On Self-Designing Learned Indexes

BAOFU HAN, Southeast University, China
GUOYU HU, National University of Singapore, Singapore
BING LI, Southeast University, China
XIAOKUI XIAO, National University of Singapore, Singapore
ZHANHAO ZHAO*, National University of Singapore, Singapore
BENG CHIN OOI, Zhejiang University, China

Learned indexes show promising performance compared with traditional indexes. However, most existing learned index structures rely on fixed heuristics, which limit their robustness under dynamic workloads. In this paper, we present SELIX, a self-designing learned index that automatically generates and optimizes index structures tailored to different workloads. We introduce a unified index template that allows different nodes to adopt structural configurations, node layouts, conflict resolution policies, and search strategies for flexible and fine-grained structural composition. On top of this template, we formulate structural optimization as a learnable function that maps workload states to index configurations, enabling automated search for optimal structures in the vast design space. We train this function using a deep reinforcement learning paradigm enhanced with meta-learning, and propose a dedicated online update strategy, which together enable fast and stable structural adaptation under workload drift. Experimental results show that SELIX consistently outperforms state-of-the-art learned indexes (ALEX, LIPP, FITing-Tree, PGM, XIndex, and FINEdex) in both in-memory and on-disk settings, achieving significantly higher throughput across diverse and dynamic workloads.

CCS Concepts: • **Data Management Systems → Storage, indexing, and physical database design**.

Additional Key Words and Phrases: Self-Designing Learned Index, Index Tuning, AIxDB

ACM Reference Format:
Baofu Han, Guoyu Hu, Bing Li, Xiaokui Xiao, Zhanhao Zhao, and Beng Chin Ooi. 2026. On Self-Designing Learned Indexes. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 219 (June 2026), 25 pages. https://doi.org/10.1145/3802100

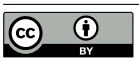
1 Introduction

Learned indexes, which employ machine learning (ML) models to predict key positions, have become an active research topic in recent years [1–6]. Compared with traditional indexes (e.g., B^+-tree) [7], learned indexes can, both empirically [8, 9] and theoretically [10, 11], achieve superior performance and lower storage costs, especially under static (read-only) workloads. Nevertheless, to the best of our knowledge, no mainstream database system has yet to adopt learned indexes [12].

Given the dynamism and variety of real-world workloads, it is challenging for any existing learned index to perform well across diverse workloads, which consequently limits their practical

*Corresponding author.

Authors' Contact Information: Baofu Han, hanbaofu@seu.edu.cn, Southeast University, Nanjing, China; Guoyu Hu, National University of Singapore, Singapore, guoyu.hu@u.nus.edu; Bing Li, Southeast University, Nanjing, China, bernie_seu@seu.edu.cn; Xiaokui Xiao, National University of Singapore, Singapore, xkxiao@nus.edu.sg; Zhanhao Zhao, National University of Singapore, Singapore, zhanhao@nus.edu.sg; Beng Chin Ooi, Zhejiang University, China, ooibc@zju.edu.cn.



ACM 2836-6573/2026/6-ART219
https://doi.org/10.1145/3802100

applicability [8, 13]. The performance of a learned index is tied to its structural design, including layout, partitioning logic, search strategies, etc. However, the structures of existing learned indexes typically rely on manually crafted rules and expert heuristics. For example, ALEX [4] separates the model nodes from the data nodes, resulting in a contiguous data layout that favors scan-heavy workloads, whereas LIPP [1] stores data along internal paths, offering superior performance for point lookups but slower scan performance than ALEX [8]. Recent work [14] investigates self-tuning techniques for specific indexes (e.g., ALEX) by adjusting parameters such as fanout and node size to accommodate varying workloads. However, their performance remains bounded by the original structure, i.e., they cannot achieve structural transformations that would outperform the underlying design. Addressing this challenge requires new strategies that enable structural adaptivity, which allows indexes to evolve in response to workload drift.

Unlike existing approaches, we aim to design a learned index whose structure can autonomously adapt to the current workload and environment without human intervention, thus expediting the practical adoption of learned indexes. However, achieving this poses three key challenges. First, it requires defining an index structure that is flexible and expressive enough to encompass diverse design choices. Given the vast design space of existing indexes, constructing such a unified structure is non-trivial. Second, identifying the optimal structure for a given workload is difficult. Mapping the characteristics of the workload and the corresponding design choices into a scalable optimization problem is essential. Third, the index must be able to adapt efficiently to workload drift. Achieving structural adjustments with minimal performance overhead is not straightforward.

In this paper, we present SELIX, a SElf-designing Learned IndeX that integrates a learning agent capable of continuously generating, evaluating, and optimizing its structural designs. We first introduce a unified index template as the foundation of SELIX, a generic tree structure in which each node can be independently configured along four structural dimensions: (1) structural configuration (node type, fanout, and capacity), (2) node layout (compact/gapped), (3) conflict resolution policy (shift/chain/rebuild), and (4) search strategy (binary/exponential search).

We then formulate index structural optimization as a learnable function. Specifically, we define a mapping $\mathcal{F}_\theta : \mathcal{S} \rightarrow \mathcal{A}$ that learns to select optimal index configurations based on the current workload state. The state $s \in \mathcal{S}$ captures node-level statistics such as read/write frequency, and I/O cost, and model prediction errors. The action $a \in \mathcal{A}$ consists of first identifying one of the top-k nodes that contribute most to latency or error, and then applying a local transformation such as adjusting the fanout, switching the layout, or retraining the local model. Given the vast action space, we introduce the deep reinforcement learning (DRL) paradigm to train the function $\mathcal{F}_\theta$, which enables efficient exploration of the design space to identify optimal index configurations under the given workload.

In real database systems, workloads evolve continuously, and retraining the policy from scratch upon each workload shift is prohibitively expensive. To this end, we ensure that SELIX can efficiently perform structural self-adaptation to handle workload drift from two aspects. First, we enhance the DRL-based learning process with meta-learning, which allows the agent to learn generalizable strategies across diverse workloads and quickly transfer prior knowledge to new scenarios. As a result, in most cases, SELIX can directly leverage the deployed agent and its trained policy to remain effective without retraining from scratch. Second, the system monitors index throughput as a drift signal, and triggers online adaptation when significant performance degradation is detected. For these cases, we propose a dedicated online update strategy based on Proximal Policy Optimization (PPO) [15], which constrains policy changes within a safe range to enable fast and stable online fine-tuning.

In summary, we make the following contributions:

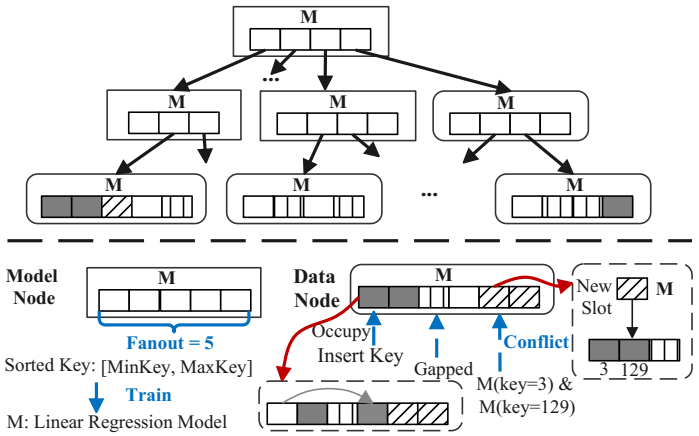


Fig. 1. The structure of learned indexes.

- We propose SELIX, a self-designing learned index, and introduce a unified index template that enables flexible and fine-grained structural composition across multiple design dimensions.
- We formulate index structural optimization as a learnable function, and adopt a DRL algorithm to train this function, enabling automated search for optimal structures in the vast design space.
- We augment the DRL-based learning paradigm with meta-learning, which enables the agent to quickly adapt to new workloads without complete retraining, and propose a dedicated online update strategy for stable structural adaptation under workload drift.
- We evaluate the performance of SELIX against state-of-the-art learned indexes under workloads with different read-write ratios in both in-memory and on-disk settings. Experimental results demonstrate consistent performance gains across all evaluated workloads, achieving up to 3.73× higher throughput and superior robustness compared to state-of-the-art learned indexes (ALEX, LIPP, FITing-Tree, PGM, XIndex, and FINEdex).

SELIX is a component of NeurDB [16, 17]. It has been integrated into NeurDB as a native index access method [1].

The remainder of the paper is organized as follows. Section 2 provides the background and presents the problem statement. Section 3 overviews SELIX. Section 4 introduces the detailed learning-based optimization strategy. Section 5 describes the implementation. Section 6 presents the experimental results, and Section 7 concludes.

2 Preliminaries

In this section, we first outline the basic concepts of learned indexes, and then formalize the problem definition. The symbols used throughout the paper are listed in Table 1.

2.1 Learned Index

For a dataset $\mathcal{D}$, a learned index M models the mapping $\mathcal{K} \rightarrow \mathcal{P}$, where $\mathcal{K}$ denotes the key space and $\mathcal{P}$ the corresponding positions in $\mathcal{D}$. To achieve better scalability, learned indexes often employ a hierarchical tree structure [1, 4] with two node types: model nodes that route queries to children, and data nodes that store keys.

Operations. A learned index typically supports four core operations: lookup, scan, insert, and delete.

[1]https://github.com/neurdb/neurdb

Table 1. Key symbols and descriptions.

Term	Description
$\mathcal{D}$	dataset of keys and their corresponding positions
$\mathcal{K}$	key space
N	number of nodes in the index
$\mathcal{S}$	state space
$\mathcal{A}$	action space
R	reward
M	learned index
w, $\mathbf{w}$	workload and workload embedding vector
ϵ	error of model predicted position
$p, \hat{p}$	position and predicted position of a key

- *Lookup.* Given a key k, lookup traverses the hierarchy from root to data node. At each model node, the model predicts which child to access next. At the data node, the model predicts a position $\hat{p} = \mathcal{M}(k)$, and a local search is performed within $p \in [\hat{p} - \epsilon, \hat{p} + \epsilon]$, where ϵ is the maximum prediction error.
- *Scan.* A range query over $[k_l, k_r]$ first locates k_l via lookup, then enumerates subsequent keys until reaching k_r.
- *Insert.* To insert a new key k_{new}, the target data node is located by traversing the hierarchy similar to the lookup procedure. The model predicts the target position $\hat{p}_{\text{new}} = M(k_{\text{new}})$. When multiple keys are mapped to overlapping positions, conflicts occur and must be resolved based on the *conflict resolution policy*.
- *Delete.* Deletion locates the target key using the same lookup procedure and removes it.

Design Primitives. Although numerous learned indexes have been proposed, their design can be categorized based on their strategies for four core primitives, as illustrated in Figure 1:

- *Structural Configuration.* This defines the structural parameters of a node, including node type, fanout, and node capacity, which together determine the tree's shape and depth.
- *Node Layout.* This defines how keys are physically organized within a data node: (1) Compact layout stores keys contiguously; (2) Gapped layout with reserved slots for insertions.
- *Conflict Policy.* During insertion, if the predicted position is empty, the key is inserted directly. Otherwise, conflict resolution is required: (1) *Shift*: relocates nearby keys to adjacent positions to make space; (2) *Chain*: links overflow nodes for conflicting keys. (3) *Rebuild*: splits the nodes and reconfigures them with new parameters, potentially triggering cascading modifications up the tree.
- *Search Strategy.* This defines how lookups are performed when the key is not found at the predicted position: (1) Exponential search and (2) Binary search.

2.2 Existing Learned Indexes

Existing learned indexes differ mainly in which primitives they use, as summarized in Table 2. Most systems use a limited subset of the design primitives. For example, RMI [3] and PGM [2] largely rely on predefined hierarchical constructions, without allowing fine-grained control over fanout or conflict handling. ALEX [4] supports adaptive fanout and node splitting but relies on hand-crafted cost models for structural adjustments. LIPP [1] achieves precise position prediction but does not support multiple node types. FITing-Tree [18] compresses index metadata using a bounded-error piecewise-linear approximation to enable fast lookup with bounded local search. XIndex [19]

and FINEdex [20] buffer incoming writes and periodically merge (and retrain when needed) to synchronize with the main structure. DILI [21] builds distribution-aware index trees by adapting node organization to local density and adjusting the structure as the distribution evolves. NFL [22] improves robustness by transforming the key distribution before constructing the index. LITune [14] partially addresses adaptation by tuning selected parameters (e.g., fanout) with reinforcement learning, but they operate on a fixed underlying index structure. To more effectively evaluate the performance of these learned indexes, NeurBench [23] provides a standardized benchmarking platform that supports comparison of major learned components under data and workload drift.

Before the emergence of learned indexes, several generic index frameworks have been explored. GiST [24] provides a unified template for implementing various index structures (e.g., B-tree, R-tree). However, its support for multiple tree types and structural variations results in an exponentially large design space, making efficient search for optimal configurations challenging. To enhance the flexibility of index design, Data Calculator [25] decouples physical layouts into design primitives and leverages a learned cost model, enabling a shift from manual expert design to automated structural synthesis. GENE [26] employs genetic algorithms to search within a hybrid design space, enabling free combination and autonomous evolution of both traditional and learned index structures. Meanwhile, several studies on adaptive indexing [27, 28] have explored index performance optimization from the perspective of runtime evolution.

Unlike existing works, we introduce a self-designing learned index that autonomously generates and updates its structure using core primitives within a unified learnable framework, enabling it to adapt to varying workloads and achieve higher performance.

2.3 Problem Definition

While different learned index designs exhibit benefits under specific scenarios, no single structure can consistently achieve optimal performance across diverse and evolving workloads [8, 13]. In real database systems, workloads are inherently dynamic, making it impractical to manually design or tune a dedicated index for each case. We therefore aim to identify a structure generation function that maps workload characteristics to an optimal index structure.

We define a learned index M with depth L containing N nodes. Each node is characterized by a tuple $m_i = \langle \tau, \phi, \mu, \eta \rangle$, where τ denotes the structural configurations, ϕ specifies the layout, μ defines the update strategy, and η denotes the search strategy. The complete index M is thus represented as the composition of all nodes $\{m_i \mid i \in [1, N]\}$ under a specific hierarchical structure. A model m_i at node i not only determines the node itself but also the subtree rooted at node i, for insertions of its data passing this node are processed by m_i. This formulation enables the representation of index structures in which different subtrees employ distinct strategies optimized for their local workload characteristics.

Let $\mathbb{W}$ denote the distribution of workloads, and w denote a specific workload sampled from $\mathbb{W}$. Given the index design space $\mathbb{M}$, we formulate the index generation problem as an optimization task to find the most efficient index under w:

$$M^* = \arg\min_{M \in \mathbb{M}} U(\mathbb{M}, w), \tag{1}$$

where $U(\cdot)$ represents the utility function that evaluates the index performance (e.g., throughput, and latency).

For realistic index structures with thousands of nodes, the search space is vast, rendering traditional heuristic algorithms inefficient. We therefore leverage deep reinforcement learning (DRL) to learn the function $\mathcal{F}_\theta$, which enables efficient navigation of the design space to discover near-optimal solutions.

Table 2. Comparison with representative learned indexes and index design/generation frameworks. Symbols: ✓ supported/configurable; ✗ not supported; – not applicable.

Method	Structural Configuration		Node Layout		Search Strategy		Conflict Policy		
	Node Type	Fanout	Dense	Gapped	Binary	Exp.	Shift	Chain	Rebuild
RMI [3]	✓	✓	–	–	✓	✓	✗	✗	✗
PGM [2]	✓	✗	✓	✗	✓	✗	✗	✓	✗
ALEX [4]	✓	✓	✗	✓	✗	✓	✓	✗	✓
FITing-Tree [18]	✓	✗	✗	✓	✓	✓	✓	✗	✓
LIPP [1]	✗	✗	✗	✓	–	–	✗	✓	✓
DILI [21]	✓	✓	✗	✓	✗	✓	✗	✓	✗
NFL [22]	✓	✗	✗	✓	✓	✗	✓	✓	✓
LITune [14]	✓	✓	✗	✗	✗	✗	✗	✗	✓
GENE [26]	✓	✓	–	–	✓	✓	✗	✗	✗
GiST [24]	✓	✓	–	–	–	–	–	–	✓
Data Calculator [25]	✓	✓	✗	✓	✓	✗	–	–	✓
SELIX (Ours)	✓	✓	✓	✓	✓	✓	✓	✓	✓

2.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) [29–31] has been widely applied in data management tasks [32], such as query optimization [33, 34], configuration tuning [14, 35], etc., where decision spaces are high-dimensional, and feedback is delayed. The core idea of DRL is to train an agent that interacts with an environment through a continuous loop of state–action–reward feedback, learning a policy that maximizes the expected cumulative reward. Existing DRL algorithms can be broadly categorized into two families: (1) value-based methods (e.g., DQN [29]), which estimate a state–action value function to guide discrete action selection, and (2) policy-based methods (e.g., DDPG [36], PPO [31]), which directly sample actions via policy probability. Value-based methods are restricted to purely discrete action spaces, whereas our index design problem involves both discrete (e.g., node layout choice) and continuous (e.g., fanout size, split threshold) actions. Similarly, off-policy methods such as DDPG, though suitable for continuous actions, are not ideal in our setting since they rely on offline replay buffers and thus cannot efficiently adapt to evolving workloads in an online environment. In contrast, on-policy methods update using trajectories generated by the current policy, which better matches dynamic workload optimization. Proximal Policy Optimization (PPO) [31] is an on-policy actor-critic algorithm designed to enhance training stability by constraining the magnitude of policy updates. It achieves this by using a clipped surrogate objective that bounds the probability ratio between the new and old policies and prevents overly aggressive updates.

We therefore adopt PPO as the core RL algorithm in SELIX. However, directly applying standard PPO presents a key challenge. Evaluating each action, i.e., a configuration change, requires running the modified index under realistic workloads, which limits frequent resampling. To address this, we extend it with a weighted sample reuse mechanism to improve data efficiency while preserving stable policy updates, as detailed in Section 4.3.

3 Overview

The overview of SELIX is shown in Figure 2, which consists of four main components: the index template, the state collector, the learning agent, and the index builder.

Index Template. At the foundation, SELIX adopts a unified index template that abstracts learned indexes as a configurable tree structure. The tree consists of two types of nodes: model nodes for routing queries and data nodes for storing keys. Each node can be independently configured with its own layout, size, search strategy, and other structural parameters. This design enables the generation of diverse index structures through node-level configuration adjustments within a unified framework.

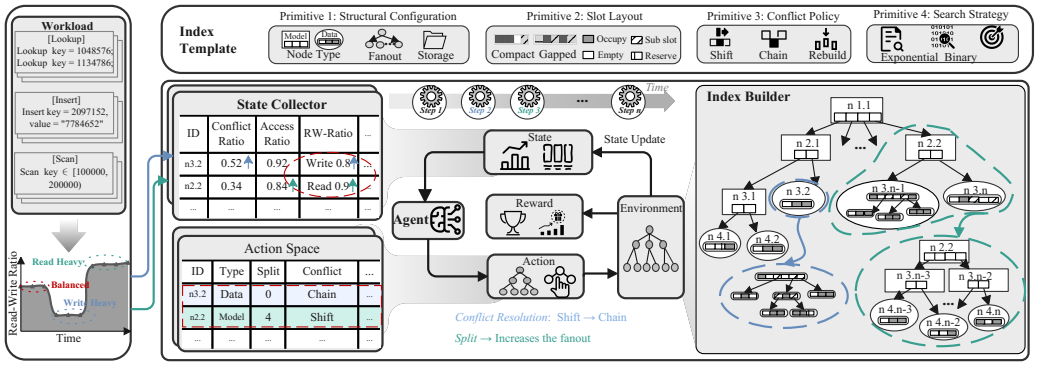


Fig. 2. The overall framework of SELIX

State Collector. The state collector constructs the state space S_t for the learning agent by capturing two types of information: workload-level information, such as query arrival rates and read/write compositions, and node-level structural information from the index's internal metadata, including access frequency, conflict degree, and model accuracy. Since a learned index may contain a large number of nodes, including all of them in the state would lead to an excessively high-dimensional input. We therefore maintain the top-k most frequently accessed nodes (i.e., those with the highest performance impact) in the state collector. The method for calculating the impact score is detailed in Section 4.2.2.

Learning Agent. Built on top of the index template, the learning agent is responsible for generating the optimal index structure for the given workload. It observes the state S_t and selects an action a_t from the action space $\mathcal{A}$ to adjust node structural parameters. The detailed design of the action space is described in Section 4.2.3. By iteratively optimizing individual nodes, the agent progressively constructs an efficient overall index structure. The agent is trained using a deep reinforcement learning framework with meta-learning, enabling fast policy adaptation across different workloads.

Index Builder. Once a_t is selected, the executor applies structural modifications to the corresponding node, and its subtree is adjusted following the change. Each update is localized so that only the affected nodes are rebuilt or reconfigured to minimize runtime interference. The state collector then captures the new performance metrics and feeds them to the learning agent to calculate the rewards, updating the state to s_{t+1} and triggering the next round, forming a closed learning loop for continuous optimization.

Execution Workflow. During the training process, we construct diverse workloads with varying read-write ratios (e.g., from read-write balanced to write-heavy, and then to read-heavy) to train the learning agent, enabling it to learn effective optimization strategies across different scenarios. Based on the trained policy, SELIX's agent generates an initial index structure tailored to the observed workload characteristics. During execution, the state collector continuously monitors runtime statistics such as access frequency, read/write ratio, and conflict degree, and constructs state vectors for each node. When the index throughput degrades beyond a predefined threshold, indicating that the current structure no longer fits the workload, the agent is triggered to perform structural adaptation. It evaluates candidate actions within its action space and selects localized structural adjustments that enhance performance while minimizing overhead. Specifically, as illustrated in Figure 2, the agent replaces the conflict resolution strategy for nodes in the subtree rooted at $n_{3,2}$ from *shift* to *chain*, reducing key relocation under write-intensive workloads. It then adjusts the fanout of the subtree starting from node $n_{2,2}$ to optimize for the read-heavy workload there. Only the affected nodes are reconfigured, while the rest of the index remains unaltered. Following the

update, new performance metrics such as write latency and insertion throughput are fed back to the agent as rewards to guide subsequent structural adjustments.

4 Design of SELIX

In this section, we first present the unified index template and its design space, and then detail the learnable function formulation and training framework, followed by theoretical analysis.

4.1 Index Template and Design Space

We introduce a unified index template that allows each node to independently select and combine design primitives, enabling fine-grained structural construction. The design space consists of four aspects: structural configuration, node layout, conflict resolution policy, and search strategy. Since each node can be independently configured, the total design space grows exponentially with the number of nodes, resulting in an enormous search space. Each aspect is defined as follows.

Node Structural Configuration Space. SELIX supports two node categories: model nodes for routing queries and data nodes for storing keys.

Model Node. A model node is characterized by a tuple $\langle \omega, \epsilon, b \rangle$, where ω represents the model type, ϵ denotes the prediction error bound, and b specifies the fanout. The model type ω can be instantiated as linear regression or piecewise linear approximation [2], mapping an input key to the corresponding child node with a prediction error bounded by ϵ. The fanout b determines the number of children, directly influencing tree depth and search complexity.

Data Node. A data node stores keys directly and is characterized by a tuple $\langle c, l, g, r, p \rangle$, where c is the node capacity, l is the node layout, g is the gap configuration, r is the insertion conflict resolution strategy, and p is the model rebuilding policy. Node capacity c balances insertion cost against lookup efficiency: smaller nodes reduce reorganization overhead during insertions, while larger nodes shorten lookup paths through reduced tree depth. The remaining parameters l, g, r and p are detailed in the *Node Layout Space* and *Conflict Policy Space* below.

Node Layout Space. The node layout space defines how keys are physically organized within a data node, specified by layout type and gap configuration. The layout type can be either *Dense* or *Gapped*. A *Dense* layout stores keys in tightly packed arrays with no empty slots, maximizing space efficiency and read performance, but insertions may require shifting existing keys to maintain sorted order. A *Gapped* layout pre-allocates empty slots to absorb insertions without triggering key movements, better suited for write-intensive scenarios. For the *Gapped* layout, two additional parameters are specified: gap ratio and gap distribution. The gap ratio in $[0, 1)$ defines the fraction of empty slots reserved for future insertions. The gap distribution has two options, *Uniform* and *Exponential*, which determine how gaps are placed within the node. In a uniform distribution, gaps are evenly distributed across the array. Under an exponential distribution, more gaps are allocated near the end of the array, suitable for append-heavy workloads.

Conflict Policy Space. The conflict policy space defines how data nodes resolve insertion conflicts. Since SELIX employs model-based insertion, the model predicts a target position for each incoming key, and conflicts arise when the predicted position is already occupied. Two resolution strategies are available: *Shift* and *Chain*. The *Shift* strategy performs in-place updates by relocating existing keys to nearby available empty slots. The *Chain* strategy performs out-of-place updates by allocating new sub-structures to accommodate keys.

Search Strategy Space. The search strategy space defines how keys are located within a data node when the model's predicted position deviates from the actual position. Two strategies are available: *Binary* and *Exponential* search. Given a predicted position $\hat{p}$ with error bound ϵ, *Binary* search scans the range $[\hat{p} - \epsilon, \hat{p} + \epsilon]$ with $O(\log \epsilon)$ complexity, suitable when the error bound is

large. *Exponential* search starts from $\hat{p}$ and expands outward with $O(\epsilon)$ complexity, which is more efficient when the prediction error is small. The choice of search strategy also depends on the node's data layout: for *Dense* layouts where keys are contiguous, both strategies perform well; for *Gapped* layouts with empty slots, *Exponential* search can skip empty positions more efficiently.

4.2 Learnable Function Formulation

4.2.1 Workload Representation. Different workloads require different configurations to achieve high performance. Since workloads change dynamically over time, we employ a lightweight Long Short-Term Memory (LSTM) [37] to encode the workload sequence into a fixed-dimensional embedding vector, enabling the RL agent to perceive workload trends and patterns. Specifically, workloads are characterized from two perspectives: (1) query arrival rate, which measures the number of incoming queries per unit time, typically expressed as queries per second (QPS), and (2) query composition, which captures the proportion and fluctuation of different query types (e.g., read/write ratio) over time. For example, the workload within a specific time window can be expressed as: $w = \{QPS = 320, QueryRatio = [0.7, 0.3]\}$, where $QueryRatio$ denotes the relative frequency of read and write operations. A sequence of such workload representations over the past x time steps, i.e. $(w_{t-x+1}, \ldots, w_t)$, is fed into the LSTM encoder to generate a compact workload embedding. The resulting embedding is then integrated into the input state of the RL agent.

4.2.2 State Space. We construct a state space based on both the index configuration and workloads. SELIX captures statistics and metadata at the node level, and the index's state is composed of the states of all its nodes. For each node n_i, its state (e.g., gapped ratio, key distribution, node fanout, access frequency) and its local workload statistics (e.g., operation counts) are maintained in a per-node metadata table. When the agent requests an observation, the state collector retrieves these metrics and transforms them into a state representation $\phi_i = (CR_i, RI_i, SK_i, AF_i, EC_i)$, where CR_i is computed from the ratio of colliding slots to total slots, RI_i is derived from the recent read/write counters within a sliding window ΔT, SK_i is obtained from the skewness of the key histogram, AF_i is the access frequency, and EC_i is the estimated restructuring cost based on node size. All features are normalized on the fly using the global min-max statistics collected in the current epoch to ensure comparability across nodes. Since an index may contain tens of thousands of nodes, incorporating states of all nodes as the index state for the learning agent would result in excessively high dimensionality and redundant information. To address this, we adopt a ranking-based sampling strategy. Specifically, each node is assigned an advantage score that quantifies how much it is expected to influence overall index performance if optimized. We compute the score as a weighted linear combination of the normalized feature vector: $score_i = \mathbf{W}^\top \phi_i = \sum_{j=1}^{5} \mathbf{w}_k \phi_{i,j}$. The top-$k$ nodes with the highest scores are selected, and their feature vectors are concatenated to form a fixed-length structural state embedding vector: $\mathbf{h}_t = [\phi_1, \phi_2, \ldots, \phi_k] \in \mathbb{R}^{k \times 5}$. Finally, we concatenate the structural state embedding and the workload embedding w_t generated by the LSTM encoder. The state at step t is thus defined as

$$\mathcal{S}_t = [\mathbf{h}_t, \mathbf{w}_t]. \tag{2}$$

4.2.3 Action Space. The action space in SELIX builds upon the index design space defined in Section 4.1. Each action corresponds to a configuration option within the design space, operating on one node and its subtree at a time. To ensure consistency in state representation, actions consider the same top-k candidate nodes identified in the state space. Formally, the action space is:

$$\mathcal{A} = \{(i, o, \mathbf{v}) \mid i \in \{1, \ldots, k\}, o \in O, \mathbf{v} \in \mathbb{R}^d\}. \tag{3}$$

Each action is represented as a triplet $a_t = (i_t, o_t, \mathbf{v}_t)$, where $i_t \in \{1, 2, \ldots, k\}$ identifies the target node, $o_t \in O$ specifies the action type to perform, and $\mathbf{v}_t$ denotes the action parameters.

The action types O correspond to the four design dimensions defined in Section 4.1: (1) structural configuration, including model type, fanout, and node capacity; (2) node layout, including layout type (Dense/Gapped), gap ratio, and gap distribution; (3) conflict policy, including resolution strategy (Shift/Chain); (4) search strategy (Binary/Exponential). For example, the action $(2, layout, Gapped)$ switches the layout of the second-ranked node from *Dense* to *Gapped*, while $(3, conflict, Chain)$ changes the conflict resolution strategy of the third-ranked node to *Chain*. Through these actions, the RL agent navigates the action space to find optimal configurations.

4.2.4 Reward. In SELIX, the reward quantifies the quality of each action to guide the agent in optimizing index structures. The reward consists of two components: performance gain and storage cost.

Performance Gain. We measure the relative throughput improvement over a baseline index configuration to encourage actions that enhance query efficiency. When the agent selects an action, it evaluates its impact on the target workload, potentially triggering restructuring operations such as data node reorganization or model retraining. Through repeated interactions and this reward formulation, the agent learns to identify high-quality actions while minimizing restructuring overhead.

$$R_{\mathrm{g}} = \eta_{\mathrm{g}} \frac{T_t - T_0}{T_0} + (1 - \eta_{\mathrm{g}}) \frac{T_t - T_{t-1}}{T_{t-1}}, \tag{4}$$

where T_0 is the baseline throughput obtained with the initial index, and T_t and T_{t-1} are those at step t and the previous step t-1. This reward captures both long-term and short-term performance improvement, and η_g is used to weigh their relative importance.

Storage Cost. For a learned index containing N nodes, and node i consumes $cost_i$. The storage cost is defined similarly as:

$$R_{\mathrm{s}} = \eta_{\mathrm{s}} \frac{C_t - C_0}{C_0} + (1 - \eta_{\mathrm{s}}) \frac{C_t - C_{t-1}}{C_{t-1}}, \text{where } C = \sum_{i=1}^{N} cost_i. \tag{5}$$

To prevent either one reward component from dominating the optimization process and destabilizing learning, we balance the contributions via a weighted combination, as shown:

$$R = \lambda_g \cdot R_{\mathrm{g}} - \lambda_s \cdot R_{\mathrm{s}}, \tag{6}$$

where λ_g and λ_s are the weighting coefficients for throughput improvement and storage efficiency, respectively, satisfying $\lambda_g + \lambda_s = 1$.

4.3 Function Training

To enable fast and stable adaptation of the learnable function under continuously evolving workloads, we integrate meta-learning with PPO to form the training framework of SELIX, as shown in Figure 3.

The key idea is to learn a meta-policy with strong generalization capability, enabling the agent to adapt quickly to new workloads with minimal updates. Our framework follows a two-level meta-training paradigm, consisting of an inner loop for task-specific adaptation and an outer loop for meta-policy optimization. In the inner loop, for each sampled workload task, the agent performs several PPO updates to adapt the meta-policy to the current task. In the outer loop, the meta-policy is updated by aggregating the post-adaptation experiences across all sampled tasks. This design ensures that SELIX can retain prior knowledge while quickly adapting to workload drift, avoiding costly retraining from scratch.

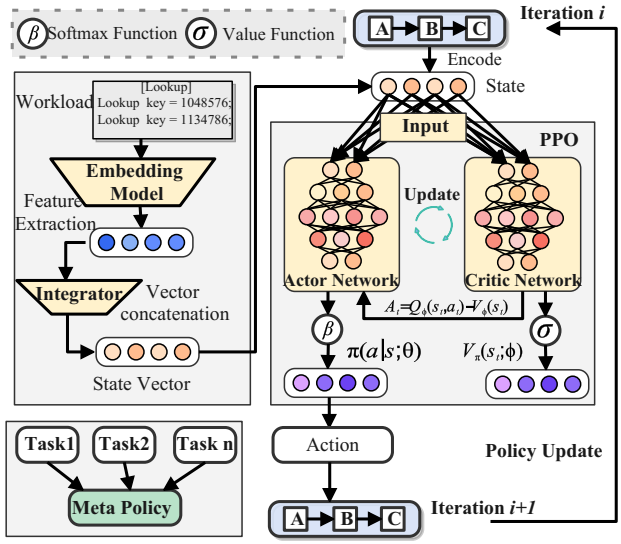


Fig. 3. The DRL network architecture.

Inner Loop (Task-Specific Adaptation). In each meta-iteration, the agent samples a batch of workload tasks from a diverse set of workloads. For each task, the agent initializes from the current meta-policy θ and performs several PPO updates to obtain a task-specific policy θ'_i. The post-adaptation trajectories and rewards are stored in a shared replay buffer for subsequent meta-policy optimization. The PPO clipped objective is defined as: $\mathcal{L}_{\text{PPO}}(\theta) = \mathbb{E}\Big[\min\big(\rho_t \hat{A}_t,\ \text{clip}(\rho_t, 1-\varepsilon, 1+\varepsilon)\hat{A}_t\big)\Big]$, where $\rho_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the current and previous policies, and $\hat{A}_t$ is the estimated advantage function. The clipping mechanism ensures bounded policy shifts during each update, preventing training instability.

Outer Loop (Meta-Policy Optimization). After completing the inner-loop adaptation across all sampled tasks, the meta-policy is updated by aggregating the post-adaptation experiences. Unlike standard PPO, which discards samples after each update, SELIX employs a weighted sample reuse mechanism that retains historical trajectories across meta-iterations, improving data efficiency while preserving stability. Each trajectory in the buffer is assigned a composite weight $w(\tau) = w_{\text{temp}}(\tau) \cdot w_{\text{IS}}(\tau)$, where $w_{\text{temp}}(\tau) = \exp(-\lambda \Delta t)$ is the temporal weight that prioritizes recent samples, and $w_{\text{IS}}(\tau)$ denotes the importance weight correcting policy mismatch, $w_{\text{IS}}(\tau) = \text{clip}\Big(\prod_t \frac{\pi_\theta(a_t|s_t)}{\mu(a_t|s_t)}, \underline{c}, \overline{c}\Big)$, where μ is the behavior policy that generated the trajectory. The meta-optimizer computes a weighted PPO surrogate objective using the mixed batch, where each historical trajectory is weighted according to its temporal recency and policy compatibility, so that outdated or policy-incompatible samples have reduced influence on the update. This allows safe sample reuse without breaking PPO's on-policy guarantees.

Algorithm 1 illustrates the pseudo-code of the training framework. In each meta-iteration, a batch of workload tasks is sampled from a diverse set of workloads (line 4). For each sampled task, the inner loop initializes from the current meta-policy and performs several PPO updates to obtain a task-specific policy (lines 6-9). The postadaptation trajectories are then stored in the replay buffer (lines 10-11). After all tasks are adapted, the outer loop constructs a mixed batch from the replay buffer using the weighted sample reuse mechanism (line 14), computes the meta-objective (line

Algorithm 1: Meta-RL Training Framework of SELIX

Input: A diverse set of workloads $\{\tau\}$
Output: Learned meta-policy θ^*

```
1  Initialize meta-policy θ;
2  Initialize replay buffer B ← ∅;
3  while not converged do
4      Sample a batch of workload tasks {τ_1, τ_2, ..., τ_B};
5      // Inner Loop: Task-Specific Adaptation;
6      for each task τ_i do
7          Initialize task policy θ_i ← θ;
8          Collect rollout data D_i using π_{θ_i};
9          Update θ_i' ← PPO(θ_i, D_i) for several steps;
10         Collect post-adaptation rollouts D_i' using π_{θ_i'};
11         Store (D_i', τ_i) into replay buffer B;
12     end
13     // Outer Loop: Meta-Policy Optimization;
14     Construct mixed batch D_mix from B with weighted sample reuse;
15     Compute meta-objective L_meta(θ) using D_mix;
16     Update meta-policy θ ← θ − α∇_θ L_meta(θ);
17 end
18 return meta-policy θ* ← θ;
```

15), and updates the meta-policy accordingly (line 16). Through iterative meta-training, the agent learns a wellgeneralized meta-policy θ^* that can be deployed for online execution (Section 4.4).

4.4 Online Execution

After meta-training, SELIX deploys the learned meta-policy θ^* for online execution, which consists of three stages: initial structure generation, runtime monitoring, and adaptive update. Each stage is described as follows.

Initial Structure Generation. Upon receiving an index building request, the agent starts from the meta-policy θ^* and performs a few PPO updates based on the observed workload characteristics to obtain an adapted policy. The adapted policy then generates an optimized index structure, which is deployed to serve workloads.

Runtime Monitoring. During runtime, SELIX continuously monitors throughput to detect performance changes. When performance remains stable, the index structure continues to execute workloads, with structure reorganization (e.g., split and merge) performed to maintain performance. When throughput degradation exceeds a predefined threshold, indicating that the workload has shifted and the current structure is no longer suitable, SELIX triggers an adaptive update.

Adaptive Update. Upon triggering, the system first takes a snapshot of the current index, including its structure and metadata. The snapshot serves as input to the RL agent, which leverages the existing structure as a starting point for optimization. The RL agent operates on the snapshot while the original index continues serving workloads. Once a new structure suited to the current workload is generated, the new index is constructed in the background. Upon completion, SELIX switches to the new index and removes the old index after no active transactions reference it.

Discussion. SELIX uses throughput degradation as triggers for adaptation. This design is based on two considerations. First, compared to monitoring workload distribution, these metrics incur minimal overhead and require no additional computation. Second, their degradation intuitively

indicates that the current index configuration no longer fits the workload and has room for optimization. Once triggered, the agent generates optimal index configurations for the current workload. We validate the effectiveness of throughput as a trigger metric in Section 6.3.7.

4.5 Theoretical Analysis

4.5.1 Time Complexity Analysis. We establish theoretical complexity bounds for learned indexes generated by SELIX to demonstrate the necessity of workload-adaptive configuration. Consider a generated index M with tree depth $L = L_m + 1$, consisting of L_m model node levels and a single data node level at the leaf.

Lookup Complexity. A lookup operation traverses the index from the root node to the leaf node. At each model node level ℓ, the query performs model inference in $O(1)$ time. Upon reaching the leaf data node, the model's prediction error ϵ necessitates searching within the range $[p - \epsilon, p + \epsilon]$, where p is the predicted position. The correction cost is $C_{\text{search}}(\mathcal{S}, \epsilon)$ where $C_{\text{search}}(\mathcal{S}, \epsilon) = O(\log \epsilon)$ for the binary search and $C_{\text{search}}(\mathcal{S}, \epsilon) = O(\epsilon)$ for the exponential search. The lookup complexity can be defined as:

$$T_{\text{lookup}}(M) = L_m \cdot O(1) + C_{\text{search}}(\mathcal{S}, \epsilon). \tag{7}$$

Insertion Complexity. An insertion first locates the target leaf data node via the lookup, then applies the configured update strategy within the leaf. The cost is dependent on the gap ratio r_g. For the dense layout ($r_g = 0$), inserting at position i requires shifting all subsequent keys to make space. For uniformly random insertions in a node with $|K|$ keys, the expected number of shifts is $|K|/2$:

$$T_{\text{insert}}(M) = T_{\text{lookup}}(M) + O(|K|/2) \tag{8}$$

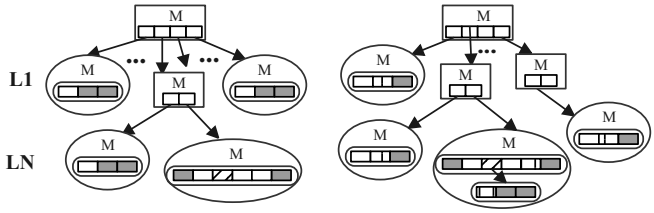
For the gapped layout with gap ratio $r_g > 0$ and the shift strategy, pre-allocated empty slots reduce the shift distance. The expected distance to the nearest available gap is $\mathbb{E}[\text{ gap-dist }] = \frac{1}{r_g}$:

$$T_{\text{insert}}(M) = T_{\text{lookup}}(M) + O\left(\frac{1}{r_g}\right). \tag{9}$$

4.5.2 Near-Optimum Analysis. Since the design space $\mathbb{M}$ is finite, the existence of a theoretical optimal index M^* is guaranteed. The theoretical optimal index M^* can be defined as

$$M^* = \arg\min_{M \in \mathbb{M}} \mathbb{E}_{w \sim \mathbb{W}}[\alpha \cdot T_{\text{lookup}}(M, w) + \beta \cdot T_{\text{insert}}(M, w)] \tag{10}$$

However, since the design space in database tuning is inherently large and finding the global optimum has been proven to be NP-hard in related cases [38, 39], directly verifying optimality is infeasible. To validate near-optimality, we reduce the configuration space and compare the index structure generated by SELIX against the optimal index structure obtained by brute-force search. As shown in Figure 4, both methods generate index structures on the OSM dataset under the read-only workload, exhibiting high structural similarity. Both produce trees of height 5 with comparable node counts: brute-force generates 3,980 model nodes and 65,845 data nodes, while SELIX generates 4,591 model nodes and 78,463 data nodes. However, brute-force concentrates approximately 62% of data nodes at Level 1, while SELIX concentrates 50.2%. Additionally, 72.5% of model nodes in the brute-force solution have fanouts in the range of 4-16, compared to 66.4% for SELIX. Despite these differences, brute-force achieves 3.32M ops/s while SELIX achieves 3.17M ops/s, a gap of only 4.8%. This demonstrates that SELIX can effectively discover near-optimal index structures in the design space.



(a) Brute-force Search. (b) SELIX

Fig. 4. Comparison of index structures generated by brute-force search and SELIX.

5 Implementation

We implement SELIX as an index library supporting both in-memory and on-disk modes, and integrate SELIX into PostgreSQL as a new index type.

Index Library Implementation. We implement SELIX using C++ and Python, and have made the source code available at [40]. SELIX index structure consists of model nodes and data nodes. Model nodes store model parameters and metadata (e.g., child-node addresses), while data nodes store actual data values. In in-memory mode, the entire index resides in main memory. Each node corresponds to a contiguous memory buffer that is allocated dynamically from a memory pool to reduce allocation overhead. In on-disk mode, model nodes and data nodes are stored in separate files so that model nodes are reached with fewer page reads, reducing I/O cost. The root model node is pinned in the first block to ensure correct access across structural modifications. Node capacities are aligned with page sizes (e.g., 8KB/16KB) to avoid partial page reads and minimize random I/O.

DBMS Integration. We integrate SELIX into PostgreSQL as a new index access method. PostgreSQL provides an extensible index framework that allows users to introduce new index types by defining a set of callback functions, including `ambuild`, `aminsert`, `amgettuple`, `amrescan`. We create an extension that implements the callbacks and registers SELIX as a new index type in the system catalog, enabling users to create and manage SELIX indexes on tables in the database via SQL commands. The workflow of using SELIX is detailed as follows:

1) Offline Pre-training. Before creating an index, users need to invoke '`SELECT` selix_pretrain (table_name, idx_column)' to trigger pre-training. The system collects workload statistics from the target table, including access patterns from `pg_stat_user_tables`, query statistics from `pg_stat_statements`, and key distributions from `pg_stats`. For newly created tables that lack historical statistics, the system first deploys a background monitor to collect workload information during actual query execution. Once sufficient data is gathered, the system constructs a workload generator and triggers pre-training. The agent generates data batches of different workloads and learns a generalized policy that enables rapid adaptation to diverse workloads.

2) Index Creation. When a user creates an index via '`CREATE INDEX` idx_column `ON` table_name `USING SELIX`', the system bulk-loads existing data and queries PostgreSQL's statistics views to obtain current workload characteristics. The pre-trained agent then generates an optimized initial index structure based on these characteristics. If no statistics are available, the agent generates a default structure and adapts online as queries arrive. 3) Online Adaptation. During execution, PostgreSQL runs a background worker that monitors SELIX index performance by collecting execution time (`mean_exec_time`) from `pg_stat_statements`. When degradation is detected, the online adaptation is automatically triggered. The pre-trained agent performs few-shot adaptation and generates an optimized index structure suited to the current workload. 4) Index Rebuilding. Once the optimal structure is determined, the system applies the new configuration by executing

'REINDEX INDEX idx_column', which rebuilds the SELIX index with the updated parameters. 5) Index Selection. We treat SELIX as a candidate index within the index selection framework [41]. To enable cost-aware selection, we provide a cost model to estimate the benefit and overhead of SELIX based on index structural properties. In particular, for a workload with read ratio r and write ratio w, the total cost is $T_{total} = r \cdot T_{lookup} + w \cdot T_{insert}$, according to the complexity analysis in Section 4.5. When multiple SELIX indexes are present, e.g., SELIX (T.A) and SELIX (T.A, T.B), the index selector compares their costs to retain the more efficient option. For example, under read-heavy workloads, we may keep the index with a lower depth for reduced traversal cost.

6 Evaluation

In this section, we evaluate SELIX by comparing it with state-of-the-art learned indexes in both in-memory and on-disk settings.

6.1 Experimental Setup

All experiments are conducted on a server equipped with an Intel(R) Xeon(R) W-1290P CPU @ 3.70 GHz (10 cores, 20 threads), 128 GB of memory, and 3 NVIDIA GeForce RTX 2080 Ti GPUs. The GPUs are solely used for RL agent training and inference, while all index operations in SELIX are executed on the CPU.

6.1.1 Datasets. We evaluate SELIX using 4 real-world datasets that are widely adopted in learned index benchmarks [8, 9, 42]: (1) **FB** [42], which contains Facebook user IDs. (2) **OSM** [42], which comprises 64-bit unsigned integer keys uniformly sampled from OpenStreetMap location data. (3) **BOOKS** [42], which consists of Amazon book popularity rankings derived from sales data. (4) **COVID** [9], which contains uniformly sampled Tweet IDs tagged with 'COVID-19'. We use OSM by default if not otherwise specified.

6.1.2 Workloads. We initialize each index by bulk-loading 50M keys, then execute 20M operations per workload. We define six workloads with varying read-write ratios: **(1) Read-Only (RO)** with 100% lookups, **(2) Read-Heavy (RH)** with 80%/20%, **(3) Balanced (BL)** with 50%/50%, **(4) Write-Heavy (WH)** with 20%/80%, and **(5) Write-Only (WO)** with 100% inserts. **(6) Scan** workload performs 20M range queries, each retrieving 100–500 consecutive keys. Following prior work [9, 43, 44], lookup and scan keys are drawn from a Zipfian distribution ($skew = 0.99$) to simulate hotspot access, while inserted keys are shuffled.

6.1.3 Training Setup. Before executing the workloads, SELIX was pre-trained following the meta-training paradigm described in Section 4.3, taking approximately 1.5 hours (1000 steps). The training tasks cover workloads with varying read-write ratios.

6.1.4 Baselines. We evaluate SELIX against state-of-the-art learned indexes under both on-disk and in-memory settings for mixed read-write workloads. In the on-disk setting, we compare with ALEX [4], LIPP [1], FITing-Tree [18], and PGM [2] using their disk-based implementations from [45]. As FITing-Tree and PGM lack multi-threaded support, all on-disk experiments run single-threaded. In the in-memory setting, we compare against ALEX+ and LIPP+, the concurrent variants of ALEX and LIPP provided by [9], as well as XIndex [46], FINEdex [47], STX B$^+$-Tree [45], Bw-Tree [48], and GENE [26]. Regarding Bw-Tree, we adopt the open-source implementation from [49]. All compared indexes adopt optimistic lock coupling (OLC) for concurrency control to ensure fair comparisons. Please note that SELIX performs structural modifications (SMOs) in all workloads with write operations, and the associated cost is reflected in the reported throughput.

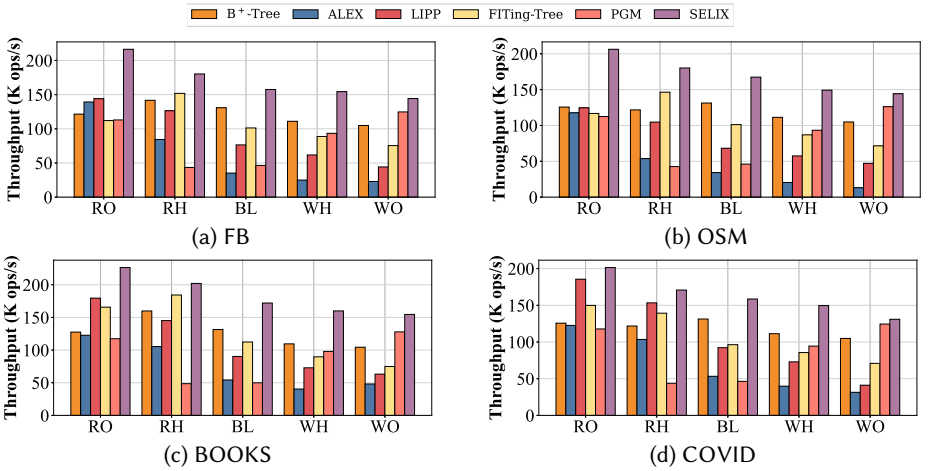


Fig. 5. Throughput under varying mixed workloads in the on-disk setting.

6.2 Overall Performance

6.2.1 Performance comparisons under the on-disk setting. We first evaluate the performance of different indexes under various workloads in the on-disk setting, and the results are presented in Figure 5. Under RO workloads, SELIX outperforms the best-performing baseline by up to 1.64× on the OSM dataset. Under RH workloads, the improvement over FITing-Tree reaches 1.23× on the OSM dataset. Under BL and WH workloads, SELIX surpasses B$^+$-Tree by up to 1.31× and 1.46× on the BOOKS dataset, respectively. Under WO workloads, SELIX achieves up to 1.21× higher throughput than PGM-index on the BOOKS dataset. As the write ratio increases from RO to WO, ALEX's throughput on the OSM dataset drops to 11.2% of its RO performance, and LIPP on the COVID dataset drops to 22.2%, while SELIX retains 64.9% of its RO throughput. This is because SELIX's adaptive structure generation mechanism configures appropriate gap ratios for leaf nodes based on workload characteristics. Under write-intensive workloads, it reserves more insertion space to reduce key conflicts and structural reorganization overhead, thereby maintaining read performance while improving write throughput.

6.2.2 Performance comparisons under the in-memory setting. To demonstrate the generality of the SELIX, we further extend our evaluation to the in-memory setting. In this experiment, we compare SELIX against concurrent versions of learned indexes, including B$^+$-Tree, Bw-Tree, ALEX+, LIPP+, XIndex, and FINEdex. Since FITing-Tree and PGM-index do not provide multi-threaded implementations, they are excluded from this comparison. All indexes are evaluated using 32 threads. As shown in Figure 6, SELIX consistently achieves the highest throughput across all workloads and datasets. On average, SELIX outperforms B$^+$-Tree, Bw-Tree, ALEX+, LIPP+, XIndex, and FINEdex by factors of 1.32×, 1.65×, 3.47×, 1.79×, 2.01×, and 3.46×, respectively. Under the write-only workload, SELIX achieves up to 1.42× improvement over the second-best baseline across all datasets.

6.2.3 Performance comparisons under varying scan workloads. We evaluate the range query performance of SELIX under both on-disk and in-memory settings, with scan ranges varying from 100 to 500 consecutive keys. Since LIPP+ does not support range scans [9], it is excluded from the in-memory experiments. As shown in Figure 7, a range scan first performs a lookup to locate the start key, then scans forward until reaching the end key, so lookup performance directly affects scan performance. As the scan range grows, the throughput of all indexes decreases due to the

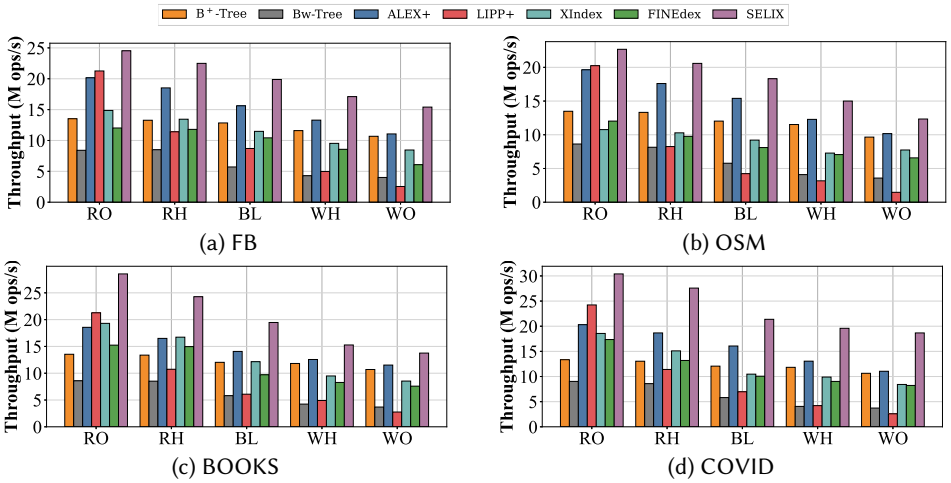


Fig. 6. Throughput under varying mixed workloads in the in-memory setting with 32 threads.

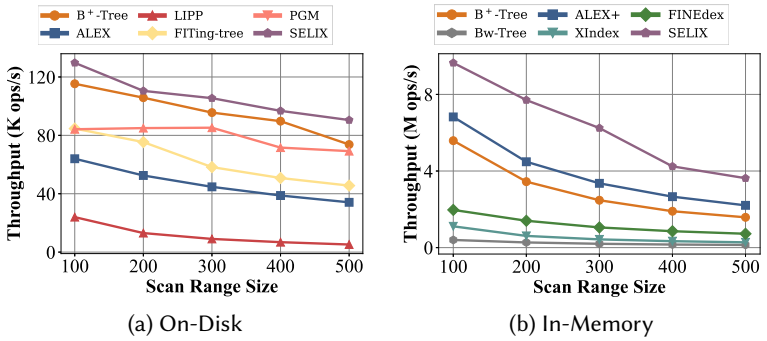


Fig. 7. Throughput under varying scan workloads.

increasing number of data nodes accessed, yet SELIX remains the best-performing method. Even at the largest scan size of 500, SELIX achieves 1.64× the throughput of ALEX in memory and 1.23× that of B-tree on disk, owing to its compact data node layout that packs keys densely within nodes and reduces node traversals during forward scans. In the on-disk setting, LIPP exhibits the lowest throughput, as it uses a single node type that resolves key collisions by inserting child nodes at the conflicting position. This forces forward scans to repeatedly descend into child nodes and return to the parent before continuing, turning sequential scans into frequent random I/O across levels.

6.2.4 Performance comparisons under varying thread counts. We evaluate the scalability of different learned indexes on the OSM dataset. As shown in Figure 8d, SELIX achieves the best scalability among all baselines, delivering up to 1.94× higher throughput than the second-best method. The improvement stems from SELIX's training that produces indexes adaptive to mixed operations in workloads at the fine-grained node level. Further, we observe that LIPP+ attains the highest throughput under read-only workloads, yet fails to maintain the same scalability as other baselines under write-heavy workloads. This limitation arises from its structural design, where each node maintains update statistics and confines updates to leaf nodes. Consequently, every insertion operation must update node-level statistics along its path, causing contention and cacheline ping-pong effects, particularly near the root. In contrast, SELIX eliminates such constraints by enabling

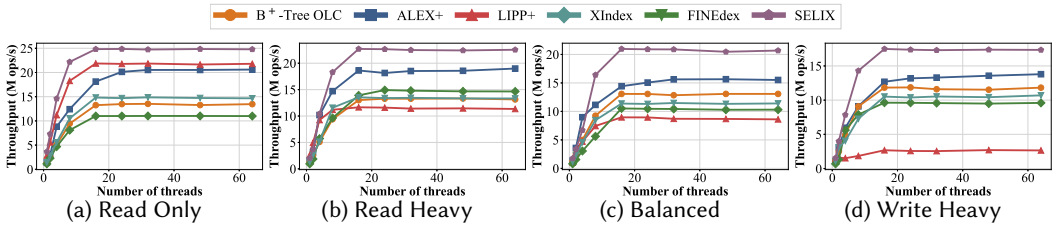


Fig. 8. Scalability under varying workloads.

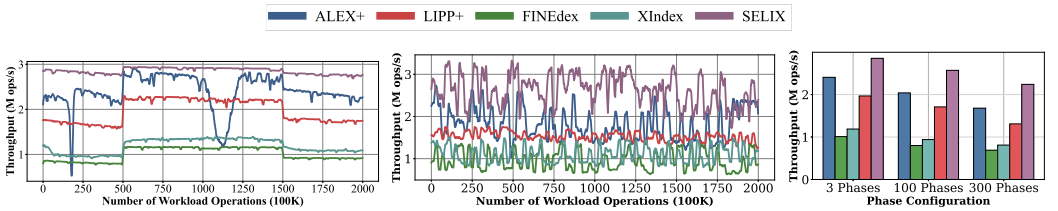


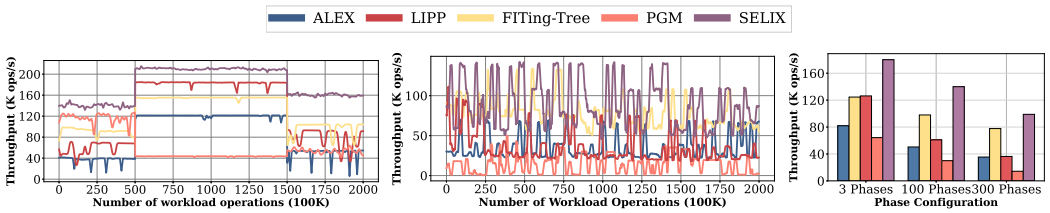
Fig. 9. Performance comparison under dynamic workloads (In-Memory).

localized structure adaptation, thereby maintaining superior scalability across both read-heavy and write-heavy workloads.

6.2.5 Performance under dynamic workloads. We evaluate the robustness of SELIX under dynamic workloads. The workload proceeds through three sequential phases: a write-heavy phase, a read-heavy phase, and a balanced phase, consisting of 50M, 100M, and 50M operations, respectively. As shown in Figure 9a and Figure 10a, SELIX maintains the highest and most stable throughput across all three phases under both settings. In the write-heavy phase, ALEX suffers sharp throughput drops due to frequent node splitting and merging, while SELIX maintains stable performance by selecting a lower fill rate that reserves sufficient gap space for insertions. In the on-disk setting, PGM experiences a pronounced performance drop during the read-heavy phase, as its LSM-inspired design accumulates multiple static segments during the preceding write-heavy phase, increasing lookup I/O overhead. These results demonstrate that SELIX's RL-driven adaptation mechanism can effectively determine both the structural configuration and the timing of adjustments, avoiding excessive rebuilding and achieving robust performance under workload shifts.

We evaluate the robustness of SELIX under fast-changing dynamic workloads. We employ a dynamic workload consisting of 100 phases, each containing 2M operations with read/write ratios uniformly sampled from $[0, 1]$. As shown in Figure 10, the throughput of all indexes fluctuates significantly with the workload drift. Although SELIX occasionally exhibits lower performance than other indexes at certain moments, it recovers quickly after adjustment without performance collapse. This robustness benefits from the SELIX's RL-driven structural optimization, which minimizes adjustment overhead under rapid workload drift.

We control the workload switching frequency by varying the number of phases while keeping the total number of operations constant at 200M. More phases result in shorter per-phase duration and more frequent workload transitions, imposing higher demands on the index's adaptation capability. From Figure 9c and Figure 10c, we observe that as the switching frequency increases, the throughput of all indexes degrades due to more frequent workload transitions. Even in the most extreme case with 300 phases, SELIX still achieves 1.33× higher throughput than ALEX+ in memory and 1.27× higher than FITing-Tree on disk.



(a) Three-Phase Dynamic Workload (b) Fast-Changing Dynamic Workload (c) Switching Frequency

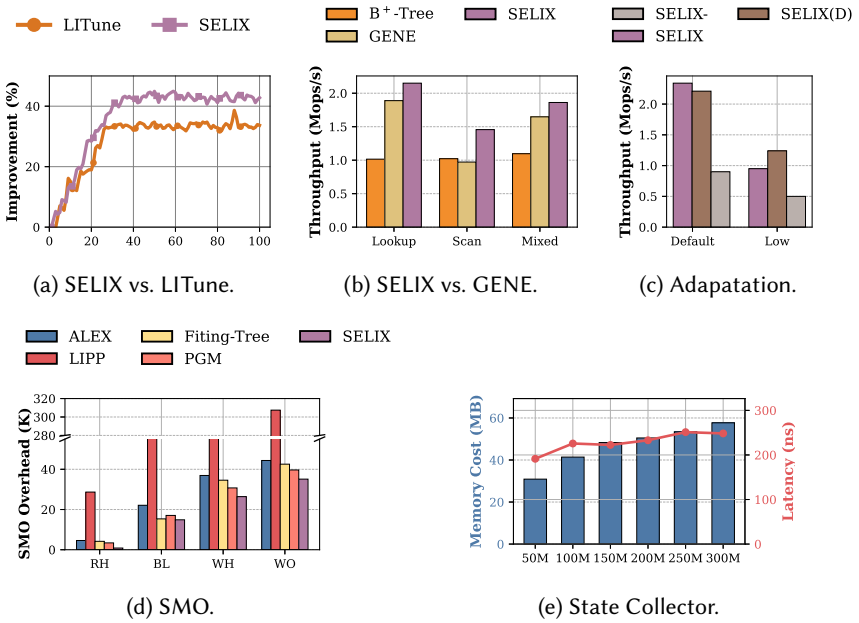
Fig. 10. Performance comparison under dynamic workloads (On-Disk).

6.3 Micro Benchmarks

6.3.1 Comparison with State-of-the-art Tuning Approach. We further compare SELIX with LITune, an existing RL-based learned index tuning framework [14]. To ensure fairness, both methods start from a common index structure and perform online fine-tuning on a balanced workload. As shown in Figure 11a, we report the throughput-based performance improvement at each tuning episode, which accounts for both operation efficiency and reorganization overhead. The results show that both methods discover optimized configurations within a small number of tuning steps, but SELIX achieves a greater improvement. This advantage is primarily because SELIX adopts a fine-grained local adjustment strategy, where each action targets a selected sub-node, enabling targeted optimizations. In contrast, LITune performs global-level tuning, identifying optimal configurations only at the overall index level. This coarse-grained approach may overlook potential optimization opportunities within local structures. Moreover, SELIX employs an on-policy RL algorithm that continuously refines its strategy based on recent interactions, enabling more stable convergence and higher adaptability during dynamic optimization.

6.3.2 Comparison with General Index Frameworks. We compare SELIX with GENE [26], a state-of-the-art general-purpose index framework. As GENE is optimized for read operations, we evaluate point queries, range queries, and mixed queries. The mixed workload follows GENE's 80/20 query ratio over a three-segment key domain (0–10%, 10–85%, 85–100%), with point lookups in the left/right segments and mixed point-range queries in the middle. All methods bulk-load 100M keys and execute 10M operations per workload. As shown in Figure 11b, while GENE selects traditional index structures based on workloads, SELIX consistently generates learned index structures and outperforms GENE across all workloads. For point queries, GENE generates a hash index, while SELIX achieves 1.25× speedup. For range queries, GENE generates a pure B^+-tree since hash does not support range scans, while SELIX achieves 1.43× speedup. For mixed workloads, GENE generates a hybrid structure combining B^+-tree and hash index, while SELIX achieves 1.27× speedup.

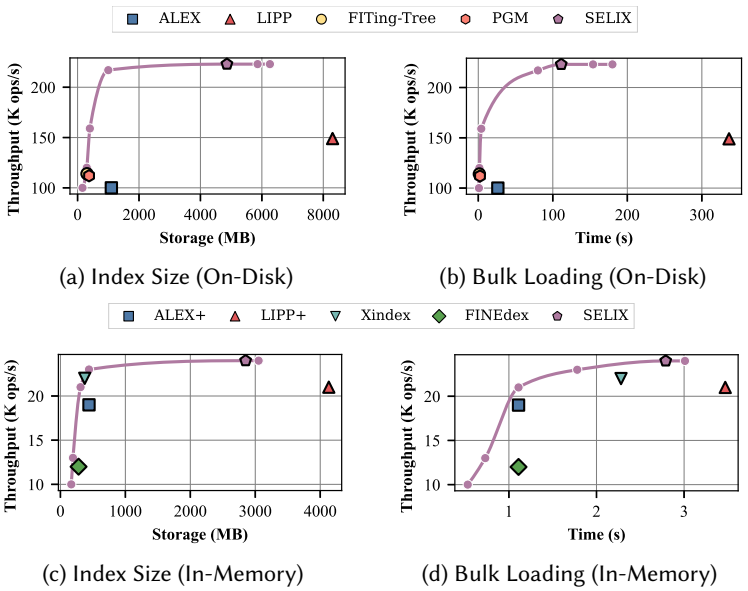
6.3.3 SELIX Design Space. We build each learned index by bulk-loading 50 million keys from the OSM dataset and measuring both construction time and storage overhead under on-disk and in-memory settings. Based on the constructed indexes, we further execute read-only workloads containing 20 million queries to assess their performance. As shown in Figure 12, the horizontal axes represent the construction time and storage overhead of each index, while the vertical axis indicates the query throughput under the evaluated workloads. From the perspective of Pareto optimality, the relationship among construction time, storage overhead, and throughput forms a classical multi-objective trade-off surface. SELIX approaches the frontier by maintaining high throughput with moderate construction time and compact index size. This demonstrates that SELIX effectively navigates on a superior tradeoff on index performance and cost in both in-memory and on-disk settings, while we prioritize achieving high throughput here.



(a) SELIX vs. LITune. (b) SELIX vs. GENE. (c) Adapatation.

(d) SMO. (e) State Collector.

Fig. 11. Micro benchmarks.



(a) Index Size (On-Disk) (b) Bulk Loading (On-Disk)

(c) Index Size (In-Memory) (d) Bulk Loading (In-Memory)

Fig. 12. Storage cost and bulk loading time analysis.

6.3.4 In-depth Analysis of SELIX Design. We analyze access statistics on the OSM dataset under the balanced workload. Table 3 compares node count per query, prediction error, and insertion conflict count. SELIX achieves the fewest node accesses under both settings (2.14 on disk and 2.5 in memory) and the lowest prediction error (27 on disk and 19 in memory). Notably, SELIX exhibits fewer node accesses but higher prediction error on disk, while the opposite holds in memory, because disk I/O latency drives SELIX to favor larger nodes and lower depth, whereas in-memory

Table 3. Fine-grained statistics under the balanced workload.

Metric	On-Disk Setting					In-Memory Setting		
	ALEX	LIPP	FITing-Tree	PGM	**SELIX**	ALEX+	LIPP+	**SELIX**
Node Count	3	2.4	3.23	4.30	2.14	3.4	2.7	2.5
Prediction Error	48	N/A	32	67	27	73	N/A	19
Conflict Count (K)	76.95	163.63	N/A	N/A	69	81.62	152.01	57

Table 4. SELIX-generated structures on OSM.

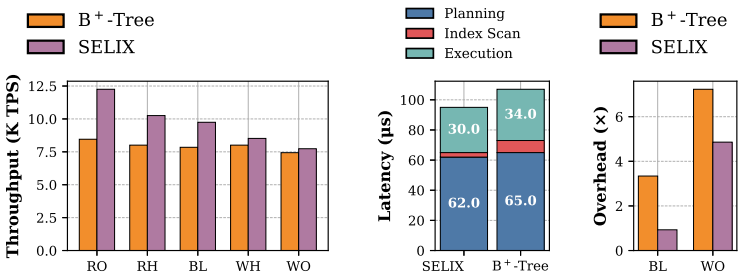
Setting	H	Fill	#Models	#Data	L1 (M/D)	L2 (M/D)	AvgKeys
Balanced	5	35.43%	6,721	82,650	4,451 / 36,787	2,176 / 38,470	604.96
Read-Only	3	60%	4,215	63,958	3,133 / 34,767	1,046 / 25,845	799.95

cache efficiency shifts optimization toward reducing prediction error. This demonstrates that SELIX can autonomously adapt to different storage environments. For insertions, SELIX incurs the fewest conflicts (69K on disk and 57K in memory), significantly lower than LIPP (163.63K) and LIPP+ (152.01K), due to its learned gap space allocation that reduces structural reorganizations.

6.3.5 Analysis of Structural Adjustments. We further analyze structure modification operations (SMOs) during execution. We evaluate on the OSM dataset under four workloads (RH, BL, WH, and WO) on disk, and exclude RO since read-only workloads do not trigger SMOs. As shown in Figure 11d, SELIX achieves the lowest SMO frequency across all workloads, with 851, 14,891, 26,401, and 35,112 SMOs respectively. This is because SELIX uses agent to find optimal node configurations that match the current workload, reducing slot collisions and minimizing reorganization overhead. In contrast, LIPP triggers an SMO upon every slot collision, resulting in the highest frequency.

6.3.6 Memory Overhead of State Collector. The state collector maintains lightweight per-node metadata for ranking purposes. For each node, the collector tracks statistics as described in Section 4.2.2, requiring approximately 189 bytes per node in our implementation. Figure 11e reports the memory usage of the state collector under different data sizes. As the system runs continuously and the index grows from 50M to 300M keys, the number of nodes increases from 167K to 362K, and the memory usage grows from 31MB to 58MB, accounting for less than 3% of the total index memory consumption. These results demonstrate that the state collector introduces acceptable overhead.

6.3.7 Analysis of Adaptation Triggers. We study the impact of different metrics that trigger the adaptation process. We compare SELIX's default throughput-based trigger with two variants: SELIX-, which uses a fixed policy without online adaptation, and SELIX (D), which triggers adaptation upon detecting workload distribution drift. We conduct evaluations under both the default stress-testing condition and a low-load scenario, simulated by reducing the operation arrival rate. For both scenarios, we drift the workloads from balanced to write-heavy. As shown in Figure 11c, under the default setting, SELIX achieves 1.13× higher throughput than SELIX (D). The throughput-based trigger only responds when performance actually degrades, making adjustments more targeted. In contrast, the distribution-based trigger may trigger adjustments prematurely and incurs additional monitoring overhead. Under the low-load scenario, however, SELIX (D) achieves 1.27× higher throughput than SELIX, since the distribution-based trigger more accurately captures performance degradation when throughput remains stable due to low system pressure. Both SELIX and SELIX (D) outperform SELIX-, achieving 2.42× and 2.25× throughput, respectively. Based on these results, we adopt throughput as the default trigger for stress tests. For low-load scenarios, we recommend switching to the distribution-based trigger.



(a) End-to-end performance (b) Breakdown (c) Overhead

Fig. 13. End-to-end evaluation of SELIX on PostgreSQL.

6.3.8 Case Study. We use the OSM dataset to illustrate how SELIX generates index structures tailored to specific workloads in memory, with detailed parameters in Table 4. For the read-only workload, SELIX generates a structure with a height of 3, containing 4,215 model nodes and 63,958 data nodes, with a fill rate of 60% and an average of 799.95 keys per data node. The majority of data nodes use a gapped array layout with exponential search, averaging 1.54 iterations per lookup. For the balanced workload, SELIX generates a structure with a height of 5, containing 6,721 model nodes and 82,650 data nodes, with a fill rate of 35.43% and an average of 604.96 keys per data node. In terms of data node distribution, under the read-only workload, SELIX concentrates 55.6% of nodes at L1, compared to 46.1% under the balanced workload, forming a shallower structure that minimizes traversal depth and optimizes lookup efficiency. In terms of fill rate, SELIX uses a higher fill rate under the read-only workload (60% vs. 35.43%), resulting in more keys per node (799.95 vs. 604.96) and thus a lower tree height. In contrast, under the balanced workload, SELIX reserves more gaps within data nodes to accommodate future insertions, trading lookup efficiency for write flexibility. These differences demonstrate SELIX's ability to adapt index structures to workload characteristics.

6.4 End-to-end Performance of SELIX

We integrate SELIX into PostgreSQL and conduct an end-to-end performance comparison against the native B$^+$-tree index. We use the OSM dataset, bulk-loading 50M rows into a table with schema (id INT, val BIGINT) and building indexes on the val column. We evaluate five representative workloads as defined in Section 6.1.2, with 1M transactions (each with one operation) executed per workload. To eliminate the impact of concurrency and ensure index usage, we disable parallel execution and sequential scans in PostgreSQL. As shown in Figure 13a, SELIX outperforms B$^+$-tree across all workloads, achieving 1.45× and 1.28× throughput for read-only and read-heavy workloads, and 1.24×, 1.06×, and 1.04× for balanced, write-heavy, and write-only workloads, respectively. In addition, we observe a smaller performance gap compared to Figure 5, where only the index component is evaluated. This is due to the fact that end-to-end query execution in a DBMS involves additional overheads, such as query planning and tuple access, that diminish the overall speedup. To confirm this, we perform a time breakdown analysis in Figure 13b. As observed, although total query latency includes multiple stages, SELIX reduces index access time. These results demonstrate the applicability of SELIX in real DBMSs.

Maintenance Overhead. We evaluate maintenance overhead under varying write pressures in PostgreSQL using the C3 suite (Update Workloads) from benchmark tool [50, 51]. We compare SELIX against B$^+$-Tree under moderate (BL, 50% writes) and high (WO, 100% writes) write intensities on a table with 50M rows. The overhead is defined as the ratio of indexed execution time to no-index

execution time. As shown in Figure 13c, under low write pressure, B$^+$-Tree and SELIX incur 3.5× and 1.3× overhead respectively. Under high write pressure, B$^+$-Tree's maintenance overhead increases sharply to 7.6×, while SELIX maintains only 5.2× overhead due to its lightweight maintenance mechanism. These results demonstrate that SELIX consistently incurs lower maintenance overhead across different write intensities.

7 Conclusion

This paper presented SELIX, a self-designing learned index that autonomously generates and optimizes its structure in response to dynamic workloads. We introduced a unified index template that enables flexible composition of node layouts, partitioning strategies, and search methods for fine-grained structural design. To achieve efficient structural adaptation, we formulated index optimization as a learnable function and trained it through a meta-enhanced DRL framework with an online update strategy for fast and stable adaptation under workload drift. Extensive experimental results demonstrate that SELIX outperforms state-of-the-art in-memory and on-disk learned indexes in both throughput and robustness.

Acknowledgments

We would like to thank the anonymous reviewers and meta-reviewer for their constructive comments, which help us substantially improve the paper. This work is partially supported by NUS Faculty Development Fund.

References

[1] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[2] 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 (2020), 1162–1175.

[3] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[4] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.

[5] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-Disk Updatable Learned Index. In *ICDE*. IEEE, 4856–4869.

[6] Yuxin Yang, Fang Wang, Mengya Lei, Peng Zhang, and Dan Feng. 2025. ALT-Index: A Hybrid Learned Index for Concurrent Memory Database Systems. In *ICDE*. IEEE, 86–98.

[7] Elisa Bertino and Beng Chin Ooi. 1999. The Indispensability of Dispensable Indexes. *IEEE Trans. Knowl. Data Eng.* 11, 1 (1999), 17–27. doi:10.1109/69.755611

[8] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004.

[9] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.

[10] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *ICML (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3123–3132.

[11] Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. 2025. Why Are Learned Indexes So Effective but Sometimes Ineffective? *Proc. VLDB Endow.* 18, 9 (2025), 2886–2898.

[12] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3 (2024), 151.

[13] Yuanhui Luo, Minhui Xie, Yiheng Tong, Shichao Jiang, and Yunpeng Chai. 2025. Understanding Robustness Issues of Updatable Learned Indexes: [Experiments & Analysis]. *Proc. ACM Manag. Data* 3, 4, Article 270 (2025), 25 pages.

[14] Taiyi Wang, Liang Liang, Guang Yang, Thomas Heinis, and Eiko Yoneki. 2025. A New Paradigm in Tuning Learned Indexes: A Reinforcement Learning Enhanced Approach. *Proc. ACM Manag. Data* 3, 3 (2025), 120:1–120:26.

[15] James Queeney, Yannis Paschalidis, and Christos G. Cassandras. 2021. Generalized Proximal Policy Optimization with Sample Reuse. In *NeurIPS*. 11909–11919.

[16] Beng Chin Ooi, Shaofeng Cai, Gang Chen, Yanyan Shen, Kian-Lee Tan, Yuncheng Wu, Xiaokui Xiao, Naili Xing, Cong Yue, Lingze Zeng, Meihui Zhang, and Zhanhao Zhao. 2024. NeurDB: an AI-powered autonomous data system. *Sci. China Inf. Sci.* 67, 10 (2024).

[17] Zhanhao Zhao, Shaofeng Cai, Haotian Gao, Hexiang Pan, Siqi Xiang, Naili Xing, Gang Chen, Beng Chin Ooi, Yanyan Shen, Yuncheng Wu, and Meihui Zhang. 2025. NeurDB: On the Design and Implementation of an AI-powered Autonomous Database. In *CIDR*. www.cidrdb.org.

[18] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD Conference*. ACM, 1189–1206.

[19] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The Concurrent Learned Indexes for Multicore Data Storage. *ACM Trans. Storage* 18, 1 (2022), 8:1–8:35.

[20] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334.

[21] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.

[22] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (2022), 2188–2200.

[23] Zhanhao Zhao, Haotian Gao, Naili Xing, Lingze Zeng, Meihui Zhang, Gang Chen, Manuel Rigger, and Beng Chin Ooi. 2025. NeurBench: Benchmarking Learned Database Components with Data and Workload Drift Modeling. *arXiv preprint arXiv:2503.13822* (2025).

[24] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB*. Morgan Kaufmann, 562–573.

[25] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 535–550. doi:10.1145/3183713.3199671

[26] Jens Dittrich, Joris Nix, and Christian Schön. 2021. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. *Proc. VLDB Endow.* 15, 3 (2021), 527–540. doi:10.14778/3494124.3494136

[27] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[28] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 665–676. doi:10.1109/ICDE.2018.00066

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* 518, 7540 (2015), 529–533.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017).

[32] Qingpeng Cai, Can Cui, Yiyuan Xiong, Wei Wang, Zhongle Xie, and Meihui Zhang. 2023. A Survey on Deep Reinforcement Learning for Data Processing and Analytics. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4446–4465. doi:10.1109/TKDE.2022.3155196

[33] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[34] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD Conference*. ACM, 931–944.

[35] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD Conference*. ACM, 415–432.

[36] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *ICLR (Poster)*.

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*. 3104–3112.

[38] Immanuel Trummer. 2021. The Case for NLP-Enhanced Database Tuning: Towards Tuning Tools that "Read the Manual". *Proc. VLDB Endow.* 14, 7 (2021), 1159–1165. doi:10.14778/3450980.3450984

[39] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2025. Wii: Dynamic Budget Reallocation In Index Tuning. *CoRR* abs/2505.02312 (2025). doi:10.48550/ARXIV.2505.02312 arXiv:2505.02312

[40] SELIX. 2025. *SELIX Implementation.* https://github.com/neurdb/selix.git

[41] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[42] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[43] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. 2025. Towards Optimizing Learned Index for High Performance, Memory Efficiency and NUMA Awareness. *ACM Trans. Archit. Code Optim.* 22, 3 (2025), 109:1–109:26.

[44] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[45] 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 (2023), 139:1–139:22.

[46] 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 *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 308–320. doi:10.1145/3332466.3374547

[47] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proc. ACM Manag. Data* 1, 4 (2023), 258:1–258:25.

[48] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. doi:10.1109/ICDE.2013.6544834

[49] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 473–488. doi:10.1145/3183713.3196895

[50] Ivo Jimenez, Jeff LeFevre, Neoklis Polyzotis, Huascar Sanchez, and Karl Schnaitter. 2011. Benchmarking Online Index-Tuning Algorithms. *IEEE Data Eng. Bull.* 34, 4 (2011), 28–35. http://sites.computer.org/debull/A11dec/online.pdf

[51] Karl Schnaitter and Neoklis Polyzotis. 2009. A Benchmark for Online Index Selection. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 1701–1708. doi:10.1109/ICDE.2009.166

Received 18 October 2025; revised 5 January 2026; accepted 24 February 2026