp. 67–73.

[30] S. Wu, Y. Cui, J. Yu, X. Sun, T.-W. Kuo, and C. J. Xue, "NFL: robust learned index via distribution transformation," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2188–2200, 2022.

[31] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 2119–2133.

[32] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: a learned multi-dimensional index for correlated data and skewed workloads," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, p. 74–86, 2020.

[33] A. Davitkova, E. Milchevski, and S. Michel, "The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries," 2020.

[34] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, 2019, pp. 569–574.

[35] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 985–1000.

[36] A. Al-Mamun, H. Wu, and W. G. Aref, "A tutorial on learned multi-dimensional indexes," in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, 2020, pp. 1–4.

[37] Y. Wang, C. Tang, Z. Wang, and H. Chen, "Sindex: a scalable learned index for string keys," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 17–24.

[38] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," 2021.

[39] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "From wisckey to bourbon: A learned index for log-structured merge trees," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 155–171.

[40] M. Mitzenmacher, "A model for learned bloom filters and related structures," *arXiv preprint arXiv:1802.00884*, 2018.