Understanding Robustness Issues of Updatable Learned Indexes: [Experiments & Analysis]

YUANHUI LUO, Renmin University of China, China
MINHUI XIE*, Renmin University of China, China
YIHENG TONG, Renmin University of China, China
SHICHAO JIANG, Renmin University of China, China
YUNPENG CHAI*, Renmin University of China, China

Learned indexes are viewed as promising substitutes for traditional indexes due to their excellent performance, especially in read-only workloads. Previous studies have shown that updatable learned indexes perform exceptionally well in many cases, suggesting they are nearly ready for real-world applications. However, unlike traditional indexes such as B+tree and ART, updatable learned indexes are prone to instability of real-time trained models, resulting in inherently uncertain structures. This raises skepticism about their robustness, hindering their broader adoption.

In this paper, we conduct a systematic benchmark and analysis to address this concern, corroborating doubts about the lack of robustness in state-of-the-art updatable learned indexes. We demonstrate that, contrary to previous findings, updatable learned indexes cannot robustly surpass traditional indexes, even losing their expected advantage under read-intensive workloads. We further reveal the root causes, including overfitted models, unbalanced structures, ineffective adjustments, and excessive space reservation. In addition, we explore potential mitigation methods to address these challenges. We hope our findings will highlight the critical importance of robustness in the design of updatable learned indexes, ultimately paving the way for their real-world adoption.

CCS Concepts: • **Information systems → Data access methods**; **Unidimensional range search**.

Additional Key Words and Phrases: Benchmarking, Robustness, Learned Index, Updatable Learned Index

ACM Reference Format:
Yuanhui Luo, Minhui Xie, Yiheng Tong, Shichao Jiang, and Yunpeng Chai. 2025. Understanding Robustness Issues of Updatable Learned Indexes: [Experiments & Analysis]. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 270 (September 2025), 25 pages. https://doi.org/10.1145/3749188

1 Introduction

Indexes play a vital role in database systems for accelerating query performance. Recently, Kraska et al. introduced learned indexes [23], a groundbreaking design regarded as a promising replacement for traditional indexes (e.g., B+tree [3, 8] and ART [27, 28]). Learned indexes leverage machine learning (ML) models to fit the mapping of indexed keys to storage locations, thereby accelerating

*Both Minhui Xie and Yunpeng Chai are corresponding authors.

Authors' Contact Information: Yuanhui Luo, Renmin University of China, Beijing, China, losk@ruc.edu.cn; Minhui Xie, Renmin University of China, Beijing, China, xieminhui@ruc.edu.cn; Yiheng Tong, Renmin University of China, Beijing, China, yihengtong@ruc.edu.cn; Shichao Jiang, Renmin University of China, Beijing, China, shichao.jiang@ruc.edu.cn; Yunpeng Chai, Renmin University of China, Beijing, China, ypchai@ruc.edu.cn.

ACM 2836-6573/2025/9-ART270
https://doi.org/10.1145/3749188

search performance through model predictions. They consistently outperform traditional indexes in static workloads (read-only), as demonstrated by both empirical evidence [35] and theoretical analysis [12, 13, 55].

Under dynamic workloads, previous works demonstrate that updatable learned indexes (ULIs) perform remarkably well in many cases, particularly in read-intensive workloads [15, 44, 48]. Therefore, their unprecedented read performance tends to overshadow other potential robustness issues, indicating that they are almost ready for real-world workloads [48]. However, none of the state-of-the-art ULIs theoretically offer better time or space complexity than B+tree [11, 16, 30, 44, 48, 49, 60].

Essentially, the data structures of ULIs are fundamentally nondeterministic, compared with traditional indexes. First, they train overfitted models to accelerate searches in large nodes (e.g., 16MB), making them susceptible to shifts in data distribution. Second, their tree structures are unbalanced and inherently uncertain, using deeper subtrees to handle difficult-to-fit data segments and adapt to skewed inserted data. In contrast, traditional indexes maintain balanced or unique structures for the same dataset, regardless of the insert order [5, 6, 8, 17, 28]. Third, the scope and type of their structural modifications (SMO) are flexible and guided by cost-model-based heuristics when nodes become full or model precision deteriorates, whereas traditional indexes employ predefined, rule-based SMOs. These design differences raise concerns about the robustness of updatable learned indexes' performance, hindering their adoption in real-world scenarios.

In this paper, we conduct a systematic study to investigate concerns about the robustness issues of updatable learned indexes. We perform a comprehensive grid test under various dynamic workloads to examine their performance fluctuations. We evaluate the performance of all open-sourced SOTA ULIs—namely, ALEX [11], LIPP [49], ALEX-OLC [48], DILI [30], DyTIS [52], SALI [16], and FINEdex [29]—comparing them against ART and B+tree in both single-threaded and multi-threaded settings. We provide a comprehensive benchmark of key metrics, including read/write throughput, tail latency, and space overhead.

Based on the benchmark, we unexpectedly and unfortunately find that **none of the SOTA updatable learned indexes can robustly outperform traditional indexes, even losing their expected advantage in read-intensive workloads**. *Our findings challenge the conclusions of previous works*, indicating that the advantages of SOTA ULIs are more limited and their performance improvements are definitely unreliable. Furthermore, we thoroughly analyze the root causes of the lack of robustness in SOTA ULIs, including overfitted models, unbalanced structures, ineffective adjustments, and excessive space reservation. We also explore potential mitigation methods for the practical deployment of updatable learned indexes.

We summarize our key findings as follows.

- **Significant performance fluctuations.** All the performance metrics of SOTA ULIs vary dramatically across workloads, whereas traditional indexes exhibit minimal fluctuations. For instance, the single-threaded lookup throughput of ALEX fluctuates by 84.4%, LIPP by 120.4%, ART by 5.5%, and B+tree by 9.7%.
- **Inferior to traditional indexes in many cases.** SOTA ULIs underperform traditional indexes in many cases across all metrics. For example, ALEX's multi-threaded lookup throughput and SALI can both be inferior to ART in up to 58.9% and 93.6% of cases.
- **Catastrophic worst-case performance.** The worst-case performance of SOTA ULIs can be unacceptably poor across metrics. For instance, the space overhead of ALEX and LIPP can both be up to 10x larger than the indexed data.

To the best of our knowledge, this is the first systematic study of the robustness of SOTA ULIs, which not only evaluates their performance but also analyzes the root causes of their limitations,

providing insights into potential mitigation for real-world deployment. In addition, we offer both experimental findings and analysis for **index designers**, as well as valuable lessons for **index users**. We limit our scope to one-dimensional in-memory updatable range indexes on numeric data types and expect future research to cover other indexes. Our benchmark **RoBin** (**Ro**bustness **B**enchmark for **in**dexes) is open-sourced at github.com/cds-ruc/RoBin.

2 Preliminaries

2.1 Updatable Learned Indexes

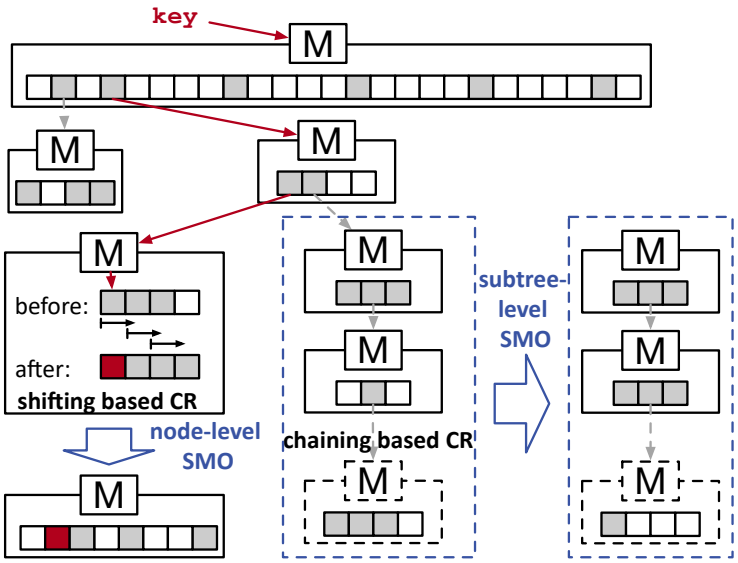


Fig. 1. The key designs of updatable learned indexes. CR refers to Conflict Resolution. SMO refers to Structural Modifications.

Previous updatable learned indexes have proposed two strategies to handle dynamic workloads. One is the delta-buffer strategy, which uses buffers at different levels—such as tree-level [13], node-level [14, 45], and record-level [29]—to accumulate insertions and periodically merges them into the main structure with model retraining. While this strategy can amortize the cost of merging, the additional search and concurrency overhead introduced by the buffers results in suboptimal performance [16, 44, 48]. Previous evaluations have shown that their performance falls short of state-of-the-art solutions [44, 48], so we omit them in our tests.

The mainstream strategy is the in-place strategy, which reserves additional space in tree nodes, determines insertion positions within nodes using model predictions, and triggers Structure Modification Operations (SMOs) to ensure sufficient space for insertions without compromising model accuracy. This approach leverages model-based insertion, as proposed by ALEX, to accelerate insertions while maintaining lookup performance through SMOs. Previous evaluations identified ALEX [11] and LIPP [49], both using this strategy, as SOTA updatable learned indexes [44, 48]. Subsequently, DILI [30], DyTIS [52], SALI [16], and Hyper [60] all follow the same in-place strategy, demonstrating superior performance.

Before benchmarking updatable indexes, we first provide a brief taxonomy of the key designs of current updatable indexes.

- **Structures**. Existing updatable learned indexes all adopt the tree structure to divide the key space into smaller sub-ranges, thereby easing ML model learning. ALEX, LIPP, DILI, SALI, and Hyper follow the unbalanced tree structure to overfit data distribution adaptively. In contrast, DyTIS utilizes a balanced tree structure with a fixed height of 3. The first two levels are radix tables, whose slots point to segments (leaf nodes). Each segment trains a piece-wise linear model for indexing. All of them employ a huge sparse node (e.g., 16MB) design to minimize tree height and ensure ample space for insertions.
- **Conflict Resolutions (CR)**. When a model-based insert targets a non-empty slot, conflict resolution is required. Existing methods can be classified into two categories: 1) shifting-based, and 2) chaining-based. Shifting-based methods (e.g., ALEX, DyTIS) shifts elements within leaf nodes to make room for the new insert, while chaining-based methods (e.g., LIPP, DILI, SALI, Hyper) construct new nodes as overflow areas to accommodate conflicting data and leave a pointer to these areas in place (chaining).
- **Structural Modification (SMO) Heuristics**. When conflict resolution strategies fail, or ML models become intolerably inaccurate, learned indexes trigger SMOs. Existing heuristics can be classified by 1) how to do SMOs and 2) when SMOs are triggered. For 1), ALEX and DyTIS adopt node-level adjustment by expanding/splitting/rebuilding nodes (and adjusting their parent nodes if necessary), while LIPP, DILI, SALI and Hyper leverage subtree-level rebuilding. For 2), common SMO triggers contain statistic heuristics (e.g., the number of insert conflicts [49], load factors [52]) or complex cost models (estimating lookup and insert costs) [11, 60].
- **Construction Methods**. When bulkloading data to construct the index, learned indexes can insert data one by one (DyTIS) or train well-fitting ML models on the entire dataset (DILI, Hyper, ALEX, LIPP, SALI). The latter can use either lightweight (ALEX, LIPP, SALI) or sophisticated (DILI, Hyper) training algorithms to build an unbalanced structure that closely fits the initial data.
- **Concurrency Control**. Previous work [48] introduced concurrent versions of LIPP and ALEX, named LIPP-OLC and ALEX-OLC, using optimistic latch coupling (OLC) [28]. SALI, based on LIPP and optimized for concurrency, also adopts OLC. DyTIS and Hyper use traditional read-write latches; Hyper further leverages the Read-Copy-Update to optimize SMOs [37]. For concurrency control granularity, LIPP-OLC, SALI, and Hyper employ per-slot latches, while ALEX-OLC and DyTIS use per-node latches.

In summary, SOTA updatable learned indexes tend to overfit the data distribution using unbalanced structures and overfitted models, relying on heuristics or cost-model-based SMOs to rescue performance degradation. However, as their design implies, they depend on implicit assumptions, such as a fixed key space and a consistent distribution between existing data and subsequent insertions [9, 11, 52]. These design choices may sacrifice robustness for potential performance gains and may not pay off outside their comfort zone (i.e., simple datasets with simple workloads).

2.2 Benchmarking Updatable Learned Indexes

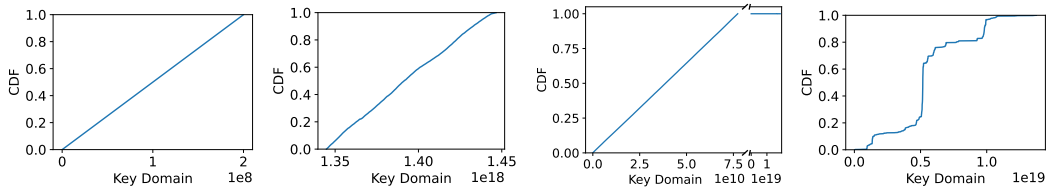
Previous studies have extensively evaluated various aspects of updatable learned indexes, primarily focusing on their strengths and performance factors [1, 15, 44, 48]. However, they typically rely on a simple, common workload to evaluate learned indexes: uniformly sampling half of the dataset to initialize the indexes and inserting the remaining data after shuffling. This workload aligns with the assumptions of learned indexes but may lead to biased results, as real-world workloads are more complex. Another recent study evaluates the impact of data sortedness on updatable learned indexes [38], but it generates data using BoDS [39], overlooking the implications of dataset distribution. In fact, we generalize the above workloads by introducing a more comprehensive benchmark. **Our findings overturn earlier, limited conclusions**, such as the claim [38, 48] that

ALEX robustly excels in throughput, tail latency, and space efficiency, revealing new complexities and challenges.

3 Benchmark Setup

We perform a comprehensive grid test consisting of 124 cases under various dynamic workloads and datasets to examine SOTA ULIs' performance fluctuations in both single- and multi-threaded settings. Our single-threaded tests are designed for deterministic comparisons and to avoid multi-threaded interference, especially since some SOTA ULIs only have single-threaded implementations. We implement RoBin, our benchmark, based on the GRE framework [48] and extend it to generate diverse workloads for robustness evaluation.

3.1 Datasets and Workloads



(a) Linear (b) Covid (c) Face (d) OSM

Fig. 2. The CDF of all four datasets

3.1.1 Datasets. We conduct tests on a synthetic dataset and three real-world datasets with increasing fitting difficulty [48], as shown in Fig. 2:

- *Linear*: a synthetic dataset consisting of consecutive integers, which can be modeled with a single linear function.
- *Covid*: real-world Tweet IDs tagged with COVID-19 [48].
- *Face*: sampled Facebook user IDs containing a small number (≈ 100) of outliers [20, 35].
- *OSM*: sampled Google S2 Cell-IDs of OpenStreetMap locations [20].

Each dataset consists of 200 million 64-bit unsigned integers. We generate the linear dataset not only to present an optimal scenario for learned indexes but also to simulate real-world use cases, such as auto-increment fields and time-series data [32, 59].

3.1.2 Workloads. For each dataset, we generate test cases by changing three configuration variables, including sampling methods, bulkload size, and insert patterns. Specifically, we construct indexes by sampling varying *bulkload sizes* of data with different *sampling methods* and inserting the remaining dataset with distinct *insert patterns*. After completing all insertions, we re-read the entire dataset via lookup operations and report the insert and lookup performance. For a given dataset, all test cases share the same key sequence of reads for fair comparison. For clarity, a test case is represented by a ⟨*sampling method, bulkload size, insert pattern*⟩ triplet.

We consider two sampling methods: 1) *Uniform sampling*: select the first few keys after shuffling the dataset. 2) *Segmented sampling*: select the first few keys after sorting the dataset. Both methods are without replacement, ensuring each sampled key is unique.

We evaluate two insert patterns: *Sorted Insert* and *Shuffled Insert*. Both patterns are employed for the remaining dataset after the bulkload is completed; for bulkloading, we consistently use the sorted insert pattern for those indexes without a construction algorithm.

The *Segmented Sampling* and *Sorted Insert* are common use cases in real-world applications, such as indexing auto-increment fields or timestamp data [39, 40].

The design goal of RoBin is not to cover all possible workloads, but to focus on representative extremes. Covering all types of workloads in testing is time-consuming and unrealistic. Therefore, we select two extreme cases from the perspective of learned indexes: one where there is no prior knowledge of future insertions (*Segmented Sampling*), and another where future distribution knowledge is maximized (*Uniform Sampling*), each combined with either skewed (*Sorted Insert*) or balanced (*Shuffled Insert*) insert patterns. We argue that the performance under other workloads should lie between these two extremes, as supported by our mitigation experiments in §6.1.

3.1.3 Measurement. For all test cases, we run each test three times and present the average results. We set a time limit of 30 minutes for each test case; if insertions are not completed within this time, we report this as a TIMEOUT error. In this case, the insert throughput is reported based on all completed operations. In addition, some test cases may cause existing learned indexes to encounter BUGs (e.g., segmentation fault), which we mark as a CRASH error.

Regarding the definition of robustness, we define it as the ability of learned indexes to consistently outperform certain traditional indexes in specific metrics. Our focus is on the lower bound of performance rather than the variance across different workloads, as fluctuations in performance are inherent to some ULIs and not necessarily undesirable. For example, when the inserted data follows the same distribution as the initial training data (*Uniform Sampling* with *Shuffled Insert*), gapped array designs [11, 49] in ULIs, which are optimized for such cases, can yield better performance.

3.2 Tested Indexes

In single-threaded settings, we tested ALEX [11], LIPP [49], DILI [30], and DyTIS [52] for learned indexes. We use their original open-source implementations provided by their authors. Hyper [60] is omitted due to the lack of an open-sourced implementation. For traditional indexes, we consider B+tree [3, 8] and ART [27] as competitors, using STX B+tree [4] and Florian's ART [43] implementations. Both are widely used in previous evaluations [44, 48].

In multi-threaded settings, we evaluated ALEX-OLC (provided by [48]), along with the authors' implementations of SALI [16] and FINEdex [29]. We also tested the multi-threaded version of ART (ART-OLC [43]) and B+tree (B+tree-OLC [47]).

3.3 Experimental Environment

All experiments were conducted on a Linux machine equipped with an Intel® Xeon® Gold 5220 processor (2.20 GHz). The Linux kernel version was 5.4.0. All tests were restricted to a single NUMA node with 64 GB of memory, and hyper-threading was disabled to ensure consistent results. All code was compiled using GCC 11.4.0 with O3 optimization.

4 Experimental Results

Our experimental results are divided into two parts. The first part focuses on the robustness of overall metrics, including single- and multi-threaded read/write throughput (§4.1), tail latency (§4.2), space overhead (§4.3), and workload sensitivity (§4.4). The second part includes the robustness evaluation of existing ULIs' out-of-bound handling mechanism (§4.5).

Unless otherwise specified, the default experimental setting (called **Baseline** in the following paper) is ⟨*uniform sampling,* 100*M, shuffled insert*⟩, which is widely adopted and regarded as a ULI-friendly configuration in previous studies [16, 30, 44, 48, 60].

4.1 Overall Throughput

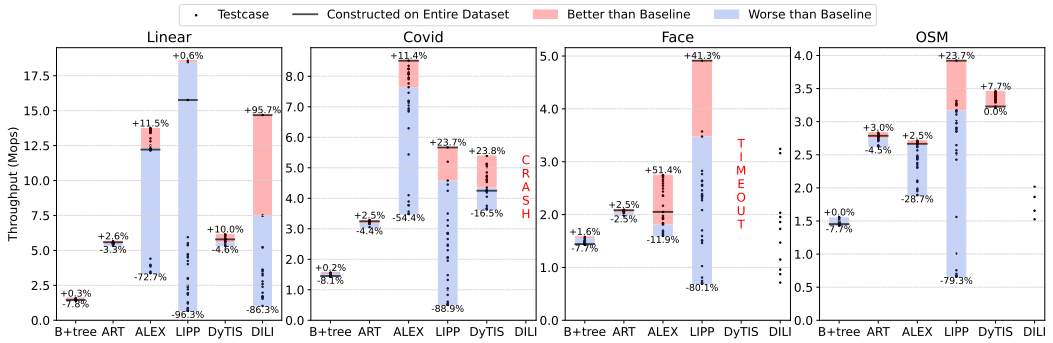


Fig. 3. Lookup throughput of all test cases (single-threaded). "Baseline" refers to the case constructing indexes with half of dataset ⟨*uniform sampling,* 100*M, shuffled insert*⟩, adopted by existing benchmark [48]. "Constructed on Entire Dataset" refers to ⟨*no sampling,* 200*M, no insert*⟩, which pre-knows the complete dataset. DILI and DyTIS failed to complete all test cases due to CRASH and TIMEOUT errors, with missing background areas indicating incomplete baseline cases.

4.1.1 Single-Threaded Throughput. We begin with the throughput results in single-threaded settings. Note that DILI and DyTIS cannot complete all tests. Specifically, DILI failed to complete 87 out of all 124 test cases, while DyTIS failed to complete 42 test cases. Figure 3 shows the lookup performance of all single-threaded indexes across four datasets. Every dot represents the performance of one test case, and the red/blue areas mark the test cases that perform better/worse than the baseline, respectively. Larger areas represent greater performance fluctuations. The maximum fluctuations of each dataset can be quantified by the relative throughput variations across best case and worst case, calculated as $(T_{best} - T_{worst})/T_{baseline}$, where T represents throughput. Figure 3 clearly shows that learned indexes have larger areas than traditional ones. Specifically, the maximum fluctuation of ART and B+tree is 9.7%. However, updatable learned indexes show significantly greater fluctuations. The maximum fluctuation of ALEX/LIPP/DILI/DyTIS is 84.4%/120.4%/186%/40.3%, respectively.

Finding 1. The performance fluctuations of all updatable learned indexes in single-threaded lookup throughput are significantly greater than those of traditional indexes.

In Figure 3, the stable performance of traditional indexes serves as a reference point. A significant number of data points representing learned indexes fall below this reference, indicating that performance fluctuations cause them to lose their advantage in read performance, resulting in inferior performance compared to traditional indexes. Across different datasets, updatable learned indexes perform worse than ART in 44.3% of cases and worse than B+tree in 10.9% of cases. Specifically, compared to ART, learned indexes show poorer performance in 53.8%, 23.1%, 33.7%, and 45.2% of cases on the Linear, Covid, Face, and OSM datasets, respectively. For the same dataset, the worst-case performance of ALEX can be up to 35.4% lower than ART, while LIPP can be up to 86.9% lower.

Finding 2. Learned indexes show inferior single-threaded lookup throughput to ART in over 44.3% cases.

Figure 3 uses black lines to indicate cases where learned indexes are constructed using the full dataset. Interestingly, despite having full knowledge of the data distribution, almost none of the existing learned indexes achieve their best performance. For instance, ALEX performs worse than its optimal case by 11.14%, 22.96%, and 2.89% on the Linear, Face, and OSM datasets, respectively. This

highlights that the heuristic-based construction algorithms of learned indexes are often suboptimal, as they often fail to achieve optimal performance even when built on the entire dataset.

Finding 3. The construction methods of updatable learned indexes are suboptimal

	B+tree	ART	ALEX	LIPP	DyTIS	DILI
Linear	1.32	4.60	2.73	0.469	0.00152	1.10
Covid	1.32	2.60	2.11	0.341	1.347	CRASH
Face	1.31	2.20	0.98	0.425	0.00201	0.832
OSM	1.32	2.42	1.22	0.407	0.142	1.12

Table 1. Insert throughput (Mops) of the slowest insert case (single-threaded). DILI crashed in all cases with the Covid dataset.

Next, we report the insert throughput of the slowest case during insertions in Table 1. Given that varying insert patterns cause significant performance fluctuations across all indexes, including traditional ones, we focus on their worst-case performance to examine their robustness. Table 1 indicates that, except ALEX, the insert performance of other learned indexes, including LIPP, DyTIS, and DILI, lags behind B+tree by up to 74.2%, 99.9%, and 36.5%, respectively, with DyTIS even exhibiting a gap close to three orders of magnitude. Under all test cases, ART and B+tree exhibit either superior or robust insert performance.

Finding 4. Except for ALEX, learned indexes may exhibit severe and even unacceptable insertion inefficiency in single-threaded worst-case scenarios.

In conclusion, single-threaded updatable learned indexes cannot robustly surpass traditional indexes (especially, ART) across different cases on all datasets. The emerging competitors (DILI and DyTIS) after previous evaluations [44, 48] did not outperform previous SOTAs (ALEX and LIPP). DILI shows performance fluctuations comparable to LIPP's, but its best performance falls short of LIPP's. DyTIS displays relatively robust lookup performance due to its balanced structure with a maximum tree height of three levels. However, maintaining this balanced structure requires frequent SMOs, resulting in terrible and impractical insert performance in some cases. For ALEX and LIPP, while LIPP achieves the highest lookup performance on the Linear, Face, and OSM datasets, its lookup performance advantage is fragile, and its worst-case insert performance can be ~2x to 7x worse than ALEX. In contrast, ALEX demonstrates relatively stable performance, generally outperforming ART on simpler datasets (Linear and Covid) but falling short on more complex datasets (Face and OSM). Additionally, ALEX's worst-case insert performance is consistently inferior to ART's.

4.1.2 Multi-Threaded Throughput. We move on to examine the throughput robustness of ULIs in multi-threaded scenarios. Notably, among all multi-threaded tests, only ART-OLC and B+tree-OLC successfully completed all 124 cases. SALI followed closely, failing in only 8 cases, while ALEX-OLC failed in 68 cases, including 8 timeouts due to deadlocks. FINEdex exhibited the highest failure rate, crashing in 105 cases. It can be reasonably inferred that the combination of more complex SMO logic and data-distribution-sensitive structures makes implementation and full coverage testing of correct concurrency control in learned indexes highly challenging.

Figure 4 shows the 16-threaded lookup throughput across all test cases. Both Figures 3 and 4 exhibit similar overall patterns, with a notable difference in the y-axis scale due to the significant throughput increase from concurrent reads. This increase occurs because concurrent inserts do not significantly affect the insertion sequence processed by the index.

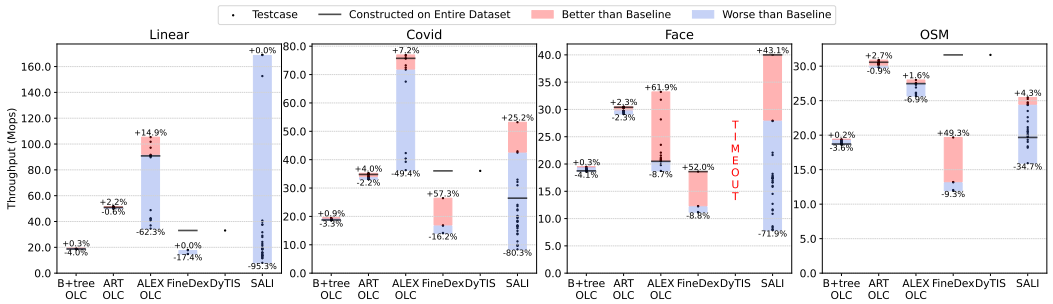


Fig. 4. Lookup throughput of all test cases (16-threads). "Baseline" refers to the case constructing indexes with half of dataset ⟨*uniform sampling*, 100*M*, *shuffled insert*⟩ and "Constructed on Entire Dataset" refers to ⟨*no sampling*, 200*M*, *no insert*⟩. DyTIS failed to complete all test cases due to TIMEOUT errors, resulting in missing test cases. The missing background areas indicate missing baseline cases.

	B+tree	B+tree OLC	ART	ART OLC	ALEX	ALEX OLC	LIPP	SALI	FINEdex
Linear	1.32	17.0	4.60	45.2	2.73	1.34	0.469	1.43	12.4
Covid	1.32	17.0	2.60	28.2	2.11	24.8	0.341	0.73	11.5
Face	1.31	17.0	2.20	25.2	0.98	1.30	0.425	0.45	9.10
OSM	1.32	17.0	2.42	28.2	1.22	9.15	0.407	3.99	9.55

Table 2. Insert throughput (Mops) of the slowest insert case (16-threaded).

Although the relative fluctuations across cases are similar, the absolute performance gaps increase significantly and are further amplified in multi-threaded scenarios. Consequently, the comparisons among indexes are similar to single-threaded scenarios, i.e., ALEX (and ALEX-OLC) consistently dominate the lookup performance of simple datasets (i.e., Linear and Face), while ART (and ART-OLC) outperforms other indexes in all other scenarios.

Finding 5. The multi-threaded lookup throughput closely mirrors the single-threaded patterns but amplifies the absolute performance gaps.

Table 2 presents the insert throughput in the worst-case scenario during multi-threaded insertions. B+tree-OLC and ART-OLC demonstrate good scalability even in these worst cases, all of which occur under *Shuffled Insert*, as *Sorted Insert* benefits from cache locality, and OLC effectively handles contention during skewed insertions of *Sorted Insert*. Because ALEXOLC and FINEdex crash in many cases, the remaining results do not offer reliable evidence of their worst-case scalability, even though they appear to scale in a few instances. For FINEdex, its performance is consistently lower than both B+tree-OLC and ART-OLC, which aligns with prior evaluations [44, 48]. However, the performance of ALEX-OLC and SALI is extremely poor and not scalable, even falling below the worst single-threaded insertion throughput reported in Table 1. Counterintuitively, the worst cases occur under *Sorted Insert*, which is typically considered cache-friendly. This is because *Sorted Insert* triggers frequent and costly SMOs, including large node creations and excessive re-insertions, which force concurrent insertions to be processed serially. Even the slot-level OLC in SALI, which incurs the finest granularity of contention, fails to mitigate the overhead of these SMOs.

We do not report the worst-case performance under mixed read-write workloads, as it falls between the read-only and write-only cases and does not alter the overall conclusions. Since OLC [28], used by all tested indexes except FINEdex, allows reads to proceed without blocking writes, the write-only workload already represents the highest level of contention.

Finding 6. In multi-threaded worst-case insert scenarios, learned indexes often exhibit poor scalability or crashes, sometimes performing worse than in single-threaded settings.

Lesson 1. Choose ALEX for generally superior single-threaded lookup performance on simple datasets; otherwise, opt for ART and ART-OLC.

4.2 Tail Latency

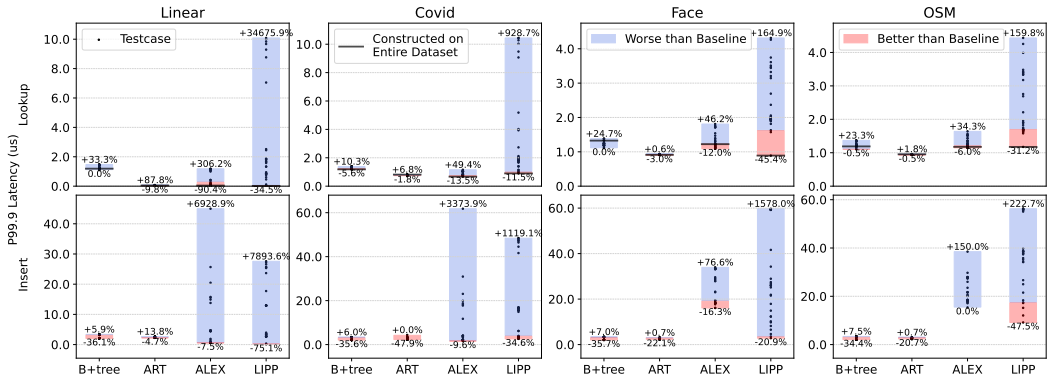


Fig. 5. Tail latency (99.9th) of all test cases (single-threaded). Top: lookup latency; Bottom: insert latency. "Baseline" refers to ⟨*uniform sampling*, 100*M*, *shuffled insert*⟩ and "Constructed on Entire Dataset" refers to ⟨*no sampling*, 200*M*, *no insert*⟩.

We report the tail latency for lookup and insertion in single-threaded (Figure 5) and multi-threaded settings (Figure 6). We sample the latency of all operations and report the 99.9th percentile. Sampling all operation latencies introduces additional recording time between operations, which can reduce contention in concurrent tests. This suggests that the actual tail latency in real-world multi-threaded scenarios could be higher due to increased contention.

In current and subsequent experimental analyses, we exclude DILI, DyTIS, and FINEdex due to their excessive number of failed test cases and poor throughput performance in §4.1 and §4.1.2.

Figure 5 shows that although the baseline cases of learned indexes exhibit low tail latency, as observed in previous tests [48], their overall tail latencies fluctuate significantly. The only exception is ALEX, which demonstrates relatively steady and low lookup tail latency. However, the worst-case lookup tail latency of LIPP and the worst-case insertion tail latency of both LIPP and ALEX are unacceptably high, reaching tens of microseconds. In contrast, the tail latency of traditional ones is remarkably and consistently low.

The general results in multi-threaded settings are similar, as shown in Figure 6. However, the insert tail latency of ALEX-OLC and SALI is amplified to hundreds of microseconds due to high concurrent contention, making them unrealistic for latency-critical applications. Notably, in the

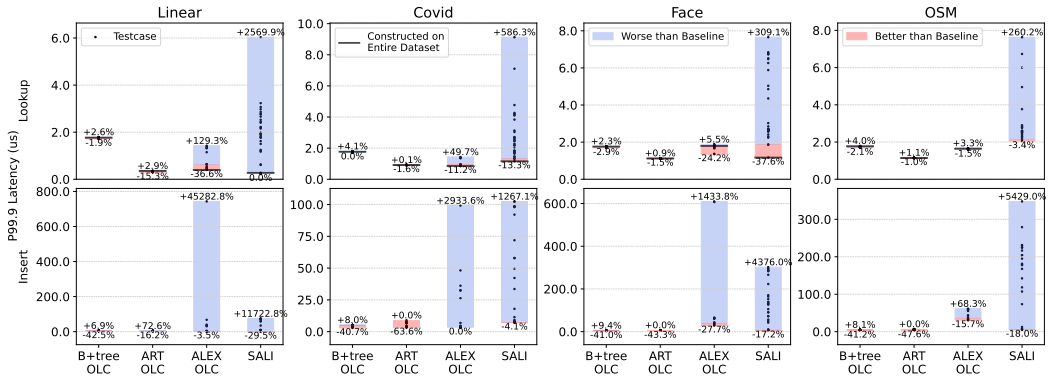


Fig. 6. Tail latency (99.9th) of all test cases (16-threaded). Top: lookup latency; Bottom: insert latency. "Baseline" refers to ⟨*uniform sampling*, 100*M*, *shuffled insert*⟩ and "Constructed on Entire Dataset" refers to ⟨*no sampling*, 200*M*, *no insert*⟩.

Linear and Face datasets, the worst-case insertion tail latency of SALI is lower than that of ALEX-OLC. This is because SALI uses record-level latching, whereas ALEX-OLC employs node-level latching, which incurs higher contention.

Finding 7. The tail latency of updatable learned indexes often fluctuates significantly and can reach unacceptably high levels in both single-threaded and multi-threaded settings.

For their application scenarios involving insertions, updatable learned indexes require systems to tolerate potentially high insertion latencies in exchange for their lookup performance advantages. As a result, they are unsuitable for latency-critical applications.

Lesson 2. Updatable learned indexes may lack the robustness required for latency-critical applications, an area where traditional indexes maintain their dominant advantage.

4.3 Space Overhead

In this experiment, we present the space overhead of indexes in both single-threaded and corresponding concurrent versions. The space overhead reflects only the memory occupied by the index structure itself, excluding unreclaimed space from the allocator or epoch-based reclamation in optimistic latch coupling (OLC) [28]. We measure the space overhead after inserting the entire dataset of 200M 16-byte key-value pairs, totaling 3200MB of data.

The space overhead of ART and B+tree is identical to that of their concurrent versions. Therefore, we omit ART-OLC and B+tree-OLC from Figure 7. In Figure 7, we first examine the single-threaded results, observing that ALEX's best-case performance consistently outperforms ART and B+tree in space efficiency, while LIPP generally lags behind both ART and B+tree. In fact, ALEX emerges as the most space-efficient index, robustly requiring less space than B+tree and ART across nearly all cases, except for one anomaly in the Face dataset. We analyze this outlier in detail in §5.4. Unlike ALEX, LIPP exhibits significant space fluctuations across workloads, with its space usage sometimes reaching 10 times the size of the original data. B+tree space usage also fluctuates by 46.6%, as Segmented-Sampling causes insertions to only affect a subset of the initial leaf nodes, leaving the remaining reserved space unused. Due to its unique data representation, ART's space

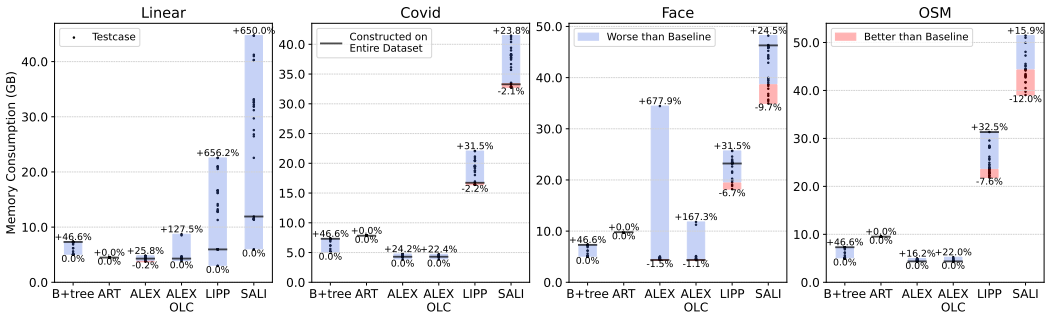


Fig. 7. Space overhead in all test cases (3200MB data). "Baseline" refers to ⟨*uniform sampling*, 100*M, shuffled insert*⟩ and "Constructed on Entire Dataset" refers to ⟨*no sampling*, 200*M, no insert*⟩. Space is measured for index structure itself only (unreclaimed space excluded).

consumption remains identical for the same dataset, regardless of the insertion order. However, it is slightly larger than B+tree by up to 96.4%.

Finding 8. The space overhead of learned indexes can vary significantly, whereas traditional indexes remain stable.

Lesson 3. ALEX is the most space-efficient index in most cases, but its lack of theoretical guarantees makes it unreliable in certain abnormal scenarios.

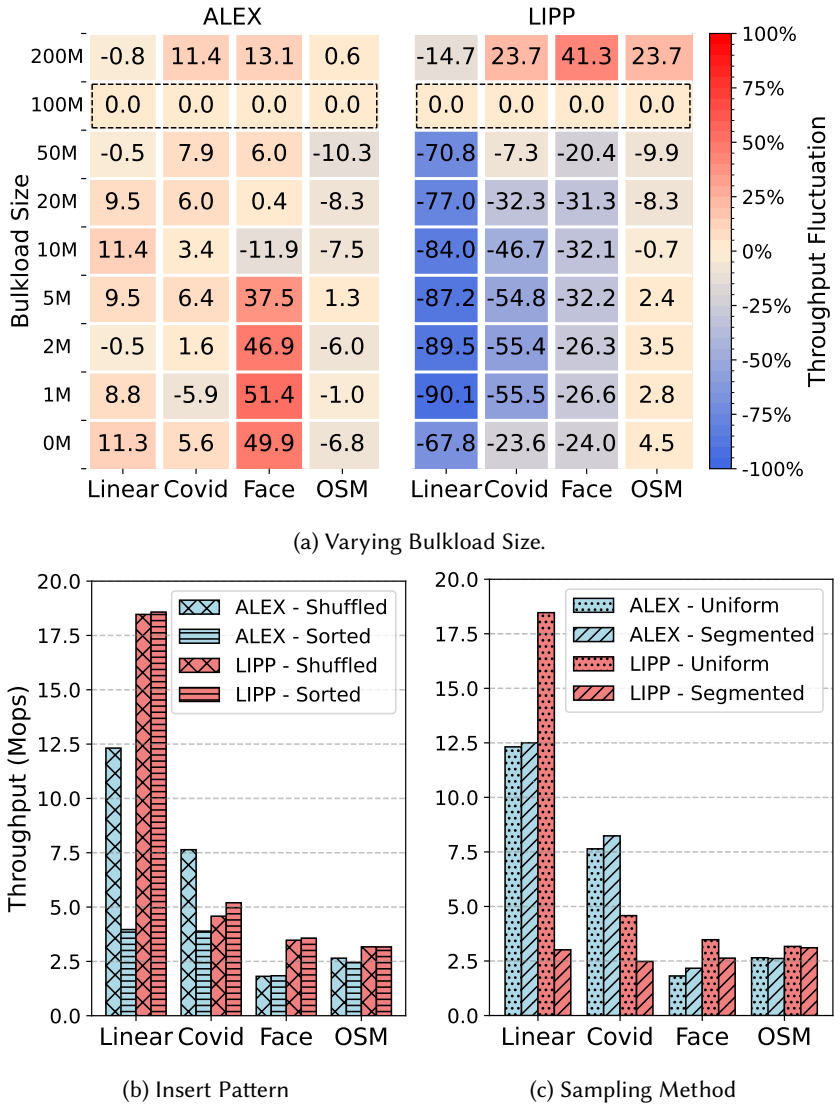
For concurrent learned indexes, ALEX-OLC exhibits performance close to that of ALEX, though it also shows significant space fluctuations in some anomalous cases on the Linear and Face datasets. In contrast, SALI consumes nearly twice the space of LIPP, even reaching up to 50GB in size. The additional space overhead caused by concurrent implementations varies across indexes. As detailed in later sections §5.4, this discrepancy is primarily due to SALI's use of record-level latching, which leads to space inflation, whereas B+tree-OLC, ART-OLC, and ALEX-OLC use node-level latching, resulting in lower space overhead for concurrency control.

Finding 9. The concurrent implementation of learned indexes can incur significant space overhead, as seen in SALI.

4.4 Sensitivity Tests

We move on to examine the sensitivity of updatable learned indexes by individually changing each variable in the baseline configuration ⟨*uniform sampling*, 100*M, shuffled insert*⟩. We focus solely on single-threaded scenarios to better control variables.

4.4.1 Impact of Bulkload Size. We first show the impact of test cases under ⟨*uniform sampling*, **0 ~ 200*M***, *shuffled insert*⟩. These cases mirror real-world scenarios, where the bulkload data and subsequent insertions are randomly sampled from the same data distribution. Figure 8a shows that learned indexes exhibit significant performance fluctuations by up to 89.5%. Notably, ALEX exhibits an unusual trend on the Face dataset, where performance decreases as the bulkload size increases. Compared to a bulkload size of 0M, ALEX's performance drops by 51.8% at 100M. Meanwhile, LIPP shows a trend of declining performance followed by improvement across all datasets, with 0M performance comparable to 50M on Linear, Face, OSM, and 20M on Covid.



(a) Varying Bulkload Size.

(b) Insert Pattern

(c) Sampling Method

Fig. 8. Lookup performance of ALEX and LIPP under varying test cases to examine their sensitivity. Each variable in the baseline configuration ⟨*uniform sampling*, 100*M*, *shuffled insert*⟩ was individually changed, specifically (a) bulkload size, (b) insert pattern, and (c) sampling method. Performance fluctuations are compared to the baseline (dashed box).

4.4.2 Impact of Insert Pattern. Next, we present the performance of test cases with ⟨*uniform sampling*, 100*M*, ***shuffled / sorted insert***⟩. In Figure 8b, ALEX exhibits significant fluctuations on the Linear and Covid datasets. Under shuffled insertions, the performance of ALEX increases by 3.1x on the Linear dataset and 2.0x on the Covid dataset compared to sorted insertions. LIPP, however, maintains stable performance with a maximum fluctuation of only 2.3%.

4.4.3 Impact of Sampling Method. We examine the lookup throughput with ⟨***uniform / segmented sampling,*** 100*M, shuffled insert*⟩. Figure 8c illustrates that LIPP's performance with segmented sampling decreased by 83.6%, 51.3%, and 23.6% compared to uniform sampling on the Linear, Covid, and Face datasets, respectively. In contrast, ALEX shows stable performance on all datasets with a maximum fluctuation of 21.0%.

To conclude, we summarize two key findings from the above experiments. The experiments reveal that learned indexes are highly sensitive to even minor changes in workload variables, especially when dealing with simple datasets. These fluctuations indicate that the performance advantage of learned indexes on simpler datasets is fragile, as small disruptions can significantly affect their efficiency. Conversely, fluctuations are minimal on more complex datasets (e.g., OSM), where learned indexes are less effective.

Finding 10. Learned indexes suffer significant performance fluctuations when facing minor changes to a single variable of workloads, especially on simple datasets.

Additionally, the performance trends of different learned indexes are highly variable and depend on specific workload characteristics. For instance, ALEX shows greater sensitivity to insert patterns, while LIPP is more affected by bulk load size and sampling method. The gapped array design of ALEX suffers from Sorted-Insert, causing the reserved gaps to be ineffective and increasing model error. And the overfitted models in LIPP make it sensitive to bulkload distribution, as discussed in a later Section §5.1. Furthermore, an unexpected phenomenon occurs where increasing the bulk load size can lead to decreased performance, making parameter tuning for learned indexes a complex task. This suggests that periodic rebuilding and refitting of datasets may not always guarantee improved performance, even with larger datasets.

Finding 11. The performance trends of learned indexes vary unpredictably based on workload parameters, with each index exhibiting distinct sensitivities to specific variables.

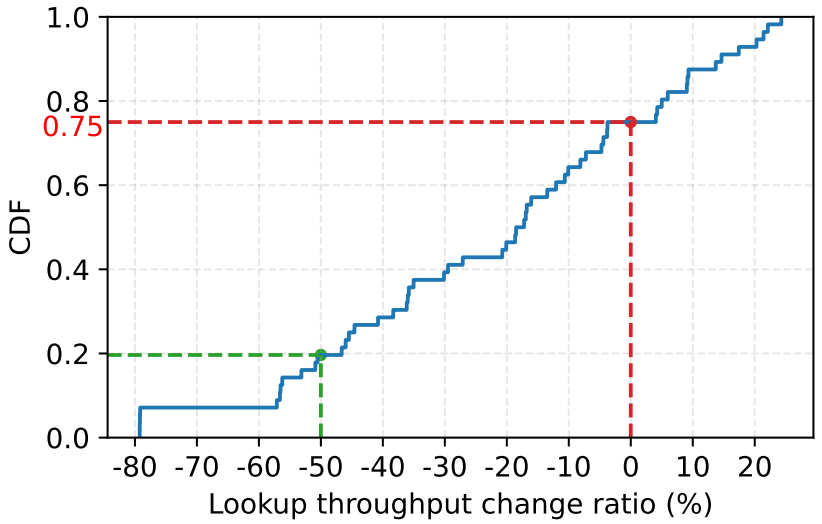


Fig. 9. **CDF of lookup throughput change ratio after bulkloading with an outlier.** For ALEX in workload configurations ⟨*segmented sampling,* 0~100*M, uniform/shuffled insert*⟩. The dashed lines indicate the proportion of cases with performance degradation and those with at least a 50% drop.

4.5 Out-of-bounds Handling

To evaluate learned indexes in scenarios that violate their key space assumption, we previously tested *segmented sampling*, where the learned index is provided with a limited key space (bounds) during bulkloading, and keys are inserted into the rightmost node. As discussed in Section §4.4.3, we found that LIPP's performance significantly degrades under *segmented sampling*. ALEX has proposed a general and effective method to expand the root node without model retraining for handling out-of-bounds insertions, successfully managing such cases. PLIN (for persistent memory) [61] and AULID (for disk) [25] also proposed similar methods.

However, as our experiments will reveal, these out-of-bounds handling methods are fragile and limited. They are only capable of detecting skewed insertions at two edges of the key space, making them highly vulnerable to disruption by out-of-bound outliers. To demonstrate this, we design an experiment based on ⟨*segmented sampling*, 0~100*M, uniform/shuffled insert*⟩, introducing an outlier (near the maximum value of an unsigned 64-bit integer) into the bulkloading data. This ensures that the remaining data is inserted into the rightmost node while still staying within the key space. As shown in Figure 9, the lookup throughput decreases dramatically after bulk loading with the outlier, with a reduction of up to 80%. In 75% of cases, the throughput drops, and more than 20% of cases experience a decrease of over 50%. This underscores the necessity for a more generalized approach to handling skewed insertions.

Finding 12. The out-of-bounds handling method in ALEX is fragile and can be easily disrupted by a single outlier.

5 Root Cause Analysis

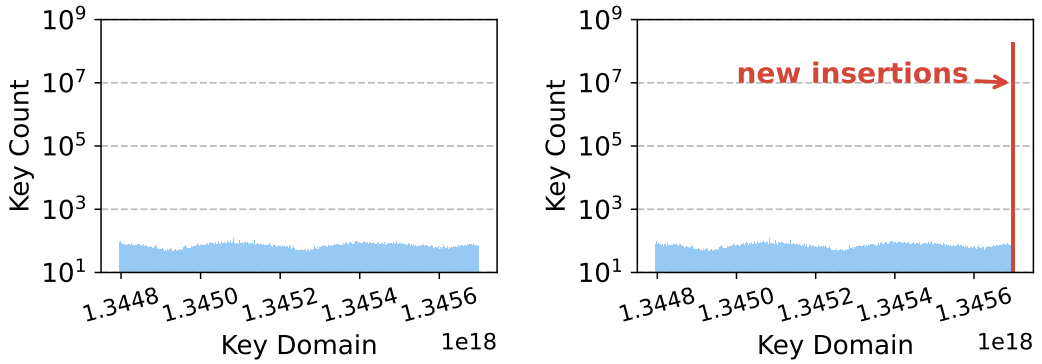
This section delves into the design principles of updatable learned indexes, identifying factors that contribute to performance fluctuations and degradation. Through case studies, we pinpoint four primary causes: overfitted models, unbalanced structures, ineffective SMO heuristics, and excessive space reservation. We filter test cases based on specific conditions to demonstrate the generality of each root cause. We report and analyze some instances, and similar observations are also found in other cases across different learned indexes.

5.1 Overfitted Models

Updatable learned indexes tend to closely fit the bulkload data to achieve better performance when the data distribution is static. However, this causes the models, particularly in the upper-level nodes, to overfit the bulkload data distribution, resulting in performance deterioration when the data distribution changes. We analyzed cases where the root model of ALEX and LIPP remained unchanged after bulk loading, finding that the root model is fixed initially in 73.4% of cases for ALEX and 80.6% for LIPP.

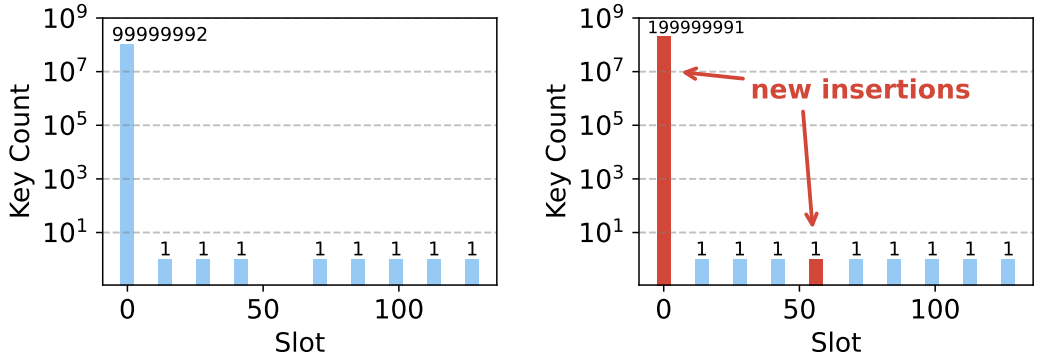
We present one case each for LIPP and ALEX to illustrate the impact of their overfitted models. Figure 10 illustrates how the model in root node maps the keys into the array slots (range partitioning) of the worst case for LIPP's lookup performance on the Covid dataset, i.e., ⟨*segmented sampling*, 2*M, sorted insert*⟩. Initially, the construction algorithm ensures an even data distribution across key domains of bulkload data. However, as new data is inserted, the initially overfitted model causes all 198M insertions to conflict at the rightmost node, resulting in a skewed, right-deep tree.

Figure 11 shows the array slot distribution of ALEX's root node during its worst lookup performance on the Face dataset, specifically under the case ⟨*uniform sampling*, 100*M, shuffled insert*⟩. The



(a) After Construction (b) After Insertions

Fig. 10. Visualization of the learned model in the root node of LIPP. Dataset: Covid. Test Case: Worst lookup throughput.



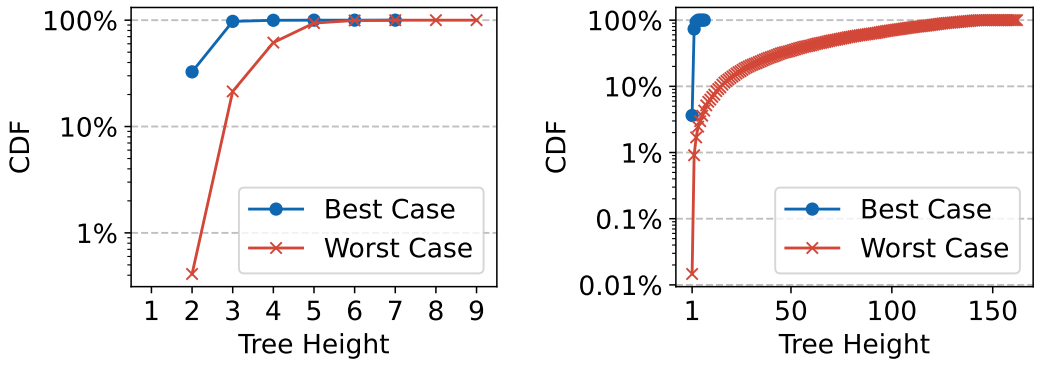
(a) After Construction (b) After Insertions

Fig. 11. Visualization of the learned model in the root node of ALEX. Dataset: Face. Test Case: Worst lookup throughput.

Face dataset contains 22 outliers near the maximum value of an unsigned 64-bit integer. In this case, bulkloading data with these outliers leads to an overfitted skewed root model, which significantly degrades ALEX's performance. This highlights the critical role of construction algorithms in the robustness of learned indexes.

5.2 Unbalanced Structures

Most updatable learned indexes adopt an unbalanced structure to limit the skewed data within a few sub-trees, allowing for finer granularity partitioning of the skewed range to improve model fitting precision. Figure 12 demonstrates that in the best cases, both ALEX and LIPP succeed in containing small amounts of data within a few skewed sub-trees, leaving the majority of data in lower levels. However, in the worst cases, the data is spread evenly across different levels, suggesting highly-skewed tree structures. The above results indicate that adopting a skewed tree height to

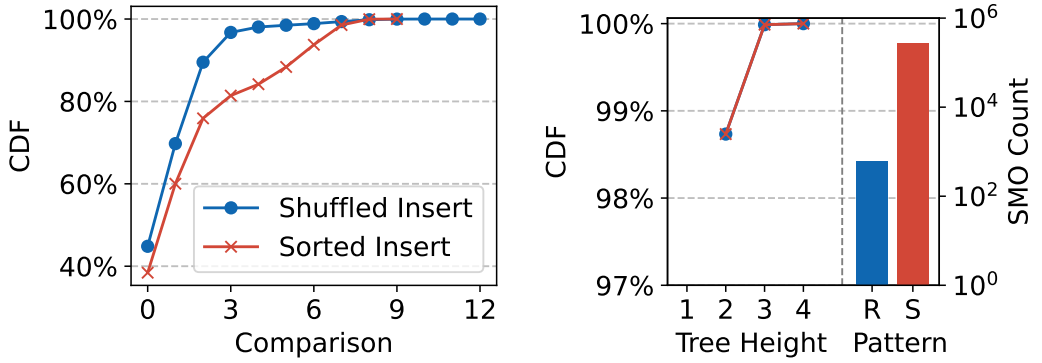


(a) ALEX (b) LIPP

Fig. 12. Tree height distribution of ALEX and LIPP. Dataset: OSM. Test Cases: Best/worst lookup throughput.

handle complex data segments is feasible, but robustly controlling the degree of skewness in the overall tree height remains a design challenge.

5.3 Ineffective SMO Heuristics



(a) Comparison of leaf nodes (b) Tree height and SMO count.

Fig. 13. ALEX statistics in ⟨*uniform sampling*, 100*M*, *sorted insert v.s. shuffled insert*⟩. Dataset: Covid. In subfigure (b), R: *Shuffled Insert*, S: *Sorted Insert*.

Updatable learned indexes gather statistics at runtime to detect performance decline caused by data distribution shifts. Based on various cost models, they decide when and how to trigger SMOs for adjusting the models and structures. However, to balance read and insert performance, the frequency and scope of statistics collection are restricted, and SMOs are only applied locally to sub-trees and nodes. By fixing the sampling method and bulk load size, we identify cases where higher SMO counts correlated with worse performance across different insert patterns, revealing that this occurred in 83.3% of cases for ALEX and 81.7% of cases for LIPP.

We provide two instances of ALEX by varying the insert pattern of the baseline as in §4.4.2. As shown in Figure 12 (a), the tree height distributions are the same, while "Sorted-Insert" is slower than "Shuffled-Insert" by 48.7%. Figure 13 (b) explains the reasons: their comparison counts differ greatly due to inaccurate models. Nevertheless, we observed that the number of SMOs in "Sorted-Insert" is 500x more than in "Shuffled-Insert", reaching 271953 operations, yet still failing to effectively mitigate the model error. As for LIPP, previous evidence from Figure 10 indicates that it fails to effectively adjust the model of the root node even after being constructed on 2M data. Because it employs a strategy of fixing nodes that exceed a certain threshold to avoid costly SMOs, such as huge subtree reconstructions. Unfortunately, this also limits its ability to adjust to data distribution changes.

5.4 Excessive Space Reservation

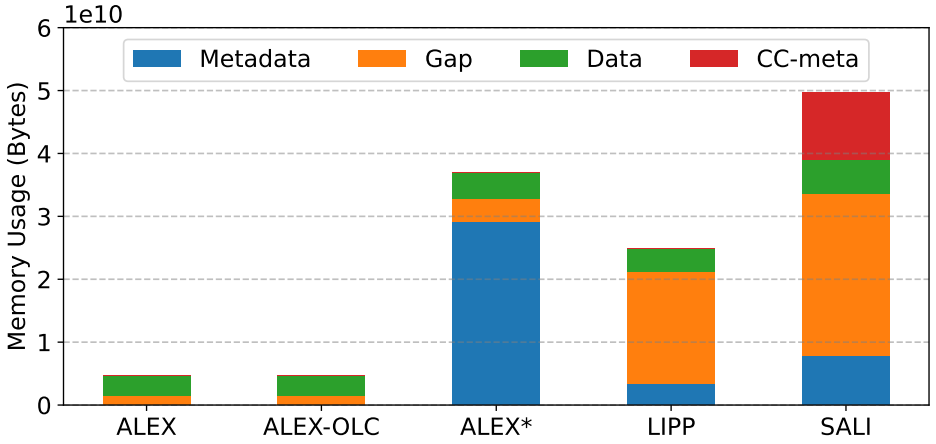
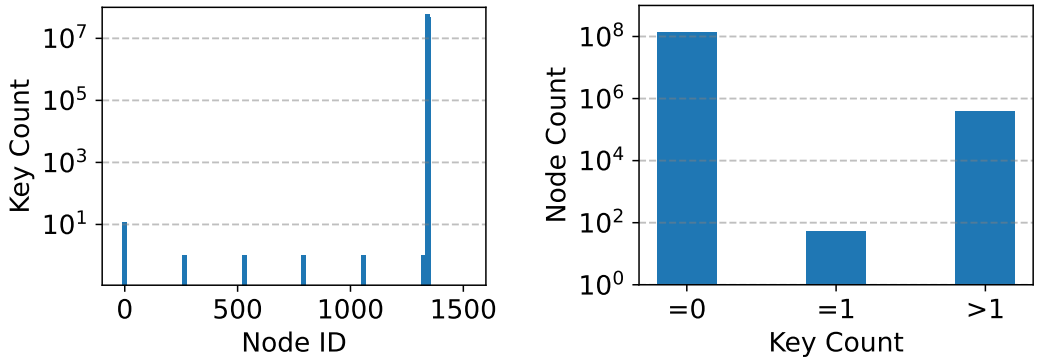


Fig. 14. **Breakdown of memory consumptions.** Dataset: Face. Test Case: ⟨*no sampling*, 200*M*, *no insert*⟩ for ALEX, ALEX-OLC, LIPP, SALI and the abnormal case for ALEX in Figure 7 (denoted as ALEX*). Gap: Reserved empty space. CC-meta: Concurrency Control-related metadata (e.g. latches).

We present the space breakdown of excessive space usage by updatable learned indexes in a test case where the entire Face dataset is bulkloaded, ensuring deterministic results by avoiding multi-threaded concurrent insertions. Additionally, we highlight the only abnormal case for ALEX on the Face dataset, where its space expands to 6x larger than normal cases, as shown in Figure 7.

Figure 14 points out that the excessive space usage of LIPP is primarily due to empty gaps, accounting for 71% of total space, highlighting its inefficient space reservation strategy. Additionally, SALI introduces a significant space overhead for concurrency control metadata on top of LIPP, reaching 22%.

Surprisingly, Figure 14 shows that the majority of the space usage is attributed to the nodes' metadata. To explain the abnormal value of ALEX's space overhead, we illustrate the statistics of its leaf nodes. Figure 15 shows that there are numerous (~100M) empty leaf nodes in this case, and all these empty nodes are concentrated on the smallest end of the entire key space. We infer that the reason behind this behavior is that ALEX attempts to split the node by equally dividing the key space, rather than splitting the node into equal-sized parts. However, when the key space is skewed,



(a) Key count per 100K nodes

(b) Node type distribution

Fig. 15. Visualization of the leaf nodes of ALEX. Dataset: Face. Test Case: the abnormal case for ALEX in Figure 6. Scanning all leaf nodes in ascending order, we merge key counts per 100K nodes and classify them into three types by key count.

this approach fails to evenly distribute the data into two nodes, which is necessary for the cost model to reduce costs and improve accuracy. This can lead to excessive, unnecessary cascading splits, as a single split cannot effectively divide the original node into two balanced parts. The outliers in the Face dataset may trigger this behavior.

6 Mitigation Methods

In this section, we explore two general and practical mitigation methods for lookup throughput that index users can adopt, while also providing design insights for index designers. However, as we reveal in Section §5, the lack of robustness in learned indexes is fundamentally rooted in their design. A black-box solution can only mitigate, not solve, the problem. Redesigning updatable learned indexes with robustness as a first-class principle is the true fix.

6.1 Data Augmentation

Data augmentation is a widely used technique in machine learning that helps mitigate model overfitting by artificially generating additional training data through various methods [41]. Inspired by data augmentation, we explore a mitigation method by adding artificial data to the initial bulkloading data. ULIs are trained on the mixed data, and the redundant artificial data is removed from ULIs before normal operations begin. We generate artificial data by uniformly sampling keys from the given key space until the artificial data size reaches a certain ratio of the original bulkloading data size.

We generate artificial data 1% the size of the bulkload data from two types of key spaces: one by uniformly sampling the entire key space of unsigned 64-bit integers, and the other by using the key space of the entire dataset. In some use cases, such as timestamps, the key space of the dataset can be provided as a hint by the index users. In addition, bulkloading with 1% more data size will have minimal impact on the bulkload time.

Figure 16 shows the CDF of the lookup throughput performance change ratio. We observe that data augmentation from both types of key spaces significantly boosts LIPP's performance, helping

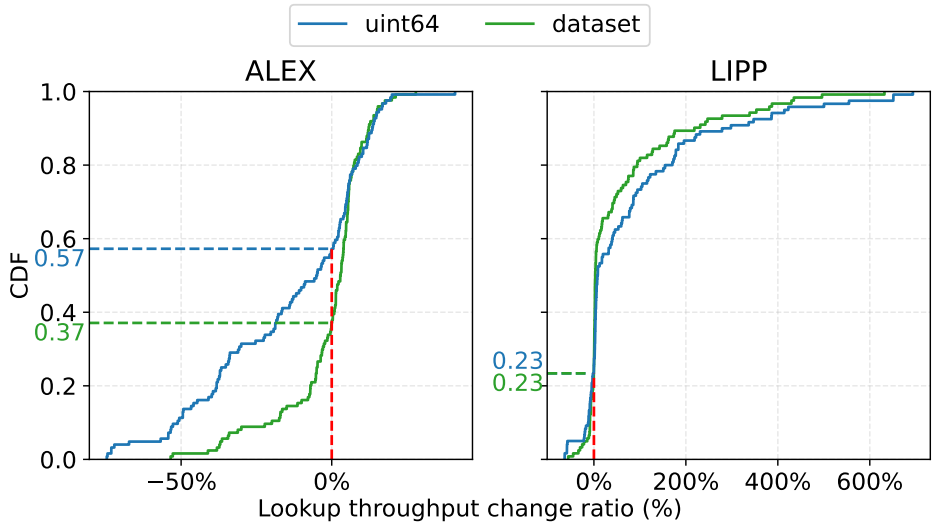


Fig. 16. **CDF of lookup throughput change ratio after bulkloading with data augmentation.** Data augmented by uniform sampled data from key space of entire unsigned 64Byte integer (uint64) or original entire dataset (dataset).

to alleviate model overfitting. However, data augmentation from both key spaces can degrade ALEX's performance by up to 50% in some cases. Overall, data augmentation based on the original dataset's key space yields a more positive effect.

Index users can adopt data augmentation for LIPP, and for ALEX, it is beneficial only when the dataset's key space is known. Future work could explore more sophisticated data augmentation techniques to consistently improve the performance of various indexes. Additionally, our experiments suggest that enhancing the robustness of model training can boost index performance, providing insights for adopting more robust models in learned index design.

6.2 Timely Re-Construction

In previous experiments, the baseline cases generally demonstrated good performance. This suggests that when the volume of inserted data is small relative to the total dataset, updatable learned indexes can efficiently handle insertions using reserved space and conflict resolution strategies. In real-world database applications, building indexes during low-traffic periods is a common optimization practice. Similarly, index users can rebuild updatable learned indexes during off-peak times to enhance their performance.

Through experiments, we investigate the ratio of bulkload data size to total data size (bulkload fraction) at which updatable learned indexes consistently outperform traditional indexes. Considering prior optimizations for interpolation-based B+tree [18], we selected thresholds of 1.5x B+tree performance and 1.1x ART performance for comparison. Figure 17 demonstrates that if the insertion ratio is less than 10% (bulkload fraction > 90%), LIPP can be consistently better than ART and B+tree in all cases. However, ALEX fails to surpass ART and B+tree in difficult datasets even with the whole datasets, but can surpass ART and B+tree if the insertion ratio is less than 40% (bulkload fraction > 60%).

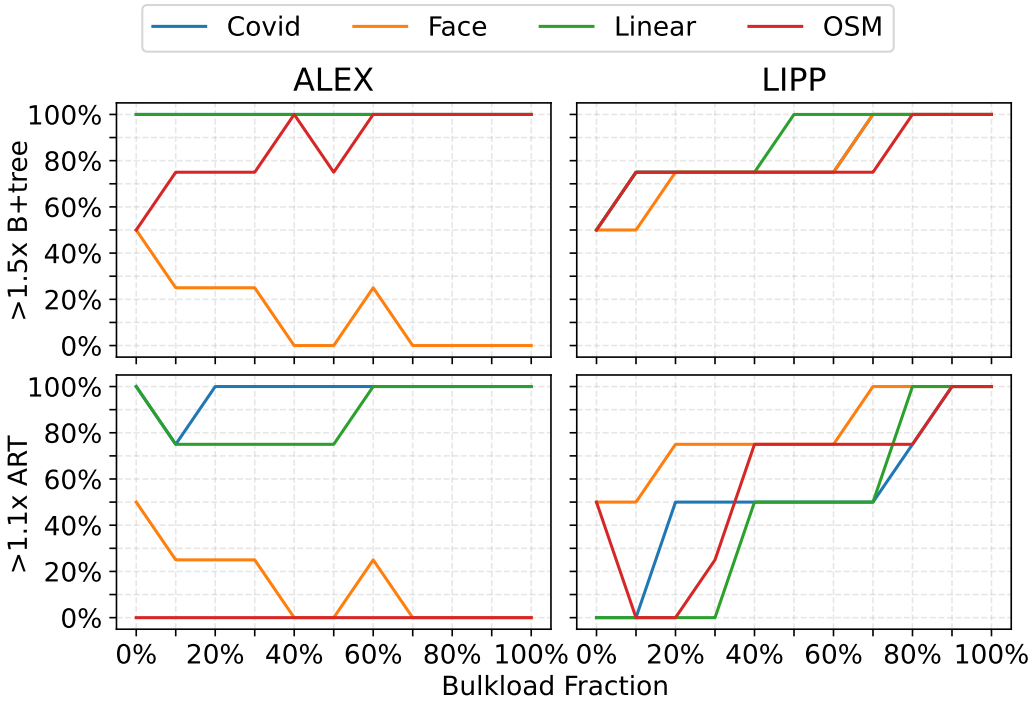


Fig. 17. **Case ratio (learned > traditional) in single-threaded lookup throughput.** Case ratio is the fraction of cases (out of total 124 cases) where ULIs exceed ART (1.1x) and B+tree (1.5x) across bulkload fractions (The bulkload-to-total size ratio).

The experiment indicates that if we can reconstruct LIPP once new inserted data reaches more than 10% of the total data, LIPP can provide a consistently superior lookup throughput, regardless of datasets and workloads. The reconstruction triggering condition can be integrated as part of the updatable learned index configuration to enhance performance during deployment, particularly by rebuilding during off-peak periods. This also hints that a hybrid indexing strategy based on insertion ratios could be beneficial for designing robust updatable learned indexes [56, 63].

7 Related Work

Learned indexes. Kraska et al. introduced the learned index framework called Recursive Model Index, using multi-level models to overfit data distributions [23]. Based on this, various static learned indexes have been developed, focusing on aspects like worst-case bounds [13], construction time [7, 21], and hyperparameter tuning [34, 36]. In parallel, numerous updatable learned indexes have emerged. Early methods used delta-buffers to handle insertions to bound model prediction errors of nodes [13, 14, 45], though they required complex concurrency control to handle concurrent insertions [45]. Recent advancements employ gapped arrays for model-based in-place inserts, optimizing conflict resolution [49], construction algorithms [30, 60], and concurrency control [16].

Additionally, several methods have been proposed to enhance the robustness of updatable learned indexes, including using balanced structure [52], transforming the distribution into uniform [51], integrating with traditional indexes [9, 19], and using dynamization technique (LSM-tree method)

to handle insertions on top of static learned indexes [13, 42, 57]. However, these approaches often fall short in achieving better performance [9, 19, 34, 44], sacrificing range query capabilities [51], or compromising insertion performance (as shown with DyTIS [52] in §4.1).

Benchmarking Learned Indexes. Marcus and Kipf et al. have tested static learned indexes in real-world datasets, which shows static learned indexes have superior performance [20, 35]. In the context of updatable learned indexes, GRE [48] and TLI [44] have tested updatable learned indexes in extensive settings. Both studies demonstrate that updatable learned indexes excel over traditional ones in most standard workloads, except in scenarios involving complex data distributions and intensive insertions.

Learned index applications. In previous real-world applications, system designers have carefully adapted static indexes to accommodate updates by selecting appropriate scenarios, thus enabling the use of learned indexes. Bourbon replaces the traditional index block of LSM-tree-based key-value stores with a static learned index [10]. FILM extends the static PGM-index [13] to efficiently handle appending insertions [33]. Treeline builds a novel key-value store that uses only the last-level PGM-index to index the data segment on disk [54]. Zhang et al. proposed a hybrid learned index framework on disk, using an in-memory B+ tree as a delta-buffer with a static on-disk PGM-index [57]. However, none of these are in-memory indexes, and optimization techniques that perform well on slower devices may not be effective in memory.

Attacking learned indexes. From a security perspective, data poisoning attacks [22] and algorithmic complexity attacks [53] are developed to deteriorate learned indexes' performance. But current data poisoning attacks show insignificant effectiveness (up to 20% drop) [2], and algorithmic complexity attacks are restricted to a specific scenario (i.e., duplicate insertions for ALEX) [53]. In fact, this attack can be easily defended by avoiding inlining duplicate keys inside the node array and instead using pointers to chain them outside the node array.

Robustness Issue of Other Learned Components. There has also been extensive research on robustness in other learned database components, including learned cardinality estimation [46, 58], learned query optimization [26], learned index advisors [62], learned multi-dimensional indexes [31], as well as general concerns related to out-of-distribution data [24] and shifting workloads [50]. Robustness analysis is thus widely recognized as an essential step toward the practical deployment of learned database components.

8 Conclusion

In this paper, we conducted a systematic benchmark and analysis to study the robustness of state-of-the-art updatable learned indexes (ULIs). While ULIs have been lauded for their potential to replace traditional indexes, our findings reveal significant limitations in their robustness: none of the existing ULIs can robustly outperform traditional indexes, even under read-intensive workloads where ULIs excel. We identified four critical root causes and explored two practical mitigation methods, offering advice for both index designers and users. We hope our benchmark and analysis will pave the way for future research to overcome this robustness issue and enable broader real-world adoption of ULIs. Our benchmark is open-sourced at github.com/cds-ruc/RoBin.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (No. 2023YFB4503600), National Natural Science Foundation of China (No. U23A20299), and Beijing Natural Science Foundation (Grant No.4254083).

References

[1] Mikkel Møller Andersen and Pinar Tözün. 2022. Micro-Architectural Analysis of a Learned Index. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 1–12. doi:10.1145/3533702.3534917

[2] Matthias Bachfischer, Renata Borovica-Gajic, and Benjamin I. P. Rubinstein. 2022. Testing the Robustness of Learned Index Structures. arXiv:2207.11575 [cs]

[3] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description , Access and Control (SIGFIDET '70)*. Association for Computing Machinery, 107–141. doi:10.1145/1734663.1734671

[4] Timo Bingmann. 2013. STX B+ Tree v0.9. https://panthema.net/2007/stx-btree/.

[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 521–534. doi:10.1145/3183713.3196896

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Transactions on Database Systems* 47, 1 (March 2022), 1–46. doi:10.1145/3506692

[7] 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 (May 2024), 116:1–116:25. doi:10.1145/3654919

[8] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. doi:10.1145/356770.356776

[9] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn.. In *CIDR*.

[10] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From {WiscKey} to Bourbon: A Learned Index for {Log-Structured} Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[11] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 969–984. doi:10.1145/3318464.3389711

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

[13] 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 (April 2020), 1162–1175. doi:10.14778/3389133.3389135

[14] 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*. ACM, 1189–1206. doi:10.1145/3299869.3319860

[15] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 315–327. doi:10.1109/ICDE55515.2023.00031

[16] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework Based on Probability Models. *Proc. ACM Manag. Data* 1, 4 (Dec. 2023), 258:1–258:25. doi:10.1145/3626752

[17] Daniel Golovin. 2008. *Uniquely Represented Data Structures with Applications to Privacy.* Ph. D. Dissertation. Carnegie Mellon University.

[18] Ali Hadian and Thomas Heinis. 2019. Interpolation-Friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. OpenProceedings.org. doi:10.5441/002/EDBT.2019.93

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

[20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[21] 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. ACM, 1–5. doi:10.1145/3401071.3401659

[22] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, 1331–1344. doi:10.1145/3514221.3517867

[23] 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*. ACM, 489–504. doi:10.1145/3183713.3196909

[24] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *Proc. ACM Manag. Data* 1, 1 (2023), 33:1–33:27. doi:10.1145/3588713

[25] Hai Lan, Zhifeng Bao, J Shane Culpepper, Renata Borovica-Gajic, and Yu Don@inproceedingslan2024fully, title=A Fully On-disk Updatable Learned Index, author=Lan, Hai and Bao, Zhifeng and Culpepper, J Shane and Borovica-Gajic, Renata and Dong, Yu, booktitle=40th IEEE International Conference on Data Engineering (ICDE). IEEE, year=2024 g. 2024. A Fully On-disk Updatable Learned Index. In *40th IEEE International Conference on Data Engineering (ICDE). IEEE*.

[26] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (2024), 1565–1577. doi:10.14778/3654621.3654625

[27] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. doi:10.1109/ICDE.2013.6544812

[28] 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 '16)*. Article 3, 8 pages. doi:10.1145/2933349.2933352

[29] 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 (Oct. 2021), 321–334. doi:10.14778/3489496.3489512

[30] 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 (July 2023), 2212–2224. doi:10.14778/3598581.3598593

[31] 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. doi:10.1007/S00778-024-00893-6

[32] Ge Luo, Ke Yi, Siu-Wing Cheng, Zhenguo Li, Wei Fan, Cheng He, and Yadong Mu. 2015. Piecewise Linear Approximation of Streaming Time Series Data with Max-Error Guarantees. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 173–184. doi:10.1109/ICDE.2015.7113282

[33] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: a fully learned index for larger-than-memory databases. *Proceedings of the VLDB Endowment* 16, 3 (2022), 561–573.

[34] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *Proc. VLDB Endow.* 15, 5 (Jan. 2022), 1079–1091. doi:10.14778/3510397.3510405

[35] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 1–13. doi:10.14778/3421424.3421425

[36] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, 2789–2792. doi:10.1145/3318464.3384706

[37] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Vol. 509518. Citeseer, 509–518.

[38] 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 '24)*. Association for Computing Machinery, 16–22. doi:10.1145/3662165.3662764

[39] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A benchmark on data sortedness. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 17–32.

[40] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for near-sorted data. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1475–1488.

[41] Sylvestre-Alvise Rebuffi, Sven Gowal, Dan Calian, Florian Stimberg, Olivia Wiles, and Timothy Mann. 2021. Data augmentation can improve robustness. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*. 29935–29948.

[42] Douglas B. Rumbaugh, Dong Xie, and Zhuoyue Zhao. 2024. Towards Systematic Index Dynamization. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 2867–2879. doi:10.14778/3681954.3681969

[43] Florian Scheibner. 2016. ART C++. https://github.com/flode/ARTSynchronized.

[44] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (April 2023), 1992–2004. doi:10.14778/3594512.3594528
[45] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 308–320. doi:10.1145/3332466.3374547
[46] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654. doi:10.14778/3461535.3461552
[47] Ziqi Wang. 2017. BTreeOLC. https://github.com/wangziqi2016/index-microbench.
[48] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (July 2022), 3004–3017. doi:10.14778/3551793.3551848
[49] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (April 2021), 1276–1288. doi:10.14778/3457390.3457393
[50] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1 (2024), 38:1–38:27. doi:10.1145/3639293
[51] 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 (June 2022), 2188–2200. doi:10.14778/3547305.3547322
[52] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. 2023. DyTIS: A Dynamic Dataset Targeted Index Structure Simultaneously Efficient for Search, Insert, and Scan. In *Proceedings of the Eighteenth European Conference on Computer Systems*. ACM, 800–816. doi:10.1145/3552326.3587434
[53] Rui Yang, Evgenios M Kornaropoulos, and Yue Cheng. 2024. Algorithmic Complexity Attacks on Dynamic Learned Indexes. *Proc. VLDB Endow.* 17, 4 (March 2024), 780–793. doi:10.14778/3636218.3636232
[54] Geoffrey X Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022).
[55] Sepanta Zeighami and Cyrus Shahabi. 2023. On Distribution Dependent Sub-Logarithmic Query Time of Learned Indexing. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 40669–40680.
[56] 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*. ACM, 1567–1581. doi:10.1145/2882903.2915222
[57] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3 (May 2024), 151:1–151:26. doi:10.1145/3654954
[58] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2024. PACE: Poisoning Attacks on Learned Cardinality Estimation. *Proc. ACM Manag. Data* 2, 1 (2024), 37:1–37:27. doi:10.1145/3639292
[59] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. 2010. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 805–814.
[60] 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 (May 2024), 145:1–145:26. doi:10.1145/3654948
[61] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2023. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *VLDB* (2023), 13.
[62] Yihang Zheng, Chen Lin, Xian Lyu, Xuanhe Zhou, Guoliang Li, and Tianqing Wang. 2024. Robustness of Updatable Learning-based Index Advisors against Poisoning Attack. *Proc. ACM Manag. Data* 2, 1 (2024), V2mod010:1–V2mod010:26. doi:10.1145/3639265
[63] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is better than one: The case for 2-tree for skewed data sets. In *CIDR*.

Received January 2025; revised April 2025; accepted May 2025