DobLIX: A Dual-Objective Learned Index for Log-Structured Merge Trees

Alireza Heidari
Huawei
alireza.heidarikhazaei@huawei.com

Amirhossein Ahmadi
Huawei
amirhossein.ahmadi@huawei.com

Wei Zhang
Huawei
wei.zhang6@huawei.com

ABSTRACT

In this paper, we introduce DobLIX, a dual-objective learned index (LI) specifically designed for Log-Structured Merge (LSM) tree-based key-value stores. Traditional LIs primarily focus on optimizing index lookups, often overlooking the critical role of data access from storage, which can become a significant performance bottleneck. In LSM-based systems, a considerable portion of the index is stored on disk, making lookups highly dependent on the efficient coordination between in-memory structures and disk-resident data. Poorly optimized access patterns can lead to excessive I/O operations, negatively impacting read latency and overall system performance. DobLIX addresses this by incorporating a second objective, data access optimization, into the LI training process. This dual-objective approach ensures that both index lookup efficiency and data access costs are minimized, leading to significant improvements in read performance while maintaining write efficiency in real-world LSM systems. Additionally, DobLIX features a reinforcement learning agent that dynamically tunes the system parameters, allowing it to adapt to varying workloads in real-time. Experimental results using real-world datasets demonstrate that DobLIX reduces indexing overhead and improves throughput by 1.19× to 2.21× compared to state-of-the-art methods within RocksDB, a widely used LSM-based storage engine.

PVLDB Reference Format:
Alireza Heidari, Amirhossein Ahmadi, and Wei Zhang. DobLIX: A Dual-Objective Learned Index for Log-Structured Merge Trees. PVLDB, 18(11): 3965 - 3978, 2025.
doi:10.14778/3749646.3749667

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ah89/DobLIX.

1 INTRODUCTION

Context. Key-value (KV) databases are crucial in various domains like cloud computing, e-commerce, big data analysis, and artificial intelligence. Among different KV store architectures, Log-Structured Merge Trees (LSMs) stand out for their exceptional write performance [50]. They are widely used in industrial applications, such as RocksDB [20], LevelDB [1], Cassandra [36], and BigTable [10].

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749667

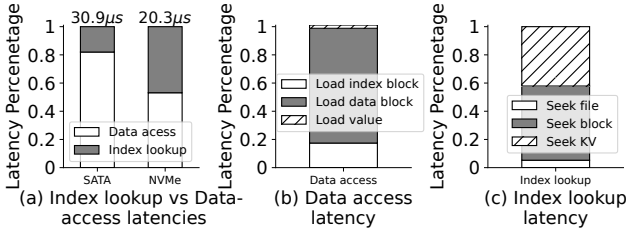


Figure 1: Lookup Latency Breakdown. Read performance on 10 million 8-byte KVs of the Wiki dataset using the native RocksDB index.

The multi-level structure of LSMs [49, 53] results in significant drops in read performance due to high read amplification [45]. The levels are divided into Sorted String Tables (SSTs) that contain KV pairs ordered by keys, with SSTs comprising fixed-size blocks (ranging from $4KB$ to $32KB$), and retrieving a specific KV pair involves an index lookup to locate the relevant level, file, block, and pair, followed by a data-access phase to load the block and fetch the desired pair. In current KV stores, with modern NVMe storage, both index lookup and data access contribute significantly to query latency (Fig. 1a).

A common strategy to enhance index lookup is Learned Indexing (LI), which involves using Machine Learning (ML) as a data structure, effectively revolutionizing data indexing by leveraging predictive capabilities and efficiently modeling records distribution patterns [18, 22, 23, 27–29]. These LI models usually reside inside memory and estimate the probable range containing a specific record by evaluating the target key through a monotonic function approximated by a cumulative distribution function (CDF) [26, 51]. However, when KV pairs are stored in storage, the LI bottleneck emerges during the data retrieval process from storage to memory [37, 44, 61, 69] (§2.2). Consequently, to enhance performance overall, it is essential to take into account these effective factors.

Current LI on LSM Research. The potential enhancement of LSM lookup performance through LI integration has been a focal point in recent studies such as *TridentKV* [45], *Bourbon* [15], *LeaderKV* [60], and *LearnedKV* [58]. These initiatives have introduced customized LI solutions for LSM structures, showing performance gains in *index lookup efficiency*. However, a **critical yet overlooked challenge** is that *indexing in LSMs is not merely about predicting data locations—it must also ensure efficient data access.* Unlike traditional small data structures, where indexes fit entirely in memory, LSM indexes often span both **memory and disk**, requiring frequent storage accesses. Consequently, **optimizing the index without considering how data is retrieved from storage can lead to inefficiencies**, such as excessive block I/O or misaligned lookups [25]. Fig. 1b shows that the retrieval of blocks from storage to memory is a dominant factor in lookup latency.

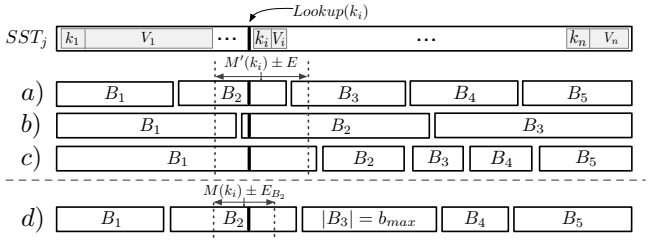


Figure 2: Comparison of LI Solutions on SSTs. Considering block partitioning Par_{Block} and the indexing $I_{IndexBlock}$ as stochastic variables. a) Small fixed-size blocks, b) Large fixed-size blocks with a guarantee on max block size. c) Variable block size with a model output ($M'(k)$) guarantee to load one block. d) Perfect solution with guarantees on model output ($M(k)$) for one block access with optimized block size.

This oversight results in a dilemma depicted in Fig. 2 when previous methodologies neglect this characteristic:

(1) *Adhering to fixed block sizes optimized for read operations*, which can result in inaccuracies in block lookup and require accessing multiple blocks during read processes. Solutions *a* and *b*, as depicted, face this challenge, a methodology also used by *Bourbon* and *LearnedKV* [58].

(2) *Opting for variable block sizes*, which can lead to significantly larger blocks, thus increasing block I/O time. Solution *c* exhibits this issue, a methodology also adopted by *TridentKV* [45] and *LeaderKV* [60].

These dilemmas arise from the **isolated optimization of LI models without considering the data access component**. Consequently, while these techniques reduce average latency compared to LSM-based KV store index lookups, they show **higher tail-latency** in cases requiring multiple or large block retrievals. This paper proposes a new LI solution to optimize **both the indexing and data access phases**, ensuring that predicted locations align with an efficient storage layout.

Furthermore, earlier studies have neglected the consideration of varying key and values lengths in their design (§2.3), as well as the enhancement of the **last-mile search**, which is the final phase to identify the target KV within the retrieved block. As shown in Fig. 1c, this step in the indexing process incurs nearly identical latency to block finding. **Thus, an LI methodology should integrate optimizations that jointly improve index lookup, block access, and last-mile search efficiency.**

Our Approach. We leverage multi-objective optimization techniques to incorporate data access overhead as an additional factor to design an LI framework [8]. To achieve a suitable configuration that optimizes index performance alongside this additional parameter, we employ an auto-tuning method based on reinforcement learning (RL). This enables us to develop configurations that generate index models capable of automatically adjusting when workloads change. Our approach specifically concentrates on constructing an index model for KVs at the SST level, leveraging their inherent sorted structure for efficient LI training. The resulting model enables **effective locating** of the desired **variable-size** KVs using merely **a single block** access with **ideal block size**, all while maintaining the LI model optimal performance.

We evaluate our framework using an array of real-world and synthetic datasets and workloads, showing a substantial reduction in indexing costs when measured against state-of-the-art indexing solutions. We show that our framework leads to better performance, with throughput improvements between 1.19× and 2.21× in various datasets and workloads.

Technical Challenges. Our approach needs to address multiple technical challenges:

- ***Index Model.*** A second parameter complicates index modeling. The model must optimize both parameters, but the cost feedback for the second one may be asynchronous.
- ***Restoration.*** During LSM query processing, data spans storage layers with varying performance. Missing metadata (e.g., the "index trajectory") can disrupt queries and require costly recovery. To avoid this, index system must preserve critical metadata to efficiently handle gaps and ensure smooth query execution.
- ***Adaptation.*** In an index framework, the parameters of the system can change due to the pattern of the incoming data or workload shifts. The design must adapt to these changes to ensure efficient performance under varying conditions.

Implementing the framework within an LSM-based KV store and conducting comprehensive benchmarking present challenges, primarily due to the incompatibility of many LI models with LSMs, as well as the inadequacies of current benchmarks (e.g., YCSB) in capturing important edge cases – including variable KV lengths, compaction overheads, and granular lookup latency breakdowns.

Contributions. We present the following contributions in this paper:

(1) ***Proposing* DobLIX*, a Dual-objective Learned IndeX framework***, for LSM-based KV stores (§3.2). This framework optimizes the performance of the LI lookup while considering any additional secondary objective parameter using two innovative LI approximation models, PLA and PRA (§3.3). In this work, our design specifically optimizes indexing and data access as a secondary parameter.

(2) ***Optimizing the last-mile search phase*** of the lookup process by incorporating the LI model traversal into the last mile search process (§3.5).

(3) ***Introducing an RL-based agent*** to dynamically adjust the indexing and data partitioning parameters in our system and to choose between the PLA and PRA methods (§3.6).

(4) ***Implementing our method on RocksDB*** and performing a comprehensive benchmark to demonstrate its superior performance compared to traditional and LI solutions (§4).

Paper Organization. The remainder of the paper proceeds as follows: In §2, we review the background concepts. §3 provides a detailed overview of our LI framework design and RL-agent. In §4, we evaluate our proposed solutions. We highlight related work in §5. We discuss our method portability and tuning agent robustness in §6, and conclude the key points of the paper and future directions in §7.

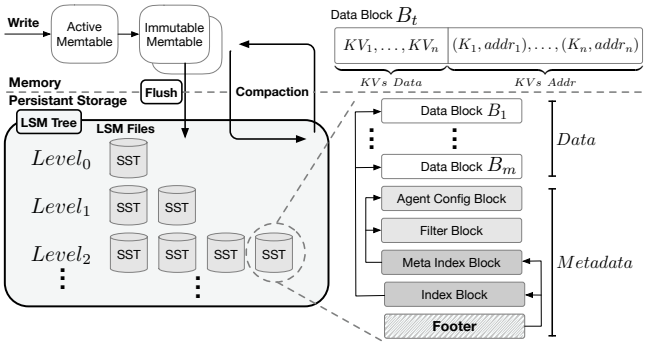


Figure 3: RocksDB Architecture.

2 BACKGROUND

2.1 Data Management and LSM-based KV Stores

To formally define the data management system's random variables in Fig. 2, we must express them mathematically [11]. First, we know two main ways to simplify big data management complexity: (1) *Data Partitioning* ($P(.)$): splits data into smaller parts [52]. (2) *Indexing* ($I(.,.)$): returns partition(s) containing the query key. $P(.)$ is unchanged by the key query but may integrate with $I_{LSM}(.,.)$ for given key k. Consequently, for simplicity in notation during the construction phase, we can represent indexing as $I(.)$. This approach enables creating indexed data structures for any dataset D. A general combination $\mathcal{F} = f_1 \circ f_2 \circ \cdots \circ f_k(D)$, where each f_i is from a set of partitions $\mathcal{P} = \{P_1, \ldots, P_n\}$, or a set of indexes $\mathcal{I} = \{I_1, \ldots, I_m\}$ forms a data structure. Some structures may be infeasible since partitions P_i or indices I_i assume specific input data properties. After construction, $\mathcal{F}(k)$ becomes the data structure's top-level index.

In the context of an LSM, the data structure is constructed from four distinct partitions defined as

$$\mathcal{P}_{LSM} = \{P_{TreeLevel}, P_{SST}, P_{Block}, P_{KV}\}.$$

Its indexing mechanism is represented by

$$\mathcal{I}_{LSM} = \{I_{LevelBloomFilter}, I_{SST}, I_{IndexBlock}, I_{KV}\}.$$

The composite function is articulated as

$$\begin{aligned}\mathcal{F}_{LSM} = & I_{KV} \circ P_{KV} \circ I_{IndexBlock} \circ P_{Block} \circ I_{SST} \circ \\ & P_{SST} \circ I_{LevelBloomFilter} \circ P_{TreeLevel}(D).\end{aligned} \quad (1)$$

$\mathcal{F}_{LSM}$ partitions data D hierarchically via tree levels ($P_{TreeLevel}$), using a level bloom filter ($I_{LevelBloomFilter}$) that marks key presence at each level [16]. Each level stores data in separate SST files (P_{SST}) with a linear-search index (I_{SST}) tracking key ranges. Each SST sorts data and splits it into blocks (P_{Block}) with key-value pairs (P_{KV}). Each SST's metadata has an index block with entries ($I_{IndexBlock}$) for data blocks enabling binary search, and KV pairs (I_{KV}) accessed via linear search in RocksDB. Fig. 2 shows four P_{Block} setups {a,b,c,d} that each yield different block divisions in one SST file; see §3.2 for details.

2.2 Learned Index (LI)

LIs [18, 22, 24, 33, 40**?**] use machine learning to efficiently map sorted keys to locations in databases. These indexes may use complex models like neural networks or simpler hierarchical linear models. Traditional LIs use ensemble learning and hierarchical model structuring. The index model $I(k)$ uses $I(k) = M(k) \times N$,

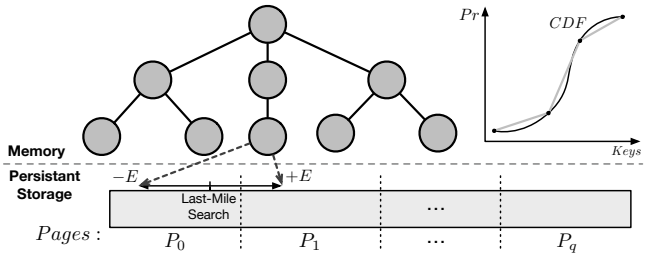


Figure 4: LI on Persistent Storage. The model last-mile search range may require loading multiple pages from the storage.

where M is the CDF estimating $p(x \leq k)$ and N is the key count, to guide queries from the root to the correct layer. Due to complex models, many LIs use piecewise linear models to approximate the CDF [43, 63]. Predict the key's position as $pos = m \times k + a$ (with error E); m, a are learned parameters, and E crucial for final target key search. These indexes are memory efficient due to lightweight parameters like intercept and slope versus conventional indexes.

2.2.1 ***LI on Persistent Storage.*** Studies [37, 44, 61, 69] show LIs fail to outperform B+tree for persistent storage KVs. These studies highlight that the design of LIs fails to leverage the characteristics of disk storage, necessitating multiple disk I/O operations during the last-mile search process as shown in Fig. 4. A similar trend is observed in LI techniques for LSMs, which also entail substantial I/O operations during their final search phase. This underscores the importance of considering I/O as a critical element in the development of LI systems.

2.2.2 ***LI on Strings.*** Most LI research focuses on fixed-size keys, with few studies on variable-length strings [54, 59, 66]. The Radix String Spline (RSS) [54] uses a Trie-based method. In RSS, each Trie node processes 8- or 16-byte keys and builds an RS model for them. Memory architecture and language usually limit integer size. In 64-bit C++, the largest built-in integer is `__uint64_t`. Some compilers offer 128-bit integers (e.g. `__uint128_t`), enabling efficient conversion of Trie nodes from strings to integers. The RS model gives monotonic CDF predictions within a set error limit. If a key exceeds the error limit, RSS adds it to a redirector map and creates a new child node. RSS compares 8- or 16-byte key segments, saving costs over full-key checks, but last-mile search is still expensive. A study [66] shows RSS spends over 70% of its time on last-mile search, suggesting major room for optimizing LIs with string keys.

2.3 Key-Value Length in LSMs

LSM storage engines are commonly used in various applications. Thus, they must handle keys and values of any type or length. The study by Cao et al. [9] examines the use of RocksDB in three different use cases at *Meta* (see Table 1): **UDB** (storage engine of a SQL database), **ZippyDB** (storage engine of a distributed KV-store), and **UP2X** (persistent storage of an AI/ML service). The study presented the average (AVG) and standard deviation (SD) of the KV length in these applications. Previous LSM LI solutions [15, 45, 58, 60] are limited to fixed key sizes, rendering them unsuitable for these applications.

2.4 Lexicographic Optimization

Lexicographic optimization [7, 12] is a multiobjective optimization approach that prioritizes objectives sequentially based on their importance. The method first optimizes the highest-priority objective,

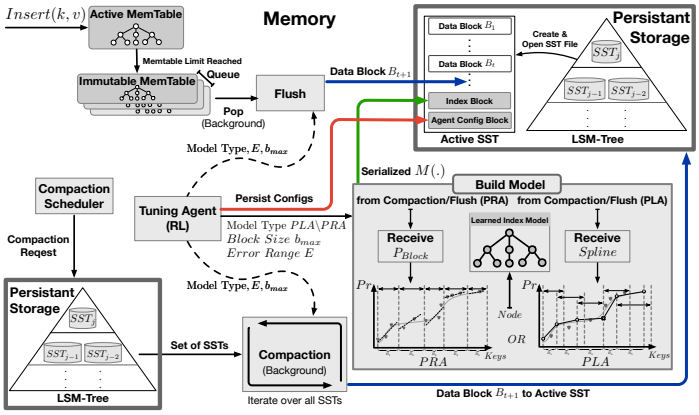


Figure 5: DobLIX Architecture in Write Flow.

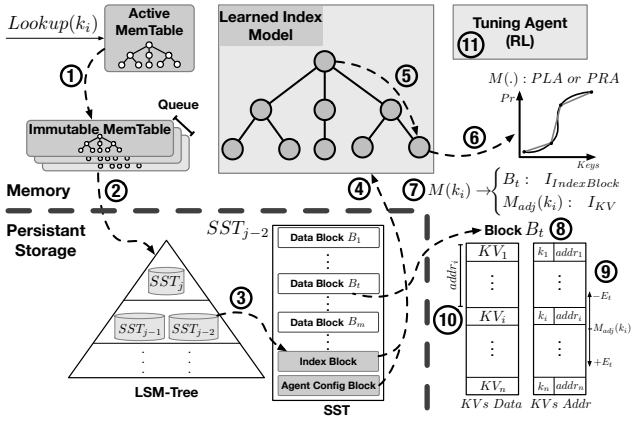


Figure 6: DobLIX Architecture in Read Flow.

Table 1: AVG and SD of key-values (in bytes) on RocksDB use cases on UDB, ZippyDB, and UP2X.

Application	Key		Value	
	AVG	SD	AVG	SD
UDB	27.1	2.6	126.7	22.1
ZippyDB	47.9	3.7	42.9	26.1
UP2X	10.4	1.4	46.8	11.6

then optimizes subsequent objectives within the progressively constrained feasible region. The optimization process follows a strict hierarchy: first, optimize f_1 to find its optimum f_1^*. Next, optimize f_2 subject to $f_1(x) = f_1^*$, and continue similarly for subsequent objectives. This guarantees that higher-priority objectives remain uncompromised by lower-priority ones.

Example 2.1. Consider a resource allocation problem with three objectives ranked by importance: (1) minimize cost, (2) maximize quality, and (3) minimize delivery time. Using lexicographic optimization, we first find the solution with the lowest cost. Among these lowest-cost solutions, we select the one with the highest quality. If there are still ties, we choose the solution with the shortest delivery time. This method is useful when objectives must be addressed in a strict order of priority. In our method, I/O has more priority over index lookup performance.

3 DOBLIX DESIGN

In this section, we describe how DobLIX is designed to speed up lookup queries. We begin by outlining DobLIX's architecture and main ideas (§3.1, §3.2). To meet dual objectives, DobLIX uses two LI methods: The Piecewise Linear Approximation (PLA) method modifies spline definition rules to consider both objectives, while Piecewise Regression Approximation (PRA) improves performance by effectively managing modeling errors (§ 3.3). DobLIX incorporates a string-compatible LI solution capable of handling variable-size KVs. To optimize the last-mile search process, it transfers model knowledge to narrow the search range and simplifies key comparisons by focusing solely on a limited part of the key bits decodable as an integer (§3.5). Furthermore, it uses an RL agent to adjust parameters like max error and block size, and to choose between PLA and PRA (§3.6).

3.1 Overall Architecture

Fig. 6 and 5 outlines the general architecture of DobLIX. This solution concentrates on learning the index at the SST level in detail. SSTs are preferred for LIs because their sorted and immutable KV structure eliminates the need for updates during their lifespan. Upon the formation of each SST, DobLIX trains an LI model based on its KVs. This model is crafted to accurately pinpoint the target block for all KVs within a specified error margin while ensuring that block sizes remain within the designated maximum limit. Consequently, DobLIX has an LI model and block partitioning, as illustrated in Fig. 2d. Subsequently, DobLIX serializes the LI model and deposits it in the index block within the SST metadata.

Write Process. Fig. 5 shows that SST creation and processing occur in the background, triggered either by a Flush to clear in-memory data or by Compaction scheduled for SST file maintenance in the LSM tree. Insert queries typically initiate the Flush process in the background. However, if the MemTable and queue are full, users may experience direct delays, as analyzed in our tail latency study (§4.2.2). Flush/Compaction receives the latest parameters—maximum error E and block size b_{max}—from the Tuning Agent for the current active SST and persists them in the Agent Config Block within the active SST for use during the lookup phase. Depending on the Model Type, PRA uses only b_{max} to create block partitioning P_{Block} and sends it to the Build Model module. If PLA is selected, it constructs block partitioning *Spline* by *Spline* and sends them to the module. The Build Model module inserts these signals to an optimized log-structured model as nodes for the lookup process described in §3.5, and finally stores the serialized model in the active SST Index Block.

Lookup Process. In Fig. 6, the stages involved in a DobLIX lookup query are outlined. ① Initially, it inspects the current MemTables; if the desired key is absent there, it looks through the unalterable MemTables. ② It then scrutinizes various levels of LSMs[17] and ③ loads the SST that may cover the target key within its range. ④ DobLIX loads the LI model from the SST metadata into memory. ⑤ It performs a search within its Trie tree to locate the node that houses the ultimate LI (§2.2.2) and ⑥ uses the trained CDF model within that node to ⑦ determine the **exact block number** that contains the key ($I_{IndexBlock}$) and **narrows down the search scope** for the last-mile search in the block using the LI model(I_{KV}).

⑧ Following this, DobLIX loads the block from storage in memory and ⑨ executes the final search within the specified range in *KVs Adrr* stored in the blocks' metadata to find the exact offset of the target KV pair in the *KVs Data* (§3.5), and ⑩ employs the retrieved address on the *KVs Data* to locate the actual KV. ⑪ Finally, DobLIX measures the *latency* of the current lookup query alongside the *index size*, incorporating these measurements as feedback to refine the tuning agent (§3.6).

3.2 Concept Overview

The management of LSM data involves data partitioning and indexing phases (§2.1), and as we established earlier, any optimization strategy, especially those involving LIs, should improve overall performance. As depicted in Fig. 2, block partitioning is intertwined with block indexing ($P_{Block} \not\perp I_{IndexBlock}$). Therefore, optimizing $I_{IndexBlock}$ requires the consideration of P_{Block}. In contrast, previous designs illustrated in Fig. 2{a,b,c} from earlier research have significant data access expenses due to the independence between the LI model and the data partitioning component.

As described in §2.2, within the traditional framework of LI modeling, $I(.)$ represents the result of approximating keys indexes drawn from an unknown distribution $\mathcal{D}_{keys}$ through practical optimization. Therefore, the classical design of the LI does not consider data access and the result coordinated by all data; however, in LSMs only a limited number of SST blocks reside in memory. In addition, the primary optimization objective is to minimize the error within the hypothesis spaces chosen, regardless of any secondary objectives.

DobLIX aims to redefine LI models by integrating efficient block-based data access as a key objective. Enhances indexing performance by ensuring that the trained model accurately maps queries to the correct block, enabling the retrieval of only a single block while adhering to the optimal block size. A critical aspect of DobLIX is the relationship between the index approximation ($I_{IndexBlock}$) and block partitioning (P_{Block}), where a one-to-one correspondence is established between the index approximation and the segments within P_{Block}. This allows DobLIX to apply LI models that partition the key domain into segments, ensuring each segment corresponds to a specific block. The system performs a binary search on an array of offsets ($I_{IndexBlock}$) to find the start of each segment (i.e. block). Within each segment, it uses an index approximation (I_{KV}) to efficiently locate the keys. Since the trained index is based on the entire data in SST, the index model used for each block requires adjustment (§3.5). This approach optimizes both block access and key retrieval, providing efficient indexing.

To achieve this, we introduce a dual-objective optimization approach for two distinct LI methodologies. The first method is based on PLA modeling [33], while the second method employs PRA, based on the recursive model index [35]. We delve into these methods in §3.3.2 and §3.3.3.

3.3 LI Approximation Methods

In this section, we present our LI algorithms that focus on dual-objective optimization to train the index model. Considering the importance of I/O performance in indexing, these algorithms are designed to partition the KV space based on their sizes and progressively systematically construct the index. Typically, we approximate the index for each segment using linear models. When it comes to searching for specific points during lookup processes, we employ a binary search on the points derived from the piecewise approximation to precisely pinpoint the required location, which is referred to as last-mile search. In the following, we elaborate on these algorithms in detail.

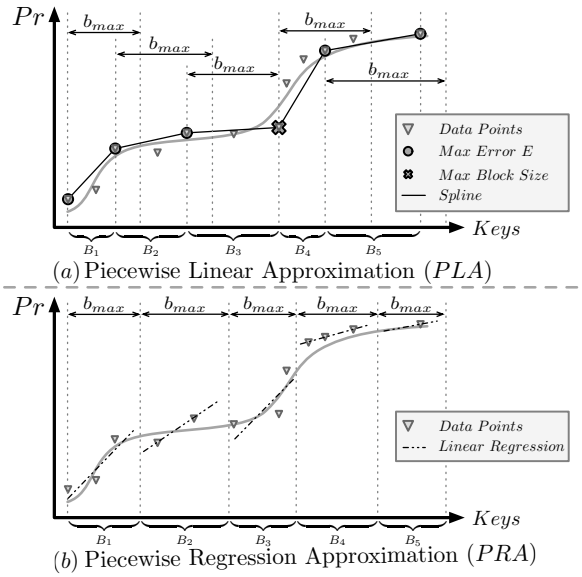


Figure 7: LI Models. B_is represent the actual blocks added to SSTs.

3.3.1 ***Dual-Objective Optimization.*** Although various methods exist to tackle multi-objective optimization problems, we aim to prioritize the data access parameter more heavily than index lookup. Thus, we always finalize a block when its size exceeds the maximum block size b_{max} by incorporating an additional pair of KVs. This ensures that the block sizes remain below b_{max}, even if the approximation error E has not yet been achieved. In another words, our optimization to uncompromise the data access over the index lookup can be defined as: $\forall B_i \in P_{Block} : |B_i| \leq b_{max}$. Note that both the configuration values b_{max} and E are given by the *Tuning Agent* (§3.6).

3.3.2 ***Piecewise Linear Approximation (PLA).*** In this method, the key space is divided into blocks B_i using a linear approximation (spline), each block containing keys that share a common prefix. For each block B_i, a spline estimate of the positions is made, defined as $M_i(x) = a_i + m_i(x - x_i)$, where a_i and m_i are coefficients derived from spline points and x_i is the initial key in block B_i. The binary search is then used around $M_i(k)$ to identify the precise index $I(k)$. For each block B_i, the next key is added to a new block (B_{i+1}) of the SST if (1) the approximation error $M_i(x)$ reaches the maximum threshold E, or (2) the condition $|B_i| \geq b_{max}$ indicates that including the new pair of KV would exceed the maximum block size allowed. This approximation process is illustrated in Fig. 7a. If the size of an added point exceeds b_{max}, a new block is created to maintain the optimal I/O (data access) performance of the previous block. This mechanism results in a new set of spline points (the cross are spline points in Fig. 7a), introducing new blocks when the secondary optimization criterion is met, in addition to the standard blocks formed by reaching the maximum approximation error E. Alg. 1 showcases this approach, where $APE(line, set)$ calculates the maximum distance the points in set can have from the given line

Algorithm 1: Dual-objective PLA

Input: Set of KVs D
Output: Radix Points $\mathcal{R}$

$\mathcal{R} \leftarrow [\,]$, $index \leftarrow 0$, offset $\leftarrow 0$
$E, b_{max} \leftarrow TuningAgent()$
$B_{curr} \leftarrow [(k_0, v_0)]$
while $(k, v) \in D$ **do**
  if $|B_{curr}| > b_{max}$ **then**
    $\mathcal{R} \leftarrow \mathcal{R} + [B_{curr}.last]$
    $B_{curr} \leftarrow [(k, index)]$
  if $|B_{curr}| > 1 \ \wedge \ APE(Line(B_{curr}.first, (k, index)), B_{curr}) \geq E$ **then**
    $\mathcal{R} \leftarrow \mathcal{R} + [(B_{curr}.\text{last}, \text{offset})]$
    offset $\leftarrow$ offset $+ |B_{curr}|$
    $B_{curr} \leftarrow [(k, index)]$
  else
    $B_{curr} \leftarrow B_{curr} + [(k, index)]$
  $index \leftarrow index + 1$
$\mathcal{R} \leftarrow \mathcal{R} + [(B_{curr}.\text{last}, \text{offset})]$

Algorithm 2: Partition $P_b(.)$

Input: Set of KVs D, Maximum Block Size b
Output: Partition $\mathcal{P}$

$t, \mathcal{P} \leftarrow [\,]$ & offset $\leftarrow 0$
while $KV \in D$ **do**
  if $|t| + |KV| > b$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} + [(t, \text{offset})]$
    offset $\leftarrow$ offset $+ |t|$ & $t \leftarrow [\,]$
  $t \leftarrow t + [KV]$
$\mathcal{P} \leftarrow \mathcal{P} + [(t, \text{offset})]$

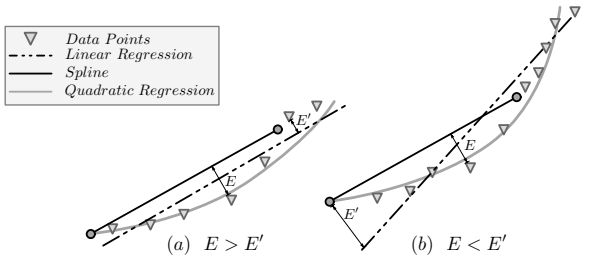


Figure 8: Comparison of PLA and PRA under different data distributions. (a) $E' < E$, indicating PRA performs better. (b) $E < E'$, indicating PLA performs better.

line. In this context, a_i is defined as $\mathcal{R}[i][0][1]$, x_i corresponds to $\mathcal{R}[i][0][0]$, and m_i is calculated as $\frac{\mathcal{R}[i+1][0][1]-\mathcal{R}[i][0][1]}{\mathcal{R}[i+1][0][0]-\mathcal{R}[i][0][0]}$, while the term offset$_i$ refers to $\mathcal{R}[i][1]$.

3.3.3 ***Piecewise Regression Approximation (PRA).*** This approach involves initially dividing the key domain into segments of size b_{max} and then approximating each segment linearly. The maximum approximation error for each segment, E', is stored and utilized during the last-mile search phase (refer to Fig. 7b). Start scanning the KVs from the beginning; if incorporating the new KV into the existing segment exceeds b_{max}, start a new segment. Alg. 2 performs this task in one pass, maintaining the integrity of each KV pair.

After partitioning the data (P_{Block}) using the optimal block size b_{max}, as outlined in Alg. 3, a new model can be constructed for each partition through a linear approximation. The boundary points are retained for search tasks to determine the appropriate model for lookup queries, and E' is stored in $\mathcal{E}$, respectively.

Algorithm 3: Dual-objective PRA

Input: Set of KVs D
Output: Radix Points $\mathcal{R}$
Model Sets $\mathcal{M}$
Maximum Segment Errors $\mathcal{E}$

$\mathcal{R}, \mathcal{M}, \mathcal{E} \leftarrow [\,]$
$b_{max} \leftarrow TuningAgent()$
$\mathcal{P} \leftarrow P_{b_{max}}(D)$ ▷ from Alg. 2
while $Partition\ par \in \mathcal{P}$ **do**
  if $|par| > 1$ **then**
    $M \leftarrow LinearRegression(par)$
    $\mathcal{M} \leftarrow \mathcal{M} + [M]$
    $\mathcal{E} \leftarrow \mathcal{E} + [APE(M, par)]$
  $\mathcal{R} \leftarrow \mathcal{R} + [par.frist]$
$\mathcal{R} \leftarrow \mathcal{R} + [par.last]$

3.3.4 ***Comparing PRA and PLA.*** In the context of building block approximations, both the PLA and the PRA rely on the scanning of data, resulting in a linear time complexity of $O(N)$, where N is the number of data points. Each method utilizes closed-form formulas with a time complexity of $O(N)$ for computations within blocks: PLA determines maximum distances, $APE(.,.)$, to construct piecewise linear segments, while PRA computes two-dimensional regression, $LinearRegression(.)$, formulas within blocks.

With respect to space complexity, PLA requires one point per spline segment to be stored since the endpoint of one line serves as the beginning of the next. The distribution of data points influences PLA memory needs; for instance, with a uniform distribution, all data might fit within a single spline (provided its size is smaller than b_{max}), thus minimizing memory usage. In contrast, PRA must store both a point and a slope for each regression line, roughly doubling the memory requirement compared to PLA. The complexity of the number of regressions is $O(\frac{N}{b_{max}})$. Consequently, the memory comparison between PLA and PRA depends on the characteristics of the data: PLA may involve fewer blocks and needs just one point per block (linear segment), whereas PRA has to retain two parameters per block.

When comparing the behavior of PRA and PLA during lookups (⑦ in Fig. 6), determining which method is superior is challenging, often leading to the interchangeable use of algorithms; in particular, we consider two scenarios that contrast PLA and PRA, as illustrated in Fig. 8, noting that both scenario (a) and scenario (b) can occur depending on the sizes of the KV pairs and the distribution of the keys. In the PRA model, the block is written in persistent storage once its size reaches the maximum block size b_{max}, whereas the PLA operates under two conditions for flushing: when the approximation error exceeds the error limit E, or when the block size reaches b_{max} (the same maximum block size used in PRA). Consequently, depending on the arrangement and distribution of KV, the spline achievable with PLA can potentially lead to a maximum error E that may be higher or lower than the maximum error E' in PRA. As shown in Fig. 8a, PRA can result in $E > E'$, indicating a more accurate approximation than PLA. This directly affects the scope of the last-mile search and the overall efficiency of each algorithm.

This implies that the behavior of KVs beyond the block boundary determined by the maximum error E plays a crucial role. If data points outside the block constrained by E align with the regression trend of the points within the block, the approximation remains accurate. However, there might be situations where data points

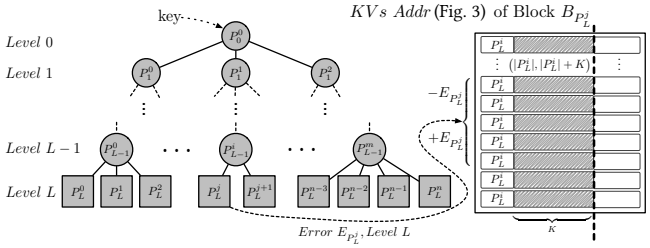


Figure 9: Last-Mile Search Optimization Flow

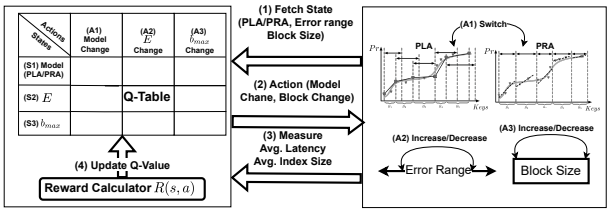


Figure 10: RL Tuning Agent Overview.

outside the block behave significantly differently from those within the block. In such cases, as clearly shown in Fig. 8b, this could lead to a less accurate linear regression in PRA, resulting in $E < E'$. This means that PRA has a lower performance than PLA in such scenarios. In some rare cases, all elements in $\mathcal{E}$ (Alg. 3), denoted as E' for each segment, are equivalent to E. In such scenarios, both algorithms exhibit identical last-mile search performance.

3.4 Serialize and Deserialize Models

After training, the LI models are serialized and stored in a metadata block called the Index Block (see Fig. 3). This process involves traversing the model tree structure via depth-first search (DFS), serializing each node byte-by-byte by writing the spline points data. During deserialization, the model reconstructs the tree by determining the type of each node (leaf or internal) and populating the corresponding data structure.

The learned indices generally outperform the binary search in terms of time complexity. Therefore, the size of the set of stored nodes for model approximation is smaller than $O(\log N)$ when reading, while writing the entire dataset takes $O(N)$. As a result, the serialized model size is asymptotically negligible (see §4.4). In practice, RocksDB writes occur in the background flush and compaction process, and since the model is deserialized only once, overall read performance is significantly improved (see §4.2.1).

3.5 Last-Mile Search Optimization

The final phase of the search process involves the last-mile search of the recovered block from storage (step ⑨ in Fig. 6). As illustrated in Fig. 1c, this step accounts for more than 40% of the indexing latency in RocksDB. Fig. 9 shows how DobLIX optimizes its performance by restoring data from the LI model computation trajectory (step ⑤ in Fig. 6) to improve the last-mile search process. Initially, the target key is searched within the string LI structure, as explained in §2.2. The LI model in DobLIX subsequently provides: *(1) Block* $b_{P_L^j}$: The block containing the target KV pair. *(2)* $M(.)$: The LI estimation of the target KV pair index in *KVs Addr* (Fig. 6). *(3) Level L*: The level at which the target key was found in the LI model. *(4) Error* $E_{P_L^j}$: The maximum range required to search for the target KV pair.

Optimizing Search Range. As described in §3.2, DobLIX necessitates a coordinate transformation to adapt the model output ($M(.)$) for indexing on the retrieved block ($B_{P_L^j}$). This is achieved by deducting the count of keys in the previous blocks (maintained as the parameter "offset" in the metadata of the block introduced in Algs. 1& 2):

$$I_{KV}(.) = M_{adj}(.) = M(.) - B_{P_L^j}.\text{offset}$$

To better illustrate the adjustment, we consider SST_j in Fig. 2, in which the model is trained on the whole SST indexes. However, using the above adjustment, the coordination of model output for our solution (i.e., Fig. 2d) is transformed to the retrieved block B_2. Subsequently, DobLIX performs a binary search within the specified model error range $E_{P_L^j}$ on $M_{adj}(.)$. This error range can be less than the maximum error E if a spline in the PLA method reaches the maximum block size b_{max}. In such cases, DobLIX calculates the spline error and incorporates it into the model, transmitting this information to the last-mile search process to streamline the number of key comparisons.

Optimizing String Comparison. DobLIX further optimizes the comparisons by avoiding full key comparisons. DobLIX provides the node level where the key is found in its string-LI structure. This level in the tree indicates the common prefix string, P_L^j, in the key (Fig. 9). Then DobLIX can ignore this common prefix, as all keys within the retrieved block share the same prefix. Additionally, DobLIX only compares string keys up to their K bytes, ensuring that the key can be identified by comparing only the K byte following the prefix within the error range. Given that K generally consists of 8 or 16 bytes as explained in §2.2.2 for shifts from demanding string comparisons to numerical comparisons. We perform an ablation study to analyze the impact of these optimization on DobLIX performance in §4.8.

3.6 Tuning Agent

DobLIX employs Q-learning [62], a lightweight Reinforcement Learning (RL) algorithm [48], to dynamically select model and fine-tune model and data access parameters. Fig. 10 shows the overview of the RL tuning Agent. DobLIX determines three key parameters in the creation of SSTs and the training model: *(1)* The choice between using the PLA vs. the PRA. *(2)* The maximum error of the LI in the PLA method (E), and *(3)* The maximum size of a block (b_{max}).

The state space for the Q-learning agent comprises the index model–Current Model which is 2 states between PLA and PRA, Error values for PLA ranging from 32 to 256, doubling at each step, resulting in 4 distinct values–and the block size–$4KB$ to $32KB$, doubling at each step, yielding 4 distinct values, as recommended by RocksDB. The action space consists of: switching between PLA/PRA models, and incrementing, decrementing b_{max}, or E. We also forbid the actions to change E when the model state is PRA as it does not consists of different E.

The reward function is defined below to consider both system latency and index storage:

$$R(s,a) = -\nu.Norm(AVG(latency)) - (1-\nu).Norm(AVG(index\ size))$$

This reward function is calculated as the negative sum of the normalized average end-to-end latency and the normalized average of SSTs index size. We use sigmoid normalization to ensure both latency and index size contribute to the reward on a comparable scale. The weighting parameter ν controls the relative importance of

Algorithm 4: System Tuning Agent

```
Q ← initializeQTable()
α, γ, ϵ, ν ← initializeVariables()
while do
    Fetch state s_{t−1}
    L_t, I_t ← fetchAverageLatencyAndSSTIndexSize()
    R_t ← calculateReward(L_t, I_t)
    Observe state s_t
    a' ← arg max_{a∈Actions} Q(s_t, a)
    Q(s_{t−1}, a_{t−1}) ← (1 − α)Q(s_{t−1}, a_{t−1}) + α(R_t + γQ(s_t, a'))
    A_t ← getAvailableActions(s_t)
    if generateRandNumber() < ϵ then
        a_t ← getRandomAction(A_t)
    else
        a_t ← arg max_{a∈A_t} Q(s_t, a)
    tuneSystem(a_t)
    s_t ← s_{t+1}
    ϵ ← updateEpsilon(ϵ)
```

latency and index size in determining the overall reward (evaluated in §4.7).

Alg. 4 outlines the agent tuning and execution process. The learning procedure begins by configuring the RL hyperparameters: α denotes the learning rate, while γ signifies the discount factor that balances immediate and future rewards. An epsilon decay strategy is used by initially setting a high exploration rate (ϵ) to investigate various actions, gradually lowering this value in successive iterations to exploit the optimal actions. At every 20 SSTs creation, the following steps occur: First, the agent obtains the average latency of the reads that have reached this level, as well as the index size of the previously created SST. Then, it measures the reward and updates it for the chosen action in the previous state. The agent then gets the available actions at that state. Then, depending on the value of ϵ, the agent explores a new action or chooses the best action from the Q-table. In addition, we implement a reset mechanism for the RL agent that reverts ϵ to its initial value if the distribution of incoming KVs undergoes a significant change. This ensures optimal exploration of the space under the new conditions.

4 EVALUATIONS

This section presents the results of the DobLIX evaluation, emphasizing its advantages over current leading methods. We design the evaluations to answer the following questions: 1) What performance advantages does DobLIX offer? (§4.2) 2) How does the length of key-value pairs affect DobLIX and other baseline systems? (§4.3) 3) What is the Storage and Write amplification of DobLIX? (§4.4) 4) How does the RL agent adjust its underlying parameters? (§4.7)

4.1 Experimental Setup

4.1.1 ***Environment.*** We implemented DobLIX in C++17 with RocksDB 8.1.1, compiled using GCC 9.4.0. We ran evaluations on Ubuntu 20.04 with a 64 core AMD Ryzen Pro 5995WX (2.7GHz) and 256GB DDR4 RAM. The storage device used was a SAMSUNG 980 Pro 2TB M.2 NVMe SSD. This NVMe SSD offers fundamental read and write performance measures as follows: sequential read speeds of 7,000 MB/s, random read speeds of 1,237K IOPS, sequential write speeds of 5,000 MB/s, and random write speeds of 172.5K IOPS. A single experiment was performed on a Seagate BarraCuda 4TB SATA SSD. We allocate 4 background threads to RocksDB and set the `max_background_compactions` and `max_background_flushes` options to 4, as prior studies [15, 45, 53] and RocksDB best practices suggest this as a balanced trade-off between compaction efficiency and I/O contention, ensuring stable read performance. Each compaction and flush task runs in a separate thread, as RocksDB dynamically assigns work within the allocated thread pools rather than using a fixed number of threads per worker.

4.1.2 ***Datasets.*** We utilize four real-world datasets sourced from the Search on Sorted Data Benchmark (SOSD)[47] along with two synthetic datasets for evaluating DobLIX. These datasets from prior studies [15, 18, 45] each hold 64M key-value pairs. The size of the key and the value are set according to the experiments used in our baselines ([15, 45]) as 8 byte and 64 byte. We also conduct experiments with other fixed key and value sizes, and also with varying key and value sizes to reflect real-world applications. The following sections provide specific information about each dataset. *WIKI [4].* Edit timestamps for Wikipedia articles. *AMZN*: Popularity of book sales collected from Amazon. *FB [2]*: An upsampled version of a Facebook user ID dataset. *OSM [3]*: Uniformly sampled locations as Google CellIds. *LOGN*: This synthetic dataset is generated from a lognormal distribution with parameters $\mu = 0$ and $\sigma = 2$, multiplied by 10^9 and rounded down to the nearest integer. UNI: Synthetic data sampled uniformly between 0 and 10^{16}.

4.1.3 ***Workloads.*** We evaluate DobLIX with four different workloads, each consisting of 10 million operations. *Read-Only (**RO**):* Focuses solely on read operations. *Read-Heavy (**RH**):* Emphasizes reads (90%) with a smaller proportion of inserts (10%). *Balanced (**BA**):* Involves an equal split between reads and inserts (50% each). Write-Heavy (**WH**): Emphasizes insert operations (90%) over read operations (10%). We also use YCSB [13] real benchmarks to evaluate DobLIX. In all workloads, the search key is selected randomly from the existing set of keys in the index with a Zipfian distribution [14], unless a different request distribution is specified.

4.1.4 ***Baselines.*** We use these schemes as comparison baselines. ***RocksDB*** [20]: An embedded high-performance KV-store used as the storage engine. We utilize the default indexing mechanism of RocksDB. ***Bourbon*** [15]: A KV store with an LI that accelerates lookups by understanding the distribution of keys. Based on WiscKey [46], a LevelDB [1] variant, Bourbon retains fixed block sizes, but might need to load several blocks (See Fig. 2A). The Bourbon source code is publicly accessible [5]. We incorporate Bourbon indexing method into our RocksDB configuration. ***TridentKV*** [45]: an LI variant of RocksDB that aims to retrieve KV pairs by loading only one data block. TridentKV modifies the size of the data blocks, potentially significantly increasing the size of the block, which can affect the lookup performance when loading a large block (see Fig. 2C). TridentKV code is available as open-source [6] and is built on top of RocksDB. We adopt their implementation in our evaluations.

4.1.5 ***Metrics.*** We use the following metrics to evaluate DobLIX and all other baselines. ***Throughput:*** Average rate of operations per second. ***Latency:*** Average end-to-end latency across all operations, excluding the slowest 1% (99^{th} percentile). ***Tail Latency:*** The average latency of the 5% slowest operations. ***Index size:*** LI and index block size. ***Compaction time:*** The average duration taken to create SSTs in the compaction process.

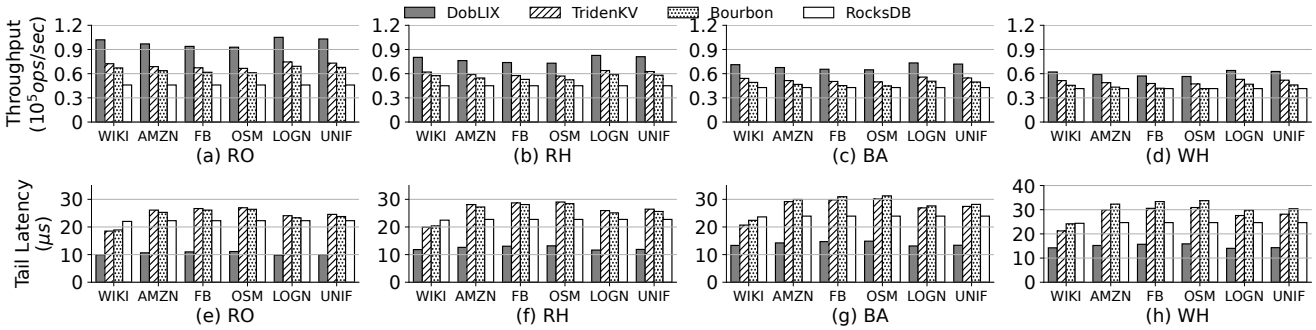


Figure 11: Throughput Comparison (upper Figure) and Tail Latency Comparison (lower Figure).

4.1.6 ***Parameters.*** By default, all methods adhere to the default configuration settings of RocksDB. The default configurations of Bourbon and TridentKV are also employed. For certain parameters such as the chosen learning method (either PLA or PRA), the maximum error bound, and the maximum block size, the tuning agent is responsible for making decisions. Initially, we use a 1% sample from the datasets to train this agent, after which we incorporate the agent into the system. In the case of the PLA method, we use the same setup as RSS [54] to configure a dynamic radix table with 18 bits for the first level, 12 bits for the second level and 8 bits for the third level and beyond.

Regarding the hyperparameters of the RL agent α and γ (see §3.6), we perform a sensitivity test with values 0.2, 0.5 and 0.8. We observe that optimal results are achieved by assigning a low value of 0.2 to α and a high value of 0.8 to γ. We also set the initial value of ϵ at 0.99, with its minimum value being 0.02, allowing for a low exploration rate even after the training period (§4.7). We set ν in the reward function as 1 to optimize solely on performance.

4.2 Performance

In this section, we detail a set of experiments designed to evaluate DobLIX performance, alongside a comparison with baseline systems. The initial experiment (§4.2.1) highlights that DobLIX improves throughput by up to 1.41×, 1.52×, and 2.21× relative to TridentKV, Bourbon, and RocksDB, respectively. In the second experiment (§4.2.2), DobLIX achieves up to 2.67× better throughput in terms of tail latency. The third experiment (§4.2.3) examines the optimization of DobLIX lookup latency components. The fourth experiment (§4.2.4) showcases DobLIX enhanced performance on the realistic YCSB benchmarks. Finally, the last experiment (§4.2.5) shows DobLIX performance across varying request distributions.

4.2.1 ***Throughput.*** This section displays the throughput when subjected to a specific workload on different datasets.

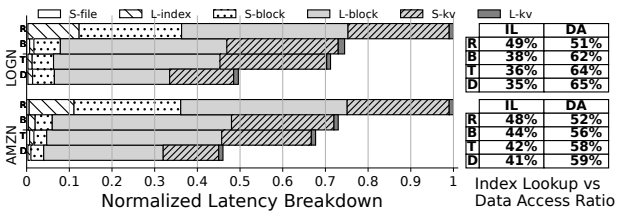
RO Workload illustrated in Fig. 11a, where DobLIX demonstrates the highest throughput, achieving a maximum of 105K ops/sec using the LOGN dataset. Compared to TridentKV, Bourbon, and RocksDB, DobLIX increases the average throughput by 1.41×, 1.57×, and 2.21×, respectively. **RH Workload** throughput depicted in Fig. 11b, where DobLIX consistently achieves the highest throughput across all datasets, peaking at 82K ops/sec. Compared to TridentKV, Bourbon, and RocksDB, DobLIX increases the average throughput by 1.27×, 1.36×, and 1.22×, respectively. Despite the marginal reduction in throughput from write operations due to the slight training overhead for new SSTs, DobLIX and other LIs still maintain superior throughput compared to RocksDB indexing approach. **BA Workload** in Fig. 11c illustrates that DobLIX consistently surpasses other systems in throughput, with an average enhancement of 1.29×, 1.27×, and 1.31× over TridentKV, Bourbon, and RocksDB, respectively. **WH Workload** in Fig. 11d shows that DobLIX achieves the highest throughput, reaching 53K ops/sec. When compared to TridentKV, Bourbon, and RocksDB, DobLIX enhances the average throughput by 1.16×, 1.14×, and 1.04×, respectively. DobLIX's slight advantage over RocksDB is explained in §3.4.

Takeaway: DobLIX enhances read throughput through co-optimized data access and index lookups. Compared to Bourbon (that loads multiple blocks per lookup) and TridentKV (features excessively large block sizes), DobLIX minimizes read amplification by ensuring single-block reads with optimal block sizes (less than b_{max}) per query, while maintaining comparable write amplification to RocksDB. DobLIX also achieves write superior to Bourbon and TridentKV due to its smaller index size (see §4.4).

4.2.2 ***Tail Latency*** Analyzing tail latencies of different methods is vital because it defines the performance expectations for users and applications regarding their back-end key-value storage engine under the most challenging scenarios.

RO Workload. Fig. 11e shows the tail latency results for RO workloads, indicating that DobLIX exhibits the lowest tail latency. Notably, DobLIX improvement in tail latency surpasses the improvements observed in throughput. Compared to TridentKV, Bourbon, and RocksDB, DobLIX achieves improvements of 1.85×, 1.89×, and 2.13×, respectively. This difference is due to the high read amplification and index lookup inefficiency of other methods, explained in §4.2.

Workloads with Write. Fig. 11{f,g,h} present the tail latency results for RH, BA, and WH workloads. These figures illustrate that DobLIX achieves the lowest tail latency across all workload types. In terms of tail latency, RocksDB indexing outperforms both Bourbon and TridentKV, which employ LIing without optimizing for the data access phase. TridentKV exhibits the highest tail latency among these workloads, showing up to 2.67× greater tail latency than DobLIX due to its large blocks, which negatively affect both read and write operations. Bourbon also experiences elevated tail latency in write-intensive workloads due to its dependence on a garbage collection mechanism[15], which degrades its performance during this period.



	IL	DA
R	49%	51%
B	38%	62%
T	36%	64%
D	35%	65%

	IL	DA
R	48%	52%
B	44%	56%
T	42%	58%
D	41%	59%

Figure 12: Latency Breakdown (Normalized) of DobLIX ('D'), TridentKV('T'), Bourbon('B'), and RocksDB ('R'). 'L' and 'S', on the legend, stand for load and seek, respectively. Tables show the relative time in Index lookup ('IL') vs. Data acess ('DA') for all methods.

Takeaway: The result shows that DobLIX has lower tail latency while competing with its opponents in the ordinary time span.

4.2.3 ***Lookup Query Latency Breakdown*** In Fig. 12, the breakdown of the latencies of systems in RH workloads is presented using the AMZN and LOGN datasets. Latencies are normalized by RocksDB latency. DobLIX achieves average latency improvements of up to 23%, 27%, and 56% compared to TridentKV, Bourbon, and RocksDB. Certain components, such as the Seek file or I_{SST} (S-file) and Load KV (L-KV), show minimal changes, as these steps are consistent across all methods. The load index block (L-index) sees an improvement of 10.35% on average compared to RocksDB, attributed to DobLIX LI methods. DobLIX also improves the lookup of blocks or $I_{IndexBlock}$ (S-block) by up to 22% compared to RocksDB. Significant enhancements compared to Bourbon and TridentKV are seen in block loading (L-block) and locating the target KV within the block or I_{KV} (S-KV). DobLIX multi-objective optimization strategy targets block size and precise block retrieval, resulting in a decrease in average block loading time by as much as 12.2% compared to other methods. In terms of the KV last-mile search (S-KV), DobLIX demonstrates an average reduction of 11.1% due to two optimizations: the elimination of key prefixes and the narrowing of the error bound range (as discussed in §3.5).

Index Lookup vs. Data Access. Tables on Fig. 12 presents the relative time breakdown between index lookup and data access across different methods. The results demonstrate that LI methods require significantly less time for indexing compared to RocksDB's native implementation, with improvements reaching up to 9% for the LOGN dataset. Also, DobLIX improvement over LI makes it achieve the lowest index lookup time among LI methods.

4.2.4 ***YCSB Macrobenchmarks*** Fig. 13a illustrates the throughput performance of DobLIX and the three baseline systems on various YCSB workloads. In particular, DobLIX consistently achieves the highest throughput on the six workloads. In the read-heavy

Figure 13: Throughput Comparison on YCSB macrobenchmarks and various distribution workloads.

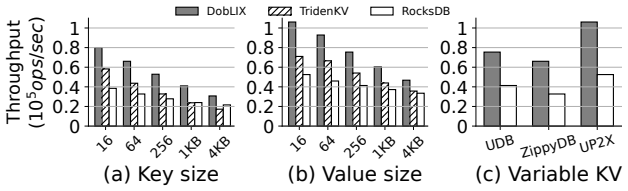


Figure 14: Impact of various key-value sizes.

YCSB{B,C,D} workloads, DobLIX demonstrates superior throughput of $481K$, $607K$, and $916K$ ops/sec, respectively. On average, DobLIX increases throughput by 1.32×, 1.42×, and 2.02× compared to TridentKV, Bourbon, and RocksDB, respectively. For balanced workloads YCSB{A,F}, DobLIX reaches throughput levels of $320K$ and $264K$ ops/sec, respectively, outperforming other techniques with an average increase in throughput of 56%. In the YCSB-E scan-focused benchmark, DobLIX achieves the highest throughput of $80K$ operations per second. However, its gains over TridentKV, Bourbon, and RocksDB are relatively modest at 9.2%, 16.7%, and 42.8%, respectively. This relative performance is because the LIs enhance point queries rather than scan operations. During the scan phase of the range queries, all techniques exhibit uniform performance.

4.2.5 ***Various Request Distributions*** Fig. 13b shows the throughput of DobLIX and the three baselines in RO workloads in the OSM dataset on six different request distributions. The figure shows that DobLIX maintains its superior read performance in the six request distributions. On average, DobLIX achieves 39.4%, 52.1%, and 78.9% higher throughput compared to TridentKV, Bourbon, and RocksDB.

4.3 Impact of Key and Value Length

In §4.2, we evaluated DobLIX performance with fixed 8-byte keys and 64-byte values. Here, we first show DobLIX performance by changing the key and value sizes while maintaining their constancy during the experiments. Then, we perform more realistic experiments in which the key and value sizes vary within each experiment (details explained in §2.3).

4.3.1 ***Fixed-sized key-values*** Fig. 14a shows how each method is affected by key size. Bourbon was excluded due to its 8-byte key limit. DobLIX delivers superior throughput on the OSM read-only workload by efficiently managing key prefixes, regardless of key size growth. However, TridentKV throughput drops drastically due to creating large block sizes by increasing the key size. The performance gap between DobLIX and RocksDB narrows with increasing size, as larger key-value loads impact both methods. Fig. 14b shows the impact of various fixed value sizes on throughput. The results show that DobLIX has the highest throughput up to a value of $1KB$. For the value size $4KB$, Bourbon works better than TridentKV as it disaggregates KVs, which is efficient for large value sizes.

4.3.2 ***Variable-sized key-values*** In this section, we conduct practical experiments utilizing variable KV sizes. We used the mean and standard deviations of the KVs of three RocksDB applications (Table 1). As none of the previously established index methods can manage variable KVs, our comparison is exclusively with RocksDB. Fig. 14c shows the experimental results for these KV sizes. It illustrates that DobLIX functions effectively with variable key-value sizes and achieves 1.32×, 1.38×, and 1.62× in key-value sizes such as UDB, ZippyDB, and UP2X.

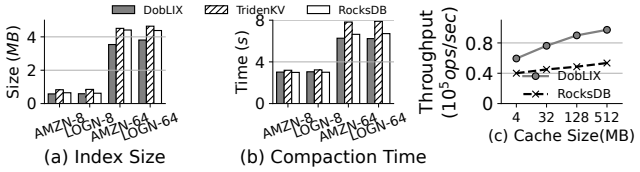


Figure 15: (a) Average index size, (b) Compaction time, (c) Effect of cache size on throughput.

4.4 Storage and Write Amplification

Fig. 15a shows the index size for the AMZN and LOGN datasets with 8- and 64-byte key sizes. We omit Bourbon as it only works with 8-byte numerical keys. The index block for each method will be written to storage beside the actual key values, hence index size has a direct relation with storage (the total space required on persistent media) and write amplification (the ratio of actual writes to storage versus logical writes requested by the system). The figure shows that in 8-byte keys, the index size of DobLIX and RocksDB has a negligible difference, while both are less than TridentKV. However, for 64-byte keys, DobLIX improves indexing by 25.9% on average. This is because DobLIX removes the longest common prefix from each cluster of keys on each node (see §3.5), and each node only keeps 8-byte numerical data. This shows that DobLIX has the lowest index size, and since it does not add any other data except the serialized LI model, it even decreases the storage and write amplification compared to RocksDB native indexing.

4.5 Compaction Time

SSTs in DobLIX and other baselines are built during background flush and compaction. [67]. Fig. 15b demonstrates the average compaction time to build SSTs on the four different workloads and datasets mentioned in §4.4. The data indicate that the DobLIX LI model does not add overhead during the SST construction compared to RocksDB. Notably, with 64-byte keys, DobLIX construction time is 5.9% shorter than RocksDB, due to the smaller DobLIX index size (Fig. 15a). Additionally, compared to TridentKV, DobLIX reduces construction time by 24.7%, attributed to TridentKV lack of prefix removal optimization in its string model. Specifically, DobLIX average SST construction time for 64-byte keys is 6.2 seconds, with model training taking 302 ms (4.8% of the total time) and serialization and metadata writing taking 237 ms (3.7% of the total time). In contrast, TridentKV model training time is 817 ms (10.4% of the total time), and serialization plus metadata writing time is 414ms (5.1% of the total time).

4.6 Effect of Cache Size

Fig. 15c demonstrates how cache size affects system throughput using the RO workload with Zipfian distribution on the AMZN dataset. While RocksDB's default configuration employs a 32MB cache, both DobLIX and RocksDB benefit from larger caches through reduced disk accesses via prefetching of data and index blocks. DobLIX achieves significant gains, outperforming RocksDB by 81% at 512MB cache size. This substantial improvement occurs because DobLIX's efficient index lookup becomes more pronounced when caching reduces data access overhead, aligning with observations by Lu et al. [45].

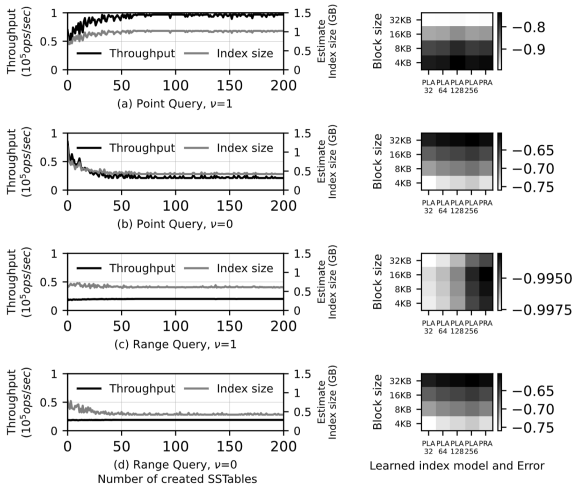


Figure 16: RL agent parameter tuning. Each row shows two plots for one specific workload and reward parameter, The right plot shows the improvement in the throughput and estimated index size during the workload. The left plot shows the reward heatmap at the end of the experiment. Darker colors have higher rewards.

4.7 Parameter Tuning

In this section, we present the results of the RL agent for parameter tuning and analyze the impact of parameters on throughput and total index size. The evaluation uses the AMZN dataset under two RH workloads: (1) point-query with a Zipfian distribution, and (2) range queries over 100 consecutive KV pairs with a uniform distribution. We also vary the ν parameter in the RL reward function (§3.6) between 0 and 1 to evaluate trade-offs between index storage footprint and throughput optimization. The left side of Fig. 16 illustrates throughput between two RL episodes, and the estimated total index size after each episode. The index size is estimated using a linear approximation based on current keys added to a total of 64 million keys in the dataset. The right side of the figure displays the RL agent Q-table rewards, highlighting the best parameter configurations for each workload.

Throughput Optimization. The throughput of point queries (Fig. 16a) increases from $45K$ to $93K$ ops/sec after 50 episodes. This improvement is achieved by tuning parameters to prioritize throughput at the cost of increasing storage size. The heatmap shows that smaller block sizes yield higher rewards because they reduce read amplification, minimizing the size of the retrieved block per query. Among the index models, PLA with a maximum error of 128 achieves the best performance. A smaller error increases the block count, while a larger error (i.e., 256) raises the cost of intra-block last-mile searches. For range queries (Fig. 16c), parameter rewards are less sensitive compared to point queries since range query performance is dominated by sequential KV iteration. However, a block size of $16KB$ paired with the PRA model achieves the highest reward. This size strikes a balance, being more efficient than $8KB$ since the data being read is more than $8KB$, so reading $16KB$ blocks lowers the amount of metadata reads. Additionally, the PRA model outperforms PLA with different error ranges at the $16KB$ block size, as this specific setting has a characteristic similar to Fig. 8b where $E' < E$.

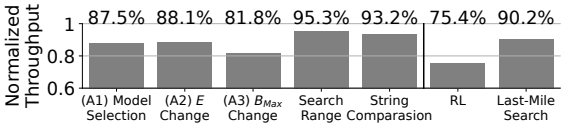


Figure 17: Ablation study. Bar plots show the normalized throughput to DobLIX full optimization.

Storage Optimization. Fig.16b and Fig.16d show results when the RL agent prioritizes minimizing the index size. The heatmaps indicate that the trends for both workloads are similar, as the stored data index is identical. Larger block sizes consistently reduce the index size by decreasing the number of blocks and associated metadata. Among the index models, PLA with a maximum error of 256 achieves the highest reward by reducing the number of splines, thus minimizing the index overhead. In both workloads, the chosen parameters achieve an estimated index size of less than $500MB$, which is half of the index size when the optimization is on the throughput. This is due to increasing the block size, which gives RocksDB a better compression ratio for the stored blocks, and also decrease the size of the LI model for each SST.

4.8 Ablation Study

We perform an ablation study to evaluate the impact of optimizations in DobLIX under an RH workload (Fig. 17). First, we tune the RL using a modified ZippyDB dataset with 16KB average values (SD = 5.2) and find the optimal state (PRA model, 8KB b_{max}, and error range $E = 8$). Then, we test these settings on the actual ZippyDB dataset while systematically disabling each optimization: (1) replacing RL-based model selection (PLA/PRA) with static PRA, (2) fixing $E = 8$ instead of RL-tuning, (3) enforcing a static 16KB block size, (4) disabling last-mile search range optimization, and (5) removing string comparison optimization. The fully optimized DobLIX serves as the baseline. Results show that disabling RL-based optimizations (A_1–A_3) reduces throughput by 12.5%, 11.9%, and 11.2%, respectively. Removing search range optimization causes a 4.7% drop due to inefficient searches, while disabling string comparison optimization lowers throughput by 6.8%, as keys often share prefixes that the optimization skips. We also disable whole RL optimization (1-3), and removing both last-mile optimizations (4, 5), which decrease the performance by 14.6% and 9.8%, respectively.

5 RELATED WORK

LSM read performance suffers from its layering; optimization focuses on Bloom filters, cache, and index. *(1) Bloom Filter Optimization.* Several works utilize filters to skip unnecessary reading since the filters do not return a false negative [42, 52, 57]. However, the filters suffer from high false positive rates and excessive memory usage. *(2) Cache Optimization.* LSbM-tree [56] adds a buffer to minimize cache invalidations due to compactions. AC-Key [64] introduces an adaptive caching algorithm to adjust the cache size based on the workload. Although caching accelerates the lookup of recently accessed data, it consumes significant storage as the cache size increases. Endure [30] optimizes the cache memory allocation adaptively based on the incoming workload. *(3) Index Optimization.* SLM-DB [32] implemented a persistent global B+tree index on NVM, and Kvell [38] adopts various memory index structures. DumpKV [71] trains a model on the KV lifetime to reduce the write amplification. Several studies [15, 45, 60] also built an LI using static

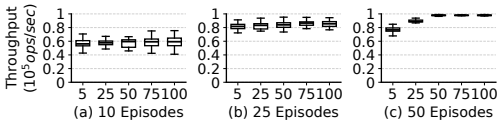


Figure 18: Final throughput robustness of the RL agent based on the percentage of initial samples to total data (x-axis), evaluated across three different total learning episodes (a, b, c). Each boxplot represents throughput distribution over 30 training runs, with the median (line), 1st quartile, and 3rd quartile (box) displayed.

data stored in SSTs to speed up read queries on SSTs. These studies improve indexing but overlook adjusting data access for persistent memory retrieval.

6 DISCUSSION

DobLIX Portability to various LSMs. All LSM variants (e.g., Cassandra, LevelDB) use lower-granularity units called blocks in their SST files to enable efficient data access. Porting DobLIX to other LSM stores is simple: use the API for block size, then save/load the model after SST writes/reads. We show LI's portability by adapting Bourbon's LevelDB [1] version to RocksDB, as discussed in §4.1.4, requiring minimal changes.

Mitigating RL Cold Start Problem. As shown in §16, the DobLIX RL agent finds near-optimal parameters after 50 episodes (1,000 SST instances), even with just 32 states and pruned actions. This underlines RL's cold start problem [19], but pre-trained models from similar workloads greatly cut training cost. For instance, transferring parameters from LOGN to AMZN (same RH point queries, both $v = 0$ and $v = 1$) requires only 5 fine-tuning episodes. Thus, pre-training transfers well across datasets for matching workloads, but switching workloads (e.g., point to range queries) on the same data still needs full retraining (30+ episodes).

RL Agent Robustness. We test our RL agent's robustness by varying initial training data (5%˜100%) and training episodes (10, 25, 50). Results (Fig. 18) show: (1) Performance variance mainly depends on episode count, and (2) the agent is robust to small initial datasets. Notably, with only 25% initial data and enough episodes, the agent reliably finds optimal solutions, indicating that our method works well even with limited data if training is sufficient.

7 CONCLUSION

We introduce DobLIX, an efficient LSM index that uses access-aware indexing on sorted SST keys during creation to enhance retrieval. We also introduce an RL tuning agent in our method to optimize the LSM I/O block size and LI error range parameters in order to improve the trained LI performance. We show that DobLIX significantly improves read performance (up to 2.21×) in RocksDB, without compromising its write intensity. DobLIX's adaptability to various workloads and data distributions allows it to be applied in a wider range of applications and scenarios.

Future Work. We aim to extend our approach with multi-objective optimization for LSM parameter tuning (e.g., write amplification) during LI training. We also improve our RL agent by adding parameters, such as adjusting the Bloom filter's false positive rate. Additionally, we investigate hybrid indexes that adaptively switch between learned and traditional types based on workload patterns for stable performance.

REFERENCES

[1] 2011. LevelDB. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html. Last accessed 2025-07-11.

[2] 2019. FB Dataset. https://doi.org/10.7910/DVN/JGVF9A/Y54SI9. Last accessed 2025-07-11.

[3] 2019. OSM Dataset. https://console.cloud.google.com/marketplace/product/openstreetmap/geo-openstreetmap. Last accessed 2025-07-11.

[4] 2019. WikiTS Dataset. https://doi.org/10.7910/DVN/JGVF9A/SVN8PI. Last accessed 2025-07-11.

[5] 2020. Bourbon Code. https://github.com/edydfang/Bourbon. Last accessed 2025-07-11.

[6] 2021. TridentKV Code. https://github.com/emperorlu/Learned-RocksDB. Last accessed 2025-07-11.

[7] Jacob D Abernethy, Robert Schapire, and Umar Syed. 2024. Lexicographic optimization: Algorithms and stability. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2503–2511.

[8] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 209–224.

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

[11] 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.

[12] Altannar Chinchuluun and Panos M Pardalos. 2007. A survey of recent developments in multiobjective optimization. *Annals of Operations Research* 154, 1 (2007), 29–50.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[14] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[15] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: a learned index for log-structured merge trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 9, 17 pages.

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–48.

[17] 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*. 505–520.

[18] 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. SIGMOD.

[19] Nan Ding and Radu Soricut. 2017. Cold-start reinforcement learning with softmax policy gradient. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 2814–2823.

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[21] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance comprehension at WiredTiger. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/3236024.3236081

[22] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (April 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[23] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *SIGMOD*.

[24] Alireza Heidari, Amirhossein Ahmadi, and Wei Zhang. 2025. UpLIF: An Updatable Self-tuning Learned Index Framework. In *Database Engineered Applications*, Richard Chbeir, Sergio Ilarri, Yannis Manolopoulos, Peter Z. Revesz, Jorge Bernardino, and Carson K. Leung (Eds.). Springer Nature Switzerland, Cham, 345–362.

[25] Alireza Heidari, Amirhossein Ahmadi, Zefeng Zhi, and Wei Zhang. 2024. Metahive: A cache-optimized metadata management for heterogeneous key-value stores. *arXiv preprint arXiv:2407.19090* (2024).

[26] Alireza Heidari, Ihab F Ilyas, and Theodoros Rekatsinas. 2020. Approximate inference in structured instances with noisy categorical observations. In *Uncertainty in Artificial Intelligence*. PMLR, 412–421.

[27] Alireza Heidari, Shrinu Kushagra, and Ihab F Ilyas. 2020. On sampling from data with duplicate records. *arXiv preprint arXiv:2008.10549* (2020).

[28] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[29] Alireza Heidarikhazaei. 2021. Structured Prediction on Dirty Datasets. (2021).

[30] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: a robust tuning paradigm for LSM trees under workload uncertainty. *Proc. VLDB Endow.* 15, 8 (April 2022), 1605–1618. https://doi.org/10.14778/3529337.3529345

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

[32] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. {SLM-DB}:{Single-Level}{Key-Value} store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.

[33] 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.

[34] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 1–15. https://www.usenix.org/conference/fast19/presentation/kourtis

[35] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*.

[36] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[37] 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 (jun 2023), 22 pages. https://doi.org/10.1145/3589284

[38] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[39] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. {ROLEX}: A Scalable {RDMA-oriented} Learned {Key-Value} Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[40] 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.

[41] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *VLDB* (2023). https://doi.org/10.14778/3598581.3598593

[42] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. {ElasticBF}: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large {Key-Value} Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 739–752.

[43] Ester Livshits, Alireza Heidari, Ihab F Ilyas, and Benny Kimelfeld. 2020. Approximate denial constraints. *arXiv preprint arXiv:2005.08540* (2020).

[44] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proc. VLDB Endow.* 15, 3 (nov 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[45] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2022. 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 (2022), 1953–1966. https://doi.org/10.1109/TPDS.2021.3118599

[46] 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.

[47] 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 (Sept. 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[48] 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. *Proceedings of the ACM on Management of Data* 1, 3

(2023), 1–25.

[49] Dingheng Mo, Siqiang Luo, and Stratos Idreos. 2025. How to Grow an LSM-tree? Towards Bridging the Gap Between Theory and Practice. *arXiv preprint arXiv:2504.17178* (2025).

[50] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. 2015. Cliffguard: A principled framework for finding robust database designs. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1167–1182.

[51] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. (2023). https://doi.org/10.14778/3587136.3587148

[52] 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.

[53] 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.

[54] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *CoRR* abs/2111.14905 (2021). arXiv:2111.14905 https://arxiv.org/abs/2111.14905

[55] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.

[56] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 68–79.

[57] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[58] Wenlong Wang and David Hung-Chang Du. 2024. LearnedKV: Integrating LSM and Learned Index for Superior Performance on SSD. *arXiv preprint arXiv:2406.18892* (2024).

[59] 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.

[60] Yi Wang, Jianan Yuan, Shangyu Wu, Huan Liu, Jiaxian Chen, Chenlin Ma, and Jianbin Qin. 2024. LeaderKV: Improving Read Performance of KV Stores via Learned Index and Decoupled KV Table. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 29–41. https://doi.org/10.1109/ICDE60146.2024.00010

[61] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. 2024. WIPE: A Write-Optimized Learned Index for Persistent Memory. *ACM Trans. Archit. Code Optim.* 21, 2, Article 22 (feb 2024), 25 pages. https://doi.org/10.1145/3634915

[62] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[63] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proc. VLDB Endow.* 15, 11 (July 2022), 3004–3017. https://doi.org/10.14778/3551793.3551848

[64] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. {AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 603–615.

[65] Giorgos Xanthakis, Antonios Katsarakis, Giorgos Saloustros, and Angelos Bilas. 2024. vLSM: Low tail latency and I/O amplification in LSM-based KV stores. *arXiv preprint arXiv:2407.15581* (2024).

[66] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3415–3427.

[67] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[68] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-Based Construction Algorithm. *VLDB* (2022). https://doi.org/10.14778/3551793.3551823

[69] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3, Article 151 (may 2024), 26 pages. https://doi.org/10.1145/3654954

[70] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. 2022. PLIN: A persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.

[71] Zhutao Zhuang, Xinqi Zeng, and Zhiguang Chen. 2025. DumpKV: Learning Based Lifetime Aware Garbage Collection for Key Value Separation in LSM-Tree. *Proc. VLDB Endow.* 18, 4 (May 2025), 1223–1236. https://doi.org/10.14778/3717755.3717778