HIRE: A Hybrid Learned Index for Robust and Efficient Performance under Mixed Workloads

XINYI ZHANG, Hong Kong Baptist University, Hong Kong SAR
LIANG LIANG, EPFL, Switzerland
ANASTASIA AILAMAKI, EPFL, Switzerland
JIANLIANG XU, Hong Kong Baptist University, Hong Kong SAR

Indexes are critical for efficient data retrieval and updates in modern databases. Recent advances in machine learning have led to the development of learned indexes, which model the cumulative distribution function of data to predict search positions and accelerate query processing. While learned indexes substantially outperform traditional structures for point lookups, they often suffer from high tail latency, suboptimal range query performance, and inconsistent effectiveness across diverse workloads. To address these challenges, this paper proposes HIRE, a hybrid in-memory index structure designed to deliver efficient performance consistently. HIRE combines the structural and performance robustness of traditional indexes with the predictive power of model-based prediction to reduce search overhead while maintaining worst-case stability. Specifically, it employs (1) hybrid leaf nodes adaptive to varying data distributions and workloads, (2) model-accelerated internal nodes augmented by log-based updates for efficient updates, (3) a non-blocking, cost-driven recalibration mechanism for dynamic data, and (4) an inter-level optimized bulk-loading algorithm accounting for leaf and internal-node errors. Experimental results on multiple real-world datasets demonstrate that HIRE outperforms both state-of-the-art learned indexes and traditional structures in range-query throughput, tail latency, and overall stability. Compared to state-of-the-art learned indexes and traditional indexes, HIRE achieves up to 41.7× higher throughput under mixed workloads, reduces tail latency by up to 98% across varying scenarios.

CCS Concepts: • **Information systems** → **Point lookups**; **Unidimensional range search**; • **Theory of computation** → *Data structures and algorithms for data management*.

Additional Key Words and Phrases: Learned indexes, machine learning for databases management, query processing

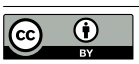
ACM Reference Format:
Xinyi Zhang, Liang Liang, Anastasia Ailamaki, and Jianliang Xu. 2026. HIRE: A Hybrid Learned Index for Robust and Efficient Performance under Mixed Workloads. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 43 (February 2026), 25 pages. https://doi.org/10.1145/3786657

1 Introduction

Indexes are essential components of modern DBMSs, and their efficiency impacts overall database performance for both retrieval and update operations [29]. The rapid advancement of machine learning has driven the emergence of learned indexes, which integrate predictive models with traditional indexing techniques [16].

Learned indexes employ approximation models to learn the cumulative distribution function (CDF) of data, predicting data locations rather than traversing fixed algorithmic paths. Early studies

Authors' Contact Information: Xinyi Zhang, Hong Kong Baptist University, Hong Kong SAR, csxyzhang@comp.hkbu.edu.hk; Liang Liang, EPFL, Switzerland, liang.liang@epfl.ch; Anastasia Ailamaki, EPFL, Switzerland, anastasia.ailamaki@epfl.ch; Jianliang Xu, Hong Kong Baptist University, Hong Kong SAR, xujl@comp.hkbu.edu.hk.



ACM 2836-6573/2026/2-ART43
https://doi.org/10.1145/3786657

demonstrated exceptional search performance, suggesting their potential to replace traditional index structures [13, 15, 30]. However, maintaining updatability remains a significant challenge. Dynamic data updates can induce model drift, degrading prediction accuracy or necessitating computationally expensive retraining, thereby reducing the efficiency of learned indexes.

To address these challenges, updatable learned indexes, such as FIT-ting Tree [11], ALEX [8], PGM [10], and LIPP [37], have been proposed, accompanied by extensive analyses [9, 22, 23, 31, 36], optimizations [3, 5, 33, 41, 42, 45], and applications [1, 2, 18, 21, 28, 38]. However, while several studies suggest that updatable learned indexes are "almost ready" for separated update and query workloads [36], our experimental results reveal that learned indexes still exhibit high tail latency and inefficient range query performance under mixed workloads. As shown in Figure 1, a comparison of state-of-the-art learned indexes with B+-tree under a mixed workload highlights three key limitations: (1) *inconsistent performance* across different workloads; (2) *suboptimal performance for range queries*; and (3) *unstable query/update latency*. A detailed discussion of these limitations is provided in Section 2.

The primary reason for these limitations is that *updatable learned index designs often prioritize optimal performance for point queries under specific conditions, at the cost of worst-case inefficiency and instability.* For example, PGM [10] employs an index-level buffer to mitigate model drift during insertions, which renders range queries highly inefficient due to the need to search multiple trees. Similarly, LIPP [37] sacrifices data locality and balanced structure to enhance model accuracy, achieving low-cost point lookups and insertions but compromising range query performance.

In this paper, we address the performance inefficiency and instability of updatable learned indexes to achieve robust and efficient performance under mixed workloads involving range queries. To this end, we propose HIRE, a hybrid in-memory index that integrates the stability of traditional indexes, such as B+-tree, with the performance advantages of learned indexes that leverage data CDFs to accelerate searches.

Specifically, HIRE employs a balanced-tree framework to ensure consistent performance across diverse workloads. At the leaf layer, we design two types of leaf nodes to maintain robust performance under varying conditions. At the internal layers, we introduce model-accelerated internal nodes, augmented by a log-based update mechanism that reduces frequent movements of keys and child node pointers. This design enhances query speed while minimizing update overhead. Furthermore, to mitigate high tail latency during retraining, HIRE adopts a non-blocking, cost-driven approach that combines multi-threaded concurrent operations with recalibration. For index construction, we propose an inter-level optimized bulk-loading algorithm that optimizes errors in multiple layers, enabling better post-construction performance. Compared to state-of-the-art updatable learned indexes and traditional indexes, HIRE demonstrates superior performance across diverse mixed workloads.

Our main contributions are summarized as follows:

- We analyze state-of-the-art updatable learned indexes and identify their trade-offs through experiments. Our analysis reveals that current designs often result in poor data locality and unbalanced structures, which degrade range query performance and increase tail latency versus traditional indexes.
- We propose HIRE, a novel in-memory index to address the performance inefficiency and instability of updatable learned indexes under mixed workloads involving range queries. To ensure high query efficiency while optimizing update and tail latency, HIRE incorporates several key components: a hybrid leaf node structure, model-accelerated internal nodes with log-based updates, a non-blocking, cost-driven recalibration algorithm, and a bulk-loading construction process optimized for low error.
- We implement a prototype of HIRE and conduct extensive experiments on multiple datasets.

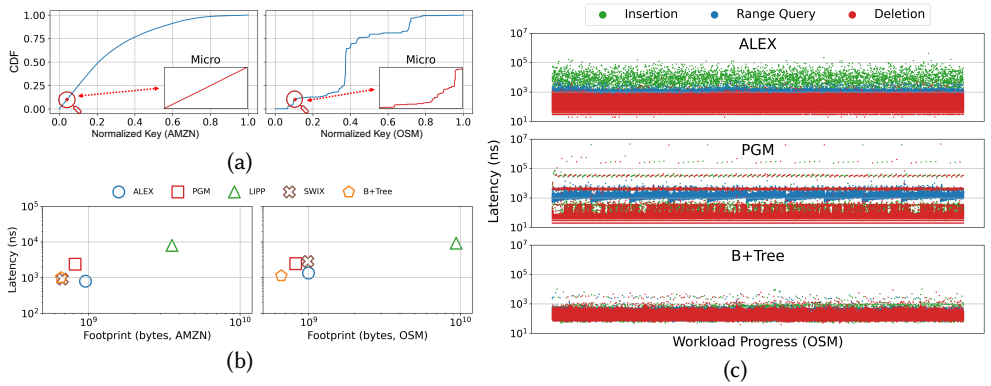


Fig. 1. Illustration of performance limitations of existing learned indexes under a balanced mixed workload (Query : Insert : Delete = 1:1:1). (a): Distributions of two SOSD datasets. (b): Performance comparisons of different indexes. (c): Latencies of insertion, deletion, and range query operations on the OSM dataset.

Compared to state-of-the-art learned indexes and traditional indexes, HIRE achieves significant performance improvements in range query efficiency and latency stability. Specifically, HIRE delivers up to 41.7× higher throughput on mixed workloads, while reducing tail latency by up to 98%. Moreover, HIRE consistently demonstrates robust performance across all evaluated scenarios.

The rest of the paper proceeds as follows. We present the background and motivation of our work in Section 2. We introduce our novel HIRE index in Section 3 and discuss its operations in Section 4. The evaluation results are shown in Section 5. Finally, we review the related work in Section 6 and conclude the paper in Section 7.

2 Background and Motivation

Consider a table $\mathcal{T}$, where each data entry $o_i \in \mathcal{T}$ is represented as a tuple $\langle k_i, v_i \rangle$, with $k_i \in \mathbb{R}$ as the key and v_i as the associated value. To retrieve the entry o_i, we search for k_i by locating its position p_{k_i} in the storage structure. This search can be accelerated by indexes. To support efficient range queries, the indexed k_i's are stored in monotonically increasing order (i.e., $\forall k_m < k_n,\ p_{k_m} < p_{k_n}$). Thus, p_{k_i} can be represented as an integer that increases monotonically with k_i. The index serves as a mapping between the search key k_i and its storage location p_{k_i}.

In contrast to traditional indexes like B+-tree, learned indexes employ machine learning models to approximate the data distribution and predict key locations [16]. Specifically, a model F learns the cumulative distribution function (CDF) of the indexed keys, enabling it to predict a search bound for a given key. Traditional indexes, such as B+-tree, use a parameter-based search bound, such as the fanout f, to guide searches. In contrast, learned indexes rely on an error-based search bound ϵ, representing the potential deviation between the predicted and actual key location. This deviation requires a "last-mile search" to locate the key precisely. Crucially, the model F must be strictly monotonic (i.e., $\forall k_m < k_n,\ F(k_m) < F(k_n)$). Without strict monotonicity, the last-mile search range could span the entire table $\mathcal{T}$, rather than being bounded by ϵ.

The predictive model is central to the design of learned indexes. For static data, a complex mapping function can effectively map keys to their positions with a small ϵ, achieving near-constant-time lookups. However, dynamic operations, such as insertions and deletions, disrupt the sorted key order, causing ϵ to grow linearly with the number of updates. As updates accumulate, the prediction error grows and degrades model accuracy. Therefore, naive designs (e.g., simply using the model in the leaf nodes of the B+-tree) are ineffective unless additional mechanisms are introduced to handle

Index	Structure	Range Query		Insertion	Deletion	Recalibration	
		Point Lookup	*Range Scan*			*Local*	*Global*
ALEX [8]	DAG	Hierarchical Model with Correction	Sequential Scan with Skip	Data-level Buffer	Mask	Merge, Split, Retrain	Top-down
PGM [10]	Mult. Balanced Tree	Hierarchical Model with Correction Tree-buffer Search	Sequential Scan with Random Scan	Index-level buffer	Mask	Merge, Retrain	Bottom-up
LIPP [37]	Imbalanced Tree	Hierarchical Model	Random Scan with Sequential Scan	Inplace	Mask	Split, Retrain	Top-down
SWIX [21]	Two-level Queue	Two-level Model with Correction	Sequential Scan with Skip	Data-level Buffer Node-level Buffer	Mask	Merge, Split, Retrain	Bottom-up
B+-tree [6]	Balanced Tree	Hierarchical Search	Sequential Scan	Inplace	Inplace	Merge, Split	Bottom-up
HIRE	Balanced Tree	Hierarchical Hybrid Search	Hybrid Sequential Scan	Data-level Buffer Inplace	Mask Inplace	Non-blocking Recalibration	Subtree Replacement

Table 1. Comparison of Index Designs

updates. Retraining the mapping function is computationally expensive, making it unsuitable for dynamic updates. Consequently, updatable learned indexes typically employ simpler piecewise linear models to construct a hierarchical structure relating keys and their positions, sacrificing some accuracy for faster updates. To mitigate the impact of dynamic operations, these indexes incorporate strategies such as update mitigation techniques, recalibration methods, and structural supports (summarized in Table 1). These design choices reflect a trade-off: *learned indexes prioritize fast lookups, often at the expense of degraded worst-case performance under highly dynamic workloads.* We discuss representative designs below:

- ALEX [8], a pioneering updatable learned index, employs linear interpolation to approximate the data distribution. This linearization enhances the generalization of the linear model and reduces the search space to a reasonable ϵ. The interpolated space also accommodates insertions without significantly affecting model accuracy. However, linearization reduces data locality. When the data distribution is highly non-linear and cannot be approximated by a single linear model within the error bound ϵ, ALEX recursively partitions the data, using multiple linear models. This top-down partitioning may result in an unbalanced index structure. While ALEX excels in point queries and updates in dynamic settings, its suboptimal data locality and potential for an unbalanced structure impair range query performance and introduce instability in operational latency.
- PGM [10] uses piecewise linear models to segment data and constructs an internal node layer from these segments. However, its update mechanism adopts a semi-dynamic approach, buffering updates before periodically merging them into existing segments, which triggers retraining. This two-stage approach requires searching both the buffer and data segments during queries, impairing data locality. Consequently, PGM prioritizes read-heavy workloads, achieving superior average query and update performance in such scenarios. However, this results in higher query and update tail latency in write-intensive environments.
- LIPP [37] aims for perfect precision by reducing ϵ to zero. However, it requires top-down node splitting for non-linear data distributions, which may result in a highly unbalanced structure. LIPP mitigates the impact of insertions on model accuracy by reserving extra space within nodes. While this enables excellent point query and insertion performance, it significantly compromises range query efficiency and introduces high tail latency.

In summary, while existing methods can constrain model error within a predefined threshold, they often compromise index balance and data locality. Balanced structures are crucial for minimizing performance variability and reducing tail latency. Maintaining good data locality is also essential for efficient range queries.

Figure 1, following the default settings in Section 5, validates our analysis using two SOSD datasets: AMZN and OSM. As shown in Figure 1a, AMZN is learned-index-friendly due to its linear micro-level distribution, despite a non-linear macro-level distribution. In contrast, OSM's

non-linearity at both macro and micro levels challenges accurate linear modeling. Figure 1b presents a comparison of various indexes on the two datasets under a mixed workload of random insertions, deletions, and range queries (returning 100 results). The lower-left region of the plot represents ideal performance (low latency and low memory overhead), while the upper-right region indicates suboptimal performance. B+-tree performs well in latency and size across both datasets, leveraging its robust design to ensure data locality and structural balance. ALEX and SWIX achieve latency comparable to B+-tree on model-friendly distributions (AMZN) but suffer significant degradation on non-linear distributions (OSM). PGM, while maintaining memory efficiency similar to B+-tree, incurs higher range query latency due to the need to search across multiple index-level buffers. LIPP's compromise in data locality and structural balance lead to poor performance. Figure 1c shows time-varied latencies on the OSM dataset. ALEX and PGM exhibit significant latency fluctuations. For ALEX, varying segment sizes lead to variable latencies during local recalibration, while structural imbalance triggers costly structural recalibration during insertions, particularly at deeper levels. PGM's high tail latency stems primarily from index buffer merging during recalibration, which is necessary to maintain model accuracy and data locality. In contrast, B+-tree's uniform leaf node size, balanced structure, and strong data locality significantly reduce tail latency.

In essence, the inconsistent performance of learned indexes is an inherent drawback: while they leverage models for fast searches under ideal conditions, their reliance on data distribution assumptions limits generality. Unlike traditional indexes, which provide robust and balanced performance, learned indexes excel only when the data closely adheres to the learned pattern. Deviations necessitate corrective searches or retraining, resulting in degraded performance for range queries, high tail latency, and increased memory consumption. This motivates a re-examination of learned index design to achieve robustness and stability under mixed workloads involving range queries. Integrating design principles from the proven B+-tree structure offers a promising direction.

3 HIRE Design

Based on observations in Section 2, we identify three key design goals for a general-purpose and updatable learned index: 1) robustness to diverse data distributions without relying on specific assumptions; 2) high efficiency for both point and range queries; and 3) stable latency for operations. To achieve these goals, we propose HIRE, a hybrid in-memory index that innovatively combines the strengths of B+-tree and learned indexes. As shown in Figure 2, HIRE adopts a traditional hierarchical index structure with a hybrid model to adapt to varying data distributions. It employs two node structures: leaf nodes with a compact data layout to accelerate range queries, and internal nodes with model-based acceleration and log-based updates for efficient traversal and update operations. To ensure stable latency, HIRE maintains a balanced tree structure. A non-blocking, cost-model-driven recalibration mechanism is designed to prevent latency spikes, ensuring the index remains fully available during maintenance. Additionally, an inter-level optimized bulk-loading algorithm is developed to enhance the initial construction efficiency. Together, these novel designs enable HIRE to deliver robust and efficient performance across diverse mixed workloads involving range queries.

3.1 Leaf Layer with Hybrid Nodes

As illustrated in Figure 2, the leaf layer of HIRE stores actual data in a linked list composed of compact nodes. We have opted for a compact layout because leaf nodes constitute the majority of nodes in an index. Alternative approaches, such as gap arrays, while enhancing model accuracy, introduce significant storage overhead and compromise data locality, which degrade range query performance.

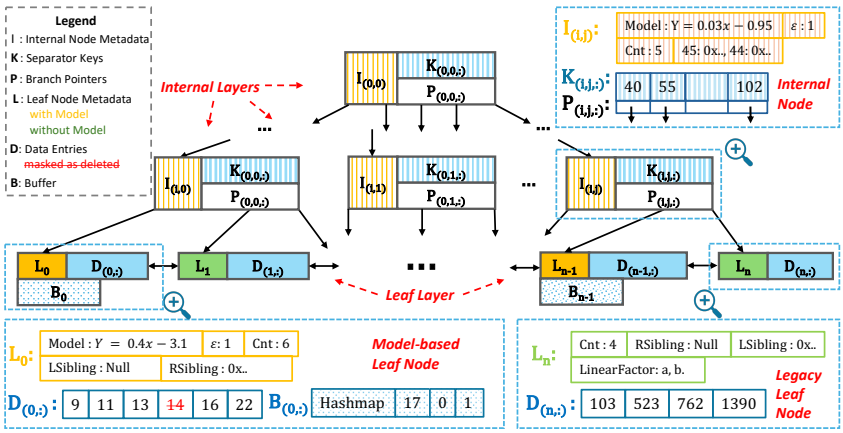


Fig. 2. Structure of HIRE

The leaf layer is hybrid, containing two different node types to adapt to various data distributions. For highly linear data segments, model-based leaf nodes employ a linear model within an error bound (e.g., ϵ) to approximate key-position relationships. This approach increases the capacity of a leaf node without compromising query performance, thereby reducing the number of nodes, index size, and tree height. To address the challenges of updates in model-based leaf nodes, we design an optimized buffer that supports $O(1)$ operations for queries, insertions, and deletions. This includes a masking scheme for deletions that preserves model validity without incurring costly data movement.

For data segments that defy linear approximation, HIRE reverts to legacy leaf nodes, which are similar to B+-tree nodes. These legacy leaf nodes feature a smaller capacity, utilize SIMD-based linear search, and support for in-place updates. To unify the design, we incorporate a dynamic optimization that records regression factors in each legacy leaf and leverages a cost model to opportunistically merge adjacent legacy nodes into efficient model-based nodes.

3.2 Model-accelerated Internal Nodes

Each internal node stores a list of keys from its child nodes along with their corresponding pointers. Unlike leaf nodes, which also need to support efficient scans, internal nodes access behave more like point lookups. To optimize internal nodes, we integrate the search efficiency of learned indexes with a log-based update mechanism inspired by LSM-trees [27]. For rapid lookups, each node employs a linear model over its keys. To improve model precision and update efficiency, the slope of the linear model is adjusted, and keys are remapped to accommodate insertions when child nodes split. Furthermore, by batching updates rather than executing them individually, the log structure serves as a buffer, avoiding the overhead of immediate retraining and delaying model invalidation after new insertions. To ensure accurate searches, the update log is accessed during each search and is designed to be lightweight and cache-efficient. As will be detailed in Section 4.2, our log-based update mechanism minimizes data movement while supporting fast lookups.

3.3 Non-Blocking Cost-Driven Recalibration

Updates in HIRE can trigger two types of recalibration. While low-cost structural adjustments, such as splits and merges, can be performed immediately, the expensive retraining of the model-based leaf node presents a significant challenge, as a naive approach would block operations and lead to high tail latency. To mitigate high tail latency and operation blocking, we propose a non-blocking, cost-model-driven recalibration mechanism. This mechanism uses background threads to execute

costly retraining and structural adjustments without stalling foreground operations. Concurrently, a built-in cost model proactively determines the optimal moment to trigger these tasks by balancing performance gains against their overhead. This approach enables HIRE to adapt to changes in data distribution while maintaining stable, low tail latency. The detailed procedures are discussed in Section 4.3.

3.4 Inter-level Optimized Bulk-Loading

To construct an efficient HIRE from scratch using existing data, we propose a bulk-loading algorithm that optimizes segment fitting across the leaf and internal layers to enhance model performance. Starting with sorted key-value pairs, the algorithm constructs an "optimal" leaf layer by using the model-based and legacy leaf nodes described in Section 3.1, minimizing the number of leaf nodes while ensuring prediction errors within ϵ. Simultaneously, splitting keys from the leaf layer are used to build upper-level nodes. During this process, the upper level may re-select splitting keys to refine lower-level segmentation, reducing prediction errors across layers. This inter-level optimization enhances the efficiency of model-accelerated internal node searches while maintaining ϵ at the leaf level. Further details are provided in Section 4.4.

4 HIRE Operations

4.1 Queries

4.1.1 Point Lookup Query. Point lookups in HIRE are performed via a depth-first traversal from the root to a target leaf node, as illustrated in Figure 3 with a query key $k_q = 56$. At each internal node, HIRE consults both the primary child list and a small log of newly added child node pointers to identify the lower-bound candidate node whose key range provides the tightest lower bound for the k_q. For our example search for key 56, the algorithm first performs a linear scan on the log. Given its small size (typically $\leq 10\%$ of the fanout f), the overhead of this scan is negligible. This scan finds a potential candidate node whose maximum key is 90. For the primary child list, the linear model's error is evaluated. If the error is less than half the child list's fanout, the model predicts k_q's position, followed by a localized correction search around the predicted position. Otherwise, a SIMD-optimized linear search is used across the child list to ensure search efficiency. In our example, this identifies the second candidate node whose maximum key is 82. Finally, HIRE compares the candidates. Between the candidate node ending at 90 (from the log) and the one ending at 82 (from the primary list), the latter provides a tighter lower bound for the key 56 and is selected for the next step in the traversal.

Upon reaching a leaf node, the search method adapts to the node type. For a model-based leaf node, HIRE checks if $k_q = 56$ falls within the model's key range. In the miss scenario depicted for key 56, this initial scan does not locate the key. The node's buffer is then queried using a hashmap, with $O(1)$ complexity. If the node is undergoing concurrent retraining, an index-level buffer associated with the retraining process is also checked. For a legacy leaf node, a single, SIMD-accelerated linear search is performed across its sorted data array to locate the entry for key 56. Once a matching key-value pair $\langle k_q, v_q \rangle$ is found, it is returned immediately; otherwise, an empty result is returned.

4.1.2 Range Query. A range query extends the point query process to retrieve key-value pairs within a range $[k_l, k_u]$, such as $[56, 156]$ in Figure 3. The process begins with a point query to locate the leaf node containing the first key greater than or equal to $k_l = 56$, identifying its storage position p_{k_l}. Subsequent steps depend on the leaf node type.

For a model-based leaf node, if $k_u = 156$ is at most the maximum key in the node's data list (i.e., k_u is equal or less than the last key in the node), HIRE collects entries from p_{k_l} to the position of the last key in range $[k_l, k_u]$. To ensure completeness, the node's buffer is also scanned for recently

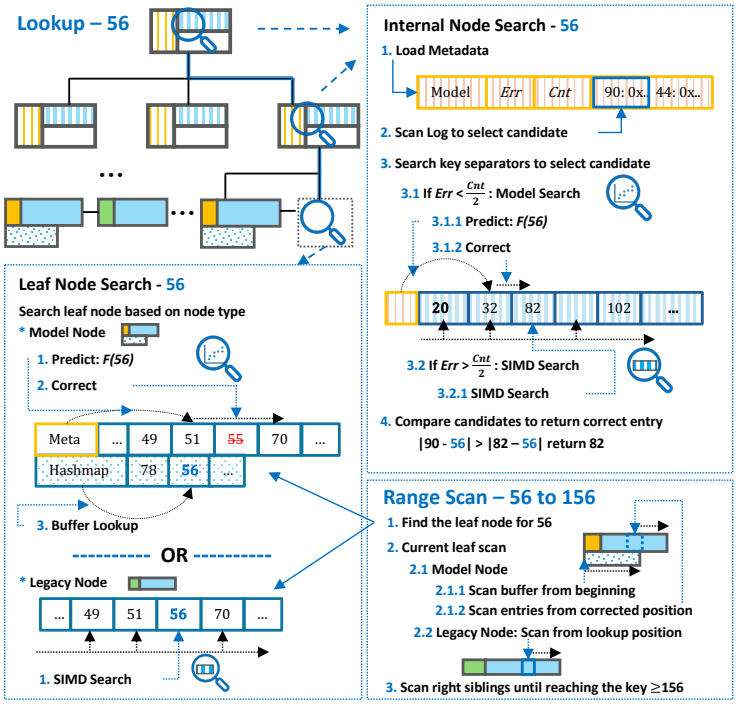


Fig. 3. Search of HIRE

inserted key-value pairs within $[k_l, k_u]$. If the range covers data within the buffer, a local sorting step is performed. This operation is confined to the results retrieved from the current leaf, merging the sorted entries from the data list with the unsorted ones from the buffer to guarantee correctly ordered output. If k_u exceeds the maximum key of the node (i.e., k_u is greater than the last key in the node), all entries from p_{k_l} to the end of the data list are collected. HIRE then traverses to the next leaf node via its sibling pointer, repeating this process until $k_u = 156$ is covered or the end of the data is reached.

For a legacy leaf node, HIRE scans the sorted data list from p_{k_l}, collecting entries until a key exceeds $k_u = 156$ or the list ends. Similar to the process in model-based nodes, if the range extends beyond the current node, the operation proceeds to the next leaf node via the sibling pointer until the query is no longer satisfied.

4.2 Updates

Update operations in HIRE, encompassing insertions and deletions, are optimized for efficiency across node types. At internal nodes, HIRE employs a mechanism based on gap array and log to ensure efficient update performance. At the leaf layer, model-based leaf nodes employ an optimized buffer mechanism to defer the need for model retraining while keeping the search efficiency. Although the buffer stores data in a vector, our hash-based optimization enables $O(1)$ complexity for lookups, insertions, and deletions within it.

4.2.1 Insertion of New Data. Algorithm 1 describes the insertion process in HIRE for a new data point $o = \langle k, v \rangle$. HIRE first locates the target leaf node by a point lookup, as detailed in Section 4.1. During traversal, if a node is marked as undergoing retraining, HIRE places o in the index-level buffer and defers insertion until retraining is complete (Line 5). Upon reaching the leaf node, the

Algorithm 1: HIRE Updates

```
1  Function insert(k, v)
       Input: Insert key k and value v
2      node ← tree.root;
3      while isLeaf(node) = false do
           // Check if the node is under retraining
4          if node.underRetrain = true then
5              tree.log[node].push(ins, ⟨k, v⟩);
6              return;
7          else
8              node ← node.nextChild(k);
9      if node is model-based leaf then
10         slot ← node.predict(k);
11         if node.data[slot] is deleted then
12             node.data[slot] ← ⟨k, v⟩;
13         else
14             node.buffer.push(k, v);
15             node.buffer.hashmap[k] = |node.buffer| − 1;
16             tree.cost.update(node) ← start thread do ;
17         leaf.cnt ← leaf.cnt + 1
18     else
19         node.LegacyInsert(k, v);
20         tree.cost.update(node) ← start thread do;
21 Function delete(k)
       Input: Delete key k
22     Do point query to k to find storage leaf and position pos ;
23     Log deletion if find nodes in the path are under retraining;
24     if leaf is model leaf then
25         if pos is in D then
               // Mask data
26             mask(leaf.data[pos]);
27             leaf.cnt ← leaf.cnt − 1
28         else
29             k_last ← leaf.buffer.last.key;
30             swap(leaf.buffer[pos], leaf.buffer.last);
31             leaf.buffer.hashmap[k_last] = pos;
32             leaf.buffer.hashmap.remove(k);
33             leaf.buffer.resize(|leaf.buffer| − 1);
34     else
35         leaf.LegacyDelete(k);
36     if leaf is underflow then
37         Do balancing operation;
```

insertion strategy depends on the node type.

For a model-based leaf node, HIRE first finds the insertion slot for o using model prediction and correction based on key k (Line 10). If the slot is marked as deleted (e.g., the slot for key 14 in Figure 4a), it can be reused without affecting the model's accuracy (Line 12). Otherwise, HIRE appends o to the node's buffer (e.g., key 7 in the example), placing it at the end and using a lightweight hashmap to track k's position (Lines 14-15). This design can achieve $O(1)$ complexity for buffer queries, insertions, and deletions. For a legacy leaf node, HIRE follows the standard B+-tree protocol (Figure 4b). The new object o is inserted directly into the sorted data list at the appropriate position. If the node overflows, it is split immediately. Finally, following any insertion,

Algorithm 2: Internal Node Operations

```
1  Function internalNodeInsert(inner, node_p)
       Input: Internal node inner and pushed up node node_p
2      slot ← inner.predict(node_p.key);
3      if inner.K[slot] is empty or deleted then
4          inner.K[slot] ← node_p.key;
5          inner.P[slot] ← node_p.ptr;
6      else
7          inner.log.append(⟨node_p.key, node_p.ptr⟩);
8      inner.cnt ← inner.cnt + 1;
9      if inner.cnt > f then
10         split inner to ⟨inner_l, inner_r⟩;
11         internalNodeInsert(inner_l.parent, inner_r);
12         Retrain inner_l and inner_l models, and remap ← start thread do;
```

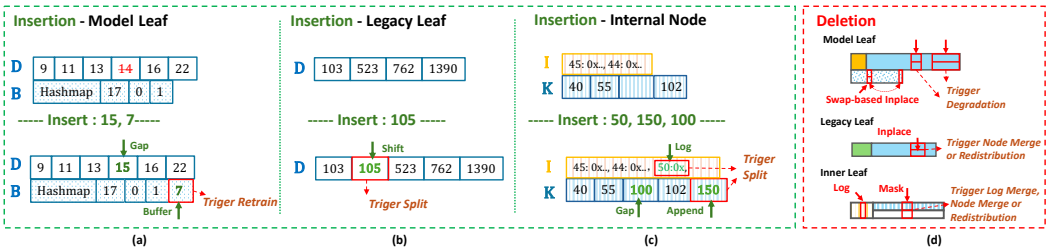


Fig. 4. Updates of HIRE

the HIRE's cost model is updated (Lines 16 and 20, to be detailed in Section 4.3). This allows HIRE to continuously evaluate whether a node requires recalibration, such as model retraining or legacy leaf node transformation.

4.2.2 Deletion of Existing Data. Figure 4d illustrates the deletion in HIRE. For model-based leaf nodes in HIRE, deletions of data integrated into the learned model use a mask-based strategy. HIRE locates the target entry via a point query, as described in Section 4.1, and flags it as "deleted" by setting the flag bit of its key through a bitwise operation (Line 26 in Algorithm 1). This preserves the key's position, maintaining the model's data distribution without requiring retraining. If the active data count falls below a predefined threshold α, HIRE recalibrates by converting the model-based leaf node into one or more legacy leaf nodes to reduce index size, as will be detailed in Section 4.3. Buffer deletions in model-based leaf nodes achieve $O(1)$ complexity: HIRE swaps the target entry with the last active element, decrements the buffer size, and updates the hashmap to reflect the new position of the swapped element, avoiding a linear-time data movement operation (Lines 29-33).

For legacy leaf nodes, HIRE performs in-place deletions using B+-tree-inspired procedures. If a deletion reduces the node's entries below the fanout f, HIRE triggers a merge with an adjacent legacy leaf sibling or redistributes entries between them to maintain performance stability.

4.2.3 Internal Node Updates. In HIRE, internal node updates occur when child nodes split (e.g., a leaf node splits or an internal node exceeds its fanout f) or merge (e.g., a node is merged into another). These processes are illustrated in Figure 4c and Algorithm 2.

An insertion is triggered when a child node splits and "pushes up" a new key-pointer pair to an internal node. HIRE first uses the internal node's model to predict the target position for the new key (Line 2). If this position is a gap, the new entry is inserted directly (e.g., key 100 in Figure 4c). If the gap is occupied, the entry is instead appended to the log (e.g., key 50 in the example). This

strategy avoids a costly $O(f)$ data movement. When the total number of child nodes in both the key-pointer (*K-P*) list and the log exceed f, the internal node splits into two B+-tree-inspired nodes. The newly created right node is then inserted into the parent node. Following the split, HIRE retrains the models for the new nodes in a background thread. Once training is complete, we scale the models' slopes and remap the child nodes, which can create new gaps for future insertions and enhances model precision (Lines 10-12).

For deletions, triggered by child node merges, HIRE applies a mask-based deletion to mark the child as deleted in the *K-P* list, preserving gaps for future insertions, similar to model-based leaf nodes (Section 4.2.2). If the child is in the log, it is removed immediately. If the number of child nodes falls below f, HIRE triggers a merge or redistribution with an adjacent sibling to maintain index balance and performance stability.

4.3 Recalibration

4.3.1 Cost Model. HIRE incorporates a cost model to determine the optimal timing for retraining a model-based leaf node. Operating concurrently in the background, this model dynamically analyzes the frequency of operations and the volume of buffered data associated with each model-based leaf node. This approach balances the performance gains from a more accurate model against the computational overhead of retraining. The decision to retrain is governed by two distinct triggers: a query-driven active trigger and a buffer-driven passive trigger.

Query-Driven Retraining (Active Trigger). This active trigger starts retraining when a model-based leaf node is both frequently queried and has accumulated a sufficient volume of new data. When a leaf node becomes a query "hotspot" and its buffer contains enough new records, the existing model may not accurately represent the data distribution, making it a prime candidate for an update.

A leaf node l is flagged for retraining when its query count, Q_l, within a recent time window T_q and its current buffer size, B_l, both exceed their respective thresholds, Q_{th} and B_{th}: $Q_l \geq Q_{th}$, $B_l \geq B_{th}$. When this condition is met, HIRE proactively initiates the retraining process. It merges the buffered entries into the data list of the leaf node and refits the model on the updated, unified data. This ensures that frequently accessed leaf nodes maintain high accuracy and search efficiency.

Parameter Tuning for Active Trigger. The effectiveness of the active trigger hinges on the appropriate tuning of its parameters: the query frequency threshold Q_{th}, the buffer size threshold B_{th}, and the time window T_q. These parameters should be set to optimize the trade-off between the performance degradation duo to outdated models and the computational cost of retraining.

The decision to retrain should be made when the expected future performance gain outweighs the immediate, one-time cost of the retraining operation. We can formalize this relationship as follows. Let $C_{retrain}$ represent the fixed, measurable cost of retraining a leaf node, as the volume of update data is usually much less than the original data. This cost includes merging the buffer and retraining the model. The performance benefit of retraining comes from moving B_l entries from the inefficient, unordered buffer into the efficient, model accelerated data list. Let $c_{buffer}(B_l)$ denote the average cost of searching for a key within a buffer of size B_l, and let c_{model} represent the cost of a search in the data list using the model. The performance gain per query after retraining is given by the difference between these costs:

$$\Delta c = c_{buffer}(B_l) - c_{model}$$

The cost to scan the buffer is roughly proportional to its size, i.e., $c_{buffer}(B_l) \propto B_l$. In contrast, c_{model} is primarily related to the model error bound ϵ, which is very low and can be treated as a constant. To estimate the total future benefit, we use the recently observed query count Q_l over the time window T_q as a predictor for future query arrivals. The total expected search cost savings, C_{gain}, before model retraining can be approximated by:

Algorithm 3: HIRE Model-based Leaf Node Retraining

```
Function retrainModelLeaf(node)
    Input: Model node node needs to be retrained
    curr ← node;
    // Maximum number of new nodes
    σ ← ⌈ (|curr.data|+|curr.buffer|) / f ⌉ − 1;
    // Snapshot nodes may affected by retraining
    snapshotPath, originalPath ← [ ]; isFinished ← false;
    while curr ≠ null and isFinished = false do
        isFinished ← σ ≤ 0;
        curr.underRetrain ← true;
        snapshotPath.push(copy(curr));
        originalPath.push(curr);
        curr ← curr.parent;
        σ ← ⌈ (σ − (f − |curr.childList|)) / f ⌉;
    Create a log at index for updates for the subtree;
    // Update pointer linkage of snapshot nodes
    UpdatePtrLink(snapshotPath);
    node ← snapshotPath.front;
    node.data ← SortMerge(node.data, node.buffer);
    node.model ← newModel(ϵ);
    cnt ← 0;
    // Attempt to retrain the model
    for d ∈ node.data do
        isAdded ← node.model.addPoint(d, cnt);
        if isAdded = false then
            Split model leaf node at cnt into ⟨node_l, node_r⟩;
            internalNodeInsert(node_l.parent, node_r;
            if cnt < α then
                covert node_l to legacy leaves;
            node ← node_r;
            cnt ← 0;
        else
            cnt ← cnt + 1;
            if cnt ≥ β then
                Split model leaf node at cnt into ⟨node_l, node_r⟩;
                internalNodeInsert(node_l.parent, node_r);
                node ← node_r;
                cnt ← 0;
    Replace snapshotPath.last with original index;
    FreeNodes(originalPath);
    Execute all updates in tree.buffer[originalPath.last];
```

$$C_{gain} \approx Q_l \cdot \Delta c = Q_l \cdot (c_{buffer}(B_l) - c_{model})$$

Thus, retraining should be triggered if and only if the expected gain surpasses the cost:

$$C_{gain} > C_{retrain}$$

Substituting in the terms gives the core decision boundary:

$$Q_l \cdot (c_{buffer}(B_{th}) - c_{model}) > C_{retrain}$$

Consequently, the cost model can adaptively tune the parameters Q_{th} and B_{th} by monitoring retraining and buffer scan costs, thereby maintaining the effectiveness of the active trigger.

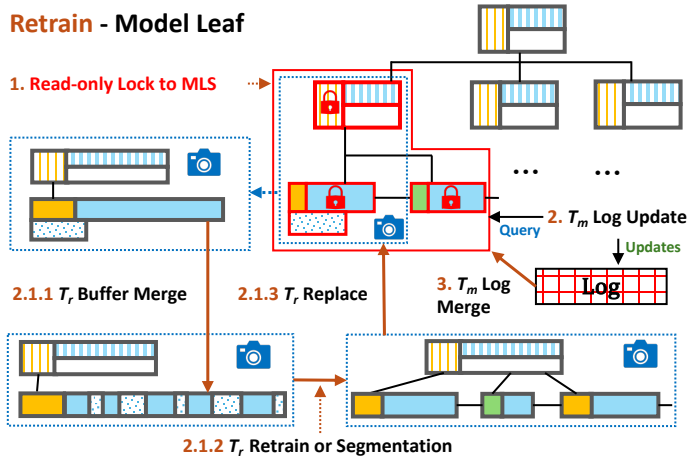


Fig. 5. Retraining of HIRE

For the time window size T_q, which is a predefined parameter by the user. It controls the stability and responsiveness of the query frequency measurement. A small T_q allows the system to react quickly to workload spikes but may be sensitive to noise. Conversely, a large T_q provides a more stable, long-term average but is slower to adapt to changes in data access patterns. The optimal value for T_q depends on the expected volatility of the workload.

Buffer Overflow Retraining (Passive Trigger). Passive trigger addresses the scenario of buffer overflow. When the buffer of a model-based leaf node reaches its maximum capacity τ, retraining becomes mandatory to integrate the buffered entries into the main data structure. Formally, retraining is initiated for a leaf node l if its buffer size B_l reaches this capacity, regardless of its query frequency:

$$B_l \geq \tau$$

In this scenario, the data from the buffer is merged into the data list, and the model is subsequently refitted on the merged data. This trigger serves as a safeguard, preventing unbounded buffer growth that would otherwise degrade search performance by necessitating linear scans over a large, unsorted set of buffered records.

4.3.2 Model-based Leaf Retraining. The balanced tree structure of HIRE facilitates index modifications, such as the replacement of specific subtrees. This inherent structural property is crucial for minimizing interference with concurrent operations accessing other regions of the index. To avoid blocking operations related to subtree reconstruction, we leverage modern multi-threaded hardware to design a non-blocking retraining mechanism based on Read-Copy-Update (RCU) [26]. This approach allows nodes to be updated in a copy and atomically replaced without impeding queries. Specifically, updates are prepared on a copy of the relevant portion of the index.[1] This approach ensures that concurrent read operations can traverse the index without acquiring locks and without observing inconsistent, intermediate states of the data structure. Consequently, HIRE can update leaf nodes and their corresponding paths to internal nodes with minimal disruption to ongoing operations, thereby maintaining high availability and throughput.

As model-based leaf nodes have no gaps in data storage unless deletions occur, new insertions must be temporarily recorded in the buffer associated with the leaf node. To maintain search efficiency, a periodic retraining process integrates these buffered data with the existing leaf data. The procedure for the retraining is detailed in Algorithm 3 and illustrated in Figure 5.

[1]More details could be found in [44]

The retraining process for a target model-based leaf node begins in a separate thread T_r, which creates a snapshot of the potential affected path (PAP) from the leaf node up to the highest node that might be altered by child node splits (Lines 2-11). During this snapshot, HIRE employs RCU to ensure safe traversal and snapshotting of the live index structure, even amidst concurrent modifications by other threads. This RCU-protected bottom-up snapshotting process considers the worst-case scenario where the model-based leaf node might split entirely into legacy leaf nodes. HIRE computes the maximum potential nodes that could be pushed up (Line 3) and identifies the full PAP whose nodes will be copied for the snapshot. Once the PAP is snapshotted, this copied path forms the basis for constructing the new subtree version and will be operated on by the retraining thread T_r. The original nodes in the main index corresponding to this PAP remain accessible to other concurrent operations, consistent with RCU. We refer to the subtree rooted at the highest node in the PAP snapshot as the minimal locked subtree (MLS). To capture concurrent modifications to the MLS while T_r is working, incoming updates to this specific subtree are temporarily recorded in a dedicated log.

Within T_r, the retraining process begins by merging the buffered data of the leaf node with its existing data entries (Line 15). A streaming linear fitting process is then applied to this merged dataset, ordered by keys, subject to a predefined fitting error threshold ϵ (Lines 18-33). If a data entry cannot be fitted within ϵ, the current model segment ends, a new leaf node is formed with the successfully fitted data, and this new node is incorporated into the PAP snapshot (Line 22). If a segment of fitted data does not meet a minimum entry count α required for a model-based leaf node, it is converted into one or more legacy leaf nodes within the snapshot (Line 24). Furthermore, even if the data fits the model well, a model-based leaf node is split if its entry count exceeds a maximum capacity β (Line 30). This capacity constraint prevents excessively large nodes, which could lead to increased retraining overhead or significant structural changes if data distributions shift dramatically.

Upon processing all merged data, the modified PAP snapshot contains the newly retrained version of the subtree. HIRE then atomically installs this new subtree into the main index by updating the parent pointer to the root of the MLS, using an RCU-safe pointer swap (Line 34). Following this, the nodes from the previous version of the subtree are freed after a grace period, ensuring no active readers are still using them (Line 35). Finally, the updates captured in the log of the MLS during the retraining period are applied to the newly installed subtree (Line 36). This two-phase approach ensures data consistency and high index availability throughout the retraining process.

4.3.3 Legacy Leaf Node Transformation. Legacy leaf nodes in HIRE can be transformed into model-based leaf nodes via two distinct mechanisms: forward merging and backward merging. Forward merging involves incrementally training an adjacent existing model-based leaf node with the data from a legacy leaf node, while backward merging consolidates multiple consecutive legacy leaf nodes into a new, single model-based leaf node. Concurrently, the cost model within HIRE continuously monitors legacy leaf nodes, computing their linear regression coefficients upon each update and tracking the position and aggregate length of consecutive legacy leaf node sequences in the index structure.

Forward merging is initiated when the linear regression coefficients of a legacy leaf node closely match those of its preceding model-based leaf node. In such cases, an attempt is made to incrementally train the existing model-based leaf node. Similar to the standard retraining procedure for model-based leaf nodes, a snapshot of potentially affected subtree nodes is taken. Using this snapshot, incremental learning is attempted: data from the legacy leaf node is streamed to the model-based leaf node for training. If the model-based leaf node can accommodate both its original data and the data from the legacy leaf node while maintaining its error within ϵ, the legacy leaf

node is merged into the model-based leaf node, and the main index is updated accordingly. To ensure safe concurrent access during this merging, the update to the main index is managed using RCU. If the error constraint is violated, the merging attempt is aborted.

Backward merging is considered when the cost model identifies a sequence of multiple consecutive legacy leaf nodes that (a) exhibit similar linear regression coefficients and (b) whose combined data volume meets the minimum requirement for creating a model-based leaf node. The merging process operates on snapshots of these legacy leaf nodes. A piecewise linear approximation (PLA) model is then employed to fit the data from these legacy leaf nodes, with the fitting error constrained to be within ϵ. If the data can be collectively fitted by the model under this error threshold, these legacy leaf nodes are converted into a new model-based leaf node, and the main index is subsequently updated to reflect this change. This structural update also leverages RCU, allowing concurrent read operations to proceed safely and without traditional locks while the index structure is modified.

4.4 Bulk Loading

The bulk-loading algorithm in HIRE is designed to optimize the index structure by strategically partitioning data into leaf nodes. The goal is to ensure that the models within the upper-level internal nodes can achieve low fitting errors, thereby improving search performance without increasing the overall index height. However, given a dataset of size N, finding a globally optimal partition of the data (i.e., minimizing the average model error along every path) requires a dynamic programming approach with $O(N^3)$ complexity, which is hard to apply to large datasets. To address this challenge, we propose an approximate algorithm that performs inter-level optimization during bulk-loading with $O(N)$ complexity.

Our bulk-loading process employs a bottom-up strategy, controlled by a key hyperparameter: the error tolerance δ. This parameter enables localized optimization during data partitioning, allowing for more efficient organization of the index structure. Given an input dataset of N ordered keys, $K = \{k_i, k_2, \cdots, k_N\}$, the construction process scans this sorted sequence to build the index level-by-level, beginning at the leaf layer. For each parent node of the leaf layer, we first populate it with an initial set of child nodes. Specifically, we generate the first $\frac{f}{4}$ leaf nodes according to the data partitioning rules of leaf retraining described in Section 4.3. The partitioning keys of these leaf nodes are then used to fit an initial linear regression model, denoted as $\mathcal{F}$. For each subsequent leaf node that can be formed under this parent, we introduce a δ-bounded optimization. Instead of using a fixed approach, we define a candidate window of δ keys around the original partitioning key. The optimal partitioning key is chosen from this window to minimize its deviation from the parent's regression model $\mathcal{F}$. The deviation for a candidate key k with an actual rank of $\mathcal{R}(k)$ is defined as a distance $\mathcal{D}(k) = |\mathcal{F}(k) - \mathcal{R}(k)|$. The key that minimizes this distance $\mathcal{F}(k)$ is selected as the final partitioning key and appended to the parent node. The model $\mathcal{F}$ is next updated in an online fashion using Recursive Least Squares (RLS) [14].

This strategy is applied recursively to higher levels of the tree. When constructing an internal node at layer $i + 1$, its children are the nodes at layer i. The "keys" used to build the model for the layer $i + 1$ node are the partitioning keys chosen for the layer i nodes. When an internal node at layer i becomes full and requires splitting, we apply a similar tolerance-based approach. The process terminates once all keys from K have been indexed, resulting in a complete and optimized HIRE index.

Complexity Analysis. Assuming the input dataset of N keys is presorted, the construction of the leaf layer requires a single linear scan over the data, which is an $O(N)$ operation. Subsequently, building the linear model for each internal node involves scanning the partitioning keys of its child nodes.

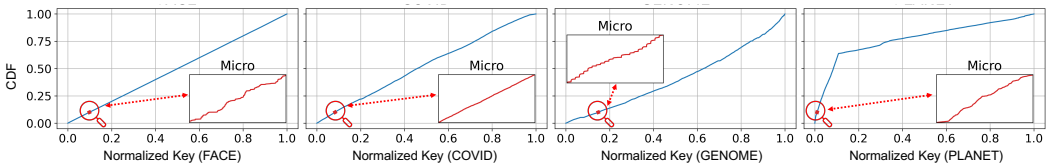


Fig. 6. CDFs of FACE, COVID, GENOME and PLANET Datasets

Since each key is used as a partitioning key for at most one parent node in the level immediately above it, the cumulative work for building all models across all layers is also proportional to N. Therefore, the total time complexity of HIRE bulk-loading algorithm is $O(N)$.

4.5 Discussion

Duplicate Keys. HIRE could handle duplicate keys using a value list approach, a strategy common in traditional index implementations. The payload associated with duplicated key is a pointer to the head of a linked list, where each node in the list holds one of the values corresponding to the duplicate key.

Multi-dimensional Keys. While HIRE is designed as an ordered index for single-attribute numeric keys, it can support multi-attribute keys using a standard lexicographical mapping, similar to that of the B+-tree [6]. Specifically, a composite key $\mathbf{k} = (k_1, \cdots, k_m)$ is transformed into a single 1D key using an order-preserving function $P(\mathbf{k})$. HIRE can then be built on these mapped keys. This approach is highly efficient for queries on the leading attribute. For example, a range predicate on the first column $k_1 \in (L, U)$ can be converted into a query for a single, contiguous interval $[P(L, min_{k_2,\cdots,k_m}), P(U, max_{k_2,\cdots,k_m})]$ on the mapped keys. HIRE then completes the search with a single range scan.

5 Evaluation

5.1 Experimental Setup

Datasets and Workloads. We conduct experiments on six real-world datasets from SOSD [25] and GRE [36]. All datasets consist of 200 million 64-bit unsigned integer keys, each paired with a random 64-bit unsigned integer value for experimental purposes. The following three datasets are selected from SOSD: OSM (uniformly sampled from OpenStreetMap locations), FACE (upsampled from Facebook user ID data), and AMZN (Amazon book sales popularity data). For GRE, we select the datasets COVID (uniformly sampled Twitter IDs labeled with COVID-19), GENOME (human chromosome gene location pairs), and PLANET (planet ID data derived from OpenStreetMap). The CDF of keys for the FACE, COVID, GENOME, and PLANET datasets is shown in Figure 6.

For all datasets, unless otherwise specified, 20% of the data is bulk loaded to construct the indexes. To simulate potential distribution shifts, a default balanced workload is applied, consisting of a sequence of random operations totaling 75% of the dataset size. These operations include queries, insertions, and deletions in a 1:1:1 ratio. For insertions, keys are drawn uniformly at random from the rest of the dataset, while deletions and queries are drawn uniformly from the dynamic set of keys currently stored in the index. All queries are executed as range queries with a default match rate (# data points falling within the range) of 256 results.

Metrics. In our experiments, we evaluate two key performance metrics: the *latency* of each operation (measured from submission to completion) and the *throughput* of the index, expressed as the number of operations processed per second (ops). For index size, we record the *memory usage* at fixed intervals throughout the workload execution and use the average of these measurements as the representative index size.

Indexes for Evaluation. We compare our proposed HIRE with three general-purpose learned

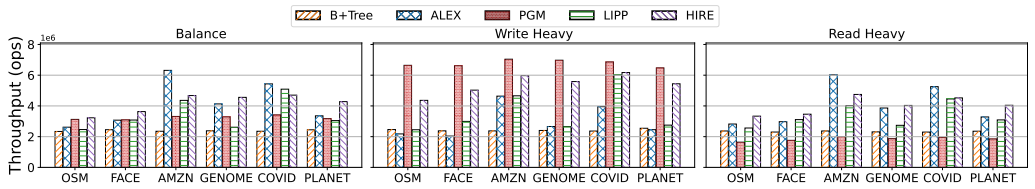


Fig. 7. Throughput on Lookup Queries

indexes previously discussed in Section 2: ALEX,[2] PGM,[3] and LIPP.[4] Note that we exclude SWIX [21] from our experiments because it is specifically designed for streaming data workloads and does not support the full set of operations required by general-purpose indexes. We also employ B+-tree[5] as a baseline representing traditional indexing structures. To ensure a fair comparison, SIMD acceleration is enabled for all baselines.

Parameter Selection.[6] For B+-tree, we use a fanout of 256, which achieves the best performance across most datasets. For PGM, we sweep its error parameter from 4 to 4,096 and report the configuration yielding the lowest latency in each experiment. For HIRE, we set $f = 256$, $\alpha = 2f = 512$, $\beta = f \times \frac{f}{2} = 32,768$, $\epsilon = \frac{f}{4} = 64$, $\tau = f = 256$, and $\delta = 8$ in all experiments.

Environment. All experiments are conducted on a workstation equipped with an AMD Ryzen 9950X and 64 GB of RAM, running Arch Linux with kernel 6.6.75. All algorithms are compiled with GCC 14.2.1, using the optimization flags -O3 -march=native -flto.

5.2 Performance on Different Workloads

This section compares the performance of various algorithms across multiple datasets and diverse workloads. In addition to the default balanced workload, two others are evaluated: write-heavy and read-heavy. The write-heavy workload emulates a write-intensive environment by adjusting the ratio of queries, insertions, and deletions to 1:8:1. Conversely, the read-heavy workload uses an 8:1:1 ratio to mimic a read-intensive environment. These workloads enable evaluation of performance variations stemming from the design decisions of each algorithm.

5.2.1 Point Lookup Query. We begin by comparing the throughput performance of each index on point lookup queries by setting the match rate to one result. Point lookup queries are fundamental to all other operations, and their performance critically impacts the overall efficiency of the index.

The results, shown in Figure 7, demonstrate that B+-tree maintains consistent performance across different datasets with distinct data distributions. In contrast, learned indexes can experience a throughput degradation exceeding 2.4× when transitioning from model-friendly distributions (e.g., AMZN) to those more challenging to learn (e.g., OSM). Despite this, learned indexes generally outperform B+-tree on point lookup queries.

Contrary to the general perception of their update inefficiency, learned indexes exhibit robust performance under the write-heavy workload, primarily due to their buffering strategies. For example, PGM employs an LSM-tree as an index-level buffer, allowing direct insertion of new keys without searching the main structure, which achieves the highest throughput. Conversely, finer-grained, data-level buffers, like that used by ALEX, require locating the correct position for each insertion, negatively impacting update throughput. This buffer granularity also introduces a trade-off between search and update performance. Under the read-heavy workload, this trade-off

[2]https://github.com/microsoft/ALEX
[3]https://github.com/gvinciguerra/PGM-index
[4]https://github.com/curtis-sun/TLI/tree/main/competitors/lipp
[5]https://github.com/tlx/tlx/tree/master/tlx/container
[6]Due to space constraints, a detailed explanation of the parameters and their selection is provided in our technical report [44].

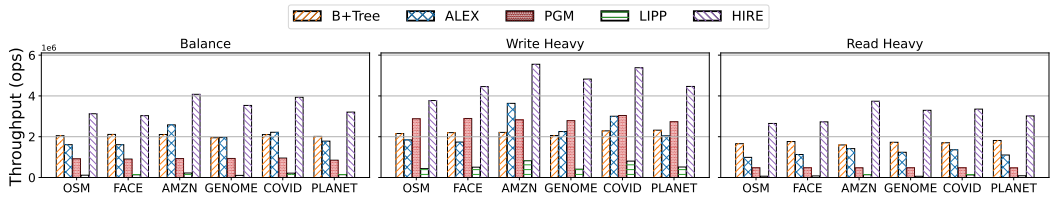


Fig. 8. Throughput on Range Queries

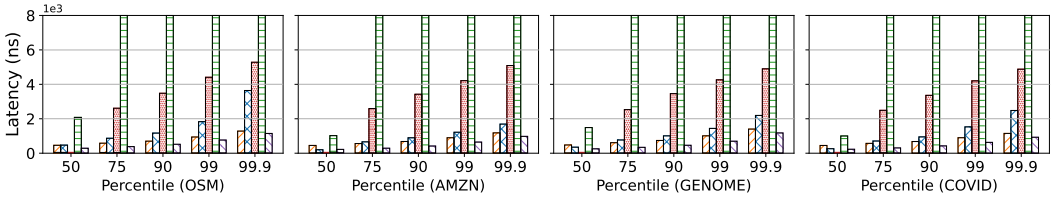


Fig. 9. Tail Latency on Different Datasets

penalizes PGM, as its need to search multiple buffers results in the lowest throughput (only 84% of B+-tree even on model-friendly distributions like AMZN and COVID). In contrast, ALEX's data-level buffer preserves data locality and confines searches to a single node, achieving significantly higher throughput – 1.2× and 2.5× throughput of B+-tree on OSM and AMZN, respectively.

HIRE and LIPP demonstrate relatively stable performance across different workloads. LIPP leverages a precise, nearly error-free model for excellent search performance. To prevent model drift from insertions, LIPP splits data nodes to maintain accuracy, which however increases pointer traversal costs and causes update performance to vary with data distributions. By contrast, HIRE uses a hybrid buffering technique at different index levels, efficiently ingesting new keys while mitigating insertion costs. Its lightweight buffering mechanism enables low-cost searches, thereby yielding strong search performance. Even on OSM, the least linearly distributed dataset, HIRE excels under the read-heavy workload, achieving 1.2× throughput of ALEX, the best-performing baseline.

5.2.2 Range Query. In our experiments, range queries use the same settings as point lookup queries, except the match rate is fixed at 256 results. Evaluating learned indexes on range queries would reveal an additional trade-off factor: data locality. Data locality is crucial because, after the index locates the first key in a range, it must sequentially scan contiguous data until it encounters a key outside the specified range. Poor data locality can force the index to search multiple buffers or perform pointer jumping, significantly increasing query latency.

Figure 8 shows the stable performance of B+-tree. For range queries, most state-of-the-art learned indexes fail to outperform B+-tree across different workloads and datasets. The performance degradation from point lookup to range queries in learned indexes arises from their inherent trade-off between data locality and update/lookup efficiency. This trade-off involves factors such as ALEX's sparse structure, PGM's index-level buffering, and LIPP's splitting strategy. Among these, LIPP exhibits the greatest performance decline, with a 24.4× performance degradation compared to point lookup queries, as its splitting strategy prevents data from being stored contiguously within a node, necessitating costly pointer traversals that reduce efficiency. In contrast, B+-tree achieves robust performance as its buffer is located only at the end of each node, thereby preserving data locality and minimizing pointer traversals.

HIRE consistently delivers the strongest range query performance. For example, it exceeds B+-tree by up to 1.9×, 2.5×, and 2.4× in throughput under the three different workloads, respectively. Compared to other learned indexes, HIRE outperforms LIPP by up to 41.7×, ALEX by 2.7×, and PGM

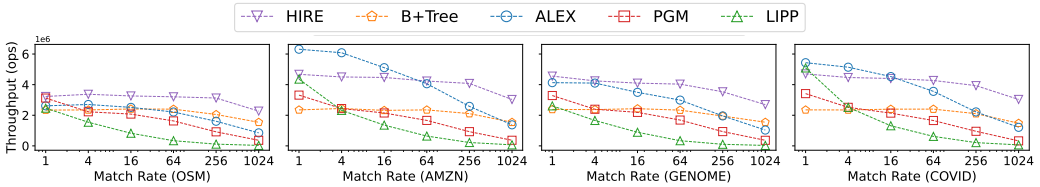


Fig. 10. Throughput on Different Match Rates

by 7.9×. This performance advantage stems from two key designs: cache-aligned lightweight buffers that reduce overhead and a compact data layout that preserves data locality within model-based nodes. For less model-friendly distributions like OSM, HIRE leverages legacy nodes to maintain data locality and ensure consistently high range query performance.

5.3 Performance on Tail Latency

Learned indexes often improve average performance by delaying expensive operations. However, tail latency is also critical as the deferred operations can negatively impact worst-case scenarios, leading to higher overall processing delays and greater uncertainty in index performance.

Figure 9 shows the 50th, 75th, 90th, 99th, and 99.9th percentile tail latencies for various indexes on four datasets. As percentiles increase, the tail latency of B+-tree rises gradually but remains relatively stable across datasets. In contrast, learned index baselines exhibit a more pronounced increase. Notably, LIPP has the highest tail latency for all datasets. PGM's tail latency pattern remains consistent across datasets, likely due to its distribution-agnostic insertion and merge strategy. ALEX achieves relatively low tail latency in model-friendly distributions, but performs poorly in model-unfriendly scenarios, where degraded structural balance increases recalibration overhead. HIRE consistently exhibits low tail latency across all datasets. Its balanced design, non-blocking cost-driven recalibration mechanism, and proactive use of legacy leaf nodes for challenging distributions effectively minimize update costs. In particular, on the challenging OSM dataset, HIRE reduces tail latency by up to 37% compared to B+-tree, 68% to ALEX, 85% to PGM, and 97% to LIPP. On the model-friendly AMZN dataset, HIRE achieves tail latency reductions of up to 51% compared to B+-tree, 56% to ALEX, 88% to PGM, and 98% to LIPP.

5.4 Performance on Different Match Rates

Range query performance is heavily influenced by the match rate. A lower match rate restricts scans to a single node, whereas a higher match rate requires traversing multiple nodes, with expensive pointer traversals. We vary the match rate from 1 to 1,024 and evaluate the indexes on different datasets, as shown in Figure 10.

Generally, increasing the match rate decreases throughput due to larger scan ranges and increased random accesses when data locality is compromised. For example, ALEX underperforms on the OSM dataset because it creates more gaps within nodes and requires extra nodes to maintain its error threshold, increasing scan costs. Similarly, ALEX excels on the AMZN dataset at low match rates but degrades significantly at higher match rates due to bypassing additional gaps. LIPP achieves optimal performance at a match rate of 1 (i.e., point lookup) across all datasets. However, even a slight increase in the match rate substantially reduces LIPP's performance due to random accesses. In contrast, HIRE consistently demonstrates robust performance due to its compact leaf node structure with model-legacy duality, which effectively maintains data locality across different data distributions. On the challenging OSM dataset, HIRE achieves up to 1.5× higher throughput than B+-tree and 2.7× higher throughput than the best-performing learned index baseline, ALEX, at a match rate of 1,024.

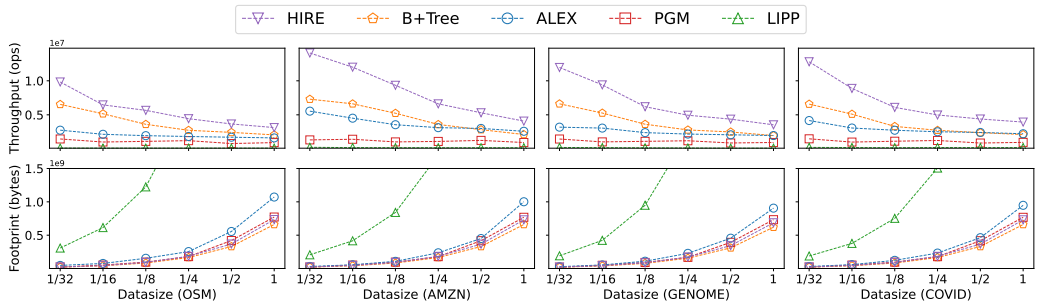


Fig. 11. Throughput and Index Size on Different Datasizes

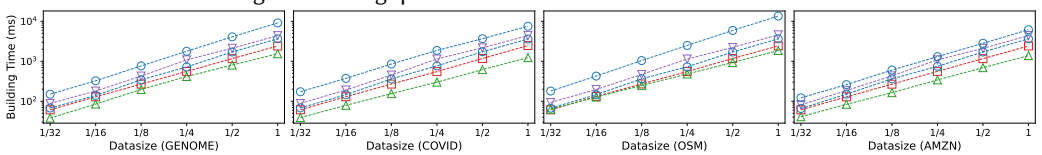


Fig. 12. Index Build Time on Different Datasizes

5.5 Performance on Scalability

Figure 11 shows the throughput (top) and memory usage (bottom) of various indexes as dataset size increases. When the dataset size grows, all indexes experience performance degradation, with throughput decreasing and memory usage increasing. However, HIRE consistently outperforms all baselines in terms of throughput while maintaining memory consumption comparable to other learned indexes, demonstrating its superior scalability. Compared to the best baseline, B+-tree, HIRE's index size is 1.1× larger, but its throughput is 1.5× higher.

5.6 Performance on Index Building

For in-memory indexes, construction time is a critical performance indicator. We present the initial build times for various indexes across different dataset sizes in Figure 12. As observed, the build time of all algorithms grows roughly linearly with dataset size.

Among them, LIPP achieves the fastest construction time, outperforming the standard B+-tree by at least 1.8×. This is because LIPP's bulk loading process computes a minimal collision mapping (i.e., multiple keys are mapped to the same slot by the model) over the entire input, which allows it to generate long, contiguous arrays and thereby accelerates construction. This design, introduces a trade-off. While it accelerating the initial construction, these large contiguous segments can incur high split cost during future updates, particularly under data distribution shifts. PGM and B+-tree exhibit similar build times, as both rely on segmenting the data before constructing their upper-level index. The construction of HIRE is moderately slower, incurring a 1.2-1.5× overhead compared to PGM. This is an expected trade-off for its strategy, which finds a more optimized index structure by exploring inter-level segmentation. ALEX also optimizes the structure during construction at the cost of build time. It consistently incurs the highest build time, taking up to 3.7× longer than B + tree in our experiments. This significant overhead stems from its complex fanout tree design, which is particularly costly for non-linear datasets.

5.7 Ablation Studies of HIRE

5.7.1 Effectiveness of Hybrid Nodes. To validate the effectiveness of our proposed hybrid node design, we conduct an ablation study comparing the full HIRE index against a variant without legacy leaf nodes. We select the model-unfriendly OSM dataset and the model-friendly AMZN dataset to evaluate the performance impact across different data distributions. The results are

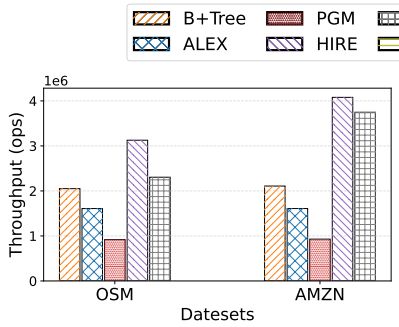


Fig. 13. Impact of Hybrid Nodes

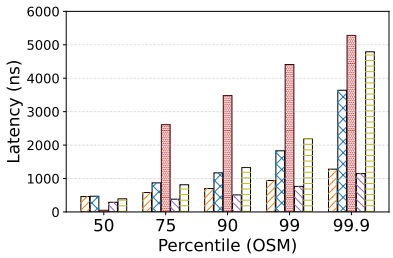


Fig. 14. Impact of Non-blocking Recalibration

shown in Figure 13. On OSM, the removal of legacy leaf nodes causes a significant 26% drop in throughput, clearly demonstrating the importance of the hybrid leaf design for ensuring robustness on skewed data distributions. The hybrid node design allows HIRE to consolidate numerous small, hard-to-learn data segments into fewer, larger legacy leaf nodes, significantly reducing the total number of leaf nodes in the index and thereby enhancing overall performance and throughput. On AMZN, the variant without legacy leaf nodes exhibits a 8% decrease in throughput comparable to the full HIRE. For such model-friendly data, most segments are linear enough to form model-based leaf nodes naturally. Nevertheless, this variant of HIRE still benefits from the remaining components of the design, making it to outperform other baselines.

5.7.2 Effectiveness of Non-Blocking Recalibration. To evaluate our non-blocking, cost-driven recalibration design, we study a modified HIRE variant which replaces our proposed mechanism with a simplified, single-threaded, and blocking recalibration process. The results on the OSM dataset, shown in Figure 14, indicate that without our non-blocking recalibration design, the HIRE variant exhibits significant latency spikes. Tail latency at the 99th and 99.9th percentiles increases by 2.9× and 4.2×, respectively, compared to the full HIRE. These spikes result from large-scale structural adjustments, such as when a large model-based leaf node splits into multiple leaf nodes, forcing the index to stall for an extended period for structure modifications. This experiment confirms that our novel non-blocking recalibration mechanism is crucial for mitigating tail latency and ensuring robust performance, particularly under dynamic workloads.

5.8 Performance on Concurrency

Although the initial design of HIRE did not support fully concurrent read-write operations, we observe that its structural modifications are localized, triggered only when legacy leaf nodes undergo splits, merges, or redistributions. This property allows us to develop a concurrent-safe version, denoted HIRE+, by incorporating a concurrency control mechanism similar to that of ALEX+ and LIPP+ in GRE [36]. This approach augmented the existing RCU mechanism in HIRE with node-level locking to ensure thread-safe access. We evaluate the performance of HIRE+ under concurrent write-heavy workloads against other concurrent indexes, including B+-tree-OLC, ALEX+, and LIPP+. As PGM lacks a concurrent implementation, we replace it with FINEdex [19], which is natively designed for concurrency. All baseline implementations are sourced from the GRE framework [36].

Figure 15 illustrates the throughput of all indexes on four datasets as the number of concurrent threads scales from 1 to 10. HIRE+ demonstrates the best performance across all scenarios. HIRE+ achieves a 5.8-6.6× speedup when concurrent threads scale from 1 to 10. At 10 threads, HIRE+ outperforms the best-performing baseline on each dataset: it is 1.3× faster than ALEX+ on AMZN,

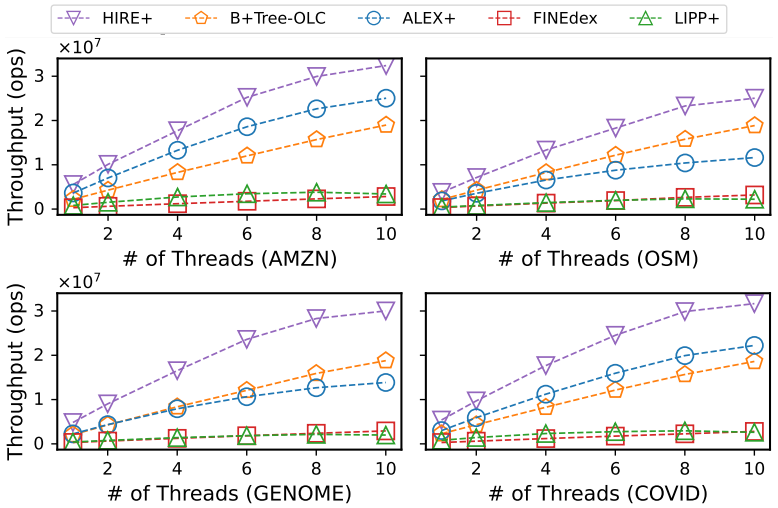


Fig. 15. Throughput Across Concurrent Threads (Write-Heavy)

1.3× faster than B+-tree-OLC on OSM, and 1.6× and 1.4× faster than the strongest baselines on GENOME and COVID, respectively.

We also observe the performance trade-offs of different concurrency control schemes. Indexes employing lock-free designs (e.g., B+-tree-OLC) or fine-grained locking (e.g., FINEdex) show sustained scalability. In contrast, other lock-based indexes, including our HIRE+, ALEX+, and LIPP+, hit a performance plateau beyond eight threads as lock contention escalates. This observation suggests that developing a more carefully designed lock-free concurrency mechanism for HIRE is a promising direction for future work.

6 Related Work

The concept of the learned index, RMI [16], emerged in 2018 as a static approach. Early research on learned indexes mainly focuses on static workload. Several works, such as RadixSpline [15], PLEX [30], and Shift-Table [13], demonstrate the potential of model-based indexing. However, a common criticism of these early designs is their inability to effectively support dynamic updates.

To overcome the limitation, updatable learned indexes have been introduced, including FITing-Tree [11], PGM, and ALEX [8]. Both FITing-Tree and PGM employ a balanced structure built bottom-up. ALEX, another notable updatable learned index, extends the RMI concept by constructing the index top-down. It sacrifices balance and a bounded structure to achieve better model fitting at different data granularities, and linearizes the data by adding gaps into segments to create a sparser structure. These pioneering updatable learned indexes delineate a design space characterized by one machine learning model (linear model), two construction methods (top-down and bottom-up), and three buffer types (index-level, segment-level, and data-level). Along with benchmarks like SOSD [25] for performance evaluation on real-world data, these approaches have triggered a "Cambrian explosion" of learned index proposals.

During this surge, numerous learned indexes have been developed, revealing three main design trends. First, some works focus on optimizing model accuracy and structure for faster lookups. For example, LIPP achieves near error-free performance using a model-based insertion and splitting approach, although it sacrifices data locality. DILI [20] employs a concise, computation-efficient linear regression model for improved efficiency, while Hyper [43] combines bottom-up and top-down construction approaches to balance latency and memory usage. SALI [12] maintains internal

statistical information to further refine its model. Second, other designs leverage modern hardware features to boost performance. For instance, Xindex [32] and FINEdex [19] are designed to support concurrent operations. CARMI [40] is the first learned index to fully exploit CPU cache characteristics. [45] extends optimizations to GPUs. Third, a shift toward case-specific learned indexes has emerged to address the limitations of general-purpose designs. Most learned indexes, including our proposed HIRE and others like ALEX, PGM and LIPP, are designed for numeric data, as modeling the CDF of strings for key lookup is a non-trivial challenge. This has led to the development of specialized solutions. For string data, SIndex [34] and LITS [39] represent notable recent progress. For other use cases, indexes like FLIRT [38] and SWIX [21] have been designed specifically for indexing streaming windows.

There are also research efforts in optimizing learned indexes, including improving persistent memory efficiency [7, 17, 18, 24, 35, 41] and parameter tuning for optimal performance [3, 4, 33].

7 Conclusion

In this paper, we propose a novel hybrid learned index, HIRE, which delivers efficient and robust performance across diverse workloads and data distributions. Compared to state-of-the-art updatable learned indexes and traditional indexes, HIRE not only exhibits higher average performance but also significantly reduces tail latency. This approach greatly mitigates the risk of drastic performance drops under mixed workloads. Currently, HIRE provides a stable and efficient index for in-memory scenarios. Its balanced tree structure and compact leaf design offer a solid foundation for future expansion to disk-based environments, enabling support for larger-scale data and more complex workloads.

Acknowledgments

We are deeply thankful to the anonymous reviewers for their constructive reviews. This work is supported by Hong Kong RGC Grants (Project No. 12202024 and 12200022). Jianliang Xu is the corresponding author.

References

[1] 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), 47:1–47:28.

[2] Yuvaraj Chesetti and Prashant Pandey. 2024. Evaluating Learned Indexes for External-Memory Joins. *arXiv preprint arXiv:2407.00590* (2024).

[3] Supawit Chockchowwat. 2022. Tuning Hierarchical Learned Indexes on Disk and Beyond. In *ACM SIGMOD International Conference on Management of Data*. 2515–2517.

[4] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. AirIndex: Versatile Index Tuning Through Data and Storage. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 204:1–204:26.

[5] Minguk Choi, Seehwan Yoo, and Jongmoo Choi. 2024. Can Learned Indexes be Built Efficiently? A Deep Dive into Sampling Trade-offs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 116.

[6] Douglas Comer. 1979. Ubiquitous B-tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[7] Lixiao Cui, Yijing Luo, Yusen Li, Gang Wang, and Xiaoguang Liu. 2024. When Learned Indexes Meet Persistent Memory: The Analysis and the Optimization. *IEEE Trans. Knowl. Data Eng.* 36, 12 (2024), 9517–9531.

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *ACM SIGMOD International Conference on Management of Data*. 969–984.

[9] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, Vol. 119. 3123–3132.

[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[11] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *ACM SIGMOD International Conference on Management of Data*. 1189–1206.
[12] 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. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 258:1–258:25.
[13] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT*. 253–264.
[14] Monson H Hayes. 1996. *Statistical digital signal processing and modeling*. John Wiley & Sons.
[15] 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, aiDM@SIGMOD 2020*. 1–5.
[16] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *ACM SIGMOD International Conference on Management of Data*. 489–504.
[17] Hai Lan, Zhifeng Bao, J Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2023. A Simple Yet High-Performing On-disk Learned Index: Can We Have Our Cake and Eat it Too. *arXiv preprint arXiv:2306.02604* (2023).
[18] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-Disk Updatable Learned Index. In *IEEE International Conference on Data Engineering, ICDE 2024*. 4856–4869.
[19] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
[20] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2212–2224.
[21] Liang Liang, Guang Yang, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. 2024. SWIX: A Memory-efficient Sliding Window Learned Index. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 41:1–41:26.
[22] Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. 2024. Why Are Learned Indexes So Effective but Sometimes Ineffective? *arXiv preprint arXiv:2410.00846* (2024).
[23] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *VLDB J.* 34, 2 (2025), 17.
[24] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proceedings of the VLDB Endowment* 15, 3 (2021), 597–610.
[25] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.
[26] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-copy update. In *AUUG Conference Proceedings*, Vol. 175.
[27] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.
[28] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024*. ACM, 347–359.
[29] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. 2011. Database system concepts. (2011).
[30] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *arXiv preprint arXiv:2108.05117* (2021).
[31] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.
[32] 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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.
[33] Taiyi Wang, Liang Liang, Guang Yang, Thomas Heinis, and Eiko Yoneki. 2025. A New Paradigm in Tuning Learned Indexes: A Reinforcement Learning Enhanced Approach. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.
[34] 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.
[35] 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 Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–25.
[36] 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.
[37] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with

Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.
[38] Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. 2023. FLIRT: A Fast Learned Index for Rolling Time frames. In *Proceedings 26th International Conference on Extending Database Technology, EDBT*. 234–246.
[39] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3415–3427.
[40] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2679–2691.
[41] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[42] Rui Zhang, Yukai Huang, Sicheng Liang, Shangyi Sun, Shaonan Ma, Chengying Huan, Lulu Chen, Zhihui Lu, Yang Xu, Ming Yan, et al. 2024. Revisiting Learned Index with Byte-addressable Persistent Storage. In *Proceedings of the 53rd International Conference on Parallel Processing*. 929–938.
[43] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[44] Xinyi Zhang, Liang Liang, Anastasia Ailamaki, and Jianliang Xu. 2025. HIRE: A Hybrid Learned Index for Robust and Efficient Performance under Mixed Workloads (Technical Report). https://arxiv.org/abs/2511.21307.
[45] Xun Zhong, Yong Zhang, Yu Chen, Chao Li, and Chunxiao Xing. 2022. Learned index on GPU. In *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 117–122.

Received July 2025; revised October 2025; accepted November 2025