



Replace or Augment: Rethinking the Role of Learned Indexes in Relational Databases

Zipeng Chen^{*}
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
zipengchen@smail.nju.edu.cn

Meng Li[†]
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
meng@nju.edu.cn

Hancheng Wang
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
hanchengwang@smail.nju.edu.cn

Junhan Lv
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
231220021@smail.nju.edu.cn

Haipeng Dai
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
haipengdai@nju.edu.cn

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

ABSTRACT

B-Trees are the predominant index structures in relational databases. However, they become a performance bottleneck in traversal-heavy workloads. Although learned indexes have emerged as a promising solution to mitigate this bottleneck, their high engineering complexity hinders their practical adoption as replacements for B-Trees. Instead, we propose a more viable alternative: augmenting B-Trees with learned indexing techniques to reduce integration costs and speed up traversals. To this end, we present MirrorBtr, a unified framework that builds upon existing learned indexes to accelerate B-Tree traversals without altering the B-Tree structure. MirrorBtr reduces space and synchronization overhead and dynamically adapts to B-Tree updates. We implement MirrorBtr in MySQL InnoDB and conduct extensive evaluations, showing that it consistently delivers traversal speedups with negligible maintenance overhead and outperforms both the conventional B-Tree and prior acceleration methods in InnoDB. In particular, MirrorBtr achieves up to 7.45 $\times$ higher query throughput than the B-Tree.

PVLDB Reference Format:

Zipeng Chen, Meng Li, Hancheng Wang, Junhan Lv, Haipeng Dai, and Siqiang Luo. Replace or Augment: Rethinking the Role of Learned Indexes in Relational Databases. PVLDB, 20(1): 1 - 15, 2026. doi:10.14778/3845598.3845599

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/unconsolable/MirrorBtr>.

^{*}Work done during internship at Alibaba Group.

[†]Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 20, No. 1 ISSN 2150-8097.
doi:10.14778/3845598.3845599

1 INTRODUCTION

B-Trees are ubiquitous in databases, but they become a performance bottleneck for traversal-intensive workloads. B-Tree indexes and their variants (e.g., B+-Trees) remain the de facto standard in mainstream database systems [25, 53, 57, 64], due to their predictable performance, robustness across workloads, and mature support for transactional and recovery features [23, 54, 55].

The rapid development of DRAM technology [48, 67] has made large-memory database instances (e.g., up to several hundred gigabytes) widely available in major cloud providers [5, 20, 49]. These instances can cache most or even the entire working set, eliminating disk I/O as the primary performance bottleneck. Nevertheless, query throughput is bounded in workloads with intensive B-Tree traversals, such as those in e-commerce, logistics, finance, and social networks [42, 61]. Specifically, these traversals dominate CPU execution time and exhibit poor scalability under high concurrency. We illustrate this bottleneck with two representative queries:

- **Q1 (Price Range Filter):** `SELECT id, price, title FROM goods WHERE price BETWEEN 20 AND 120;`
- **Q2 (Join with Date Filter):** `SELECT users.id, users.name FROM users JOIN shipments ON users.id = shipments.user_id AND shipments.date = '2026-01-01';`

Q1 performs range scans and repetitive primary index lookups on the goods table to fetch matching products. Q2 conducts a join with date filtering and probes the primary index of users for each matching record. Both queries rely on frequent root-to-leaf B-Tree node accesses and intra-node binary searches, incurring substantial cumulative overhead. In addition, node-level latches ensure thread safety, but their atomic operations frequently invalidate CPU cache lines, leading to costly synchronization overhead. As a result, such frequent lookups encounter a performance wall: (1) B-Tree traversal overhead increases from 20% to over 80% of total query execution time as the result set size increases (detailed in Section 3.1) and (2) repeated latch contention degrades concurrent scalability.

To address the above bottlenecks, learned indexes accelerate index lookups, while optimistic latching B-Trees mitigate read latch contention. Despite their potential, both suffer

from high integration complexity in real-world relational databases. Learned indexes [11, 18, 32, 72] and optimistic latching B-Trees [37] are promising solutions to the traversal bottleneck. The pioneering work on learned indexes [32] exploits data distribution to achieve lower query latency than traditional B-Trees. This initial proposal catalyzes a surge of research that extends the learned paradigm to broader scenarios, including dynamic updates [11, 18, 50, 72, 83] and disk-based storage [34, 44, 81]. Meanwhile, optimistic latching B-Trees employ version-based validation to eliminate read-side contention [3, 37]. Given these benefits, both techniques are viable and effective alternatives to address the traversal bottleneck.

Despite these compelling advantages, the deployment of learned indexes and optimistic latching B-Trees in relational databases remains limited, primarily due to excessive integration complexity. First, B-Tree indexes are deeply coupled with core database components such as crash recovery [27, 51, 52, 85] and transaction management [4, 14, 58, 74]. Migrating from B-Trees to learned indexes thus requires comprehensive system redesign and rigorous validation, going far beyond simple API replacements. Notably, the PostgreSQL community has faced similar challenges during trials of learned index integration [12]. In addition, incorporating optimistic latching B-Trees complicates the control flow. Any detected concurrent write triggers an operation restart [37, 38], demanding safe resource reclamation to avoid leaks from incomplete cleanup during repeated retries. This mechanism entails extensive code modifications to core execution paths [3]. As a result, mainstream relational databases, including MySQL and PostgreSQL [12, 26], have not adopted these techniques into their core architectures, while most deployments are confined to key-value stores [2, 9, 28, 36, 43, 68, 70].

Proposed design. To tackle the aforementioned integration challenge, we advocate a pragmatic alternative paradigm: augmenting conventional B-Trees with learned models rather than replacing them. Specifically, a replacement design fully substitutes the native B-Tree with a new structure that independently controls search logic and data layout. In contrast, an augmentation approach preserves the existing B-Trees for compatibility and leverages auxiliary structures to assist with specific subtasks, such as guiding search paths and alleviating concurrency contention. We choose concurrent learned indexes for augmentation, motivated by their high read performance, compact memory footprint [46, 62, 71], and support for optimistic latching [18, 71, 83]. However, learned indexes evolve continuously toward more advanced designs [11, 18, 50, 83], and different index designs exhibit distinct performance characteristics [62, 71], resulting in varying benefits and overheads within MirrorBtr. We thus construct a flexible framework compatible with diverse index implementations, along with a dedicated mechanism to choose the most suitable one.

We instantiate this paradigm in MySQL InnoDB and present MirrorBtr, a lightweight learned mirror that augments B-Trees. MirrorBtr offers the following key features: (1) A non-intrusive framework that enables seamless integration of various learned indexes into B-Trees; (2) An offline selection pipeline that identifies suitable learned index candidates; (3) A lightweight synchronization scheme between the learned index and the underlying B-Tree with low runtime overhead; (4) Effective acceleration of B-Tree traversals.

While learned indexes are continuously evolving toward hardware limits, demonstrating their practical value via real-world system integration remains relatively underexplored. Rather than proposing new learned indexes, our work focuses on deploying existing ones to relational databases through a unified framework.

Contributions. Our main contributions are summarized as follows:

- **B-Tree augmentation with learned indexes.** We propose MirrorBtr, a lightweight framework that augments B-Trees with learned indexes to speed up B-Tree lookups. This approach reduces space and synchronization overhead while preserving the original physical layout of B-Trees.
- **Privacy-preserving offline pipeline for learned index selection.** We develop an offline pipeline to select a learned index candidate across synthesized datasets with varying distributional complexity. Decoupled from the database instance, this design avoids accessing user data and complies with privacy policies.
- **Lightweight synchronization for coexisting indexes.** We design a synchronization protocol based on learned index updates and rightward B-Tree leaf-sibling searches, ensuring correctness without global latches on the learned index.
- **Integration and evaluation in a real-world database.** We implement MirrorBtr in the MySQL InnoDB storage engine and evaluate it with multiple real-world datasets [46, 71]. Experimental results show up to 7.45 $\times$ throughput improvement over vanilla B-Tree on traversal-intensive queries, while consuming only 0.51% of the B-Tree index size.

2 BACKGROUND

This section reviews the on-disk B-Tree as the foundation of MirrorBtr design, taking InnoDB as a representative example. Note that although InnoDB’s index structure is technically a B+-Tree, we use “B-Tree” to also refer to the B+-Tree throughout this work, following the convention in [16, 54].

Search modes and separator keys. Point and range queries adopt dedicated search modes to retrieve matched records via B-Tree traversal. During this process, separator keys in inner nodes govern the traversal path. Specific to InnoDB, four directional search modes [65] are provided:

- **LT and LE.** Search for records **less than (LT)** or **less than or equal to (LE)** the query key, with backward scanning.
- **GT and GE.** Search for records **greater than (GT)** or **greater than or equal to (GE)** the query key, with forward scanning.

In terms of separator key implementation, InnoDB stores the minimum key of each child subtree inside inner nodes [8]. As illustrated in Figure 1, 10 and 8613 serve as separator keys, which are minimum keys of their respective child subtrees. Given a query key k , inner nodes route the traversal to the leaf node with the largest separator key that satisfies $< k$ for LT and GE modes, and $\leq k$ for LE and GT modes. After locating the target leaf node, the search continues to find the exact record within the node.

Sorted offset arrays in nodes. On-disk B-Trees maintain an array of record offsets sorted by key, enabling binary search during lookups [54]. Such offset arrays can be dense with one slot per record, as in PostgreSQL’s line pointer array [25], or sparse with one slot per group of records, as in InnoDB’s page directory [64].

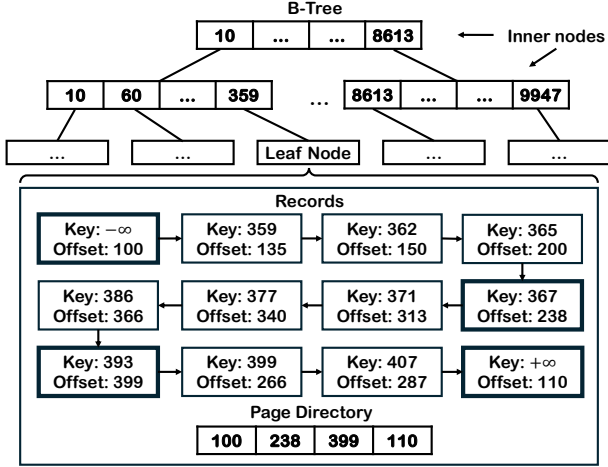


Figure 1: InnoDB B-Tree and its leaf node layout.

Accordingly, intra-node searches in InnoDB first perform a binary search on the page directory and then proceed with a linear scan on the record group. Figure 1 illustrates the layout of an InnoDB leaf node, including its records and the associated page directory. Adjacent user records form an ordered linked list anchored by two markers: the infimum and supremum records. These marker records are allocated during node initialization with lower offsets than user records. The page directory stores offsets to the infimum record (100), two user records (238 and 399), and the supremum record (110) as sparse landmarks. This arrangement bounds the size of each record group to a fixed threshold (1 to 3 in this example), thereby limiting the scan distance.

3 MOTIVATION

In this section, we analyze the root causes of B-Tree inefficiencies and examine why integrating existing index optimizations remains nontrivial for relational databases.

3.1 When B-Trees Become a Bottleneck?

B-Trees deliver high efficiency for queries with few index searches: the number of traversals is small, and the shallow tree height (typically 3 to 4 levels) resulting from kilobyte-sized nodes (e.g., 16KB in InnoDB, 8KB in PostgreSQL) keeps traversal overhead modest in total query execution time.

However, B-Trees become bottlenecks in traversal-heavy queries that repeatedly perform root-to-leaf lookups, including secondary-index range scans (Q1) and index nested-loop joins (Q2) in Section 1. We illustrate this with two cases returning at most hundreds of rows on a table with 100 million records. Although index lookups access far fewer pages than a full table scan, the cumulative traversal overhead dominates execution time and limits concurrent scalability.

Case #1: B-Tree Traversal Dominance. As the result set size grows, B-Tree traversal increasingly dominates query execution time. As shown in Table 1, when the range expands from 1 to

Table 1: Traversal time of secondary index range queries.

Range size	1	10	25	50	100	200
Percentage (%)	21.80	45.82	61.72	71.91	79.08	83.02

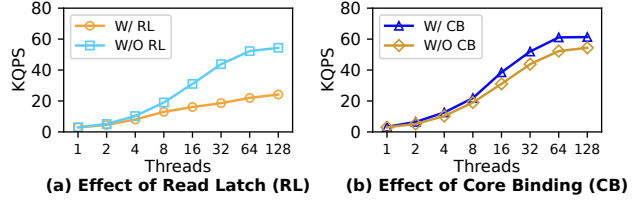


Figure 2: Secondary index range query scalability under different settings with 100 range size. W/ RL: with read latch, W/O RL: without read latch, W/ CB: with core binding, W/O CB: without core binding.

200 keys, the time share of B-Tree traversal surges from 21.80% to 83.02%, confirming traversal overhead as the primary bottleneck.

Case #2: Scalability Barriers in Concurrent Workloads. Beyond accumulated traversal overhead, B-Trees suffer from scalability limitations under concurrent access, as validated by the throughput results in Figure 2. In Figure 2(a), the standard B-Tree with read latches (W/ RL) plateaus at 24 KQPS, while the version without read latches (W/O RL) scales to over 50 KQPS at 128 threads. This gap stems from cache line ping-pong induced by latch contention [38]: even for read-only workloads, acquiring a read latch entails atomic updates to latch states (e.g., reader counts), which in turn trigger expensive cross-core cache coherence traffic. In addition, Figure 2(b) shows that binding threads to a single socket (W/ CB) improves throughput by up to 13%. This gain stems from the elimination of cross-socket node fetching, which reduces the CPU cycle share of node fetching from 13.04% to 4.51%. These results confirm that latch contention is the primary scalability barrier, while cross-socket node fetching further reduces throughput.

3.2 Adoption Barriers of Existing Techniques

Despite substantial progress in learned indexes, which span dynamic updates [11, 13, 15, 18, 50, 72, 80, 83], on-disk storage [33, 34, 44, 81], and scalability optimizations [18, 50, 83], mainstream databases hesitate to adopt them as replacements for B-Trees. Similarly, optimistic latching B-Tree variants mitigate the scalability bottleneck [3, 37, 38], yet see limited deployment in relational databases. These adoption gaps stem primarily from prohibitive engineering complexity. We elaborate on the key reasons below.

Reason #1: Indexes are deeply coupled with key DBMS components such as recovery and concurrency control, making full replacement highly nontrivial. For crash recovery, ARIES-style write-ahead logging is the de facto standard in mainstream databases [27, 51, 52, 85]. Under this scheme, index updates generate physiological redo logs (e.g., "insert record at position X in node Y") and logical undo logs. Furthermore, modern databases extensively employ multi-version concurrency control (MVCC) for

transaction management [4, 14, 58, 74], which maintains a version chain of older values for each index entry. Given the fundamentally different node layout of learned indexes, adapting them requires a full redesign of both log replay and version storage mechanisms. Attempts to integrate ALEX into PostgreSQL [12] further confirm this engineering burden: beyond logging and MVCC, such efforts also face challenges in disk persistence and optimizer cost estimation. Hence, index replacement is not a drop-in swap but entails holistic re-engineering.

Reason #2: Integrating optimistic latching B-Tree also demands extensive code changes. Optimistic latching effectively relieves latch contention, but requires operation retries upon read-write conflict [3, 37, 38]. To enable efficient retries across multiple call stacks, LeanStore proposes manual stack unwinding using “longjmp” in C [3]. Since “longjmp” bypasses C++ destructors, manual stack unwinding explicitly cleans up resources by wrapping resource allocation, destruction, and control-flow transition with hooks. Such pervasive, fine-grained modifications to the existing codebase incur a prohibitive implementation effort.

3.3 A Practical Alternative to Full Replacement

Given the high engineering complexity, replacing existing B-Tree indexes entirely remains prohibitive. A more practical alternative is augmentation: preserving existing infrastructure while employing learned models as auxiliary structures.

Augmentation overcomes replacement barriers and gains benefits at a lower cost. First, augmentation ensures compatibility with other modules: the learned model only assists traversals, while the B-Tree still handles critical responsibilities such as crash recovery and transaction control. In addition, we encapsulate optimistic retries within the auxiliary structures, thereby avoiding the massive intrusive code changes required by optimistic latching B-Trees. Traversals access the auxiliary structure once to obtain a hint cursor and validate the cursor position, thereby confining changes to a few hook points. Hence, augmentation entails far less integration effort than replacing with learned indexes and adopting optimistic variants. Finally, augmenting the B-Tree with auxiliary structures reduces the number of nodes accessed per traversal, thereby mitigating index traversal overhead and latch contention.

Requirements for index augmentation. However, applying learned models as auxiliary structures to B-Trees is nontrivial. Therefore, to be both practically viable and effective, auxiliary structures must satisfy the following stringent requirements:

- **R1: Lightweight memory footprint.** Given that indexes already dominate memory consumption [79], the auxiliary structure must be space-efficient. Otherwise, excessive memory usage reduces the number of cached index nodes, increasing I/O overhead.
- **R2: Correctness with low synchronization overhead.** When auxiliary structure updates are not synchronized with B-Tree updates, traversals may encounter stale mappings. Enforcing strong consistency eliminates this staleness but incurs substantial synchronization overhead. Thus, the auxiliary structure must minimize synchronization while ensuring correctness.
- **R3: Non-intrusive integration.** As stated in Section 3.2, index replacements can alter the physical B-Tree format, incurring data

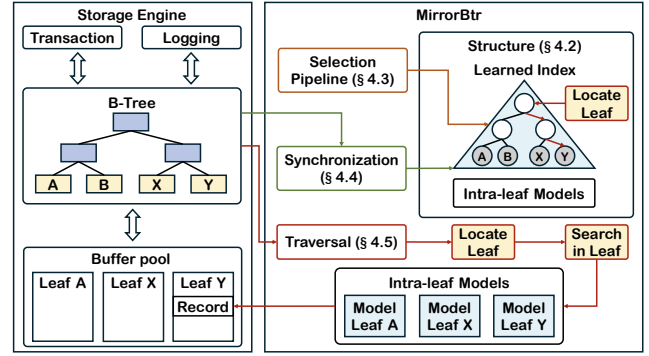


Figure 3: MirrorBtr design overview.

migration and engineering efforts on other modules (e.g., recovery, concurrency control). By non-intrusive integration, we preserve the on-disk layout while confining changes within the B-Tree.

4 DESIGN

This section presents MirrorBtr, a non-intrusive framework that constructs auxiliary structures to accelerate B-Tree traversals.

Architecture Overview. As illustrated in Figure 3, MirrorBtr employs a learned index to locate the target leaf for a given key, skipping inner node accesses during B-Tree traversals. To effectively meet the requirements for index augmentation (R1-R3), its core design principles are summarized as follows:

- **Leaf-node-level mapping.** To reduce the memory footprint and simplify synchronization, MirrorBtr mirrors key-to-leaf-node mappings rather than exact record positions (satisfying R1 and R2).
- **Non-intrusive auxiliary structure.** Building on leaf-node-level mappings, MirrorBtr employs an in-memory learned index without modifying the B-Tree node layout (satisfying R3).
- **Offline learned index selection.** An offline pipeline helps to identify a learned index candidate that offers gains with low overhead (satisfying R1 and R2).
- **Correct and efficient synchronization.** MirrorBtr allows B-Tree traversals to proceed concurrently with learned index updates. By enforcing rightward node splits and merges, stale results can be repaired by moving right along the leaf nodes. Thus, correctness and efficiency are ensured without global latches (satisfying R2).

Collectively, these design decisions enable MirrorBtr to deliver low overhead, seamless B-Tree integration, and traversal speedups across diverse workloads. We elaborate on each component below.

4.1 Leaf-node-level Mapping

A straightforward B-Tree augmentation scheme maintains record-level mappings, incurring substantial space and synchronization costs. Specifically, this scheme stores correspondences between keys and their exact record positions. Despite locating records with a single lookup, it incurs excessive space overhead: for an index with N records, it requires $O(N)$ extra storage, which becomes prohibitive at scale. Moreover, each record modification triggers an update to the auxiliary structure, raising maintenance overhead.

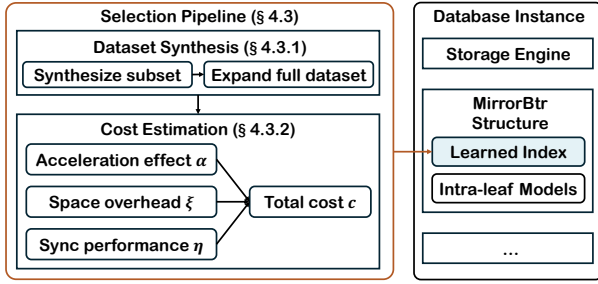


Figure 4: Selection pipeline overview.

To alleviate these problems, MirrorBtr adopts leaf-node-level mappings instead. The auxiliary structure tracks key-to-leaf-node mappings rather than individual record locations. Since a B-Tree with N records and a leaf fanout of m contains (N/m) leaf nodes, the space complexity reduces to $O(N/m)$. Synchronization is also simplified because updates are only triggered by B-Tree structural modifications (e.g., node splits and merges). Nevertheless, leaf-node-level mapping entails an additional intra-leaf search to locate the record. To compensate, MirrorBtr employs leaf-node-level mapping with lightweight intra-leaf models.

4.2 Non-intrusive Auxiliary Structure

Building upon leaf-node-level mappings, MirrorBtr employs a non-intrusive auxiliary structure. As shown in Figure 3, this auxiliary structure is a learned index constructed on the original B-Tree. It is maintained entirely in memory and leaves the on-disk node layout unmodified. The learned index stores mappings from separator keys to leaf node IDs, thus supporting the same search modes as the vanilla B-Tree. In our InnoDB instantiation, the separator key is the minimum key of each leaf node. To reduce the latency of intra-leaf searches, a per-leaf model learns the keys in the offset array. This model resides in node metadata for cached leaf nodes, thereby preserving on-disk node layouts and avoiding construction overhead for infrequently accessed nodes.

MirrorBtr provides a unified framework for integrating diverse learned indexes. Given emerging advances in learned indexes [11, 18, 50], the framework enables MirrorBtr to adopt suitable future ones without extensive modifications. As the learned index resides in memory, such adoption incurs no disk-format compatibility issues. Specifically, the framework requires the integrated learned indexes to support the following concurrent operations:

- **last_lt(k):** Find the last key-value pair whose key is less than k .
- **last_lt_eq(k):** Find the last key-value pair whose key is less than or equal to k .
- **insert(k, v):** Insert the key-value pair (k, v) to the learned index.
- **update(k, v):** Update the corresponding value of k to v .
- **delete(k):** Remove the key-value pair corresponding to key k .

Accordingly, read-only [32] and single-threaded updatable learned indexes [11, 13, 15, 72, 80] do not meet the requirements.

Table 2: Notation used in selection pipeline.

Notation	Definition
ϵ	Absolute model prediction error
M_l	Number of PLA models to keep a large ϵ
M_s	Number of PLA models to keep a small ϵ
n	Dataset size
d	Synthetic dataset stored in the index
i	B-Tree index
l	Learned index
$Thds$	Thread count set
t	Thread count in set $Thds$
w	Operation executed on the index
r_w	The ratio of w
$TP_w(i, \emptyset, d, t)$	Throughput of w with index i under d, t
$TP_w(i, l, d, t)$	Throughput of w when l augments i under d, t
$S(i, d)$	Space cost of i under d
$S(i, l, d)$	Space cost of i and l when l augments i under d
$\alpha(i, l, d)$	Acceleration effect when l augments i under d
$\xi(i, l, d)$	Space overhead when l augments i under d
$\eta(i, l, d)$	Sync performance when l augments i under d
$c(i, l, d)$	Total cost when l augments i under d

4.3 Learned Index Selection Pipeline

Existing benchmarks [62, 71] show that the performance diversity of learned indexes demands dedicated selection. However, they evaluate standalone learned indexes, ignoring the interactions with B-Tree during augmentation. We thus design an offline selection pipeline that identifies the most suitable index under augmentation-specific metrics, as illustrated in Figure 4. This selection is decoupled from the online instance, offloading heavy evaluations to a development environment and avoiding interference with foreground workloads. However, this design restricts direct access to raw user data for privacy compliance: exporting user data outside the database instance introduces leakage risks. It may violate regulations such as HIPAA and GDPR [1, 82]. To address this, MirrorBtr employs two key steps: (1) Dataset synthesis that generates diverse key distributions without violating privacy rules, (2) Cost estimation that combines augmentation-specific metrics into a unified cost. The notations in this section are summarized in Table 2.

4.3.1 Dataset Synthesis. Since the selection pipeline is isolated from the database with user data access restricted under privacy policies, we synthesize evaluation datasets that mirror empirical behavior. Prior work has established that learned index performance depends primarily on distributional hardness rather than specific values [46, 62, 71]. Based on this insight, we adopt a controllable synthesizer [71], quantifying the hardness using the number of piecewise linear approximation (PLA) models [75]. Specifically, M_l and M_s denote the number of PLA models for large ($\epsilon = 4096$) and small ($\epsilon = 32$) error thresholds, capturing global nonlinearity and local fine-grained fluctuation, respectively.

Conventional synthesizers directly generate full datasets for end-to-end index evaluation [13, 71]. However, this approach is infeasible for MirrorBtr, which is built atop the original B-Tree and trains its learned index on a subset of B-Tree leaf minimum keys

rather than the full dataset. This raises a critical observation: the distributional hardness of this subset is not uniquely determined by that of the full dataset. For example, under fixed global hardness M_l , increasing local hardness M_s tends to reduce the hardness of the subset. A higher M_s shrinks the span of local PLA models and decreases the number of keys per model. When the sampling stride exceeds this key count, only a few minimum keys lie within each model. Hence, intra-model slope differences and local perturbations are heavily undersampled, filtering out most local variations. In addition, even with identical M_s and M_l , different placements of PLA breakpoints alter the alignment between sampling positions and breakpoints, yielding minimum key subsets with distinct hardness.

Therefore, we first synthesize the subset for the learned index, then expand it into a full dataset for the B-Tree, as formalized in Algorithm 1. First, we fix the number of keys per leaf node k , which is used only within Algorithm 1, to determine the size of d_{min} and ensure every k -th key in d corresponds to a leaf minimum key (lines 1-2). We then generate the target leaf minimum-key subset d_{min} with hardness M_s and M_l using a controllable synthesizer [71] (line 3). To reserve space for intra-leaf keys in the B-Tree, we scale each minimum key for disjoint gaps. These gaps are then filled with consecutive keys to construct the full dataset d (lines 4-8). The result preserves the intended hardness for subsequent evaluation.

4.3.2 Cost Estimation. With the synthetic dataset generated in the previous step, we evaluate candidate learned indexes along three augmentation-specific metrics that align with the design goals:

- **Acceleration effect α .** As reads do not trigger synchronizations, α quantifies the average read throughput gain relative to the B-Tree across thread counts in set $Thds$:

$$\alpha(i, l, d) = \frac{1}{|Thds|} \sum_{t \in Thds} \frac{Tp_{read}(i, l, d, t)}{Tp_{read}(i, \emptyset, d, t)} \quad (1)$$

- **Space overhead ξ .** This metric measures the size ratio of the combined B-Tree and MirrorBtr to the vanilla B-Tree:

$$\xi(i, l, d) = \frac{S(i, l, d)}{S(i, d)} \quad (2)$$

- **Sync performance η .** Since synchronization with the learned index is only triggered by writes, η captures the average write throughput gain across thread counts in set $Thds$:

$$\eta(i, l, d) = \frac{1}{|Thds|} \sum_{t \in Thds} \frac{Tp_{write}(i, l, d, t)}{Tp_{write}(i, \emptyset, d, t)} \quad (3)$$

Algorithm 1: Dataset synthesis for selection pipeline

Input: dataset size: n , hardness parameters: M_s, M_l

Output: full synthetic dataset: d

```

1  $k \leftarrow$  fixed number of keys per leaf node
2  $n_{leaf} \leftarrow \lceil n/k \rceil$ 
3  $d_{min} \leftarrow$  hardness_synthesis( $n_{leaf}, M_s, M_l$ )
4  $d \leftarrow \emptyset$ 
5 foreach  $min\_key \in d_{min}$  do
6    $scaled\_key \leftarrow min\_key \times k$ 
7   for  $j = 0$  to  $k - 1$  do
8      $d \leftarrow d \cup \{scaled\_key + j\}$ 
```

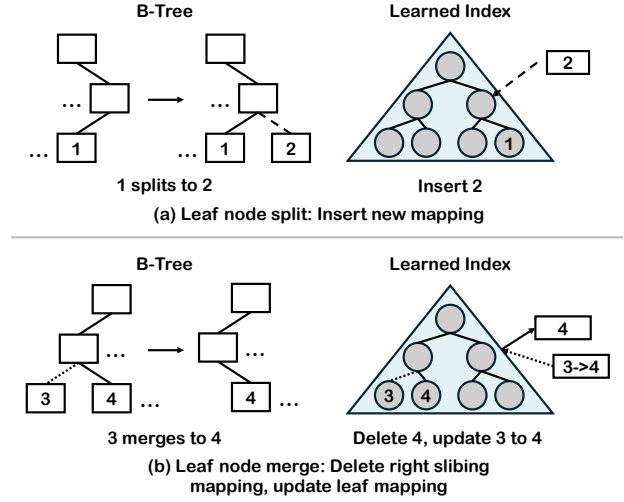


Figure 5: Leaf node split and merge with MirrorBtr.

Then we combine these metrics into a unified cost model:

$$c(i, l, d) = \frac{\xi(i, l, d)}{\alpha'(i, l, d) \cdot \eta'(i, l, d)} \quad (4)$$

where $\alpha'(i, l, d) = 1 + r_{read} \cdot (\alpha(i, l, d) - 1)$ and $\eta'(i, l, d) = 1 + r_{write} \cdot (\eta(i, l, d) - 1)$ are the adjusted acceleration effect α and sync performance η , weighted by workload ratios r_{read} and r_{write} . Specifically, under read-intensive workloads ($r_{write} \rightarrow 0$), $\eta' \rightarrow 1$, and the cost is dominated by the acceleration effect α' . Conversely, under write-intensive workloads, the cost is dominated by sync performance η' . A lower cost c indicates that the learned index l is better suited for MirrorBtr on dataset d . Section 5.2 applies this pipeline to evaluate candidate learned indexes.

4.4 Correct and Efficient Synchronization

In MirrorBtr, the learned index maintains an independent key-to-leaf mapping that is updated separately from B-Tree structural modifications. This creates a time window in which the B-Tree and the learned index are temporarily inconsistent, introducing two synchronization challenges: (1) *correctness*: a concurrent traversal may use a stale result from the learned index to locate a now-split or merged node, and (2) *efficiency*: the mapping update may become a centralized bottleneck that serializes concurrent traversals. To address (1), we adapt the rightward-only invariants from B-link trees [21, 35] to the inter-index setting: B-Tree nodes split and merge only to the right, and traversals resolve staleness via rightward sibling search. To address (2), we update mappings in the learned index without serializing concurrent traversals. The remainder of this subsection details synchronization mechanisms for initialization and B-Tree structural modifications.

4.4.1 Initialization. The learned index in MirrorBtr is initialized with all minimum keys and their corresponding leaf node IDs from the B-Tree. Specifically, this process traverses the lowest-level inner nodes, collects the minimum keys and their associated node IDs, and loads these mappings into the learned index. A singleton pattern is employed to prevent redundant index population.

4.4.2 Handling leaf node splits. A leaf node splits upon overflow: a new leaf node is created, records are redistributed, and inner nodes are updated to reference the new node. Since the learned index maintains its own mapping from keys to leaf nodes, a concurrent traversal may query the learned index before the mapping update. The traversal then retrieves a stale leaf node ID, only to find the target record relocated to the newly created node. MirrorBtr addresses this by applying rightward-only splits; thus the target record can only reside in the originally retrieved node or its right siblings. We then insert the mapping for the newly created node into the learned index so that future traversals can directly locate it. As detailed in Figure 5(a), when leaf node 1 is split into node 2, a mapping for the new leaf node 2 is inserted into the learned index. The mapping update on the learned index does not need to serialize concurrent traversals, since a traversal that observes a stale result can locate the correct leaf via rightward sibling search.

4.4.3 Handling leaf node merges. A leaf node merges upon underflow by consolidating with a sibling, triggering updates to child references in inner nodes. As with node splits, when a concurrent merge removes a leaf node from the B-Tree, traversals may retrieve this reclaimed node. Similarly, MirrorBtr applies rightward-only merges and rightward sibling search from B-link trees [21, 35]. Since merge always consolidates with the right sibling, a traversal that arrives at the reclaimed node can locate the target record by scanning right. To reflect the B-Tree leaf node merge, the learned index is updated accordingly. As illustrated in Figure 5(b), merging node 3 with its right sibling node 4 first removes the mapping of node 4 from the learned index and then updates the mapping of node 3 to reference node 4. Together with the rightward sibling search, a deferred reclamation mechanism is employed to ensure memory safety for concurrent traversals. This mechanism marks the merged leaf node as pending reclamation, unlinks it from the B-Tree, but does not recycle its storage. Reclamation is deferred until all pre-merge transactions are complete, since traversals cannot access the reclaimed node via the learned index. When a traversal encounters a node marked as pending reclamation, it searches the right siblings directly since the records have been merged there.

Deferred reclamation ensures safe concurrency with negligible overhead. A long-lived transaction delays merged node reclamation but does not amplify traversal cost. Only traversals that queried the learned index before the update may locate the pending-reclamation node, independent of transaction duration. Other B-Tree operations (e.g., leaf node scans) remain unaffected, as they never access unlinked nodes. Since OLTP transactions are typically short-lived [4], merged nodes are reclaimed promptly.

In addition, MirrorBtr correctly resolves an intermediate state during learned index updates. Consider the transient interval after deleting the mapping for node 4 but before updating the mapping for node 3. During this interval, traversals for keys in node 4 temporarily resolve to node 3. Upon reaching node 3, the traversal acquires a read latch and finds it marked as pending reclamation. The traversal then accesses its right sibling node 4, where the merged record resides. It thus automatically recovers from this intermediate state and ensures correctness.

Algorithm 2: Locate the leaf node with learned index

Input: search key: k , search mode: m
Output: leaf node containing the target record: $leaf$

```

1  $li$  denotes the learned index
2  $bp$  denotes the buffer pool
3  $leaf\_id \leftarrow m \in \{LT, GE\} ? li.last\_lt(k) : li.last\_lt\_eq(k)$ 
4  $leaf \leftarrow bp.get(leaf\_id)$ 
5  $next\_leaf\_id \leftarrow leaf.next\_leaf\_id$ 
6  $move\_right \leftarrow \text{true}$ 
7  $op \leftarrow m \in \{LT, GE\} ? ">" : "\geq"$ 
8 while  $move\_right$  and  $next\_leaf\_id$  is not null do
9    $move\_right \leftarrow k \text{ op } last\_index\_key(leaf)$ 
10  if  $move\_right$  then
11     $next\_leaf \leftarrow bp.get(next\_leaf\_id)$ 
12    if  $m \in \{GE, GT\}$  then
13       $move\_right \leftarrow \text{true}$ 
14    else
15       $move\_right \leftarrow$ 
16         $k \text{ op } first\_index\_key(next\_leaf)$ 
17    if  $move\_right$  then
18       $leaf \leftarrow next\_leaf$ 
19       $next\_leaf\_id \leftarrow leaf.next\_leaf\_id$ 
19 return  $leaf$ 

```

4.5 Accelerated B-Tree Traversal

MirrorBtr augments B-Tree with a two-step acceleration scheme: (1) Leaf node lookup. The learned index maps search keys to leaf node IDs, skipping inner node probes. (2) Intra-leaf model-guided search. The intra-leaf model predicts a slot in the offset array to narrow the search range. This subsection details this scheme.

4.5.1 Locate leaf node with learned index. Read and optimistic write operations do not incur structural modifications and can directly locate leaf nodes via the learned index. In contrast, pessimistic writes require latch coupling [21] from the root node to modify the B-Tree, thus unable to skip inner nodes with the learned index.

As detailed in Algorithm 2, this process first queries the learned index to obtain a leaf node ID (line 3), then performs a rightward sibling search to locate the correct leaf (lines 8–19). Specifically, it compares k against $last_index_key(leaf)$ to decide whether to access the right sibling (line 9), using $>$ for LT/GE and $\geq$ for LE/GT. If movement is needed, the algorithm fetches the right sibling and validates the move by comparing k against $first_index_key(next_leaf)$; GE and GT modes always move because the current leaf does not contain the first record satisfying these modes (lines 12–15). If the move is confirmed, the algorithm advances to the right sibling and iterates; otherwise, it terminates and returns the current leaf (lines 16–18). Nodes marked as pending reclamation redirect traversals immediately to their right siblings.

Correctness analysis. We prove correctness via a loop invariant: at the start of each while-loop iteration (line 8), the target record resides in $leaf$ or one of its right siblings. **(1) Initialization.** Without concurrent modifications, the learned index returns the exact

leaf_id via error correction in learned indexes [11, 32]. With concurrent splits or merges, the rightward-only invariants from B-link trees [21, 35] ensure the target can only reside in the retrieved node or its right siblings. **(2) Maintenance.** Each iteration compares k against $last_index_key(leaf)$ and $first_index_key(next_leaf)$. If the target lies beyond *leaf*, the algorithm advances to the right sibling, preserving the invariant. **(3) Termination.** The loop exits when (a) *move_right* is false, confirming the target falls within *leaf*'s key range, or (b) *next_leaf_id* is null, meaning no right sibling exists. In both cases, the invariant eliminates the “right siblings” possibility, so the target must reside in the returned *leaf*.

Efficiency trade-off. The rightward sibling search trades rare superfluous sibling fetches for a non-intrusive augmentation requirement. For example, consider an LT query with key k that falls between a leaf's last key and its right sibling's first key. The sibling is fetched for validation, but no movement is performed since its first key exceeds k . Storing the right sibling's first key in the current node eliminates these redundant fetches. Nevertheless, it necessitates modifying the B-Tree's physical layout, violating the non-intrusive integration requirement (R3 in Section 3.3).

4.5.2 Search leaf node with intra-leaf model. Upon receiving a new query, the intra-leaf model predicts a position within the offset array, and an exponential search accurately locates the target slot. In a sparse layout (e.g., InnoDB [64]), each slot covers a group of records, requiring a linear scan to find the record. In a dense layout (e.g., PostgreSQL [25]), the slot directly yields the target record.

Intra-leaf model selection. We begin with linear models due to their computational efficiency [11, 17] and wide adoption [9, 13, 40]. While complex models such as neural networks are also explored in learned indexes [32, 73], their added complexity is rarely justified by marginal performance gains [11, 40]. We evaluate three representative candidates to identify an effective one: linear regression (LR) and two piecewise linear regression (PLR) variants [75], namely OptimalPLR and GreedyPLR. The evaluation uses three real-world datasets: genome [71], osm [46], and libio [71] (see Section 5.1 for details). We first build each model and measure its space and construction cost. We then perform 50 deletes followed by 50 inserts, recording the average prediction error after each round and the average search latency across all rounds. As illustrated in Figure 6(a)–(c), all three models exhibit a sawtooth error pattern rather than monotonic degradation. Although LR exhibits the highest error, Figure 6(d) shows that it achieves the lowest average search latency, which is consistently lower than that of binary search. This is because PLR's error-bounded segmentation yields multiple segments per leaf node, forcing each search to locate an applicable segment. Moreover, Figure 6(e)–(f) shows that PLR incurs higher space and construction costs. Overall, LR is preferred due to its faster search, fixed space, and simple construction.

Intra-leaf model construction. Intra-leaf models are built lazily upon first access to that leaf, using a singleton guard to avoid concurrent redundant training. This process scans records in the offset array and trains a model that maps record keys to slot positions.

Intra-leaf model updates. The intra-leaf model is rebuilt upon changes to the offset array structure. For sparse offset arrays (e.g., page directory in InnoDB), a new slot is created when the size of a

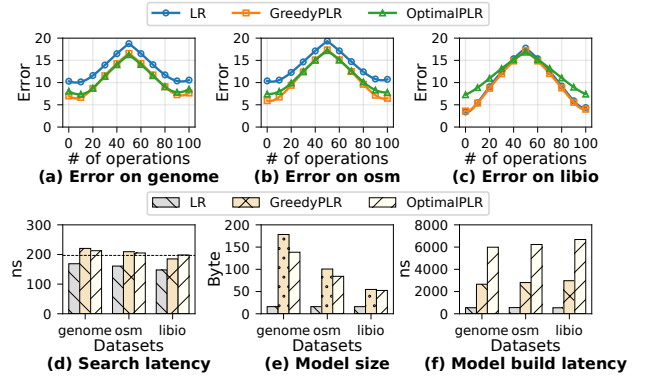


Figure 6: Intra-leaf model evaluation results. The dashed line in subfigure (d) represents binary search latency.

record group exceeds the maximum threshold, and adjacent slots merge if the size falls below the minimum. When triggered, the intra-leaf model is retrained by scanning records in the offset array. For dense offset arrays, entry count tracks the record count directly, thus requiring more frequent retraining.

5 EVALUATION

5.1 Experimental Setup

Platform. All experiments are run on an Ubuntu 22.04 server equipped with two 24-core Intel Xeon Gold 5318Y processors (hyper-threading enabled) and 512GB of RAM. We implement MirrorBtr on the InnoDB storage engine in MySQL 8.0.41 and make the source code publicly available ¹.

Baseline. We compare against two InnoDB built-in baselines.

- **B-Tree** is the default index in InnoDB, which is a balanced tree for efficient point and range queries.
- **Adaptive Hash Index (AHI)** [63] is an in-memory, hash-based auxiliary structure that accelerates lookups by mapping keys to record pointers, thus avoiding multi-level B-Tree traversal.

Datasets. The evaluations use the following datasets, most of which are real-world datasets from SOSD [46] and GRE [71] benchmarks. Among them, osm and genome exhibit higher nonlinearity, posing greater challenges for learned models [71].

- **autoinc** [66] refers to the auto-incremented sequence numbers, which serve as the default index key in Sysbench [30].
- **libio** [71] contains the Repository IDs from libraries.io.
- **osm** [46] is derived from uniformly sampled OpenStreetMap location coordinates.
- **genome** [71] is created from loci pairs in human chromosomes.

Workloads. We extend the Sysbench [30] workloads as follows. In each experiment, we populate the database with 100 million records and run benchmark operations for 30 seconds.

- **Secondary index range query** issues range queries on the secondary index with a range size of 100.

¹<https://github.com/unconsolable/MirrorBtr>

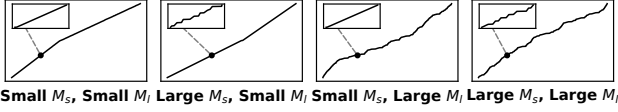


Figure 7: Synthetic subset CDFs. M_s and M_l control local and global hardness, respectively.

Table 3: Average speedup ratio and learned index size. The best values are marked in bold.

Dataset	libio			osm			genome		
Algorithm	ALEX+	SALI	LOFT	ALEX+	SALI	LOFT	ALEX+	SALI	LOFT
Speedup Ratio	1.62	1.59	1.08	1.28	1.02	0.84	1.42	1.38	0.97
Size (MB)	71.4	52.7	81.4	87.9	226.8	76.2	72.2	84.2	79.9

- **Primary index nested-loop join** performs index joins on the primary index using 100 randomly selected join keys.
- **Read-write on indexes** runs a transaction of point queries, secondary index range queries, updates, and paired delete-inserts.
- **Write-only on indexes** inserts to both the primary and secondary indexes.
- **Skewed inserts** preloads two non-contiguous key segments and concentrates concurrent inserts in the gap between them [18].
- **Intrinsic-order inserts** issue inserts under partial data sortedness using the (K, L) -sortedness metric [59].

5.2 Learned Index Selection

This experiment identifies the learned index used in subsequent evaluations. We consider ALEX+ [71], SALI [18] and LOFT [50] as candidates, while Hyper [83] is excluded due to unavailable source code. We integrate the selection pipeline into an in-memory B-link tree [7] and evaluate cost over four synthetic datasets whose subset CDFs are shown in Figure 7. As reported in Figure 8, ALEX+ achieves the lowest cost across most settings because it stores mappings solely in data nodes, enabling sibling scans without parent backtracking. SALI incurs parent backtracking for sibling scans due to co-located mappings and subtree pointers, along with memory overprovisioning [40], while LOFT must scan extra mappings due to their unsorted order, so both incur higher cost. Consistent performance trends on synthetic and real-world datasets (libio and osm) validate the use of synthetic data for cost estimation.

To validate the cost estimation, we measure the average speedup ratio and learned index size with write ratios from 0 to 1 (step 0.2). As shown in Table 3, ALEX+ achieves the highest speedup ratio across all three datasets with competitive space overhead. Thus, the cost model correctly identifies the optimal candidate, and we select ALEX+ for subsequent evaluations. Notably, while we evaluate state-of-the-art indexes, MirrorBtr can easily incorporate future advanced designs and identify the suitable one.

ALEX+ lacks support for structural modifications on the root node, making insertions beyond key boundaries prone to deadlocks [50]. To resolve this issue while preserving latch-free inner-node reads of ALEX+, we adopt a copy-on-write scheme for root

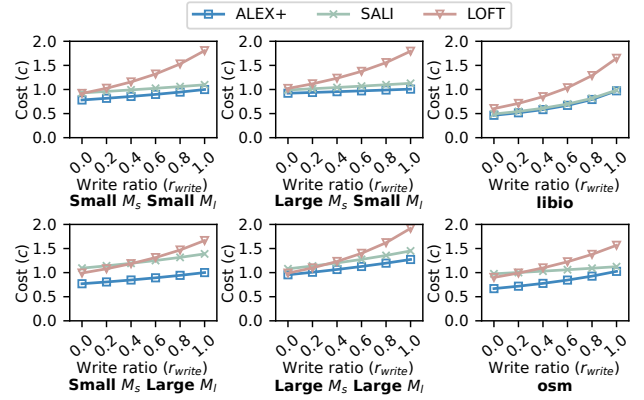


Figure 8: Cost model evaluation results.

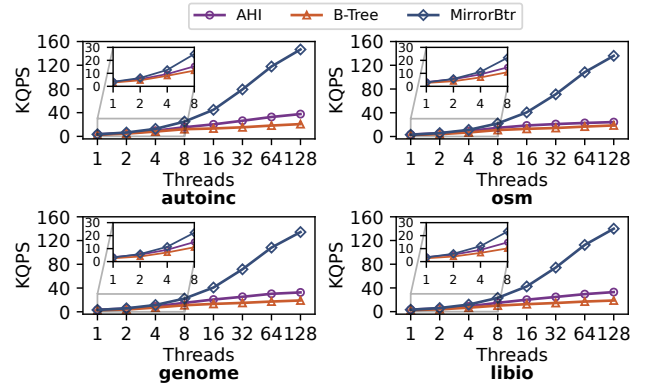


Figure 9: Throughput of secondary index range query.

expansion. We duplicate the root and boundary leaf nodes, modify the copies, then atomically write back pointers, with all modifications serialized via per-node latches.

5.3 In-memory Workloads

5.3.1 Traversal-intensive queries. This experiment measures the throughput under workloads dominated by repeated primary index traversals, using secondary index range queries and primary index nested-loop joins as representative cases. Figure 9 and Figure 10 report that MirrorBtr consistently outperforms all baselines under these queries. For range queries, MirrorBtr outperforms AHI and B-Tree by up to 5.60 $\times$ and 7.45 $\times$, respectively. For nested-loop joins, the performance gains reach 5.81 $\times$ and 7.26 $\times$, respectively. This stems from MirrorBtr’s traversal with ALEX+, which employs latch-free reads and optimistic latching on data nodes, so the traversal only acquires a read latch on each B-Tree leaf node. Nevertheless, B-Tree remains competitive at low concurrency (*i.e.*, < 8 threads) since latch contention is minor. Even so, MirrorBtr still delivers consistent gains by reducing traversal latency, as shown in the zoomed inset of Figure 9 and 10. Specifically, MirrorBtr improves throughput over B-Tree by an average of 1.47 $\times$ for secondary index range queries

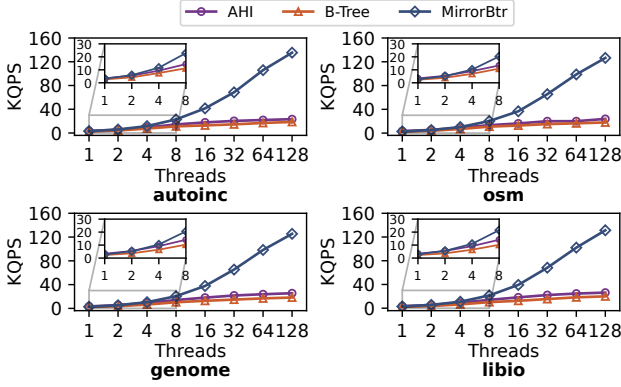


Figure 10: Throughput of primary index nested-loop joins.

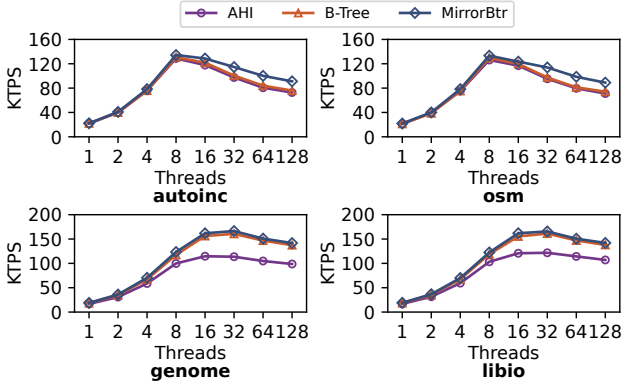


Figure 11: Throughput of write-only on indexes.

with 2 threads. Among the baselines, AHI outperforms B-Tree by bypassing tree traversal, but its single read-write latch per hash table limits scalability. B-Tree suffers from latch contention at every level of the root-to-leaf path.

5.3.2 Write-only on indexes. This experiment evaluates insert performance under distinct patterns: autoinc and osm produce sequential inserts, while genome and libio produce random inserts. Figure 11 shows that MirrorBtr outperforms baselines, though the improvement is modest (reasons detailed in Section 5.6.2). MirrorBtr benefits sequential inserts more than random inserts, because sequential inserts confine splits to the rightmost leaf, causing most threads to acquire latches on the same inner nodes. MirrorBtr’s learned index bypasses inner nodes and locates the target leaf directly, relieving such latch contention.

5.3.3 Learned index stress test. Since learned indexes may degrade under skewed [18, 62] and intrinsic-order inserts [59], we evaluate MirrorBtr under these two challenging patterns with 64 threads. Skewed inserts first sort the entire dataset and preload two non-contiguous segments (0–50 million and 75–125 million), then insert a segment (50–75 million) as a concentrated hotspot. Intrinsic-order inserts use the (K, L) -sortedness metric, where K denotes

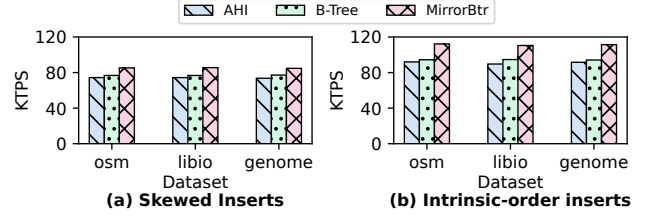


Figure 12: Throughput of skewed and intrinsic-order inserts.

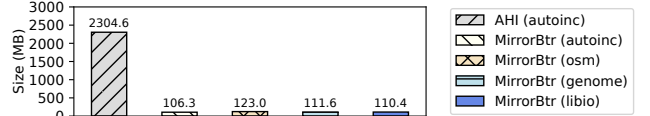


Figure 13: Space overhead of AHI and MirrorBtr.

the fraction of out-of-order entries and L denotes the maximum displacement. We set $(K = 25\%, L = 25\%)$, under which the write throughput of ALEX drops by 36% [59]. As shown in Figure 12, MirrorBtr outperforms the best-performing baseline, B-Tree, by 11.5% under skewed inserts and 18.9% under intrinsic-order inserts. The advantage of MirrorBtr is not diminished because updates to the learned index are only triggered by B-Tree leaf node splits and merges. Hence, the overhead of learned index updates is effectively amortized by individual inserts.

5.3.4 Space overhead. Figure 13 compares the memory overhead between AHI and MirrorBtr. AHI’s space is constant across distributions, so the result on the autoinc dataset suffices for others. In contrast, the memory footprint of MirrorBtr varies slightly with dataset hardness. At minimum, MirrorBtr incurs only 4.61% of AHI’s size, equivalent to 0.51% of the 20.44GB B+-Tree index size. This substantial saving originates from their divergent mapping granularities. AHI caches record-level mappings proportional to the number of records, whereas MirrorBtr maintains leaf-node-level mappings, resulting in far fewer entries.

5.4 Larger-than-memory Workloads

This section evaluates performance when the dataset exceeds buffer pool capacity. We evaluate secondary index range queries and read-write queries using 16 threads while varying the buffer pool size. As shown in Figure 14, MirrorBtr outperforms all baselines, achieving up to $2.29\times$ higher throughput over B-Tree for range queries and $1.53\times$ for read-write queries. AHI consistently underperforms B-Tree since its record-level mappings consume excessive memory, thus competing with the B-Tree for limited buffer pool space and incurring additional I/O. In contrast, MirrorBtr adopts leaf-node-level mappings, significantly reducing space overhead and leaving more memory for B-Tree. Moreover, its accelerated traversal further improves throughput. As the buffer pool size grows, reduced I/O shifts the bottleneck to CPU, further amplifying MirrorBtr’s traversal acceleration.

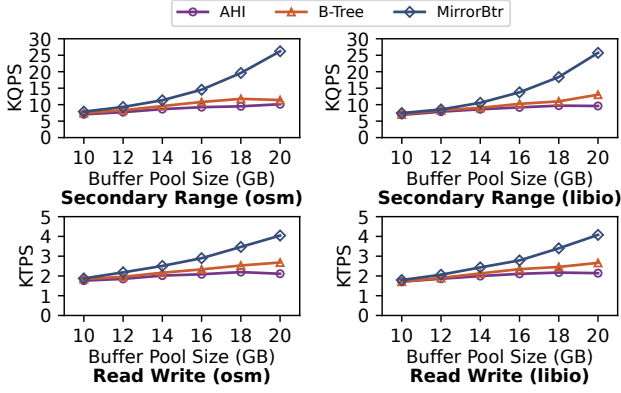


Figure 14: Throughput with different buffer pool sizes.

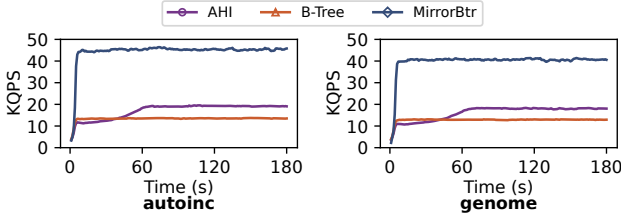


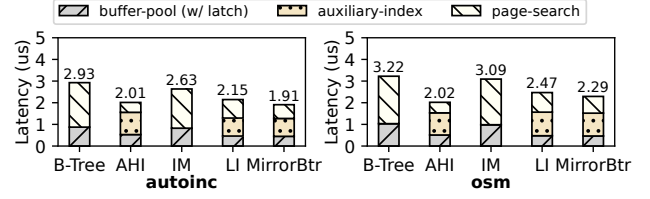
Figure 15: Throughput on a fresh instance over time.

5.5 Construction Time

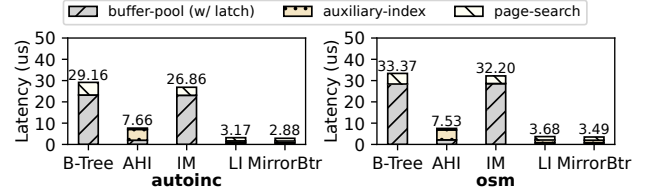
This experiment evaluates the warm-up time to reach steady-state throughput from a cold start. We execute secondary index range queries on freshly initialized instances with 16 threads and track throughput over time. As shown in Figure 15, MirrorBtr reaches stable performance within 20 seconds, comparable to the baseline B-Tree. This efficiency is attributed to its lightweight construction, which only scans the lowest-level inner nodes to build the learned index in less than 2 seconds. In addition, intra-leaf models are lazily trained upon their first access, avoiding centralized bottlenecks. In contrast, AHI takes more than 1 minute to warm up because it populates a hash table with massive record-level mappings. Its throughput rises gradually as entries are imported.

5.6 Performance Breakdown

5.6.1 Primary index traversal latency breakdown. We measure primary index traversal latency on secondary index range queries, since each query traverses the primary index to fetch a full record for each matching key, making it the dominant cost. To isolate component contributions, we further evaluate two ablation variants, LI (learned index only) and IM (intra-leaf model only). As illustrated in Figure 16, learned index is the primary contributor to latency reduction. On autoinc, LI reduces single-threaded traversal latency by 26.6% over B-Tree by skipping buffer pool accesses and searches on inner nodes. Under 64 threads, LI achieves 89.1% reduction by eliminating latch contention on inner nodes. IM predicts positions within the offset array of leaf nodes, reducing latency by 10.2%

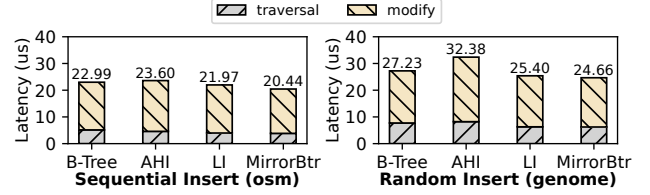


(a) Single-threaded execution.

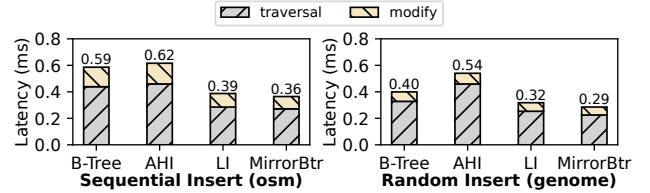


(b) 64-threaded execution.

Figure 16: Breakdown of primary index traversal. MirrorBtr ablations: LI (learned index only), IM (intra-leaf model only).



(a) Single-threaded execution.



(b) 64-threaded execution.

Figure 17: Breakdown of insertion. MirrorBtr ablations: LI (learned index only).

under 1 thread and 7.8% under 64 threads. Combining both components, MirrorBtr reduces single-threaded traversal latency by 34.8% and 64-threaded latency by 90.1% on autoinc. The latency reduction is smaller on osm (28.8% under 1 thread and 89.5% under 64 threads), as its less linear CDF increases both the tree height of ALEX+ and the prediction error of intra-leaf models. Given the modest contribution of IM, subsequent analyses omit this variant.

5.6.2 Insertion latency breakdown. To analyze the reasons behind the modest improvement of MirrorBtr under insertion workloads, we decompose its latency in Figure 17. Under a single thread, modification dominates the latency, leaving little room for traversal

acceleration. Under 64 threads, node splits block concurrent traversals due to InnoDB’s B-Tree concurrency control. MirrorBtr reduces insertion latency by accelerating traversals of optimistic insertions. However, it cannot mitigate the write-side blocking during node splits, offering limited improvement to concurrent scalability.

6 DISCUSSION AND FUTURE DIRECTIONS

Extend to other databases. Although MirrorBtr is implemented in InnoDB, its design is database-agnostic. Porting MirrorBtr requires: (1) inner-node separator keys to initialize the learned index, (2) right-sibling links in leaf nodes for rightward sibling searches, (3) per-leaf sorted offset arrays for intra-leaf model training and prediction. These properties are readily available in mainstream databases, such as PostgreSQL [25], MariaDB [47], and SQL Server [31].

Monetary efficiency. MirrorBtr reduces the overall monetary cost. With a memory footprint of only 0.51% of the B-Tree index size, MirrorBtr achieves higher throughput with fewer threads than the B-Tree. Given that 1 vCPU core is 8× more expensive than 1 GB memory in public clouds [45], this CPU efficiency leads to a net reduction in total cost of ownership.

Lightweight construction. As an in-memory auxiliary structure, MirrorBtr is reconstructed during database initialization. However, the overhead is reduced by two key strategies: (1) **Scanning fewer nodes.** The learned index is constructed by scanning the lowest-level inner nodes, which are fewer than the leaf nodes. (2) **Building intra-leaf models for cached nodes.** The intra-leaf models are built lazily for nodes in the buffer pool, bypassing cold nodes. As shown in Section 5.5, MirrorBtr achieves stable throughput after a warm-up period similar to the baseline B-Tree. Moreover, this one-time cost is further amortized by the entire instance lifetime (often months in production), making the overhead negligible.

Support for variable-length keys. MirrorBtr currently supports fixed-length keys. To handle variable-length string keys, learned indexes for strings [69, 78] can be incorporated through the unified interface of MirrorBtr. For the selection pipeline, existing hardness metrics fail to capture the highly skewed prefix commonly found in strings [78]. Instead, the Group Partial Key Length (GPKL) metric and the corresponding synthesizer can be adopted [78]. Intra-leaf models are trained on poor man’s normalized keys [24, 54]: extract 8 bytes after the node’s common prefix as an unsigned integer.

7 RELATED WORK

Learned index deployment: KV stores vs. relational databases. Recent works [2, 9, 28, 43, 68, 70] and a benchmark [41] use learned indexes as lightweight helpers in LSM-based key-value stores such as RocksDB or LevelDB. However, augmenting B-Trees, the primary index in relational OLTP databases, is harder because they require careful update handling, while immutable SSTables are ideal for learning [9]. In addition, the effectiveness of the learned index in relational databases remains uncertain. MirrorBtr fills this gap by exploring a non-intrusive method that layers learned indexes on existing B-Trees in relational databases.

Integrating AI4DB techniques in relational databases. AI4DB techniques have shown promise in autonomous database optimization [6, 19, 50, 77]. Yet its practical impact hinges on system-level

integration, shown by recent efforts in relational DBMSs [56, 86]. PilotScope [86] provides a general framework on PostgreSQL for knob tuning, index tuning, cardinality estimation, and query optimization. NeurDB [84] incorporates learned concurrency control and query optimization into PostgreSQL. Redshift MDDL [10] learns frequent scan predicates to reorganize data layout for block skipping. In contrast, the integration of learned indexes remains underexplored. MirrorBtr supports diverse updatable learned indexes to accelerate B-Tree traversal and alleviate latch contention.

Modern B-Tree variants. Several works replace B-Tree with redesigned mechanisms to address bottlenecks such as latch contention [39], point-range tradeoffs [76], and insertion efficiency [60], as surveyed in [22, 23]. MirrorBtr instead adopts an augmentation strategy that preserves the existing B-Tree and layers learned models as auxiliary structures. Since MirrorBtr accelerates traversal via locating the leaf node directly, it applies to variants that keep a standard root-to-leaf B-Tree traversal. For example, BP-Tree [76] and QuIT [60] retain standard traversal in reads, making augmentation with MirrorBtr feasible. In contrast, Bw-Tree [39] introduces latch-free traversal that MirrorBtr does not guarantee, making it unsuitable for augmentation with MirrorBtr.

Database cracking. Database cracking [29] shares a non-intrusive augmentation philosophy with MirrorBtr. However, database cracking follows a query-driven update scheme, whereas MirrorBtr adopts a write-driven one. For database cracking, each query triggers a physical data reorganization. In contrast, MirrorBtr is updated under B-Tree structural modifications or changes on the leaf node offset array.

8 CONCLUSION

We propose MirrorBtr, a framework that speeds up B-Tree traversal without intrusive integration. It uses learned models for inner and leaf nodes to save space and simplify synchronization. A selection pipeline helps choose the most suitable learned index; ALEX+ is used for InnoDB integration. Experiments show that MirrorBtr improves traversal performance with low overhead.

ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China under Grant No. 2023YFB4502400, in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant No. JYB2025XDXM901, in part by the National Natural Science Foundation of China under Grant No. 62272223, U22A2031, U24B20153, 62402212, in part by the New Generation Information Technology Innovation Project 2023 (2023IT196), in part by the Fundamental Research Funds for the Central Universities under Grant No. 2024300349 and 2025300309, in part by the Natural Science Foundation for Young Scientists of Jiangsu Province under Grant No. BK20241245, in part by the “111 Center” (No. B26023), in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, in part by Singapore MOE AcRF Tier-2 Grant (MOE-T2EP20224-0005), in part by Alibaba Group through the Alibaba Research Intern Program. We also thank Zheyu Miao, Yuming Yang, and Mingwei Xu from Alibaba Group for the valuable help throughout the project.

REFERENCES

- [1] Sepideh Abedini, Shubhankar Mohapatra, DB Emerson, Masoumeh Shafieinejad, Jesse C Cresswell, and Xi He. 2025. MaskSQL: Safeguarding Privacy for LLM-Based Text-to-SQL via Abstraction. In *NeurIPS 2025 Workshop on Regulatable ML*.
- [2] Hussam Abu-Libdeh, Deniz Altinbükten, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. *CoRR* abs/2012.12501 (2020). arXiv:2012.12501 <https://arxiv.org/abs/2012.12501>
- [3] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 06.-10. März 2023, Dresden, Germany, Proceedings (LNI)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.), Vol. P-331. Gesellschaft für Informatik e.V., 259–281. <https://doi.org/10.18420/BTW2023-13>
- [4] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (2023), 1426–1438. <https://doi.org/10.14778/3583140.3583157>
- [5] AWS. 2025. Amazon Aurora Instance Types. Retrieved August 22, 2026 from <https://aws.amazon.com/rds/aurora/instance-types/>
- [6] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek R. Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data* 2, 1 (2024), 50:1–50:26. <https://doi.org/10.1145/3639305>
- [7] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1235–1248. <https://doi.org/10.14778/3583140.3583143>
- [8] Jeremy Cole. 2013. B+Tree Index Structures in InnoDB. Retrieved August 22, 2026 from <https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/>
- [9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnathan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 155–171. <https://www.usenix.org/conference/osdi20/presentation/dai>
- [10] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 55–67. <https://doi.org/10.1145/3626246.3653379>
- [11] 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 *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984. <https://doi.org/10.1145/3318464.3389711>
- [12] Gary Evans and Nischay Kothari. 2025. Exploring learned indexes in PostgreSQL: Theory and Implementation. Retrieved August 22, 2026 from <https://www.postgresql.fastware.com/blog/exploring-learned-indexes-in-postgresql>
- [13] 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. <https://doi.org/10.14778/3389133.3389135>
- [14] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *Proc. VLDB Endow.* 15, 11 (2022), 2797–2810. <https://doi.org/10.14778/3551793.3551832>
- [15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1189–1206. <https://doi.org/10.1145/3299869.3319860>
- [16] Chuqing Gao, Shreya Ballijepalli, and Jianguo Wang. 2024. Revisiting B-tree Compression: An Experimental Study. *Proc. ACM Manag. Data* 2, 3 (2024), 169. <https://doi.org/10.1145/3654972>
- [17] Jiak Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 315–327. <https://doi.org/10.1109/ICDE55515.2023.00031>
- [18] Jiak Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALL: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proc. ACM Manag. Data* 1, 4 (2023), 258:1–258:25. <https://doi.org/10.1145/3626752>
- [19] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1 (2025), 2:1–2:26. <https://doi.org/10.1145/3709652>
- [20] Google. 2025. Cloud SQL Pricing. Retrieved August 22, 2026 from <https://cloud.google.com/sql/pricing>
- [21] Goetz Graefe. 2010. A Survey of B-tree Locking Techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26. <https://doi.org/10.1145/1806907.1806908>
- [22] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402. <https://doi.org/10.1561/19000000028>
- [23] Goetz Graefe. 2024. More Modern B-Tree Techniques. *Found. Trends Databases* 13, 3 (2024), 169–249. <https://doi.org/10.1561/1900000070>
- [24] Goetz Graefe and Per-Ake Larson. 2001. B-Tree Indexes and CPU Caches. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 349–358. <https://doi.org/10.1109/ICDE.2001.914847>
- [25] The PostgreSQL Global Development Group. 2025. PostgreSQL: Documentation: 17.64.1 B-Tree Indexes. Retrieved August 22, 2026 from <https://www.postgresql.org/docs/17/btree.html>
- [26] Jonah H. Harris. 2021. Re: ML-based indexing ("The Case for Learned Index Structures, a paper from Google") - Mailing list pgsql-hackers. Retrieved August 22, 2026 from <https://postgrespro.com/list/id/CAH2-WzkwoPr028p+3d=kNjd6hoA3V+Zy6hou-LcHaV8FymH9Q@mail.gmail.com>
- [27] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 877–892. <https://doi.org/10.1145/3318464.3389716>
- [28] Alireza Heidari, Amirhossein Ahmadi, and Wei Zhang. 2025. DobLIX: A Dual-Objective Learned Index for Log-Structured Merge Trees. *Proc. VLDB Endow.* 18, 11 (2025), 3965–3978. <https://www.vldb.org/pvldb/vol18/p3965-heidari.pdf>
- [29] 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. <https://cidrdb.org/cidr2007/papers/cidr07p07.pdf>
- [30] Alexey Kopytov. 2004. Sysbench: A System Performance Benchmark. Retrieved August 22, 2026 from <https://github.com/akopytov/sysbench>
- [31] Dmitri Korotkevitch. 2016. *Pro SQL Server Internals*. Apress.
- [32] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504. <https://doi.org/10.1145/3183713.3196909>
- [33] 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. <https://doi.org/10.1145/3589284>
- [34] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-Disk Updatable Learned Index. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4856–4869. <https://doi.org/10.1109/ICDE60146.2024.00369>
- [35] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670. <https://doi.org/10.1145/319628.319663>
- [36] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (2024), 4536–4545. <https://doi.org/10.14778/3685800.3685915>
- [37] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42 (2019), 73–84. <https://api.semanticscholar.org/CorpusID:83459602>
- [38] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*. ACM, 3:1–3:8. <https://doi.org/10.1145/2933349.2933352>
- [39] 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, USA, 302–313. <https://doi.org/10.1109/ICDE.2013.6544834>
- [40] Meng Li, Huayi Chai, Siqiang Luo, Haipeng Dai, Rong Gu, Jiaqi Zheng, and Guihai Chen. 2025. VEGA: An Active-tuning Learned Index with Group-Wise Learning Granularity. *Proc. ACM Manag. Data* 3, 1 (2025), 86:1–86:26. <https://doi.org/10.1145/3709652>

//doi.org/10.1145/3709736

- [41] Junfeng Liu, Jiarui Ye, Mengshi Chen, Meng Li, and Siqiang Luo. 2026. Evaluating Learned Indexes in LSM-tree Systems: Benchmarks, Insights and Design Choices. In *Proceedings 29th International Conference on Extending Database Technology, EDBT 2026, Tampere, Finland, March 24-27, 2026*, Wolfgang Lehner, Vanessa Braganholo, Kostas Stefanidis, Zheyang Zhang, Alexander Krause, and João Felipe Nicolaci Pimentel (Eds.). OpenProceedings.org, 183–195. <https://doi.org/10.48786/EDBT.2026.16>
- [42] Mingyu Liu, Junbin Kang, Kai Wang, Lu Zhang, Haibo Chen, Xiuchang Li, and Tianhong Ding. 2025. ScaleCache: Scalable and Production-grade Buffer Management for Disk-based Database Systems. *Proc. VLDB Endow.* 18, 12 (2025), 5073–5085. <https://doi.org/10.14778/3750601.3750628>
- [43] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2022. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning. *IEEE Trans. Parallel Distributed Syst.* 33, 8 (2022), 1953–1966. <https://doi.org/10.1109/TPDS.2021.3118599>
- [44] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maolinayazi. 2022. FILM: a Fully Learned Index for Larger-than-Memory Databases. *Proc. VLDB Endow.* 16, 3 (2022), 561–573. <https://doi.org/10.14778/3570690.3570704>
- [45] Ziming Mao, Jonathan D. Ellithorpe, Atul Adya, Rishabh Iyer, Matei Zaharia, Scott Shenker, and Ion Stoica. 2025. Rethinking the Cost of Distributed Caches for Datacenter Services. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks, HotNets 2025, UMD Campus, College Park, MD, USA, November 17-18, 2025*, Alan Zaaxing Liu, Philip Brighten Godfrey, and Barath Raghavan (Eds.). ACM, 317–325. <https://doi.org/10.1145/3772356.3772388>
- [46] 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. <https://doi.org/10.14778/3421424.3421425>
- [47] MariaDB. 2025. MariaDB Enterprise Open Source Database. Retrieved August 22, 2026 from <https://mariadb.com/>
- [48] John C. McCallum. 2024. Memory Prices 1957 - 2024. Retrieved August 22, 2026 from <https://web.archive.org/web/20250423035422/https://jcm.it.net/memoryprice.htm>
- [49] Microsoft. 2025. Overview of Azure SQL Managed Instance Resource Limits. Retrieved August 22, 2026 from <https://learn.microsoft.com/en-us/azure/azure-sql/managed-instance/resource-limits/>
- [50] Yuxuan Mo and Yu Hua. 2025. LOFT: A Lock-free and Adaptive Learned Index with High Scalability for Dynamic Workloads. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, ACM, 475–491. <https://doi.org/10.1145/3689031.3717458>
- [51] C. Mohan. 1999. Repeating History Beyond ARIES. In *VLDB '99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 1–17. <http://www.vldb.org/conf/1999/P1.pdf>
- [52] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. <https://doi.org/10.1145/128765.128770>
- [53] MongoDB. 2025. WiredTiger Storage Engine. Retrieved August 22, 2026 from <https://www.mongodb.com/docs/manual/core/wiredtiger/>
- [54] Marcus Müller, Lawrence Benson, and Viktor Leis. 2025. B-Trees Are Back: Engineering Fast and Pageable Node Layouts. *Proc. ACM Manag. Data* 3, 1 (2025), 14:1–14:26. <https://doi.org/10.1145/3709664>
- [55] Thomas Neumann. 2017. The Case for B-Tree Index Structures. Retrieved August 22, 2026 from <https://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html>
- [56] 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). <https://doi.org/10.1007/S11432-024-4125-9>
- [57] Oracle. 2025. Indexes and Index-Organized Tables. Retrieved August 22, 2026 from <https://docs.oracle.com/en/database/oracle/oracle-database/23/cnclpt/indexes-and-index-organized-tables.html>
- [58] Shujian Qian and Ashvin Goel. 2024. Massively Parallel Multi-Versioned Transaction Processing. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 765–781. <https://www.usenix.org/conference/osdi24/presentation/qian>
- [59] Aneesh Raman, Andy Huynh, Jinqi Lu, and Manos Athanassoulis. 2024. Benchmarking Learned and LSM Indexes for Data Sortedness. In *Proceedings of the Tenth International Workshop on Testing Database Systems, DBTest 2024, Santiago, Chile, 9 June 2024*, ACM, 16–22. <https://doi.org/10.1145/3662165.3662764>
- [60] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2025. QuIT your B+-tree for the Quick Insertion Tree. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, 451–463. <https://doi.org/10.48786/EDBT.2025.36>
- [61] Jiachen Shi, Jingyi Yang, Gao Cong, and Xiaoli Li. 2025. NEXT: A New Secondary Index Framework for LSM-based Data Storage. *Proc. ACM Manag. Data* 3, 3 (2025), 193:1–193:25. <https://doi.org/10.1145/3725330>
- [62] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004. <https://doi.org/10.14778/3594512.3594528>
- [63] MySQL Team. 2025. Adaptive Hash Index. Retrieved August 22, 2026 from <https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html>
- [64] MySQL Team. 2025. Introduction to InnoDB. Retrieved August 22, 2026 from <https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html>
- [65] MySQL Team. 2025. page0types.h File Reference. Retrieved August 22, 2026 from <https://github.com/mysql/mysql-server/blob/9fc8d6003b4ee32767c1e30942f22c3508c31/storage/innobase/include/page0types.h>
- [66] MySQL Team. 2025. Using AUTO_INCREMENT. Retrieved August 22, 2026 from <https://dev.mysql.com/doc/refman/8.0/en/example-auto-increment.html>
- [67] Kefei Wang and Feng Chen. 2023. Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems. *Proc. VLDB Endow.* 16, 13 (2023), 4339–4352. <https://doi.org/10.14778/3625054.3625068>
- [68] Wenlong Wang and David Hung-Chang Du. 2024. LearnedKV: Integrating LSM and Learned Index for Superior Performance on SSD. *CoRR abs/2406.18892* (2024). <https://doi.org/10.48550/ARXIV.2406.18892>
- [69] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, August 24-25, 2020*, Taesoo Kim and Patrick P. C. Lee (Eds.). ACM, 17–24. <https://doi.org/10.1145/3409963.3410496>
- [70] Yi Wang, Jianan Yuan, Shangyu Wu, Huan Liu, Jiaxian Chen, Chenlin Ma, and Jianbin Qin. 2024. LeaderKV: Improving Read Performance of KV Stores via Learned Index and Decoupled KV Table. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, IEEE, 29–41. <https://doi.org/10.1109/ICDE60146.2024.00010>
- [71] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzhong Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.
- [72] 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. <https://doi.org/10.14778/3457390.3457393>
- [73] 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. <https://doi.org/10.14778/3547305.3547322>
- [74] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792. <https://doi.org/10.14778/3067421.3067427>
- [75] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *VLDB J.* 23, 6 (2014), 915–937. <https://doi.org/10.1007/S00778-014-0355-0>
- [76] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. *Proc. VLDB Endow.* 16, 11 (2023), 2976–2989. <https://doi.org/10.14778/3611479.3611502>
- [77] Xianghong Xu, Tieying Zhang, Xiao He, Haoyang Li, Rong Kang, Shuai Wang, Linhui Xu, Zhimin Liang, Shangyu Luo, Lei Zhang, and Jianjun Chen. 2025. AdaNDV: Adaptive Number of Distinct Value Estimation via Learning to Select and Fuse Estimators. *CoRR abs/2502.16190* (2025). <https://doi.org/10.48550/ARXIV.2502.16190>
- [78] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proc. VLDB Endow.* 17, 11 (2024), 3415–3427. <https://doi.org/10.14778/3681954.3682010>
- [79] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1567–1581. <https://doi.org/10.1145/2882903.2915222>
- [80] Jiaoyi Zhang and Yihan Gao. 2022. CARM: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691. <https://doi.org/10.14778/3551793.3551823>
- [81] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3 (2024), 151. <https://doi.org/10.1145/3654954>
- [82] Rui Zhang and Shenghui Zhao. 2025. SafeSQL-LLM: A Synthetic Data Approach to Privacy-Preserving Text-to-SQL. In *31th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2025, Hefei, China, December 14-18, 2025*, IEEE, 1–8. <https://doi.org/10.1109/ICPADS67057.2025.11323205>

- [83] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proc. ACM Manag. Data* 2, 3 (2024), 145. <https://doi.org/10.1145/3654948>
- [84] 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 *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. <https://vldb.org/cidrdb/papers/2025/p29-zhao.pdf>
- [85] Bolong Zheng, Yongyong Gao, Jingyi Wan, Lingsen Yan, Long Hu, Bo Liu, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. 2023. DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems. *Proc. VLDB Endow.* 17, 1 (2023), 1–14. <https://doi.org/10.14778/3617838.3617839>
- [86] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *Proc. VLDB Endow.* 17, 5 (2024), 980–993. <https://doi.org/10.14778/3641204.3641209>