Response Letter for EDBT Submission 111 "Evaluating Learned Indexes in LSM-tree Systems: Benchmarks, Insights and Design Choices"

We would like to express our gratitude to all reviewers for their meticulous review and constructive comments. We dedicated substantial efforts to investigate and have addressed all the issues raised by reviewers. The revisions made in the paper are marked in blue, and some highlights are outlined below:

• **New experiments:** We have significantly enhanced our experimental evaluation by adopting a more practical and comprehensive setup, which includes the following improvements: (1) We extended the dataset size to 100GiB and use 3 synthetic YCSB-Generated datasets [3, 4] and 3 real world datasets [43]. (2) We incorporated four prevalent unclustered learned indexes into our evaluation: ALEX [14], DILI [35], LIPP [58], and LITS [60]. (3) Additional experiments and analysis on compaction overhead, range lookup, and variable length strings. (4) Error bars for point lookup and range lookup are added. Modified/added figures and tables are listed below: Figure 4, 5, 6, 7, 8, 13, 11, 14, 12 and Table 1, 2.

Meta-Reviewer

1. Better articulate the goals and experimental setup (R1D1,D2).
2. Explain the rationale behind selected indexes or broaden the comparison set (R1D3; R2D1).
3. Discuss how your method handles string or variable-length keys.
4. Include results from at least one or two additional datasets with interesting patterns beyond the one currently reported.
5. Repeat experiments to show variance and include error bars in performance figures.
6. Provide deeper analysis of range query performance (R2D5).
7. Incorporate more datasets to strengthen experimental robustness (R2D6).
8. Replace the vague “Wisely allocate the memory budget” with a more concrete concluding statement (R2D7).
9. Correct broken citations and links (R2D8).

We thank the meta-reviewer and all reviewers for their constructive feedback. We have added all required experiments and discussion to address these concerns. For detailed response please refer to: M1 corresponds to R1A1; M2 to R1A2; M3 to R1A3; M4 to R1A1 and R2A1; M5 to R5A2; M6 to R2A4; M7 to R1A1 and R2A1; M8 to R2A5; and M9 has been addressed.

To further improve clarity and presentation, we added more discussion and examples in Section 1 to better explain our motivation, and summarized our key findings and propositions in Section 6.3 and Table 2.

Reviewer #1

R1.W1. R1.D1. and R1.D2 The paper could describe the goals and setup and use of learned indexes more clearly.
The original SOSD benchmark generates datasets with 8B integer keys, but Section 5 mentions the datasets use 24B keys generated by the SOSD benchmark. It'd be good to clarify this change.

R1A1. We thank the reviewer for this helpful suggestion. Our paper aims to address two key problems: suitability and tuning guide. To improve clarity, we have added further illustrations and explanations in Section 1, along with additional details on the experimental setup at the beginning of Section 5. Regarding the dataset clarification, we now evaluate both the 8B SOSD real-world datasets and the 24B YCSB-Generated datasets.

R1.W2 and R1.D3 The rationale for evaluating only clustered indexes is unclear. Section 3.3 claims unclustered indexes like ALEX alter the storage layout, but this is unconvincing—both fence pointers and learned indexes map keys to segments. The paper should clarify this reasoning or include unclustered indexes in the evaluation.

R1A2. We have added 4 prevalent data-unclustered indexes into the experiment and demonstrate how to integrate them in Section 4.

R1.W3 and R1.D4 The current evaluation is limited to fixed-length keys, with little discussion on variable-length or string keys—an important area addressed by omitted unclustered indexes such as LITS (VLDB 2024) and NFL. Including discussion or evaluation on supporting variable-length keys in LSM-based learned indexes would strengthen the paper.

R1A3. We have included LITS in all experiments and added an evaluation of variable-length key queries in Section 5.3 and Figure 12.

Reviewer #2

R2.W1 and R2.D6 The paper claims broad evaluation but only presents results on randomly generated data (Uniform/YCSB). The link to the full report is broken. Including results from additional datasets would offer a more complete view and better support the claim of a comprehensive evaluation.

R2A1. The Pareto analysis of memory usage and performance across all six datasets is now included in Section 5. We have also fixed the report link, which now provides additional experiments on range lookups and compaction overhead.

R2.W2. Only data-clustered indices are considered in the experimental evaluation.

R2A2. Refer to R1A2. 4 unclustered indexes are added.

R2.W3 Settings of the experiments could be more clear (e.g. cold or hot cache for the experiments)

R2A3. We have clarified the cache settings for LSM-tree at the beginning of Section 5. Additional details about the datasets and implementation have also been included in Sections 5 and 4.

R2.D5. The "short range queries" in Fig.11 show improved memory/latency trade-offs, but the range result size is unknown a priori. It's unclear if queries span multiple segments; if only one is fetched, what happens when ranges exceed a segment? Will additional I/O degrade performance?

R2A4. We have added a detailed analysis of range lookups in Figures 11 and 12. This includes separating the lookup into seeking and scanning phases, measuring each individually, analyzing segment retrieval, and examining how the position boundary influences performance.

R2.D7. "Wisely Allocate the Memory Budget." is vague and does not represent the knowledge

R2A5. We have refined the statement in Section 6.2 to make it more precise and now explicitly reference the corresponding observation to clarify the intended takeaway.

R2.D8. [27] [28] citations are duplicate, same as [41] and [42]. Citation [1] does not work

R2A6. All the citations are fixed.

R2.D1. At least one index in the evaluation and the difference in performance.

R2A7. Data-unclustered indexes have been added.

Reviewer #3

R3.W1 The paper's core proposition is unclear. Memory use of a few MBs seems negligible on a 128GB system; the focus on read-heavy scenarios overlooks LSM-trees' design for write-heavy workloads; and the evaluation is limited to just 6.4M key-value pairs.

R3A1. Thanks for your constructive advice. We re-structure Section 6 and specifically summarize our findings in two aspects: compatibility with LSM-tree and tuning guides. Table 2 and Section 6.3 are added to provide a clearer summary. The experiment is extended to 100GiB datasets with up to 800 million key-value pairs and test the compaction overhead on write-heavy scenario with up to 50GiB entries are compacted as shown in Figure 7, 11, 13.

R3.W2. Several observations, such as the memory–selectivity trade-off with the "position boundary," seem fairly obvious.

R3A2. We thank the reviewer for this comment. As discussed in Section 5.1, the memory–performance trade-off varies significantly across index types, position boundaries, key distributions, and index granularities. Many of these trade-offs behave quite differently in LSM-tree systems compared to prior work [43, 57]. In addition to performance, we also consider other critical factors such as rebuild cost and seeking efficiency in LSM-trees. To make these insights clearer, we have consolidated our key observations in Section 6.3 and summarized them in Table 2.

R3.W3. and R3.D2. The effectiveness of learned indexes is highly data-dependent. Synthetic datasets often oversimplify the problem, making them less suitable than real-world, more irregular data.

R3A3. We have incorporated three real-world datasets: Facebook ID, OSM Cell ID, and Wikipedia Timestamp, into our experiments. In addition, we provide a more detailed analysis across different data scales, entry sizes, and key distributions in Section 5.1, Figure 7, and Figure 8.

R3.W4. It's unclear how the authors controlled for implementation effects. Index traversal speed can depend on code quality—how was this accounted for in the measurements and conclusions?

R3A4. We use the official implementations released by the respective authors, or those provided in well-established benchmark frameworks [42], which ensures efficient and representative traversal performance. In addition, we have added Figures 4 and 5 to illustrate how learned indexes are integrated into the LSM-tree system in different ways in Section 3.3 and Section 4. Performance of different implementations are included in Section 5.1 and Figure 7.

R3.D1. The motivation for combining LSM-trees with learned indexes is unclear and appears somewhat ad hoc, lacking a concrete real-world use case. The dataset used (6.4M keys) is modest by today's standards, and the need for such indexing complexity is questionable. It's also unclear how representative the evaluated workloads are of real-world scenarios.

R3A5. We have clarified the motivation for integrating learned indexes into LSM-trees with a concrete use case in Section 1. The dataset scale has been substantially extended to larger sizes. Furthermore, beyond operation-wise evaluation, we now include six YCSB workloads, a widely used key-value store benchmark, to better capture real-world workload characteristics.

Reviewer #5

R5.D1, R5.D2, R5.D3 Section 1, and in Section 3.3, 4.1, and 6.1 could be shortened.
Figure 5 needs to be clarified.
learned indexes can improve memory usage but not latency compared to FP. This point should be further discussed in the paper.

R5A1. Thanks for the constructive advice. We have re-organized the relevant sections, refined the figure, and added a table to better illustrate the details. In addition, Figure 8 has been included to justify the observed latency, and Section 5.1 now provides further discussion to explain the memory usage.

R5.D4 The experiments should be repeated multiple times and the figures should include error bars. This is particularly important since the latencies measured are very small (microseconds).

R5A2. Error bars are added by repeating the experiment of range lookup and point lookup (Figure 7 and Figure 11).

R5.D5. The URL provided for the technical report is broken.

R5A3. All the citations are fixed.

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

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

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

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

Meng Li
Nanjing University
China
meng@nju.edu.cn

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

Abstract

LSM-tree-based data stores are widely adopted for their high performance, but efficient querying becomes increasingly challenging as data scales. Recent efforts to integrate learned indexes into LSM-trees have shown promise in improving lookup performance. However, only a narrow range of index types has been explored, and their relative strengths and limitations remain unclear, hindering practical adoption. To address this, we present a comprehensive benchmark for systematically evaluating learned indexes in LSM-tree systems. We summarize the workflows and theoretical costs of eight learned indexes, identify key factors affecting their performance, and introduce a novel configuration space covering index types, boundary positions, and granularity. We implement and evaluate these designs on a unified platform across diverse configurations. Our findings reveal surprising insights, such as limited lookup gains from large memory budgets and low retraining overhead. Finally, we provide practical guidelines to help developers effectively select and tune learned indexes for real-world use.

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 [8], Apache Cassandra [30], and MongoDB WiredTiger [7]. 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 can substantially improve lookup performance, their memory usage grows linearly with the data volume. For example, indexing 10TiB of data with 128B keys can consume up to 320GiB of memory. Although disk space is typically abundant, memory remains a constrained resource—particularly in cloud-native, multi-tenant environments [45]. Excessive memory consumption by fence pointers may limit the space available for other critical in-memory components, such as Bloom filters, write buffers, and block caches, ultimately degrading overall system performance.

To address this issue, replacing fence pointers with learned indexes offer a promising solution due to their strong query performance and superior memory efficiency. For example, the PGM index has been shown to grow in space complexity at only $O(\log \log N)$ [37], in contrast to the linear growth of fence pointers, highlighting their potential to further enhance LSM-tree systems. Building on this insight, several studies [9, 39] have explored integrating learned indexes into LSM-tree architectures to improve both query latency and memory utilization. Although numerous evaluations [43, 57] have demonstrated the performance of learned indexes in both in-memory and on-disk environments, their effectiveness within LSM-tree systems remains largely unexplored. This gap arises from the unique hierarchical data structure of LSM-trees, compaction strategies, and the interplay among various in-memory components, which together influence overall performance in complex ways. To better understand the potential of learned indexes in this context, we identify two key questions that are critical for evaluating their effectiveness in LSM-tree systems:

Are all learned indexes suitable for LSM-tree systems? Recent studies have explored integrating learned indexes into LSM-tree systems. For example, Dai *et al.* [9] applied a PLR model to LevelDB, while Lu *et al.* [39] incorporated RMI, both showing improvements over traditional fence pointers. However, these efforts raise a critical question: are all modern learned indexes inherently compatible with LSM-tree architectures? Many learned indexes are designed for specific goals—such as supporting variable-length strings [60], improving updatability [14], or refining segmentation

*Corresponding Author

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

algorithms [17]—but their impact on key LSM-tree challenges like memory efficiency, compaction-induced rebuilding costs, and various types of query performance remains underexplored. Moreover, range queries in LSM-trees require locating the first key not greater than the range's start key and merging sorted results across levels. Many learned indexes [35, 58, 60] are not specifically designed for such range semantics, making their performance in LSM-tree settings even less predictable.

How can we integrate and tune learned indexes in LSM-tree systems? Since most learned indexes are designed for in-memory settings, integrating them efficiently into LSM-trees is non-trivial. Tuning is even more challenging due to the interplay of multiple LSM components—such as Bloom filters, caches, and write buffers—that compete for limited memory and jointly impact both update and lookup performance. How to allocate memory effectively between learned indexes and these system components remains an open question. In addition, the configuration space of learned indexes is highly diverse, with different models requiring distinct parameters and optimization strategies. Developing a universal tuning guideline would therefore be valuable, yet difficult, given the heterogeneity of index designs and their complex interactions with system-level trade-offs.

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 affecting their compatibility with LSM-trees. We begin by summarizing the structures and underlying algorithms of existing learned indexes, then classify them based on data layout to assess their suitability for LSM-tree architectures. For example, representative indexes such as ALEX [14] and LIPP [58] are primarily designed for in-memory lookups and rely heavily on pointer chasing, making their layouts less compatible with the contiguous, sorted structure of LSM-tree levels. In contrast, RMI [29] and PGM [17] align more naturally with sorted runs, as they store data in contiguous segments. Based on these characteristics, we categorize learned indexes into data-clustered and data-unclustered groups, and subsequently provide a detailed compatibility analysis for each.

We identify three unified key parameters that affect performance for learned 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 and rebuilding cost. 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 ten 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 provide several key insights. First, data-clustered indexes generally offer a better memory–performance trade-off for lookups and more intuitive range lookup semantics than data-unclustered indexes across diverse datasets and key sizes. Second, while model learning and writing during compaction are modest relative to key-value I/O for large entries, learning time becomes significant with smaller entries, particularly for data-unclustered indexes. Finally, although increasing index granularity has little effect on performance, it can reduce memory consumption for data-clustered indexes. 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 several tuning principles for integrating learned indexes into LSM-trees:

- Data-clustered indexes generally have better memory efficiency than data-unclustered indexes.
- A smaller position boundary improves the performance of data-clustered indexes.
- Larger SSTables enhance lookup performance by lowering index memory overhead and allowing smaller position boundaries within the same memory budget.
- Memory allocation shows diminishing returns: once segment size falls below the I/O block size, additional memory provides little improvement.

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 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:

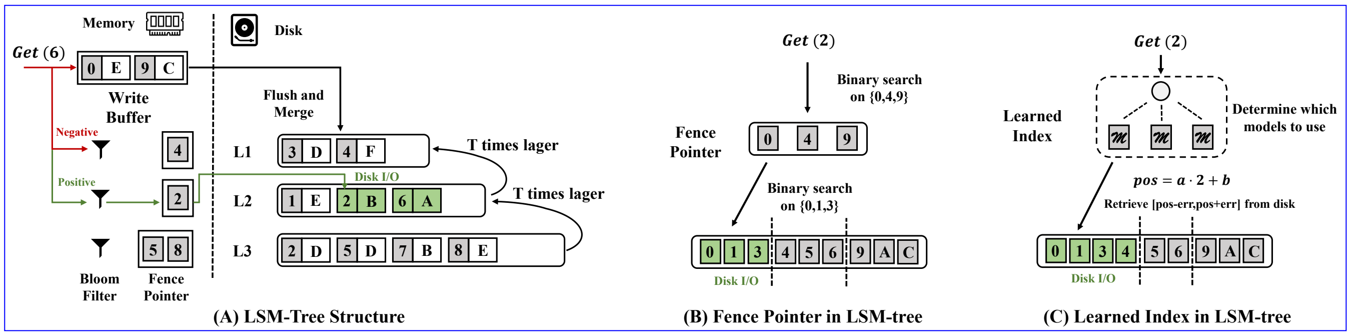

Figure 1: (A) presents the general structure of an LSM-tree; (B) presents the structure of fence pointer and (C) shows an example of how learned index replaces fence pointers

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 [16, 20] 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 [16, 20, 30], 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 [40], developing Bourbon [9], 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 the compatibility of these indexes with LSM-tree regarding the memory efficiency and several fundamental operations such as point lookup, range lookup and compaction.

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) [9], 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

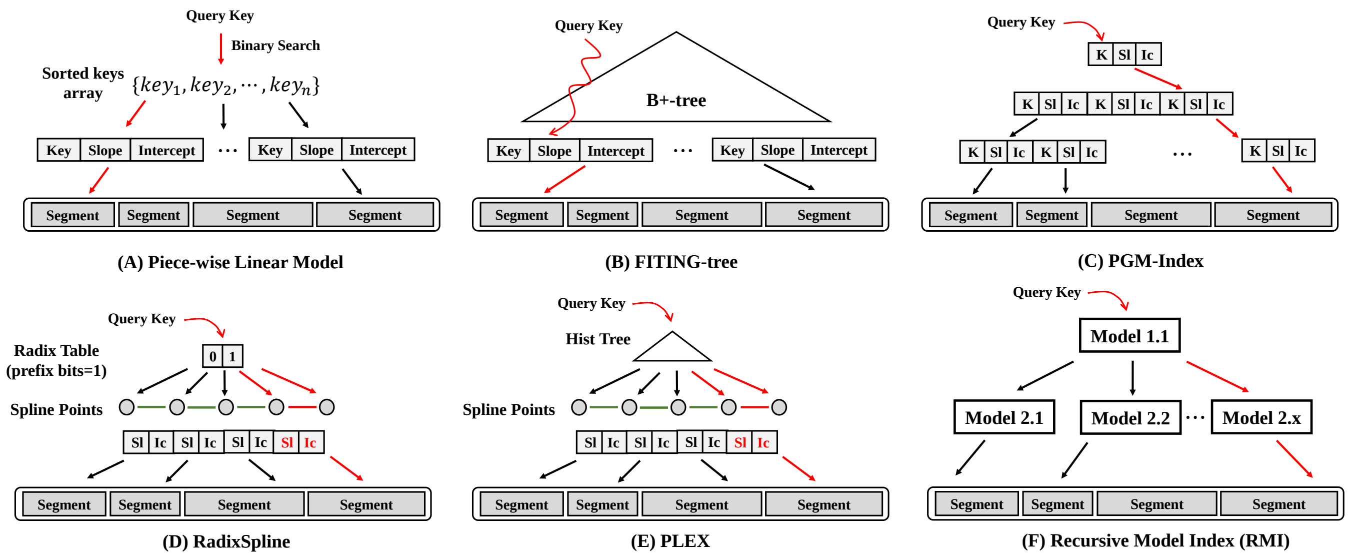

Figure 2: (A) to (F) present the data structures of data-clustered learned indexes and the general lookup procedure. Ic represents the intercept of the linear model, Sl is the slope, and K is short for the key.

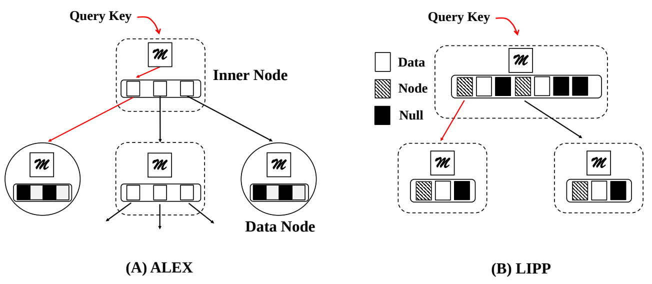

Figure 3: (A) and (B) present the structure of ALEX and LIPP. The data (key-value pairs) are not stored continuously to accommodate incoming new entries.

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 [18], 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) [17], 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) [28], 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) [54], 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.

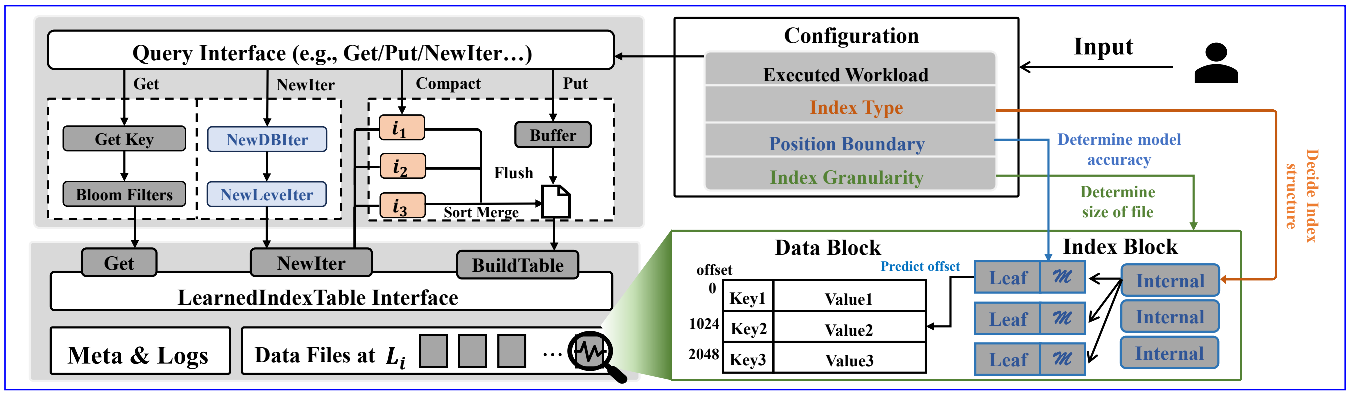

Figure 4: **The figure demonstrates the architecture of our testbed system. The left-hand side is the detailed conduction of different operations to test while the right-hand side is how the three configuration impact the system.**

Recursive Model Index (RMI) [29], shown in Figure 2 (F), is a learned index that recursively applies machine learning models to 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 [14], shown in Figure 3(A), uses inner and data nodes, each combining arrays with linear models to predict key positions. Inner nodes hold pointers to children; data nodes use gapped arrays for efficient insertions. Lookups traverse inner nodes via model predictions, then use exponential search within data nodes. ALEX adapts to data changes by dynamically splitting or merging nodes, maintaining efficient performance under dynamic workloads.
LIPP [58], shown in Figure 3(B), uses a linear model in each node to predict key positions. Each node contains a data array and a bitmap indicating three slot types: DATA, NULL, and NODE. The FMCD algorithm is used to minimize key conflicts. Insertion into a NULL slot converts it to DATA and stores the key-value pair. If the slot is already DATA, it becomes a NODE pointing to a new child node, built recursively with the conflicting keys. During lookup, the root's linear model predicts the slot. If it's a NODE, the search follows the pointer to the child; if it's DATA, the key is checked and returned if matched.
DILI [35] 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 [59] 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.
LITS [60] is a learned index optimized for string keys. It combines a trie-based structure with linear models to efficiently predict key positions. LITS partitions the key space using a compact trie, where each leaf node stores a linear model trained on the suffixes of keys within that partition. During a lookup, LITS traverses the trie using the key prefix to locate the corresponding model, which then predicts the approximate position of the full key in a sorted array. This hybrid design allows LITS to handle variable-length string keys efficiently while maintaining low memory overhead. It also supports dynamic updates by retraining local models when prediction errors exceed a threshold.

3.3 Compatibility Analysis

While both data-clustered and data-unclustered learned indexes offer strong read performance and memory efficiency in in-memory settings, their behavior differs significantly in LSM-tree systems. Below, we compare them across four key dimensions:
Memory Efficiency. Data-unclustered indexes allocate empty slots during construction to support future inserts, whereas data-clustered indexes segment existing sorted arrays and build models directly on them. In LSM-trees, where on-disk data is immutable (SSTables), the updatable structure of data-unclustered indexes introduces unnecessary memory overhead, reducing memory efficiency.

As illustrated in Figures 3 and 5, data-unclustered indexes typically store mapped integer keys in both internal and leaf nodes, while data-clustered indexes store them continuously in leaf nodes only. This design allows data-clustered indexes to omit storing data segments, since SSTables already preserve the key-value order. In contrast, data-unclustered indexes must maintain all mapped keys in memory: if only data block offsets were stored, lookups would require numerous random I/Os, resulting in worse performance than a simple binary search on the data blocks. In our experiments, we further analyze this trade-off by comparing memory usage when storing all keys versus omitting leaf segments for both clustered and unclustered indexes.

Point Lookup. Like fence pointers, data-clustered indexes predict a key's position and retrieve a small range from disk, followed by a binary search. In contrast, data-unclustered indexes often avoid reading unnecessary data by directly predicting a precise location in the structure, optimizing the "last-mile" search.

Range Lookup. LSM-trees perform range queries in two steps: a *seeking phase* (locating the start key in each level) and a *scanning phase* (retrieving and merging entries across levels). Data-clustered indexes naturally support the seeking phase by predicting the start key's position within a bounded error. Most data-unclustered indexes, however, do not support seeking efficiently—especially when the start key is absent—except for ALEX. Additionally, since their leaf nodes are typically unlinked, traversal may require scanning large portions of the structure.

Compaction and Index Rebuild. During compaction, LSM-trees merge levels and rebuild indexes for the new SSTable. Both index types we revisit support bulk loading. However, some data-unclustered indexes—like LIPP—require additional preprocessing (e.g., FMCD optimization) before data insertion, which can increase rebuild time.

4 Implementation and Configuration

In this section, we are going to introduce how to integrate the before-mentioned indexes to a unified LSM-tree testbed–LevelDB, a well-known LSM-tree key-value store. Based on the implementation, we then identify three important configuration that could significantly affect the query performance, memory efficiency, and compaction cost.

4.1 Implementation

To systematically evaluate the effectiveness of the three aforementioned configurations, we build a benchmark system based on LevelDB. The overview of our system is presented in Figure 4. 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 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. As shown in Figure 4, an SSTable consists of data blocks and index blocks: the data blocks store the original key-value pairs, while the index blocks store either fence pointers or learned indexes. For fence pointers, the smallest key of each data block is recorded in ascending order. For learned indexes, the key space is normalized into an integer domain, and models are trained on this transformed set.

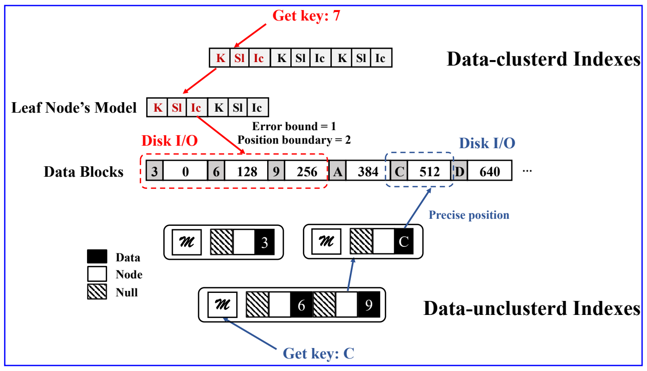

Figure 5: Implementation of two kinds of indexes.

Because data-clustered and data-unclustered indexes differ in design, we implement them slightly differently (Figure 5). For data-clustered indexes, the model predicts a position guaranteed within a bounded error. Thus, it suffices to retrieve entries in the range $[pos - err, pos + err]$ and check for the target key. Since keys are stored contiguously, mapped key segments can be omitted. By contrast, data-unclustered indexes typically consist of inner nodes and leaf data nodes. At a leaf, the trained model predicts the target's position directly, which contains both the mapped key and its location in the corresponding data block.

NewIter. The iterator interface is essential in LSM-tree systems, supporting both range queries and compaction. For data-clustered indexes, we follow a process similar to `InternalGet`: the trained model predicts the position of the first key not larger than the sought key during the seeking phase. Even if the key itself is absent, the correct starting key is guaranteed to fall within the error bound. In contrast, data-unclustered indexes do not store mapped keys in strictly ascending order within leaf nodes. As a result, finding the first key not larger than the sought key may, in the worst case, require traversing the entire tree.

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.

4.2 Learned Index Configuration

From our implementation, we identify three key factors that influence the query performance, memory efficiency, and compaction cost of learned indexes:

Index Type. Data-clustered and data-unclustered indexes behave quite differently for point and range lookups, leading to varying query performance depending on the choice. Even within each

Table 1: Testing Dataset

Dataset	Size	Entry Number	Key Size	Value Size
Uniform	100GiB	104,857,600	24B	1000B
Zipfian ($\theta = 0.9$)	100GiB	104,857,600	24B	1000B
Zipfian ($\theta = 0.99$)	100GiB	104,857,600	24B	1000B
Facebook ID	2.98GiB	200,000,000	8B	8B
OSM Cell ID	11.9GiB	800,000,000	8B	8B
Wikipedia Timestamp	2.98GiB	200,000,000	8B	8B

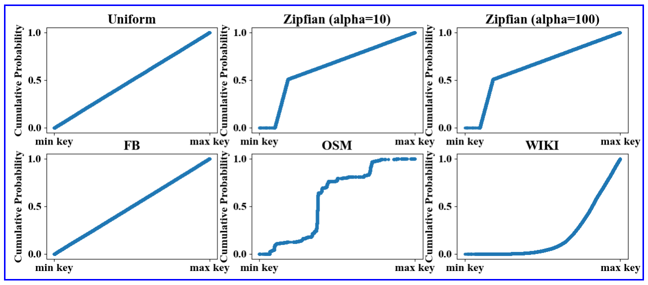

Figure 6: The key distribution of 6 different datasets. We display the skewness parameter $\alpha = 1/(1 - \theta)$ to replace the original parameter in YCSB-Gen project.

category, different index designs can yield distinct trade-offs in memory efficiency and prediction cost (i.e., index lookup time).

Position Boundary. For data-clustered indexes, lookup and iteration performance depend heavily on the prediction model's accuracy. Larger errors increase I/O overhead and degrade query performance. Prior work [43] also shows that higher accuracy generally requires more memory. Whether such additional memory investment is worthwhile in LSM-tree systems remains an open question.

Index Granularity. Modern LSM-tree implementations (e.g., LevelDB, RocksDB, PebblesDB [16, 20, 49]) 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. Dai *et al.* [9] 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.

5 Evaluation

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. Following the practical tuning guide [47], we limit OS page buffer usage to 70% of total memory (75 GB) using cgroup and employ the default 8 MB LevelDB cache as a faster in-memory cache.

Datasets and Workload Setup We evaluate all 10 indexes on YCSB-Generated datasets [3, 4] with 24 B keys and 1000 B values, following prior LSM system studies [11, 36, 44]. The synthetic datasets include three key distributions: Uniform and two Zipfian distributions with skewness parameters $\theta = 0.9$ and $\theta = 0.99$. While the classical Zipf distribution is parameterized by α, YCSB-Gen instead uses $\alpha = 1/(1 - \theta)$ to control skewness. Each dataset contains over 100 million entries, totaling 100 GiB, with integer keys and values zero-padded to the target size. To further evaluate real-world effectiveness, we also test on three open-source datasets [43]: the Facebook ID dataset (**FB**, 200 million keys), the OSM Cell ID dataset (**OSM**, 800 million keys), and the Wikipedia Timestamp dataset (**WIKI**, 200 million keys). The CDFs of all tested datasets are shown in Figure 6.

Settings of Learned Indexes. We integrate the following baselines into our system. The implementations of PGM[1], RadixSpline[2], PLEX[3], ALEX[4], LIPP[5], DILI[6], LITS[7], and PLR[8] [9] 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[9] [42] and the FITing-Tree[10] implementation from the SOSD benchmark [43]. 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 [42] 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 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.

[1]https://github.com/gvinciguerra/PGM-index
[2]https://github.com/learnedsystems/RadixSpline
[3]https://github.com/stoianmihail/PLEX
[4]https://github.com/microsoft/ALEX
[5]https://github.com/Jiacheng-WU/lipp
[6]https://github.com/pfl-cs/DILI
[7]https://github.com/schencoding/lits
[8]We contacted the author to obtain the code
[9]https://github.com/BigDataAnalyticsGroup/analysis-rmi
[10]https://github.com/RKolla99/FITing-Tree

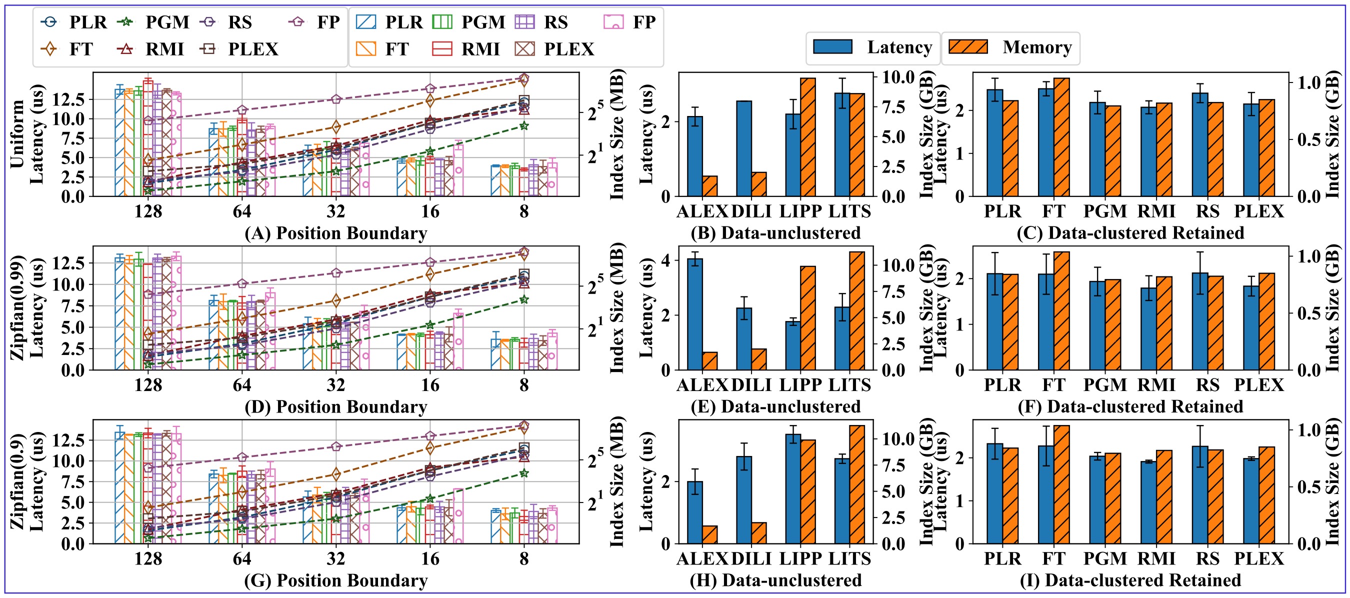

Figure 7: Evaluation of 10 learned indexes on YCSB-Gen datasets with three skew levels. In (A), (D), and (G), lines represent memory usage, while bars indicate query latency. Error bars denote standard deviation.

		Data-clustered Index							Data-unclustered Index			
		FP	PLR	FT	PGM	RMI	RS	PLEX	ALEX	DILI	LIPP	LITS
FB	Latency	2.13	2.0	2.12	2.50	2.0	2.6	2.8	1.01	1.04	1.03	1.03
	Size	372MB	1.6GB	750MB	66MB	17.7MB	127MB	201MB	3.2GB	3.6GB	23GB	8.1GB
OSM	Latency	1.4	2.1	2.0	2.02	8.33	2.0	1.18	1.41	-	-	1.3
	Size	1.5GB	1.9GB	2.5GB	204MB	70.2MB	449MB	770MB	12GB	-	-	32GB
WIKI	Latency	2.0	2.12	2.13	2.13	2.13	2.12	2.12	1.0	1.12	1.92	1.38
	Size	372MB	330MB	184MB	10.9MB	1.7MB	32MB	54.5MB	3.2GB	1.7GB	4.3GB	2.9GB

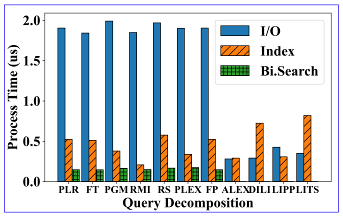

Figure 8: The table (left) reports index performance on three real datasets with the position boundary fixed at 16, while the figure (right) shows the breakdown of a point lookup. LIPP and DILI fail on OSM due to excessive memory requirements (LIPP) and unsupported key distribution (DILI).

5.1 Pareto Analysis

Observation 1: Data-clustered indexes generally consume less memory than fence pointers on the three YCSB datasets, but in the real-world datasets, only PGM, RMI, RS, and PLEX achieve consistently better memory efficiency than fence pointers.

In Figure 7(A)–(I), we compare 10 learned indexes with fence pointers across three datasets of varying skewness using one million point lookups.

Overall, data-clustered indexes achieve better memory efficiency than fence pointers and consume less memory than data-unclustered indexes, while maintaining comparable query performance when the position boundary is small on the YCSB datasets. This advantage arises because fence pointers must store the starting key of each segment (at least 24 B per key), and the number of segments grows linearly with the dataset size. Learned indexes, by contrast, map the key space to an integer domain and store only lightweight model parameters (e.g., two 8 B integers per linear model). Moreover, in indexes such as PGM, RMI, RS, and PLEX, the number of segments can grow sublinearly with data size, further reducing memory consumption.

When the key size shrinks to 8 B in the real-world datasets, however, PLR and FITing-Tree consume more memory than fence pointers. Fence pointers' memory cost decreases linearly with key size, while PLR and FITing-Tree still require same number of integers per model, offsetting their efficiency.

Finally, key distribution strongly affects memory efficiency. Although both the Facebook and Wikipedia Timestamp datasets contain 200 million keys, learned indexes on Facebook consume more memory. This is because Facebook keys are uniformly distributed, whereas Wikipedia keys are skewed, allowing learned indexes to exploit the skew for better compression and lower memory usage.

Observation 2: Allocating more memory to reduce the position boundary improves the performance of data-clustered indexes, but the benefit becomes marginal once the boundary drops below 16.

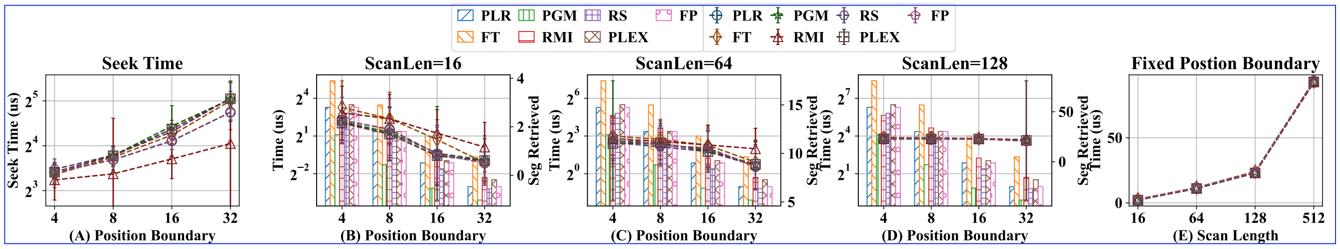
Figure 9: Performance of range lookup in different phases under YCSB-Gen Uniform dataset. The bars in (B) to (D) indicates how many divided segments are accessed during the query while the lines indicate how much time are consumed.

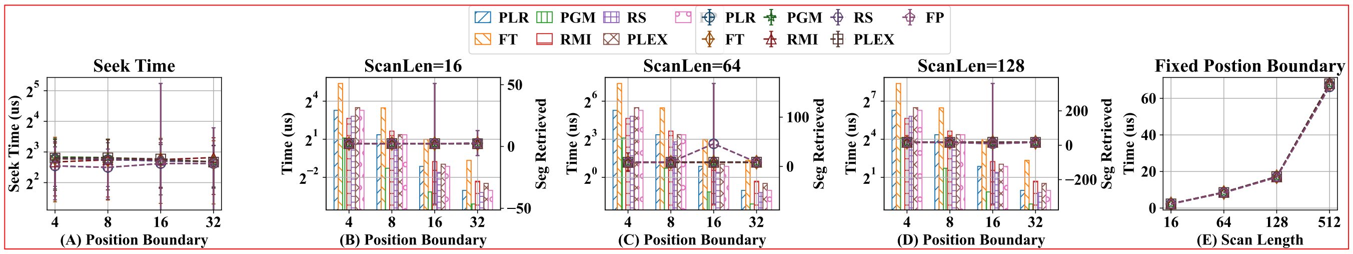
Figure 10: Additional result on Zipfian ($\theta = 0.9$) dataset

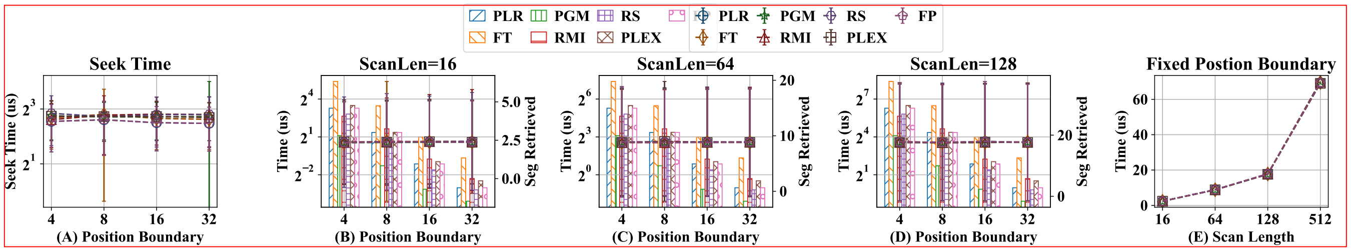
Figure 11: Additional result on Zipfian ($\theta = 0.99$) dataset

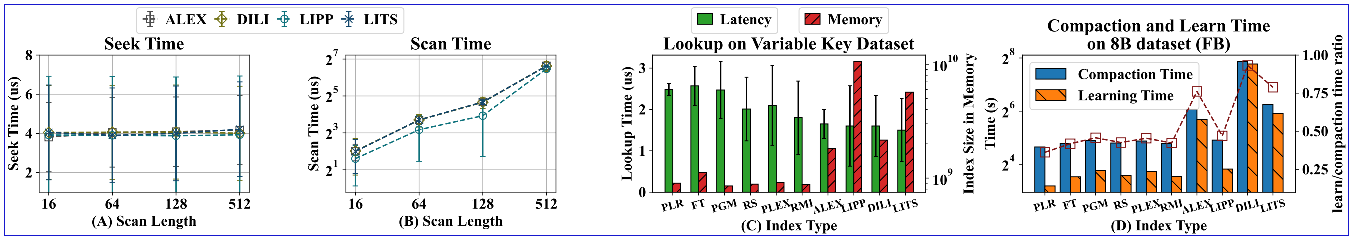
Figure 12: (A) and (B) show range lookup performance of data-unclustered indexes; (C) shows the performance of learned indexes with variable length key lookups; (D) presents the compaction time and learning time under dataset with small key size.

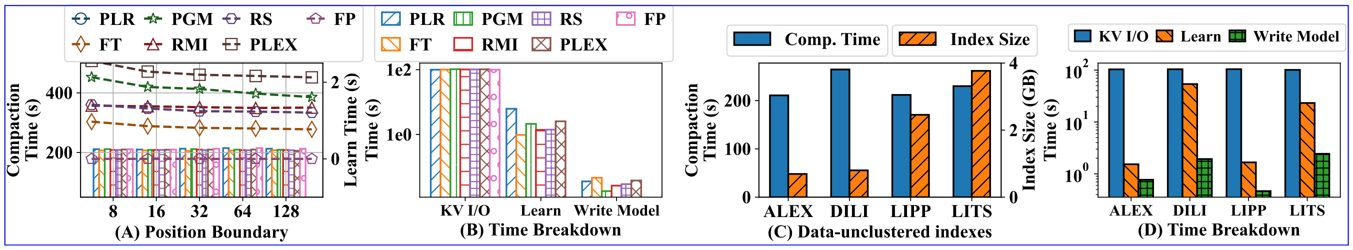
Figure 13: Compaction time, training time, and I/O time. The bars in (A) shows the time of finishing the compaction while the line presents the time used for training the models.

In Figure 7, we observe that increasing the memory budget to lower the position boundary significantly improves query performance, since I/O time is the dominant source of latency (Figure 8). However, once the boundary reaches 16, further gains become marginal: retrieving 16 entries takes only about 2us longer than retrieving a single entry, while memory consumption grows disproportionately. For example, retaining all mapped keys in memory consuming hundreds of megabytes improves performance by just 2us, whereas reducing the position boundary from 128 to 16 yields nearly 10us improvement at the cost of only tens of megabytes.

Observation 3: Data-clustered indexes typically achieve higher memory efficiency than the data-unclustered indexes.

Data-unclustered indexes require at least 2 GiB of memory to store mapped keys, whereas data-clustered learned indexes use only a few megabytes with a position boundary of 8. This difference arises because data-unclustered indexes must explicitly store all mapped keys in their structures, as keys are scattered across a tree (Figure 5), while data-clustered indexes can omit mapped integer keys since they follow the sorted order in data blocks. Instead, clustered indexes retrieve entries within the error bound and perform a binary search to locate the target. With a small position boundary (e.g., 8), their latency is 1–2us slower than data-unclustered indexes while saving gigabytes of memory.

We also evaluate data-clustered indexes that retain mapped key segments in memory, allowing them to perform I/O as efficiently as data-unclustered indexes. As shown in Figure 7(C), (F), and (I), these achieve nearly identical performance to unclustered indexes while still using less memory. A key reason is that unclustered indexes like ALEX, LIPP, and DILI are designed for updatable settings, employing mechanisms such as gapped arrays to support future inserts. In immutable SSTables, however, such designs reduce memory efficiency. Similarly, LITS relies on Tries to store full key strings, which further increases memory usage.

5.2 Impact on Range Lookup

Observation 4: The position boundary remains critical for the seeking phase in range lookups with data-clustered indexes, but its impact diminishes as the retrieved range length increases.

Range lookups in LSM-trees involve two steps: a seeking phase to locate the starting key in each sorted run, and a scanning phase to retrieve key-value pairs sequentially. We evaluate this by performing one million consecutive range lookups of varying lengths on the Uniform dataset.

As shown in Figure 11(A), the cost of seeking grows with the position boundary, similar to point lookups, since it involves locating the first key not smaller than the start key. In contrast, when the scan length is short, larger position boundaries can slightly reduce latency. This occurs because larger boundaries correspond to longer segments, increasing the likelihood that consecutive range lookups fall within the same segment, thereby benefiting from cache hits and reducing model-access overhead. However, this effect diminishes as the scan length increases, since retrieving many segments from disk lowers the cache hit rate. As illustrated in Figure 11(B)–(E), the scanning phase time is generally proportional to the scan length, with the position boundary influencing performance only when the scan length is small.

Additionally, most data-unclustered indexes, except ALEX, lack a suitable interface for locating the first key not smaller than the start key during the seeking phase, requiring extra modifications to adapt them for LSM-trees. Because their keys are not stored contiguously, finding the first key may sometimes require traversing the entire tree, especially when the start key is absent. Nevertheless, Figure 12(A) and (B) show that seeking performance remains comparable to data-clustered indexes, despite the slower tree traversal. This is because the dominant cost of seeking lies in accessing keys across all levels of the LSM-tree, while in-memory traversal is relatively inexpensive. Similarly, the scanning phase of range lookups scales proportionally with the query range length. More results on other datasets are included in the technical report [1].

5.3 Variable-Length Dataset

Beyond fixed-size strings used in graph and relational databases, we also evaluate learned indexes with variable-length keys. Using YCSB-Gen with a Uniform distribution, we generate keys from 8 B to 24 B, fix the position boundary at 8, and execute one million point lookups. During training, keys are padded to the maximum length and mapped to integers, except LITS, which uses a Trie to index variable-length keys. Since entry sizes vary, data-clustered indexes cannot directly fetch ranges like before. Instead, we retain all the mapped keys and position like in Figure 7(C).

As shown in Figure 12(C), query latency remains nearly identical to the fixed-key case. LITS, despite being tailored for variable-length strings, does not significantly outperform other unclustered indexes, since in memory index access dominates, while in LSM-trees disk I/O—even for a single entry—accounts for nearly half of query time (Figure 8). Learned indexes with mapped keys retained can directly locate entries, avoiding read amplification.

5.4 Compaction Overhead

Observation 5: Unexpectedly, the learning overhead is modest in long-entry datasets, contributing less than 5% of total compaction time. However, when entry sizes are small, the overhead grows significantly, up to 20% for data-clustered indexes and as high as 80% for data-unclustered indexes.

Figure 13 shows compaction and learning time on the YCSB-Gen dataset (Uniform), while Figure 12(D) presents results on the Facebook dataset. In both cases, we continuously insert 50 million new keys, triggering compactions that merge 50 million existing keys, corresponding to 50GiB and 5GiB respectively.

Unexpectedly, the learning overhead for both data-clustered and data-unclustered indexes is modest compared to the cost of reading and writing keys to disk. This is because large entry sizes dominate disk I/O time, whereas learning time scales only with the number of entries. Consequently, when entry size decreases from 1024B to 16B, I/O time is greatly reduced (Figure 12(D)), while learning time remains nearly unchanged.

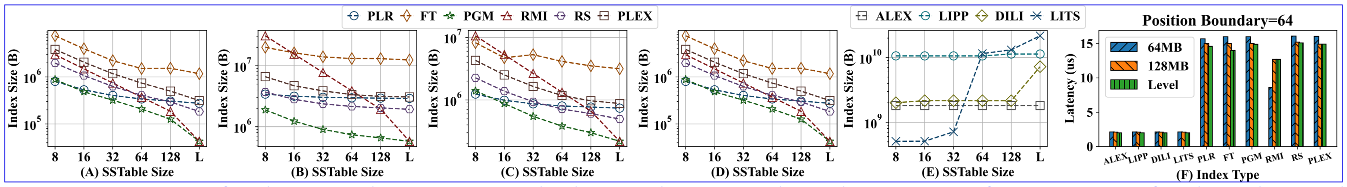

Figure 14: Impact of index granularity on point lookup with position boundary ranging from 128 to 16 for data-clustered indexes.

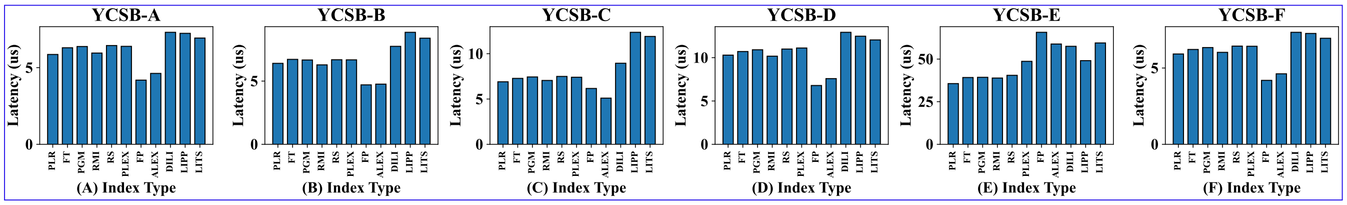

Figure 15: Average operation time of indexes under six YCSB workloads. Data-clustered indexes use position boundary 16.

In detail, data-clustered indexes generally require less time to build models, while data-unclustered indexes, such as DILI, take longer. This is because they insert keys individually into tree structures, which incur additional costs from splitting and balancing operations. More results on other datasets are included in the report [1].

5.5 Index Granularity

Observation 6: Data-clustered indexes consume less memory as the granularity grows while it does not significantly affect the query performance.

To study index granularity, we vary SSTable size from 8 MB to 128 MB and also test the level-granularity model of Dai *et al.* [9]. Using one million point lookups, we measure latency and memory usage (Figure 14, rightmost). Lookup latency changes little, varying only a few microseconds across configurations. Memory, however, is strongly affected: coarser granularity yields substantial savings, with over 10× reduction when moving from 8 MB SSTables to the level model.

For data-unclustered indexes, ALEX, LIPP, and DILI sustain stable lookup performance with only modest increases in memory usage as SSTable size grows. In contrast, LITS maintains a trie structure in memory, causing its memory consumption to grow more rapidly with the size of the indexed data.

5.6 Impact on Mixed Workload

To assess performance under more realistic conditions, we evaluate the learned indexes using six YCSB workloads: A (read-write balanced), B (point lookup–heavy), C (point lookup–only), D (recent point lookups), E (range lookup–heavy with ranges under 100), and F (read-modify-write, with 50% lookups and 50% updates). These workloads are conducted sequentially. Results are shown in Figure 15 with position boundary fixed to 16.

Overall, index performance closely mirrors that of the basic point and range lookup experiments, suggesting that learned indexes adapt well to mixed read-write scenarios without significant performance degradation.

Table 2: Four key factors influencing the suitability of learned indexes in LSM-tree systems. ✓ indicates efficiency, ⚠ highlights potential issues, and ✗ denotes severe limitations.

	Methods	Memory Efficiency	Key Lookup	Index Rebuilding	Key Seeking
Data -clustered Indexes	PLR [9]	⚠	✓	✓	✓
	FITing-Tree [18]	⚠	✓	✓	✓
	PGM [17]	✓	✓	⚠	✓
	RadixSpline [28]	✓	✓	⚠	✓
	PLEX [54]	✓	✓	⚠	✓
	RMI [29]	✓	✓	⚠	✓
Data -unclustered Indexes	ALEX [14]	✗	✓	✗	✓
	DILI [35]	✗	✓	✗	⚠
	LIPP [58]	✗	✓	⚠	⚠
	LITS [60]	✗	✓	✗	⚠

6 Discussion

In this section, we summarize the key insights from our evaluation and address the two central questions posed in Section 1: (1) Are all learned indexes suitable for LSM-tree systems? and (2) How can they be efficiently tuned within LSM-tree systems?

6.1 Compatibility

Are all learned indexes suitable for LSM-tree systems? We examine this question through four key aspects required for integration into LSM-trees: memory efficiency, key lookup, rebuilding during compaction, and seeking the first key not smaller than the range-start key. These factors are summarized in Table 2 and discussed below.

Memory efficiency. Observation 1 shows that **not all learned indexes save memory compared to fence pointers under all conditions.** This largely depends on how each model represents data segments and stores parameters. Among the evaluated methods, PGM, RMI, RadixSpline, and PLEX consistently provide the best memory efficiency, whereas the performance of others varies with key size and distribution. In general, data-clustered indexes are more memory-efficient than data-unclustered ones. Data-unclustered indexes, requiring maintaining mapped keys and empty slots for future updates, consume too much memory than data-clustered indexes. Furthermore, when the key distribution is skew, the learned indexes can usually save more memory than uniform ones.

Key lookup. Prior work on learned indexes has focused on minimizing lookup time within the index itself, as this dominates in-memory query costs. In LSM-trees, however, disk I/O becomes a critical factor. When the position boundary is large, I/O overhead dominates total query latency, diminishing the relative benefits of faster index lookups.

Rebuilding cost. Observation 5 reveals that rebuilding learned indexes accounts for only about 5% of compaction time on datasets with long entries, but becomes significant when entries are small. This occurs because I/O time decreases with smaller entries, while learning time scales with the number of entries. Data-clustered indexes generally maintain acceptable rebuild times even with 16 B entries. By contrast, data-unclustered indexes designed for updatability, incur high rebuild costs due to expensive insertion, splitting, and balancing operations. Although they are efficient for heavy update workloads in memory, they incur high costs in compaction-heavy scenarios within LSM-tree systems.

Seeking keys. As noted in Section 4, most data-unclustered indexes lack native support for seeking the first key not smaller than a given key, requiring code modifications to traverse the tree. In contrast, data-clustered indexes naturally support this operation since keys are stored contiguously in segments, enabling straightforward traversal even when the target key is absent. Performance-wise, the gap is small: although unclustered indexes require extra in-memory traversal, the dominant cost of seeking still comes from disk access across levels, making their seeking time comparable to clustered indexes.

6.2 Tuning Guide

Select the right index types. Both performance and memory efficiency depend heavily on the choice of index. In general, data-clustered indexes consume less memory than data-unclustered ones while achieving comparable lookup performance and naturally supporting seeking operations. Thus, data-clustered indexes are generally better suited for LSM-tree systems. Among them, PGM, RadixSpline, RMI, and PLEX consistently use less memory, even on datasets with very small entries.

Position boundary enhances both point and range lookups. For point lookups, the position boundary determines how many bytes must be retrieved from disk, and for data-clustered indexes, seeking a key also requires scanning within this range. Consequently, reducing the position boundary is the most direct way to improve lookup performance.

Avoid over-allocating memory to the position boundary. According to Observation 2, reducing the position boundary by allocating more memory can improve lookup performance by lowering disk I/O. However, the benefit becomes marginal once the boundary falls below 16, as I/O time is already minimal. In such cases, it is more effective to allocate memory to other in-memory components—such as Bloom filters, write buffers, or caches—to enhance overall system performance.

Increase index granularity. While index granularity (i.e., SSTable size) has less impact on performance than the position boundary, increasing granularity can still improve the memory–performance trade-off by up to 10%. Larger SSTables reduce the memory required for index structures, allowing more memory to be devoted to lowering the position boundary. However, adopting a level-granularity model requires caution, as it is only feasible when performing full merges (i.e., merging an entire level into the next). Although full merges do not increase overall write amplification [13], they may cause short-term spikes in resource usage and temporarily degrade foreground performance.

6.3 Takeaways

In summary, we provide the following guidelines for integrating learned indexes into LSM-tree systems:

① Use learned indexes when entry sizes are large or key distributions are skewed.

② Prefer data-clustered indexes such as PGM, RadixSpline, RMI, and PLEX for their superior memory efficiency and balanced performance.

③ Reduce the position boundary to improve query performance, but only when it is larger than 16, as further reductions yield marginal gains.

④ Increase index granularity (e.g., larger SSTables) to reduce memory consumption and improve the memory–performance trade-off.

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 [10–13, 23, 24, 36, 41, 44]. These studies have significantly improved the performance of LSM-tree systems. Additionally, works like Dostoevsky [11], Wacky [12], and Moose [36] 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 [5], Data Calculator [26], Design Continuums [25], and Limousine [6] 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 [5, 6]. 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 [22] redesigns B+-tree nodes, incorporating CDF and correction models to enhance point lookups. RUSLI [22] modifies RadixSpline [28] to support updates, while FINEdex [33] introduces a buffer, building on XIndex [55], to handle updates more efficiently. LSI [27] is the first to model unsorted data. Some learned indexes, such as AULID [32], are specifically designed for disk-based systems. Additionally, recent studies have applied learned indexing to string, spatial, and multi-dimensional queries [15, 21, 34, 46, 48, 53, 56, 59]. Several evaluations [31, 57] and surveys [19, 38] 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, 9, 39, 50–52] 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.* [9] integrate piecewise linear regression models into their LSM-tree system [40] and propose Bourbon, achieving significant lookup improvements. Lu *et al.* [39] propose TridentKV, which integrates RMI [29] as the learned index and claims better performance than Bourbon in read-heavy workloads. Ramadhan *et al.* [50] 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/buchuitoudegou/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] 2025. Learned-Index-for-LSM-tree technical report. https://github.com/buchuitoudegou/LearnedIndexInLSM/blob/master/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] brianfrankcooper. 2010. YCSB Workload Generator. https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads.

[4] brianfrankcooper. 2010. YCSB Workload Generator for C++. https://github.com/basicthinker/YCSB-C.

[5] 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.

[6] 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.

[7] Source Code. 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[8] 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.

[9] 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.

[10] 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.

[11] 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

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] Facebook. 2024. RocksDB. https://github.com/facebook/rocksdb.

[17] 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.

[18] 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.

[19] 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.

[20] Google. 2024. LevelDB. https://github.com/google/leveldb/.

[21] 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.

[22] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures.. In *AIDB@ VLDB*.

[23] 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).

[24] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24.

[25] 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*.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[31] 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

[32] 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).

[33] 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.

[34] 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.

[35] 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).

[36] 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.

[37] Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. 2024. Why are learned indexes so effective but sometimes ineffective? *arXiv preprint arXiv:2410.00846* (2024).

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *Proc. VLDB Endow.* 15, 5 (2022), 1079–1091.

[43] 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.

[44] 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

[45] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 726–737. https://doi.org/10.14778/2752939.2752942

[46] 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.

[47] PingCAP. 2025. TiKV Tuning Guide. https://docs.pingcap.com/tidb/stable/tikv-configuration-file/.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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).

[54] 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

[55] 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.

[56] 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.

[57] 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).

[58] 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

[59] 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).

[60] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proc. VLDB Endow.* 17, 11 (July 2024), 3415–3427. https://doi.org/10.14778/3681954.3682010

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009