[Experiments & Analysis] Evaluating Learned Indexes in LSM-tree Systems: Benchmarks, Insights and Design Choices

Junfeng Liu
Nanyang Technological University
junfeng001@e.ntu.edu.sg

Jiarui Ye
Nanyang Technological University
jiarui005@e.ntu.edu.sg

Mengshi Chen
Nanyang Technological University
mengshi002@e.ntu.edu.sg

Meng Li
Nanjing University
meng@nju.edu.cn

Siqiang Luo*
Nanyang Technological University
siqiang.luo@ntu.edu.sg

Abstract

LSM-tree-based data stores are widely used in industry due to their exceptional performance. However, as data volumes grow, efficiently querying large-scale databases becomes increasingly challenging. To address this, recent studies attempted to integrate learned indexes into LSM-trees to enhance lookup performance, which has demonstrated promising improvements. Despite this, only a limited range of learned index types has been considered, and the strengths and weaknesses of different learned indexes remain unclear, making them difficult for practical use. To fill this gap, we provide a comprehensive and systematic benchmark to pursue an in-depth understanding of learned indexes in LSM-tree systems. In this work, we summarize the workflow of 8 existing learned indexes and analyze the associated theoretical cost. We also identify several key factors that significantly influence the performance of learned indexes and conclude them with a novel configuration space, including various index types, boundary positions, and granularity. Moreover, we implement different learned index designs on a unified platform to evaluate across various configurations. Surprisingly, our experiments reveal several unexpected insights, such as the marginal lookup enhancement when allocating a large memory budget to learned indexes and modest retraining overhead of learned indexes. Besides, we also offer practical guidelines to help developers intelligently select and tune learned indexes for custom use cases.

ACM Reference Format:
Junfeng Liu, Jiarui Ye, Mengshi Chen, Meng Li, and Siqiang Luo. 2025. [Experiments & Analysis] Evaluating Learned Indexes in LSM-tree Systems: Benchmarks, Insights and Design Choices. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

*Corresponding Author

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Log-structured Merge Trees (LSM-trees), have already been widely used as the fundamental storage structure underpinning many key-value stores like Google Spanner [6], Apache Cassandra [29], and MongoDB WiredTiger [5]. These key-value stores play pivotal roles in various applications in social media, streaming processing, and file systems.

Index in LSM-tree stores. As illustrated in Figure 1(A), a typical LSM-tree consists of both memory and disk components. The memory components include a write buffer, bloom filters, and fence pointers, while the disk components are sorted key-value arrays organized across multiple levels. To prevent the need for sequential scanning of these arrays on disk, the *fence pointers* in memory help quickly locate the specific data block where the target key is stored. This allows only the relevant data block to be read, thereby significantly reducing the number of I/Os required.

While fence pointers have successfully reduced the number of I/Os, the question still arises: can we push the performance further within the same memory constraints? Recent advances in machine learning have led to the development of several learned indexes, which have demonstrated superior memory efficiency through extensive evaluations [41]. These learned indexes achieve impressive results using much less memory space, presenting an exciting opportunity to further optimize indexing in LSM-trees. However, two main questions remain unanswered when applying them to LSM-tree systems:

Are all learned indexes suitable for LSM-tree systems? Recent years have seen a surge of interest in integrating learned indexes into LSM-tree systems. Dai *et al.* [7] are among the first to explore this integration by applying a Piece-wise Linear Regression (PLR) model to LevelDB [18], achieving significant improvements over traditional fence pointers. Similarly, Lu *et al.* [37] demonstrate promising results using the Recursive Model Index (RMI) within an LSM-tree framework. While these studies show the potential of learned indexes in LSM-tree systems, they also prompt a natural question: are all recently proposed learned indexes well-suited for use in LSM-tree architectures? Evaluating this is nontrivial. Given

the large and growing number of proposed learned index designs, implementing all of them within a single system for empirical comparison is infeasible. A more practical approach is to first identify the distinctive storage characteristics of LSM-tree systems, such as the hierarchical level and immutable files, then classify learned indexes based on their compatibility with these characteristics. This allows us to systematically determine which types of learned indexes are most appropriate for LSM-tree environments.

Is there a universal guideline and configuration space for tuning learned indexes in LSM-tree systems? While prior benchmarks [27, 41, 54] have outlined practical principles for deploying learned indexes in both in-memory and on-disk systems, tuning them within LSM-tree systems remains significantly more complex. This complexity arises from the presence of multiple interdependent components, such as Bloom filters, caches, and write buffers, which all compete for the same limited memory budget and jointly influence both update and lookup performance. How to effectively allocate memory between learned indexes and these LSM-specific components is still an open question. Furthermore, the configuration space of learned indexes is highly diverse, with different models requiring different parameters and optimization strategies. While developing a universal tuning guideline would be highly valuable, it is also inherently challenging due to the variability in index structures and the interplay with system-level tradeoffs.

Motivated by these questions, in this paper, we conduct a comprehensive study on applying learned indexes in LSM-tree systems. Our contributions are concluded as the following:

We revisit several prevalent learned indexes and identify the key factors that impact the compatibility between learned indexes and LSM-tree. We start with revisiting the existing learned indexes, summarizing their structures and underlying algorithms. Next, we classify these indexes based on their data layout to determine which ones are most compatible with LSM-tree architectures. Particularly, some representative learned indexes, such as ALEX [12] and LIPP [55], are primarily designed for in-memory lookups and involve less efficient pointer jumping, making their data layout incompatible with the continuous and sorted structure of LSM-tree levels. A detailed categorization and compatibility study of learned indexes tailored for LSM-trees would offer valuable insights for designing more suitable models for LSM-tree systems.

We identify three unified key parameters that affect performance across all compatible indexes and propose a comprehensive configuration space for LSM-trees. Our theoretical analysis reveals three core tuning parameters that significantly influence the performance of learned indexes in LSM-tree systems: index type, position boundary, and index granularity. Index type refers to the specific learned index model employed, each characterized by unique segment partitioning strategies and inner index structures, leading to different memory-performance tradeoffs. Position boundary denotes the final search range that the LSM-tree retrieves from disk. This parameter directly affects I/O cost and is a crucial tuning knob for many learned index designs. Index granularity determines the number of entries over which a learned index is constructed, influencing both lookup accuracy and index overhead. These three parameters define a unified and inclusive configuration space that enables systematic experimentation and performance evaluation. This framework provides a solid foundation for understanding and optimizing the integration of learned indexes in LSM-tree systems.

We develop a unified testbed LSM-tree system with a learned-index-compatible interface that enables seamless integration and fair comparison of six representative indexes. In Section 4, we detail the implementation of a universal interface that allows diverse learned indexes to be easily integrated into LevelDB. We also illustrate how the three key parameters—index type, position boundary, and index granularity—affect LSM-tree performance. Using this platform, we conduct a comprehensive evaluation of six representative learned indexes compatible with LSM-trees, testing them under various configurations and datasets to assess their impact on core system operations such as point lookups, range lookups, and compaction. Our findings yield several important insights. First, all evaluated learned indexes exhibit a superior memory-latency tradeoff compared to traditional fence pointers, offering significantly lower lookup latency for the same memory usage. This underscores the potential of learned indexes in LSM-tree systems. Among the tested models, RMI, and PGM consistently demonstrate strong performance and dominate other approaches in most scenarios. We also observe that the overhead introduced by model training and write-time indexing during compaction is modest relative to the overall compaction cost. Additionally, increasing index granularity reduces the number of learned indexes required, thereby lowering overall memory consumption. Beyond point queries, we further evaluate the effects of learned indexes on range lookups and mixed workloads involving both reads and updates. Based on these experiments, we identify three key design principles for integrating learned indexes into LSM-trees:

- Position boundary is the most critical factor for lookup performance. While the choice of index type mainly affects the memory-latency tradeoff, prioritizing models with smaller error bounds under a fixed memory budget provides greater benefits than optimizing internal index structures.
- Increasing SSTable size improves lookup performance under a fixed memory budget by reducing index memory overhead and enabling the use of smaller position boundaries.
- Memory allocation exhibits diminishing returns: the performance improvement from increasing the memory budget plateaus once the segment size becomes smaller than or equal to the I/O block size.
- The construction and training of learned indexes introduce minimal overhead compared to traditional fence pointers during compaction.

2 Background

This section discusses the background knowledge about the LSM-tree systems and the indexing schemes over it.

2.1 Log-Structured Merge Trees

An LSM-tree efficiently manages data across multiple disk components, organizing data into sorted arrays at different levels. It also maintains an in-memory component, where recent updates are stored in a write buffer. To enhance lookup performance, each

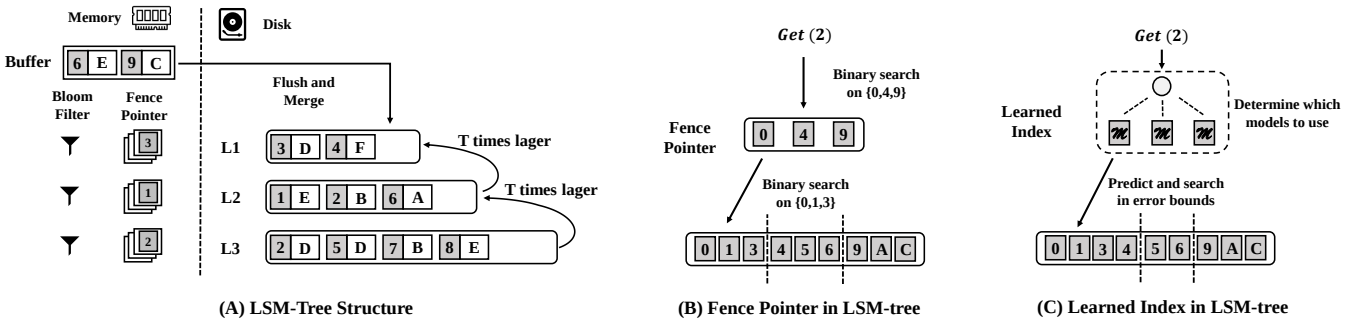

Figure 1: (A) presents the general structure of an LSM-tree; (B) and (C) illustrate how original fence pointer and learned indexes work in an LSM-tree, respectively

sorted array is associated with Bloom filters and indexes. The capacity of each level increases exponentially by a size ratio of T, meaning the total number of levels required to store N entries is approximately $L = \lceil \log_T \frac{N \cdot e}{F} \rceil$, where F is the size of the write buffer, and e is the size of individual entries. Each level consists of key-value pairs stored in a sorted array, referred to as a *sorted run* in some works. LSM-trees primarily support three operations:

Updates. Updates in an LSM-tree are initially written to the in-memory write buffer. Once the buffer is full, the key-value entries are flushed to disk and merged into the sorted array at level-1, as shown in Figure 1 (A). If a level exceeds its capacity, a compaction operation is triggered, merging entries into the next level to maintain efficient storage and query performance. Alternatively, to mitigate resource consumption spikes, some databases [14, 18] perform partial compactions by merging only a subset of entries into the next level, rather than compacting the entire level at once. In these systems, a sorted run is divided into sorted files, known as *SSTables*, and only a subset of these SSTables is selected for merging during each compaction.

Point Lookups. A point lookup searches for the value of a specific key by checking levels sequentially until the key is found. To expedite this process, Bloom filters and indexes are employed. When a Bloom filter indicates a potential match, the LSM-tree locates the approximate range within that level and performs a binary search to verify the key's presence.

Range Lookups. Range lookups collect entries from each LSM-tree level and use a sort-merge process to remove duplicates, returning only the most recent values.

2.2 Indexes in LSM-tree

Traditional index structures in LSM-trees often rely on *fence pointers*, as shown in Figure 1(B). These pointers store the smallest (or largest) key of a fixed range of key-value pairs, allowing the LSM-tree to quickly narrow down the search to a small data range when locating a specific key. During compaction, the smallest or largest key of each newly created data range is stored in memory. By querying these index structures, the LSM-tree can skip unnecessary searches through large amounts of data on disk, significantly reducing I/O operations and improving lookup efficiency.

Learned Indexes in LSM-tree. Although fence pointers are widely used in most LSM-tree systems [14, 18, 29], there remains an opportunity to improve memory efficiency by replacing them with more advanced learned index structures. This potential arises for two key reasons: (1) The sorted arrays on disk are immutable, meaning they are only created and deleted during compactions, making them well-suited for even non-updatable learned indexes, and (2) since the entries are already stored in a naturally sorted order on disk, learned indexes can efficiently map the data, potentially reducing the overhead of sorting. As shown in Figure 1 (C), by training the learned model during compactions, we can easily replace the fence pointers with learned indexes.

Abu-Libdeh *et al.* [2] were the first to evaluate the feasibility of using a linear regression model in LSM-tree systems, finding a positive impact when replacing traditional fence pointers with learned index structures. However, their study did not systematically explore the performance variations across different types of learned indexes or how various configurations might affect results. Building on this idea, Dai *et al.* integrated learned indexes into a key-value separated LSM-tree system, Wisckey [38], developing Bourbon [7], an LSM-tree system equipped with a piecewise linear learned index. While Bourbon demonstrated notable performance improvements, it still did not thoroughly investigate all possible design choices for learned indexes, such as experimenting with different index types.

3 Learned Indexes Revisited

We revisit eight learned indexes in detail and assess their compatibility with LSM-tree systems. Broadly, we classify these indexes into two categories based on their data layout: **data-clustered indexes**, such as FITing-Tree, PGM, and RMI, and **data-unclustered indexes**, including LIPP and ALEX. As illustrated in Figure 2, data-clustered indexes store key-value pairs in physically continuous blocks, whereas, as shown in Figure 3, data-unclustered indexes do not. Instead, data-unclustered indexes require additional steps, such as traversing via pointers, to retrieve continuous key-value pairs. In the following sections, we first review the structures of the most representative data-clustered and data-unclustered indexes. We then analyze which indexes are most compatible with LSM-tree systems, followed by a theoretical analysis of the cost associated with each index.

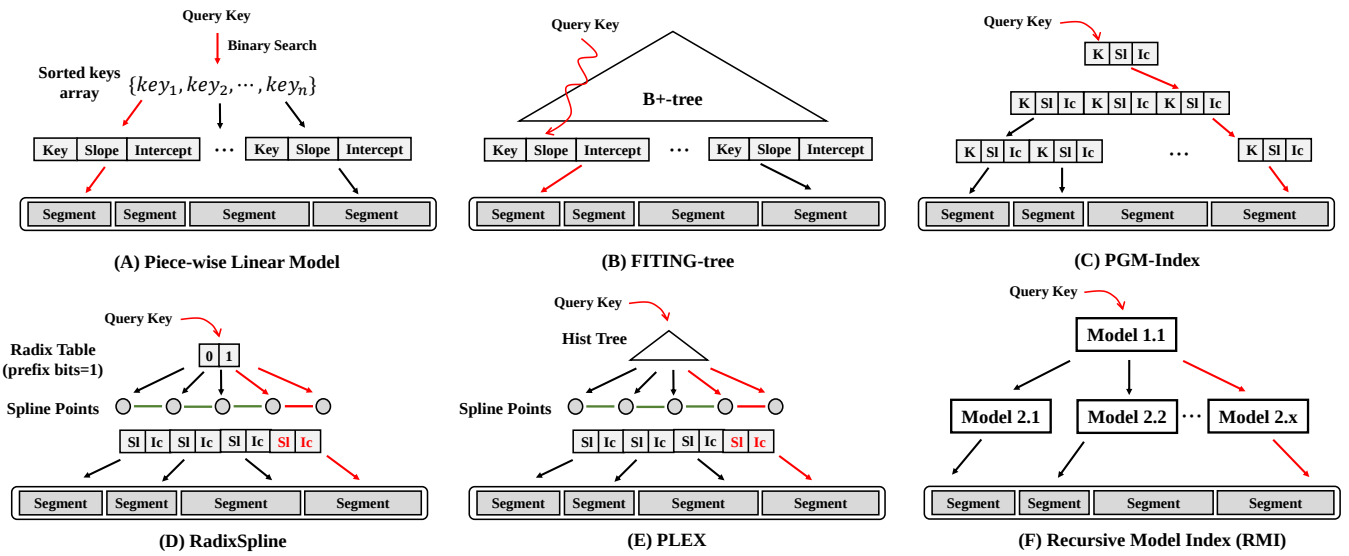

Figure 2: (A) to (F) present the data structures of data-clustered learned indexes and the general lookup procedure. Ic represents the intercept of the linear model, Sl is the slope, and K is short for the key.

3.1 Data Clustered Indexes

In this subsection, we review six well-known data-clustered learned indexes and their respective lookup procedures.

Piece-wise Linear Regression (PLR) [7], shown in Figure 2 (A), uses a greedy algorithm to divide a sorted array into segments based on a specified error bound, which represents the maximum allowable difference between the estimated and actual key positions. For each segment, PLR builds a linear model to predict the approximate position of a key. During a lookup, PLR first locates the segment containing the key by performing a binary search over the segments. It then uses the corresponding linear model to estimate the key's position, denoted as *appx_pos*. Since the linear model ensures that the true key position lies within a range of [*appx_pos - error*, *appx_pos + error*]–where the *error* reflects the prediction tolerance–a binary search is performed within this range to find the exact key location.

FITing-Tree [16], shown in Figure 2 (B), uses a greedy algorithm to divide a sorted array into segments similarly based on a specified error bound. A linear model is built for each segment to predict the position of keys. To efficiently search through these segments, FITing-Tree uses a B+-tree to index the segments. When looking up a key, FITING-tree first traverses the B+-tree to locate the segment containing the key. It then uses the corresponding linear model to predict the approximate position. Since the linear model ensures that the true key lies within the range [*appx_pos - error*, *appx_pos + error*], a binary search is performed within this range to find the exact key location.

Piecewise Geometric Model index (PGM) [15], shown in Figure 2 (C), takes a different approach by using a streaming algorithm, rather than a greedy one, to divide the array into segments and build linear models with a given error bound. PGM further applies this streaming algorithm recursively to construct parent nodes and build linear models for these higher-level nodes. To look up a key, PGM predicts an approximate position *appx_pos* using its model, then recursively performs a binary search within [*appx_pos - error*, *appx_pos + error*] until the exact key is found in the leaf node.

RadixSpline (RS) [26], shown in Figure 2 (D), selects a subset of keys from the sorted array as spline points and uses linear interpolation models to estimate the positions of keys between any two spline points. RadixSpline ensures the accuracy of its spline layer by imposing error bounds on the approximations. If the error exceeds a predefined threshold, additional spline points are added to improve the approximation. To index these spline points, RadixSpline constructs a radix table. When looking up a key, RadixSpline first uses the radix table to locate the correct spline segment, then applies the linear interpolation model to predict the key's approximate position. A binary search is then performed within the range [*appx_pos - error*, *appx_pos + error*] to locate the exact key.

Practical Learned Index (PLEX) [51], shown in Figure 2 (E), is an improved version of RadixSpline that employs a hierarchical Hist Tree (or Radix Tree) to index spline points and reduce search space. Like RadixSpline, PLEX uses spline points and linear interpolation models to estimate key positions but improves lookup efficiency by leveraging this hierarchical structure. A key feature of PLEX is its self-tuning capability, which dynamically adjusts the number of spline points based on data distribution and workload. This optimization ensures a balance between prediction accuracy and memory usage, improving overall performance. During lookup, PLEX first uses the Hist Tree to locate the segment containing the key, then applies the corresponding linear model to predict the approximate position. A binary search is then performed within [*appx_pos - error*, *appx_pos + error*] to find the exact key. With its hierarchical structure and self-tuning, PLEX efficiently adapts to large datasets and varying workloads, offering better scalability than RadixSpline.

Recursive Model Index (RMI) [28], shown in Figure 2 (F), is a learned index that recursively applies machine learning models to

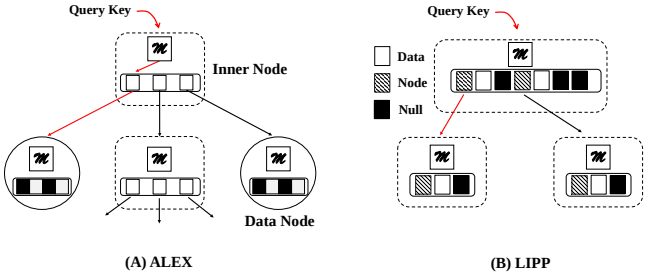

Figure 3: (A) and (B) present the structure of ALEX and LIPP. The data (key-value pairs) are not stored continuously to accommodate incoming new entries.

approximate the position of keys in a sorted array. It organizes models in a hierarchical structure, where upper-level models predict the position of keys for the next layer, progressively refining the prediction until the lowest layer estimates the key's position. RMI is built in a top-down manner, where the top-level model is trained first to give a coarse estimate of key positions. Based on this, the dataset is divided, and lower-level models are trained on smaller subsets, improving accuracy as you move down the hierarchy. This approach allows RMI to tailor the complexity of each model to the portion of the data it handles, optimizing both performance and memory usage. During lookup, RMI first uses the top-level model to make a rough prediction, then refines this through subsequent layers. The final model predicts the approximate key position, followed by a binary search within a small range to find the exact key. RMI's error is not predefined by the user but rather recorded during the training process, adapting to the data's characteristics.

3.2 Data Unclustered Indexes

ALEX [12], shown in Figure 3 (A), uses two types of nodes: inner nodes and data nodes, both combining arrays with linear models to predict key positions. Inner nodes contain an array of pointers to other nodes, while data nodes use gapped arrays that store key-value pairs, interleaving empty slots to support efficient insertions. During a lookup, ALEX traverses from the root through inner nodes, using the model to predict the appropriate child node. Once at a data node, the model predicts the position of the key, followed by an exponential search to locate the exact key. ALEX dynamically adjusts its structure by splitting or merging nodes as data evolves, ensuring efficient handling of both lookups and updates while maintaining optimal performance for dynamic workloads.

LIPP [55], shown in Figure 3 (B), uses a linear model in each node to precisely predict key positions. Each node contains a data array and a bitmap to distinguish between three slot types: DATA, NULL, and NODE. The linear model predicts which slot should be accessed during a lookup. LIPP employs the Fastest Minimum Conflict Degree (FMCD) algorithm to minimize conflicts (multiple keys mapped to the same slot). When a key is inserted into a NULL slot, it is marked as DATA and stores the key-value pair. If inserted into an occupied DATA slot, the slot becomes a NODE, pointing to a new child node created with the conflicting keys. The child node is built using the same algorithm. During lookup, LIPP uses the root node's linear model to predict the key's position. If the predicted slot is NODE, it follows the pointer to the child node and repeats the process. If the slot is DATA, it checks for a match and returns the key-value pair if found.

DILI [34] uses a two-phase approach to build the index: a bottom-up tree-building process using linear regression models based on global and local key distributions, followed by a top-down refinement where the fanout of internal nodes is customized according to local key distributions. This design strikes a balance between the number of leaf nodes and the tree height, both crucial factors in minimizing key search time. Additionally, DILI includes flexible algorithms for efficient key insertion and deletion, allowing the index to dynamically adjust its structure when necessary.

NFL [56] introduces a new approach to addressing the challenges of learned indexes by transforming complex key distributions before constructing the index. NFL uses a two-stage framework: first, it applies Numerical Normalizing Flow (Numerical NF) to transform the key distribution into a near-uniform one. Then, it builds a learned index using a specialized After-Flow Learned Index (AFLI), optimized for the normalized data.

3.3 LSM-Compatible Indexes

Though data-unclustered indexes offer excellent read performance and memory efficiency, we argue that data-clustered indexes are a more suitable choice for LSM-trees. This is because LSM-trees rely on the sorted and contiguous storage of data across levels (i.e., SSTables and sorted runs) to optimize read and write performance. Data-clustered indexes naturally maintain this physical continuity, allowing them to replace existing fence pointers with minimal engineering effort. By contrast, data-unclustered indexes like ALEX and LIPP disrupt the current data layout (i.e., SSTable), making integration with LSM-trees more complex. Moreover, range lookups, a critical operation in LSM-trees, benefit from the sequential access provided by data-clustered indexes. These indexes ensure fast access to key-value pairs stored contiguously, while data-unclustered indexes, which scatter data, require additional memory and disk jumps, significantly increasing the overhead of such queries.

While integrating data-unclustered indexes into LSM-trees is a promising and intriguing topic, this integration is not feasible without significantly altering the widely recognized LSM-tree storage architecture for the following reasons:

(1) Need for non-compact and uncontinous storage layout replacement: Successful integration would necessitate replacing the current compact data layout (*i.e.*, SSTables) with a discontinous data-unclustered structures, which represents a considerable undertaking.
(2) Re-Implementation of Basic Functions: Basic functions, such as range lookup and compaction iterators, would need to be re-implemented when using data-unclustered indexes.

Given these considerations, this paper primarily focuses on the performance of data-clustered indexes within existing LSM-tree systems, while leaving the exploration of data-unclustered indexes for future research.

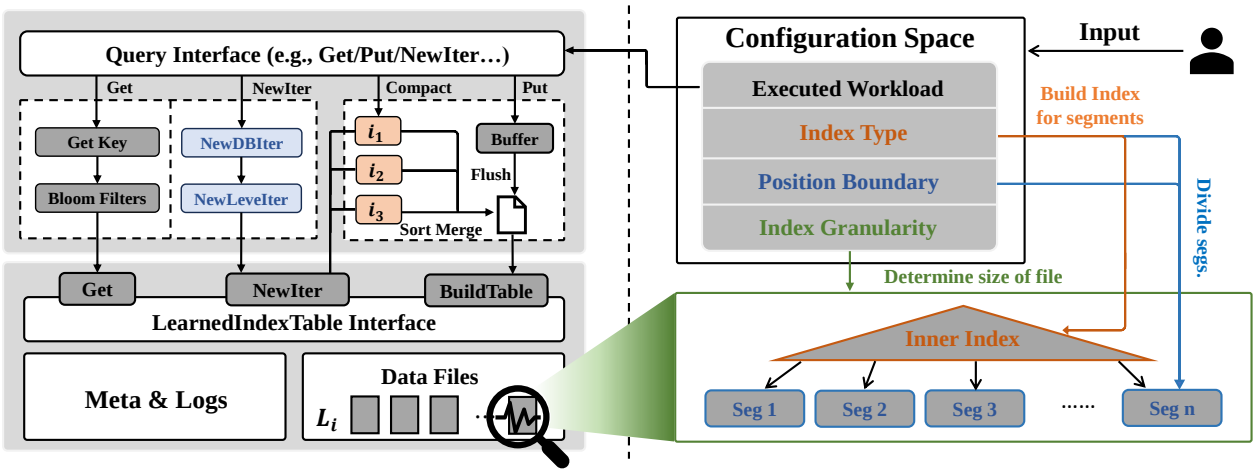

Figure 4: The figure demonstrates the architecture of our testbed system. The left-hand side is the detailed conduction of different operations to test while the right-hand side is how the three configuration impact the system.

4 Exploring the Configuration Space of Indexing in LSM-trees

In this section, we are going to analyze the cost of LSM-compatible indexes to identify the tuning options that affect the read performance and memory consumption when applied to LSM-tree systems.

4.1 Configuration Space

To begin with, data-clustered indexes locate a key by first identifying the segment likely to contain it, then reading that segment from disk, and finally searching within it under a bounded error range, as illustrated in Figure 4. The cost of the first step depends on the type of index used, since different index structures organize segment metadata differently. The second cost, related to I/O, is primarily determined by the error bound. Data-clustered indexes ensure that segment lengths do not exceed 2× the error bound (ϵ), and thus, the I/O cost would not exceed $O(\frac{2\epsilon}{B})$, where B is the size of an I/O block. The final cost stems from the in-segment search, typically performed using binary search, and is also governed by the error bound. Based on this process, the two most critical factors to tune are:

Index Type. The choice of index structure has a direct impact on lookup speed and memory overhead. For example, FITing-tree uses a B+-tree to index segments, which offers faster lookups but incurs higher memory consumption than simpler alternatives like the sorted arrays used in PLR. Thus, selecting the appropriate index type involves balancing performance and memory usage.

Position Boundary. Once the approximate segment is identified, the LSM-tree must read it from disk. We define the "position boundary" as the range length of this segment, which is usually 2× the error bound in the learned indexes. A smaller position boundary reduces I/O cost, which is often the dominant overhead in LSM-trees, but increases the number of segments, which raises memory usage. Tuning this boundary is therefore crucial for achieving an optimal tradeoff between I/O efficiency and memory consumption.

Beyond these two main factors, **Index Granularity** also plays an important role in determining the performance of learned indexes within LSM-tree systems. Modern LSM-tree implementations (e.g., LevelDB, RocksDB, PebblesDB [14, 18, 46]) often apply partial compaction strategies, where sorted runs are divided into multiple files (SSTables), and only some are compacted into the next level. In such cases, learned indexes are typically built at the SSTable level, meaning their position boundary is bounded by the SSTable size. Dai *et al.* [7] suggest that coarser-grained indexing—such as level-grained models like LevelModel—can yield performance improvements of around 10% under read-heavy workloads. To examine this claim, we evaluate the performance of learned indexes built at varying SSTable sizes, as well as those constructed across entire levels instead of individual files.

Configuration Tradeoff. Selecting an index type entails a tradeoff between memory consumption and lookup latency. To fairly evaluate index types, it is important to compare them under consistent configurations. Position boundary is likely the most influential parameter for read performance due to its impact on I/O cost. While reducing position boundaries can improve I/O efficiency, this often leads to increased memory usage by creating more, smaller segments. The memory overhead introduced by this tradeoff remains an open question and warrants empirical evaluation. Moreover, although increasing SSTable size (i.e., index granularity) reduces the number of indexes, it may also lead to less efficient compactions by increasing the volume of data processed per compaction round. Therefore, careful tuning of index granularity is also essential for optimizing read and compaction performance.

4.2 Implementation

To systematically evaluate the effectiveness of the three aforementioned configurations, we build a benchmark system based on LevelDB, a well-known and streamlined LSM-tree implementation. To integrate learned indexes without disrupting the system's core functionality, we implement a new class, `LearnedIndexTable`, which inherits from and replaces the original `Table` class. We override

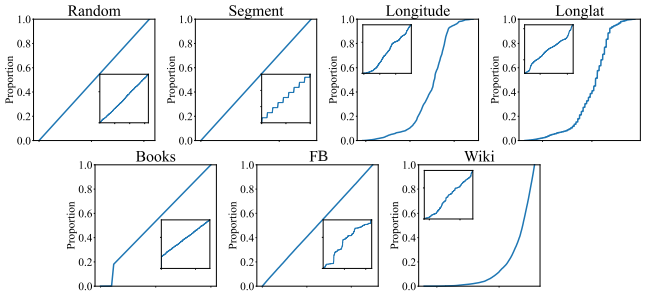

Figure 5: The CDF of different datasets

three key functions: `InternalGet` (Get), `NewIterator` (NewIter), and `TableBuilder` (BuildTable). Below, we describe each implementation in detail and explain how these functions interact with the rest of the system.

Get. The `InternalGet` function handles point lookups by locating a specific key in a data file. In our implementation, key-value pairs are sorted and stored in segments, each indexed by a learned model. To perform a lookup, the learned index first consults its internal model to identify the corresponding segment. The segment is then fetched from disk using the Linux `pread` interface, and a binary search is conducted within the segment to retrieve the target key.

NewIter. The iterator interface is essential in LSM-tree systems, supporting both range queries and compaction. Our learned index iterator begins by seeking to a target key using the same procedure as `InternalGet`, and then proceeds to iterate over subsequent key-value pairs within the segment. Once all entries in the current segment are exhausted, the iterator advances to the next segment.

BuildTable. The `TableBuilder` interface is responsible for constructing learned-index-based tables. During flushes or compactions, the builder receives sorted key-value pairs and constructs a learned index over them, which is similar to how traditional fence pointers are built in baseline LevelDB. Additionally, the original on-disk SSTable format is replaced by the `LearnedIndexTable` format, in which the inner index and data segments are serialized separately, with their offsets recorded in the file header.

Specifically, by using the interface, some commonly used LSM-tree processes and operations are implemented as the following:

Compaction. The process of building learned indexes is similar to that of fence pointers: when an new table is created through a flush or compaction, the sorted key-value pairs are used to train the corresponding learned index. The index is then serialized, written to disk, and marked as "learned". Once the table is complete, the learned index is linked to it.

Point Lookup. When a file is marked as "learned", LevelDB first locates the table containing the key and accesses the learned index to get a prediction for the target key. An I/O operation is then performed to fetch the approximate segment, followed by a binary search to retrieve the exact key.

Range Lookup. A range lookup involves two phases: (1) identifying the starting point of the range at each level, and (2) retrieving the key-value pairs at each level and merging them until the entire range is fetched. The first phase is similar to a point lookup, involving access to the learned index and a binary search. The second phase differs slightly from LevelDB's default behavior. We retrieve one I/O block at a time for the starting segment (typically 4096B) for all indexes until all key-value pairs in the target range are fetched from disk.

We integrate the following baselines by implementing the above interface into our system. The implementations of PGM[1], RadixSpline[2], PLEX[3], and PLR[4] [7] are based on versions released by the respective authors. For RMI and FITing-Tree, as no suitable C++ versions are available, we used the RMI implementation from a benchmark paper[5] [40] and the FITing-Tree[6] implementation from the SOSD benchmark [42].

5 Evaluation

Experiment Design. In this section, we evaluate the performance of learned indexes across various scenarios. First, we assess the memory-latency tradeoff in lookups to determine the effectiveness of replacing fence pointers with learned indexes and to identify the key factors affecting lookup performance. Next, we address a common concern about the additional training time introduced during compaction by running a write-only workload and measuring compaction time. Following this, we evaluate unique features of LSM-tree systems, such as the asymmetric read overhead across different levels and the impact on range lookups. Finally, we test the performance under mixed workloads using the YCSB benchmark to assess their effectiveness in real-world applications.

Datasets and Workload Setup We evaluate all the indexes on seven different datasets generated by the SOSD benchmark [42]: Random, Segment, Longitude, Longlat, Books, FB, and Wiki with their cumulative distributions shown in Figure 5 (a). Due to space constraints, we present results for the Random dataset in this paper. For a complete set of results on all datasets, please refer to our technical report [1]. Each dataset consists of 6.4 million key-value pairs, with 24-byte keys and 1000-byte values. In each experiment, we perform 1,000,000 operations.

Running Environment. We conduct our experiments on a machine running Ubuntu 22.04, equipped with an Intel Core i9-13900K CPU (36 MB L3 cache), 128 GB of memory, and a 2 TB NVMe SSD. All learned indexes are integrated into LevelDB, with the LSM-tree configured to use a leveling compaction policy, a size ratio of 10, and a 10-bit-per-key Bloom filter.

Settings of Learned Indexes. To evaluate the performance of fence pointers under different position boundaries, we adjust the data block size in LevelDB to generate varying numbers of fence pointers **(abbr. FP)**. For PLR, FITing-Tree **(abbr. FT)**, and PLEX, we directly vary the error bounds to control the position boundaries. For RMI, we follow the guidelines in [40] and use their *RMILabs* implementation, as recommended in the paper. This setup uses a two-level model tree. To vary the position boundary, we adjust the size of the second level, which in turn affects the position boundary. For both RadixSpline **(abbr. RS)** and PGM, the error bounds can be adjusted to control the position boundaries. However, since

[1] https://github.com/gvinciguerra/PGM-index
[2] https://github.com/learnedsystems/RadixSpline
[3] https://github.com/stoianmihail/PLEX
[4] We contacted the author to obtain the code
[5] https://github.com/BigDataAnalyticsGroup/analysis-rmi
[6] https://github.com/RKolla99/FITing-Tree

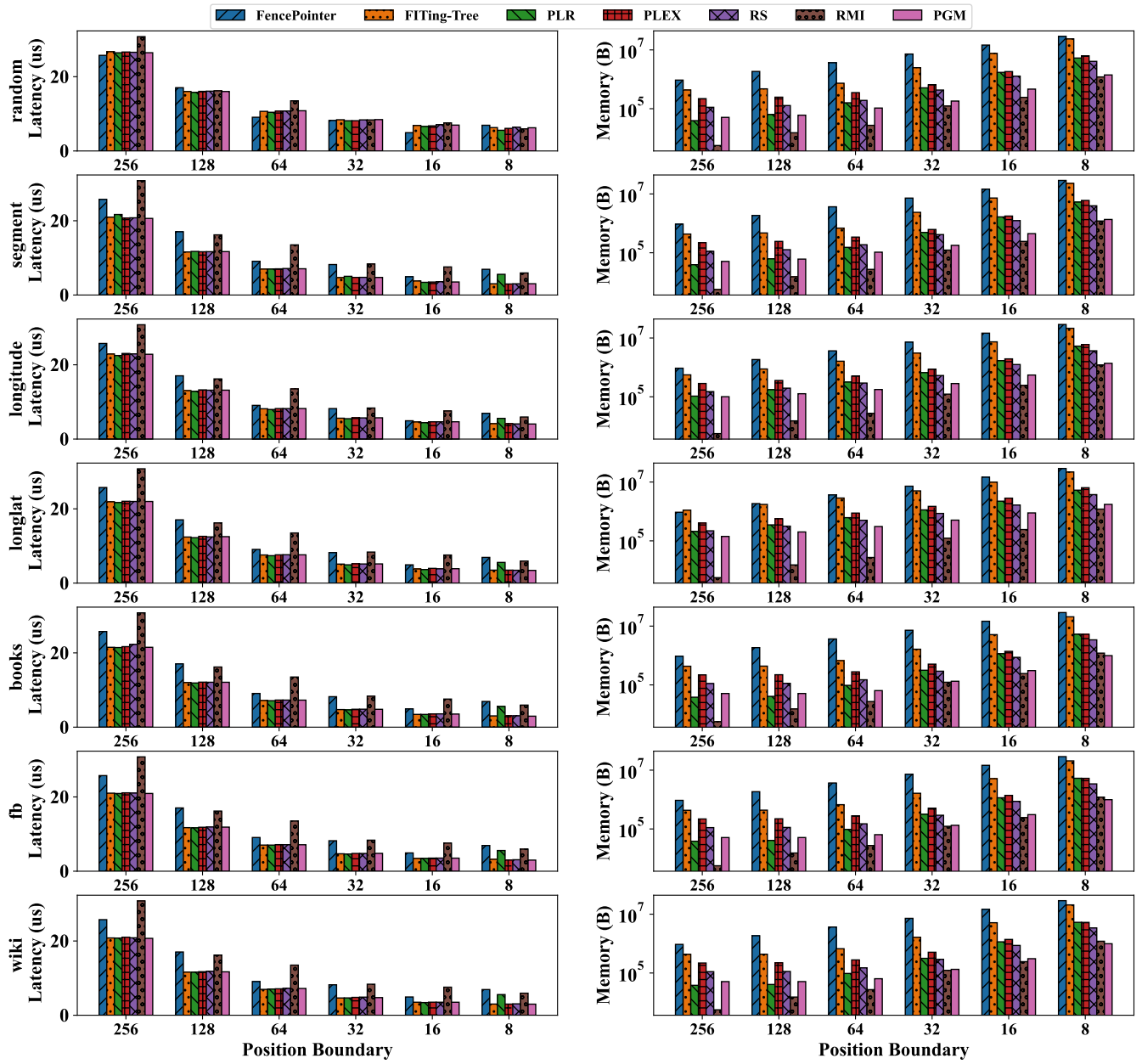

Figure 6: Latency and memory usage of different indexes under different position boundary under different datasets.

both have additional parameters that affect their internal structure, these must also be fine-tuned for optimal performance. In PGM, the *EpsilonRecursive* parameter defines the error bound for internal nodes. We test various values, and find that *EpsilonRecursive* has little impact on PGM's performance in LSM-tree systems. Therefore, we retain the default setting of *EpsilonRecursive* = 4. For RS, the *RadixBits* parameter controls the size of its radix table. After varying this value, we determine that *RadixBits* = 1 offers the best tradeoff in LSM-tree systems, reducing memory usage while maintaining satisfactory performance.

5.1 Analysis on Position Boundary

Observation 1: Smaller position boundary positively reduces the latency for all indexes at the cost of increasing the memory usage. The efficiency of improving performance by adding memory budget varies from indexes, indicating the memory-latency tradeoff is very different.

We vary the position boundary from 256 to 8 for each index and evaluate their performance under a point lookup workload consisting of 1,000,000 queries. We record both the lookup latency and

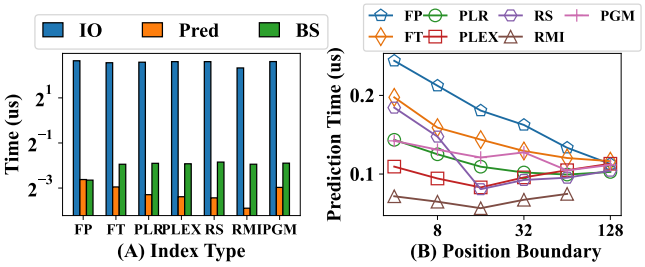

Figure 7: Query time breakdown

memory usage, as shown in Figure 6 (A) and (B). Overall, decreasing the position boundary reduces lookup latency but increases memory consumption.

When the position boundary is held constant, all indexes exhibit nearly identical lookup latencies. This suggests that I/O dominates the cost of point lookups, while the time spent accessing the inner index and searching within a segment is relatively minor. As illustrated in Figure 7(A) and (B), the I/O time required to fetch the segment from disk is approximately 10 times greater than the combined time for model prediction (including inner index access) and binary search within the segment. Although model prediction time increases slightly as the position boundary decreases (Figure 7(B)), this increase is outweighed by the I/O reduction, resulting in an overall latency improvement for all indexes.

Despite similar lookup performance, memory usage varies significantly across index types. Traditional fence pointers exhibit the worst memory-latency tradeoff: as the position boundary decreases, their memory usage grows rapidly compared to learned indexes. This aligns with one of the core design goals of learned indexes—to reduce memory overhead. However, the specific design of a learned index also impacts its memory-latency tradeoff. For instance, while FITing-Tree outperforms fence pointers, its memory consumption still increases quickly. This is due to its use of a B+-tree structure, which improves prediction accuracy at the cost of additional memory. RadixSpline (RS) and PLEX demonstrate comparable tradeoffs; both divide sorted data using spline points and use a radix table (RS) or radix tree (PLEX) to index those points. Although PLEX incorporates a self-tuning mechanism to optimize radix tree performance, this benefit is less pronounced in LSM-tree systems. This is likely because PLEX's optimizations are most effective under skewed key distributions, which are uncommon in LSM-trees—especially at levels with fewer keys. PLR shows a memory-latency tradeoff similar to RS and PLEX, largely due to its lightweight inner index structure for leaf segments, which helps conserve memory. Among all the evaluated learned indexes, PGM and RMI achieve the best memory-latency tradeoffs. PGM employs an optimized segmenting algorithm that reduces the number of segments needed for a given error bound, thus lowering memory usage. RMI stands out for its flexible configuration: by setting a large second-level index, it can achieve extremely low error bounds (as small as 1), enabling high precision with minimal memory. In contrast, other indexes are constrained by their minimal achievable error bounds, which limits their precision under fixed memory budgets.

Observation 2: The improvement in read performance becomes increasingly marginal as the memory budget grows, suggesting that allocating additional memory to the index may not always yield proportional benefits to the system.

While increasing the memory budget for indexes generally enhances performance, we observe diminishing returns as index size continues to grow. As shown in Figure 6(A), lookup latency decreases substantially when the position boundary is reduced from 256 to 128 and then to 64. However, beyond this point, further reductions in the position boundary yield little to no performance improvement for most learned indexes, even though their memory usage grows exponentially, as shown in Figure 6(B). This phenomenon occurs because, initially, additional memory allows the index to model the key space with higher precision, effectively narrowing the search range and reducing I/O cost. But once the precision reaches the granularity of one or two I/O blocks, the dominant cost—disk I/O—can no longer be significantly reduced. At this stage, further memory investment offers minimal performance gain.

As a result, the growth in memory consumption does not translate into a proportional improvement in lookup latency. This trend mirrors findings from prior benchmarks on learned indexes in in-memory systems [42], underscoring a fundamental characteristic of the memory-latency tradeoff inherent to learned index designs.

5.2 Impact of Index Granularity

Observation 3: Learned indexes consume less memory as the granularity grows while it does not significantly affect the query performance.

To evaluate the impact of index granularity, we vary the SSTable size from 8MB to 128MB and also include the level-granularity model proposed by Dai *et al.* [7]. We run a point lookup-only workload with 1,000,000 queries and record the latency and memory usage of each index configuration. As shown in Figure 8 (rightmost), lookup latency remains largely unaffected by granularity, varying by only a few microseconds across all configurations. In contrast, granularity has a significant impact on memory usage: increasing SSTable size (i.e., coarser granularity) leads to substantial memory savings, with over a 10× reduction in memory usage when moving from 8MB SSTables to the level model.

Specifically, when the position boundary is greater than 64, memory usage decreases exponentially as SSTable size increases. This is because larger SSTables reduce the total number of SSTables in the system (for a fixed dataset size), thereby lowering the number of inner indexes and associated overhead. However, when the position boundary is smaller than 32, memory usage becomes relatively stable for most learned indexes—except for RMI. In this regime, although fewer inner indexes are needed, the number of segments created by the models remains high, and segment-level overhead becomes the dominant contributor to memory consumption. RMI behaves differently: its memory usage consistently decreases regardless of the position boundary. This is because RMI's memory cost is primarily driven by its inner index (i.e., the first-stage model), which remains the dominant component of memory usage across configurations, regardless of how many segments are created.

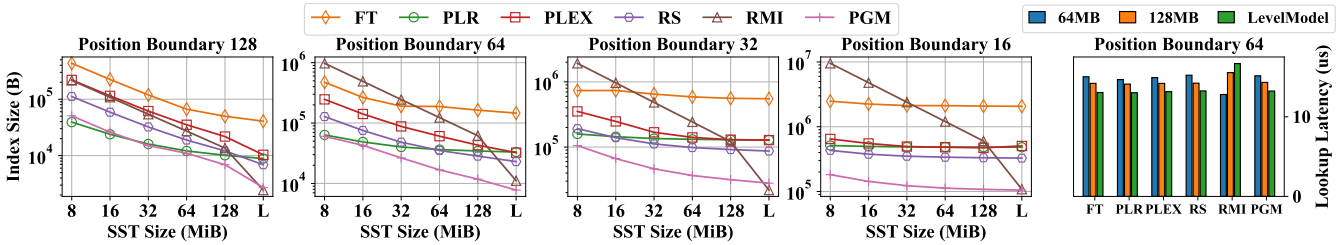

Figure 8: Impact of index granularity on point lookup.

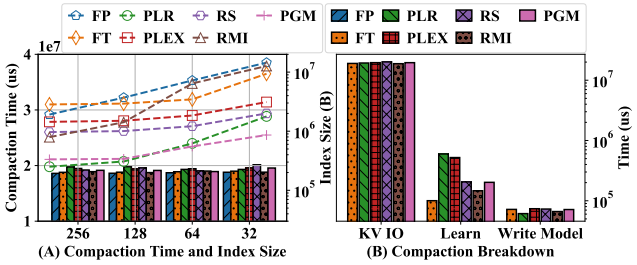

Figure 9: Compaction time and breakdown.

5.3 Compaction Overhead of Learned Indexes

Observation 4: Unexpectedly, the learning overhead introduced by learned indexes can be considered modest compared to the overall compaction overhead. Index learning and writing time account for less than 5% of the total compaction time in most of the cases.

As discussed in Section 2, compaction is triggered when a level reaches its maximum capacity, merging the data from that level into the next. In some systems[14, 18], only a portion of the data is merged to the next level to reduce compaction overhead and prevent write stalls. The compaction process involves reading the sorted run into memory, merging the data, writing it back to disk, and building an index for the new sorted run. When using a learned index, as described in the implementation section, this process also includes segmenting the sorted run and training a model for each segment. Previous research [7] highlights that the training time for learned indexes introduces significant overhead, particularly in write-heavy workloads. Therefore, in this section, we evaluate the overhead of learned indexes in a write-only workload with 1,000,000 operations with setting the write buffer to 64MB.

As shown in Figure 9, the compaction time remains almost unchanged as the index size grows for all the indexes. This is because the primary cost of compaction comes from reading and writing key-value pairs to and from disk. Additionally, the compaction time for learned indexes does not significantly exceed that of fence pointers. Specifically, most learned indexes show less than a 5% increase in compaction time, while PLEX exhibits around a 10% increase.

To break this down, we examine the time spent on learning and writing the model. For most learned indexes, model training takes less than 5% of the compaction time, which explains the minor increase in total compaction time. However, PLEX uses around 10-15% of the compaction time for training due to its self-tuning algorithm. As for model writing, most learned indexes consume less than 5% of compaction time. This demonstrates that the overhead introduced by learned indexes during compaction is modest.

The lower training overhead compared to Bourbon [7] result from advancements in hardware. Since the training process occurs in memory, CPU performance is a key factor. Using a high-end CPU, such as the i9-13900K, helps reduce this overhead. Additionally, Bourbon does not pipeline the training process with compaction. Instead, it reads data from the disk first and then performs the training using a background thread, likely leading to higher resource consumption.

5.4 Read Overhead at Different Levels

Observation 5: Evenly assigned position boundary across different LSM-tree levels may lead to suboptimal memory-latency tradeoff especially for skewed workload.

The LSM-tree has a layered structure, with each level's capacity growing exponentially. This leads to varying read overhead at each level, raising the question: is setting the same position boundary for learned indexes across all levels optimal?

As shown in Figure 10, we evaluate both uniformly and non-uniformly distributed workloads to measure the read overhead (i.e., lookup time) at each level when using the same position boundary. In the case of uniformly distributed workloads, the proportion of read overhead closely follows the level size, and the index size scales similarly with the level capacity due to the consistent position boundary across levels. However, for skewed workloads (i.e., non-uniform distribution), the read overhead is no longer directly proportional to the level capacity, resulting in an imbalance between index memory allocation and read overhead.

This imbalance suggests that allocating more memory to levels where read overhead exceeds the index memory proportion could improve the memory-latency tradeoff. In this sense, setting a uniform position boundary across different levels is not always a good choice. This insight is aligned with setting a different memory budget (i.e., bit-per-key) for bloom filters across levels in the LSM-tree [8].

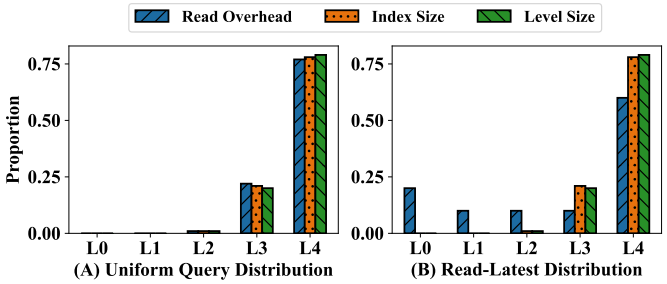

Table 1: The table presents the processing time for each stage of point lookup in PLR, with the position boundary set to 10.

Statistics of Point Lookup In Detail			
Process	SST=4MB	SST=32MB	SST=128MB
Table Lookup	0.19 us / op	0.11 op / us	0.07 us / op
Prediction	0.17 us / op	0.15 us / op	0.15 us / op
Disk I/O	2.12 us / op	2.10 us / op	2.16 us / op
Binary Search	0.16 us / op	0.15 us / op	0.16 us / op

Figure 10: The figure (left) shows the read overhead, the index size, and entries at different levels; the table (right) presents the detail of point lookup.

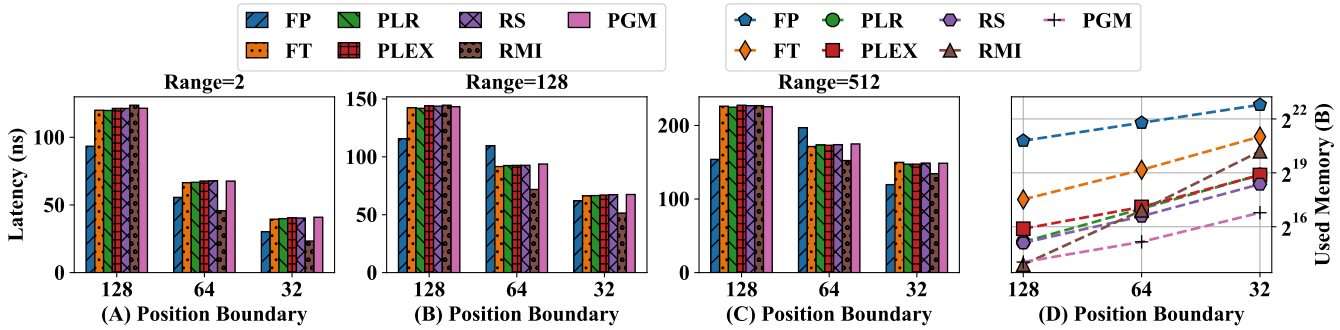

Figure 11: Performance of range lookup under different lookup range and position boundary.

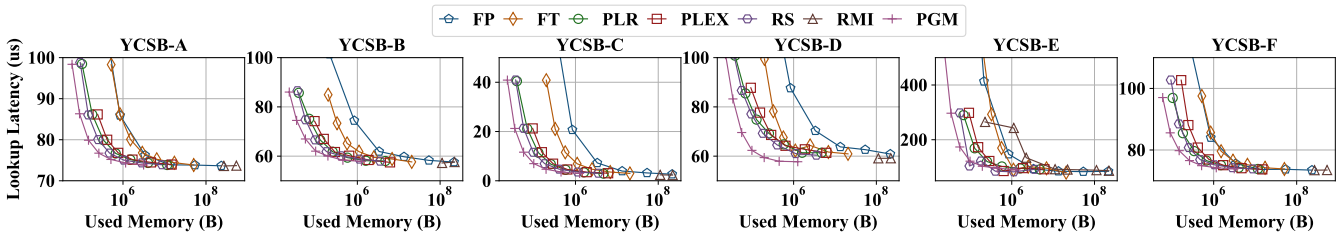

Figure 12: Average operation time of indexes under six YCSB workloads.

5.5 Impact on Range Lookup

Observation 6: Learned indexes offer a superior memory-latency tradeoff compared to fence pointers during short-range lookups. However, this advantage diminishes as the length of the lookup range increases.

Range lookups are a fundamental operation in LSM-tree systems, responsible for retrieving a specified span of key-value pairs. The process begins by locating the starting key of the range at each level, a task facilitated by learned indexes. Once the starting point is identified, the system sequentially retrieves the remaining entries within the target range.

To evaluate the effectiveness of learned indexes for range queries, we test various index types using 64MB SSTables under different range lengths. As shown in Figure 11, the memory-latency tradeoff for small ranges resembles that of point lookups. In these cases, all learned indexes outperform the traditional fence pointer in terms of memory efficiency while maintaining comparable query latency. However, as the range length increases, the latency across different indexing methods begins to converge, reducing the performance advantage of learned indexes. For example, in Figure 11(A), increasing the position boundary significantly improves performance in short-range queries. In contrast, as shown in Figure 11(C), the same adjustment yields little benefit for longer ranges. This is because, for short ranges, the dominant overhead lies in seeking the initial block, where learned indexes are most effective. In long-range queries, however, the main cost shifts to scanning and retrieving a large number of entries, which diminishes the relative impact of index precision. As a result, the memory-latency benefits of learned indexes diminish as range length increases, highlighting a limitation of their effectiveness in long-range query scenarios.

5.6 Impact on Mixed Workload

Observation 7: The memory-latency tradeoff does not vary a lot under update-lookup mixed workload compared to read-only workloads. Prioritizing read performance is a practical approach when evaluating a learned index in LSM-tree systems.

To evaluate the performance of learned indexes under more complex and real-world conditions, we test them with six YCSB workloads: A is read-write balanced, B is point lookup heavy, C is point lookup only, D focuses on recent point lookup, E is range lookup heavy (with ranges less than 100), and F is read-modify-write (50% lookup and 50% update). The result is shown in Figure 12.

Overall, the memory-latency tradeoff remains consistent with the results from the point lookup workloads tested earlier. Specifically, across all workloads, PGM continues to offer the best tradeoff, while FITing-tree lags behind other learned indexes in performance.

6 Discussion

In this section, we are going to conclude about the performance tradeoff of learned indexes on LSM-tree systems and how we can make them suitable for LSM-trees. The evaluation and analysis in the previous section have already found several observations regarding learned indexes in various aspects in the LSM-tree systems. To conclude, learned indexes showcase satisfactory memory-latency tradeoff in both range lookup and point lookup–using less memory and delivering stronger throughput, while the extra computation consumption for training is not remarkable, which demonstrates a superior potential of learned indexes to integrate learned indexes into LSM-tree systems. To help users understand more about having the best learned indexes in their LSM-tree systems, we are going to give three decision guidelines in the following.

6.1 Insights and Tuning Guide

Prioritize Position Boundary. In Section 4, we identify three key factors that influence the performance of learned indexes in LSM-tree systems, with position boundary having the most significant impact on read performance while different index types only impact the memory-latency tradeoff. Though some learned indexes focus on optimizing prediction or segment lookup, which can show excellent results in memory-based systems, these advantages become less noticeable in LSM-tree environments. For example, even though PLR employs the simplest inner index structure, its memory-latency tradeoff remains desirable in most cases. This underscores that position boundary plays a more critical role in overall performance compared to prediction optimizations, suggesting that optimizing the model position boundary to a smaller range within the same memory budget is more important than improving the efficiency of indexing these learned models in LSM-trees.

Increase the Index Granularity. While index granularity (i.e., SSTable size) has less impact on performance than the position boundary, increasing granularity can still yield up to a 10% improvement in the memory-performance tradeoff across all learned indexes. By reducing the memory required to store index structures, larger SSTables make it possible to allocate more memory toward decreasing the position boundary, thereby enhancing lookup performance. However, adopting a level-granularity model must be approached with caution, as it is only feasible when full merges are performed (i.e., merging an entire level into the next). Although full merges do not increase the overall write amplification [11], they can lead to short-term spikes in resource usage, which may temporarily degrade foreground performance.

Wisely Allocate the Memory Budget. Since the performance gains diminish once the index size surpasses a certain threshold, allocating excessive memory to learned indexes is not optimal. Instead, allocate a balanced portion of memory to the indexes based on your total budget, and dedicate the remainder to other in-memory components, such as Bloom filters and write buffers, to enhance overall performance. Additionally, because read overhead at each level is not always proportional to the level's capacity, using a uniform position boundary across all levels may not be the best approach. It is more effective to adjust the boundaries according to the specific query distribution.

6.2 Future Direction

Simply integrating existing learned indexes into LSM-tree systems is not the final step in learned index research. Based on our findings, there are two promising directions for further exploration: (1) developing a more sophisticated algorithm for dynamically allocating memory budgets for learned indexes, taking into account workloads, query distribution, and dataset characteristics to optimize the memory-latency tradeoff, and (2) incorporating learned indexes into the broader optimization of LSM-tree design space, such as in systems like Dostoevsky [9], Wacky [10], and Moose [35]. The first direction arises from our observation of the imbalance between read overhead and memory consumption at different levels. The second direction addresses a gap in recent LSM-tree design studies, which largely overlook the role of indexes, especially learned indexes. Given the strong memory-latency tradeoff demonstrated by learned indexes, considering them in the LSM-tree design configuration may lead to valuable optimization insights.

7 Related Work

LSM-tree Stores. Extensive research has focused on optimizing LSM-tree stores through comprehensive theoretical analysis and parameter tuning, such as size ratio, compaction policies, and Bloom filters [8–11, 21, 22, 35, 39, 43]. These studies have significantly improved the performance of LSM-tree systems. Additionally, works like Dostoevsky [9], Wacky [10], and Moose [35] define distinctive LSM-tree structures and derive optimal configurations by theoretically modeling the cost of various LSM-tree operations. Integrating learned indexes into these designs could offer valuable insights. Furthermore, self-tuning systems such as Cosine [3], Data Calculator [24], Design Continuums [23], and Limousine [4] model the costs of different index structures, including learned indexes and LSM-tree indexes, to calculate the optimal storage structure within a given budget. While these works provide broad insights into storage design, they lack fine-grained guidelines specifically tailored for LSM-tree storage and learned indexes, with some focusing primarily on cloud storage [3, 4]. Our study aims to complement this

research by offering more targeted design insights for LSM-tree systems and learned indexes.

Learned Indexes. Our study lies in the improvement of learned indexes techniques. In addition to the learned indexes discussed earlier, we review other notable approaches. MADEX [20] redesigns B+-tree nodes, incorporating CDF and correction models to enhance point lookups. RUSLI [20] modifies RadixSpline [26] to support updates, while FINEdex [32] introduces a buffer, building on XIndex [52], to handle updates more efficiently. LSI [25] is the first to model unsorted data. Some learned indexes, such as AULID [31], are specifically designed for disk-based systems. Additionally, recent studies have applied learned indexing to string, spatial, and multi-dimensional queries [13, 19, 33, 44, 45, 50, 53, 56]. Several evaluations [30, 41, 54] and surveys [17, 36] provide valuable insights into tuning issues and the evolving landscape of learned indexes. Integrating a wider variety of learned indexes into LSM-trees in the future could provide us with more valuable insights.

Learned Indexes in LSM-tree Systems. Due to the compatibility between learned indexes and LSM-tree systems, and the promising memory-latency tradeoff they offer, several recent studies [2, 7, 37, 47–49] explore integrating learned indexes into LSM-trees to improve lookup performance. Abu-Libdeh *et al.* [2] are the first to evaluate the feasibility of learned indexes in LSM-tree systems, though their study does not fully cover different configuration options or index types. Dai *et al.* [7] integrate piecewise linear regression models into their LSM-tree system [38] and propose Bourbon, achieving significant lookup improvements. Lu *et al.* [37] propose TridentKV, which integrates RMI [28] as the learned index and claims better performance than Bourbon in read-heavy workloads. Ramadhan *et al.* [47] further improve Bourbon by replacing binary search with exponential search, yielding moderate performance gains. However, these works do not fully explore the entire configuration space that affects learned index performance in LSM-trees, nor do they thoroughly investigate the memory-latency tradeoff. Our work aims to bridge this gap, providing additional insights and extending these foundational studies.

8 Conclusion

In this study, we have conducted a comprehensive theoretical and practical evaluation of integrating learned indexes into LSM-tree systems. We begin by revisiting existing learned indexes and analyzing their expected costs to identify key factors that influence LSM-tree performance. Through rigorous evaluations under various conditions, we have derived several design guidelines tailored to LSM-tree systems and provide practical insights for optimizing their performance.

9 Artifacts

To facilitate reproducibility and further exploration, we provide the full implementation, including source code, workload generators, and experiment scripts, in our public GitHub repository: https://github.com/qingshanlanshan/LearnedIndexInLSM. The repository includes detailed instructions on how to configure, build, and run the experiments described in this paper. Please refer to the `README.md` file in the repository for setup instructions, system requirements, and usage examples.

References

[1] -. Learned-Index-for-LSM-tree technical report. https://github.com/qingshanlanshan/LearnedIndexInLSM/TechnicalReport.pdf.
[2] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H Chi, Lyric Doshi, Tim Kraska, Andy Ly, Christopher Olston, et al. 2020. Learned indexes for a google-scale disk-based database. *arXiv preprint arXiv:2012.12501* (2020).
[3] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.
[4] Subarna Chatterjee, Mark F Pekala, Lev Kruglyak, and Stratos Idreos. 2024. Limousine: Blending Learned and Classical Indexes to Self-Design Larger-than-Memory Cloud Storage Engines. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–28.
[5] Source Code. 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger.
[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[7] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.
[8] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[9] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927
[10] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.
[11] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.
[12] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.
[13] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).
[14] Facebook. 2024. RocksDB. https://github.com/facebook/rocksdb.
[15] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.
[16] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 international conference on management of data*. 1189–1206.
[17] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 315–327.
[18] Google. 2024. LevelDB. https://github.com/google/leveldb/.
[19] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[20] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures.. In *AIDB@ VLDB*.
[21] Andy Huynh, Harshal Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).
[22] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24.
[23] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.
[24] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.
[25] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: a learned secondary index structure. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management.*

1–5.
[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*. 1–5.
[27] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
[28] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[30] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proc. ACM Manag. Data* 1, 2, Article 139 (June 2023), 22 pages. https://doi.org/10.1145/3589284
[31] Hai Lan, Zhifeng Bao, J Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2023. A simple yet high-performing on-disk learned index: Can we have our cake and eat it too? *arXiv preprint arXiv:2306.02604* (2023).
[32] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
[33] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.
[34] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index (Extended version). *arXiv preprint arXiv:2304.08817* (2023).
[35] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[36] Yu Liu, Hua Wang, Ke Zhou, ChunHua Li, and Rengeng Wu. 2022. A survey on AI for storage. *CCF Transactions on High Performance Computing* 4, 3 (2022), 233–264.
[37] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2021. TridentKV: A read-optimized LSM-tree based KV store via adaptive indexing and space-efficient partitioning. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1953–1966.
[38] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
[39] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.
[40] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *Proc. VLDB Endow.* 15, 5 (2022), 1079–1091.
[41] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *arXiv preprint arXiv:2006.12804* (2020).
[42] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.
[43] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proc. ACM Manag. Data* 1, 3, Article 213 (Nov. 2023), 25 pages. https://doi.org/10.1145/3617333
[44] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 985–1000.
[45] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.
[46] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
[47] Agung Rahmat Ramadhan, Min-guk Choi, Yoojin Chung, and Jongmoo Choi. 2023. An Empirical Study of Segmented Linear Regression Search in LevelDB. *Electronics* 12, 4 (2023), 1018.
[48] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, designing, and optimizing LSM-based data stores. In *Proceedings of the 2022 International Conference on Management of Data*. 2489–2497.
[49] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.
[50] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the last mile: Efficient learned string indexing. *arXiv preprint arXiv:2111.14905* (2021).
[51] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *CoRR* abs/2108.05117 (2021). arXiv:2108.05117 https://arxiv.org/abs/2108.05117
[52] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 308–320.
[53] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 17–24.
[54] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *arXiv preprint arXiv:2207.02900* (2022).
[55] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. 14, 8 (April 2021), 1276–1288. https://doi.org/10.14778/3457390.3457393
[56] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: robust learned index via distribution transformation. *arXiv preprint arXiv:2205.11807* (2022).