Are Updatable Learned Indexes Ready?

Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, Tianzheng Wang[†]

The Chinese University of Hong Kong [†]Simon Fraser University

ABSTRACT

Recently, numerous promising results have shown that updatable learned indexes can perform better than traditional indexes with much lower memory space consumption. But it is unknown how these learned indexes compare against each other and against the traditional ones under realistic workloads with changing data distributions and concurrency levels. This makes practitioners still wary about how these new indexes would actually behave in practice. To fill this gap, this paper conducts the first comprehensive evaluation on updatable learned indexes. Our evaluation uses ten real datasets and various workloads to challenge learned indexes in three aspects: performance, memory space efficiency and robustness. Based on the results, we give a series of takeaways that can guide the future development and deployment of learned indexes.

1 INTRODUCTION

Recent advances in machine learning (ML) have sparked a flurry of research on using ML to improve various database engine components. Many learned components are reported to perform significantly better than traditional indexes. However, it remains unclear (1) what the relative merits are of the various techniques and (2) how those impressive results will hold up under various workloads. Practitioners are often wary about using these new techniques in their systems without knowing how they would perform in practice. In this paper, we hope to fill the gap by studying *learned indexes* [11, 13–16, 18–20, 23, 30, 33, 43, 46, 52, 54, 55, 57, 60, 62], the arguably most well-studied learned database engine component.

Although preliminary evaluations on learned indexes exist, they largely focused on read-only workloads [36, 38]. A system built for practical adoption, however, must also consider dynamic workloads. Hence, we focus on *updatable* learned indexes. Moreover, to provide a holistic evaluation, we focus not only on (1) performance (throughput and latency), but also (2) space efficiency and (3) robustness under dynamic workloads. Next, we briefly introduce our evaluation methodology and highlight our findings.

1.1 Performance

Ideally, a performance study about learned indexes should involve real data since learned indexes can easily overfit to a particular synthetic distribution [2]. The SOSD benchmark [21] has taken the initiative with four real datasets for evaluating learned indexes. Recent evaluations have started using SOSD's real datasets [36, 38], but they only focused on read-only workloads. Among the four real datasets in SOSD, osm and facebook are known to be more challenging for learned indexes [36, 38]. In contrast, learned indexes are able to outperform traditional indexes on the wiki and books datasets [36, 38]. However, it is unclear how these results (i.e., which type of index performs better under which scenarios)

may extend beyond the four concrete datasets, leading to the question: can a learned index still outperform all traditional indexes on a variety of data? We observe the fundamental reason is the lack of a common, *quantitative* metric to evaluate how "hard" (or "easy") a dataset is for a learned index to handle. Furthermore, without a quantitative measure, it is hard for practitioners to know whether their target workload/production data is "easy" or "hard", which may eventually impact which index should be chosen to achieve the best performance.

In this paper, we propose to use *piecewise linear approximation* (PLA) [45] as an approximate metric to quantify the hardness of a dataset. PLA has a clear theoretical foundation from computational geometry, which captures the minimal number of linear models required to fit a data distribution. It has been used to build a space-optimal learned index [16]. We use PLA to approximate the hardness of a dataset in two dimensions: *global hardness* and *local hardness*. The former challenges the index's structure design choices and the cost models that govern structural modification operations (SMOs); the latter challenges the accuracy of various ML models.

The global and local hardness metrics based on PLA enable us to conduct a comprehensive study that pragmatically examines updatable learned indexes along three aspects: (1) **data** (from *easy* to *hard*), (2) **workload** (from *read-only* to *write-only*), and (3) **concurrency** (from *single-threaded*/*single-socket* to *multi-threaded*/*NUMA*). Our results revealed a number of interesting insights. For example, although our results show that recent updatable learned indexes (ALEX [14], LIPP [57]) perform better than traditional indexes (ART [25], Masstree [37], HOT [4] and Wormhole [58]) in over 80% of the data-workload space under a single thread, some learned index design choices are actually at odd with concurrency control and show regression under multiple threads.

1.2 Space Efficiency

Memory consumption is an important factor in production environments. Learned indexes have been reported to use 4×-2000× less memory compared to traditional indexes because they store succinct models instead of keys in nodes [14, 23]. While their space advantage has been proven in read-only environments, we found that it might not hold up in a realistic environment with data modifications. In read-only workloads, the data array to be indexed can be fully ordered and packed, so even the leaf nodes can store no keys: they can store and use a model to point straight to the underlying data array. In contrast, a real system often needs to handle dynamic workloads and builds indexes on non-primary keys. In both cases, the underlying data array could be unsorted or the nodes have to leave spaces to accommodate insertions. Consequently, the index often needs a leaf layer that explicitly stores the key-position information, bloating memory usage.

In this paper, we examine the *end-to-end* memory space efficiency of learned indexes including their leaf layer with key-position pairs.

We found that learned indexes are only 3.2× smaller than traditional indexes at best and might even use more space than some traditional indexes. Note that traditional indexes can consume around 55% of the total memory in transactional in-memory databases [61]. Therefore, our results suggest that memory saving is not a definitive advantage of the current generation of updatable learned indexes and highlight the need for further improvements in this area.

1.3 Robustness

Modern data systems often require predictable performance. Low tail latency is particularly important for user-facing applications [28]. In this paper, we examine whether the high performance of learned indexes comes with a price on poor tail latency. Furthermore, many workloads require efficient and robust support for range queries. In addition to point queries, we therefore also test learned indexes' behavior under varying range scan sizes and compare them with state-of-the-art traditional indexes. Finally, the power of learned indexes is that they allow specialization to a given data distribution, hence becoming *instance-optimized* [2]. However, in a realistic environment, the data can continuously evolve. It then becomes important to study how well and how fast a learned index can adapt to changes. Yet existing performance studies [21, 36, 38] have not looked into this issue. Some recent updatable learned indexes, such as ALEX [14] and XIndex [52], have briefly touched upon this issue but did so without comparing with a wide range of traditional/learned indexes as well as on different data hardness. In this paper, we study the impact of data distribution changes on learned indexes and compare their robustness with that of the traditional indexes. Our results show that although a few learned indexes generally showed good resilience to changes in data distribution, most traditional indexes exhibit even better, extremely robust performance across different workloads and data distributions.

1.4 Contributions, Limitations, and Roadmap

We have created and open-sourced a benchmarking suite, GRE[1] which makes it easy to compare against learned and traditional indexes under the evaluation spectrum proposed in this paper. To facilitate future studies, GRE includes scripts to run the benchmark and visualize all experiments presented in this paper. While earlier effort [38] has presented a leaderboard for learned indexes, GRE further expands the dimension of the comparisons by presenting *heatmaps* to visualize which "area" in the data-workload spectrum has already been "conquered" by learned indexes and which area the traditional indexes are still in possession.

To the best of our knowledge, this is the first comprehensive study of updatable learned indexes' performance, space efficiency and robustness. However, since learned indexes are still fast evolving with new proposals on handling multi-dimensional data [13, 15, 30, 43, 46, 54, 60], persistence [33] and string keys [50, 55], we focus on one-dimensional updatable indexes on numeric data and anticipate future work to cover the other aspects.

In the rest of this paper, Section 2 provides the necessary background on updatable learned indexes. In Section 3, we present our

experimental setup, including how we approximate the hardness of a dataset. Sections 4–7 then present the detailed empirical results. Section 8 summarizes our lessons learned and then we answer the theme question *Are updatable learned indexes ready?* in Section 9.

2 LEARNED INDEXES

The main intuition behind learned indexes is that if the data is a set of continuous integer keys (e.g., the keys 0 to 100M) in an array a, the key itself can be directly used as an offset into the array (e.g., the value of key 77 can be accessed at a[77]). This can potentially allow O(1) rather than O(log n) lookup complexity and significantly reduce index storage overhead. However, implementing a practically useful learned index is challenging in terms of supporting non-contiguous data, dynamic workloads, and concurrency.

2.1 Structure and Last-Mile Search

The ideal case with O(1) complexity is based on a crucial assumption that keys are continuous, which is not necessarily true in reality. Therefore, learned indexes usually train multiple ML models to approximate the cumulative distribution function (CDF) of the data, forming a hierarchy of models akin to the structure of B+-trees with inner and leaf nodes. However, instead of laying out keys physically, inner nodes in learned indexes store models. The model guides traversals to reach leaf nodes which store the actual data. If simple linear models are used, the node only needs to store the slope and intercept, reducing storage footprint. There are two inner node designs for updatable learned indexes:

ML for Child Search. In this design, an inner node *I* uses a model to predict which child node of *I* contains the key. Hence, the model of *I* is trained using the information from its children (e.g., maximum and minimum keys of each child). Since a model might be imperfect and give wrong predictions, it is necessary to search around the predicted position to reach the exact child. PGM-Index [16], XIndex [52] and FINEdex [29] adopt this design.

ML for Subspace Lookup. Indexes using this design (ALEX [14], APEX [33] and LIPP [57]) recursively partition the key space, and a partition is represented by a slot in an inner node. A model in each inner node is used to *decide* which partition a key belongs to. Thus, a traversal simply uses the model to *compute* which child (partition) shall be visited next, instead of predicting.

When reaching a leaf node, there is always a "last-mile" search around the model predicated position because models may not be perfect. Last-mile search often is a performance bottleneck. Different learned indexes use different designs to mitigate it:

Error-Driven Designs. This type of updatable learned indexes requires as input an error threshold to bound the distance of the last-mile search; it is adopted by PGM-Index [16], XIndex [52], FITing-Tree [18] and FINEdex [29]. In case a model's error margin goes beyond the given threshold, the index will carry out adjustment such as increasing the granularity of the models [20] or increasing the number of nodes (and hence increasing the number of models since each node hosts one model) [16, 18, 52].

Performance-Driven Designs. ALEX [14] and APEX [33] maintain runtime statistics (e.g., the number of keys searched in the last-mile and tree height) to detect suboptimal behaviors due to model inaccuracies. These indexes maximize the accuracy by

---

[1]Akin to the GRE test for a learned index to "pass and graduate" for practical use. Available at https://github.com/gre4index/GRE.

triggering the best action (e.g., SMOs) based on an empirical-based cost model.

Collision-Driven Designs. The key idea is to model inaccuracies as "collisions." Specifically, a model-based CDF can be seen as an order-preserving function where different keys could be "hashed" (predicted) to the same "bucket" (array position), causing collisions. Hence, a last-mile search could be handled by an open-addressing collision resolution scheme that gets or puts a collided key elsewhere in the index. Based on this idea, LIPP [57] attempts to eliminate last-mile search by finding a model that minimizes collisions. When a collision is inevitable, it uses a chaining scheme that creates new nodes (or recursively a sub-tree of new nodes) to transform the last-mile search problem to be a sub-tree traversal problem.

2.2 Dynamic Workloads

Beyond lookups, updatable learned indexes also need to support inserts/deletes/updates which can change data distributions. Existing solutions usually use tree-merge, delta-merge or sparse nodes.

Tree-Merge. As represented by PGM-Index [16], tree-merged based approaches are inspired by LSM-trees [44] which build multiple sub-indexes, each of which covers a subset of keys. Inserts are then handled by creating new sub-indexes by merging smaller sub-indexes with the new key to ensure the keys stay sorted. Updates are done in-place and deletes are implemented as inserts of tombstones. Consequently, tree-merge based designs inherit the performance characteristics of LSM-trees, for example, searching for a key may visit multiple sub-indexes of different sizes.

Delta-Merge. Unlike tree-merge designs, in XIndex [52], FITing-Tree [18] and FINEdex [29], new keys are collected in delta nodes and are merged periodically with the main data array with potential model retraining. Updates and deletes are done in-place. The main difference among the various indexes under this design is their delta granularity. For XIndex, new inserts are absorbed by a per-node delta. FINEdex [29] maintains a delta per record to reduce conflicts within a node and facilitate parallel retraining.

Sparse-Nodes. To absorb future inserts, ALEX [14], APEX [33] and LIPP [57] leave space in tree nodes, making them sparse. Updates and deletes are also done in-place. The index then needs to conduct SMOs such as node merges and splits, similar to traditional indexes. Moreover, to balance between the tree's fill factor and model accuracy, the index should consider learned-index specific operations, such as node resizing and model retraining when the workload and data distribution change.

2.3 Concurrency

Since most learned indexes employ a hierarchy of models, classic concurrency control approaches for trees can be adapted to work on learned indexes. Among the surveyed in-memory learned indexes, only FINEdex [29] and XIndex [52] support concurrency. To cope with the in-memory environment, both of them use optimistic locking, which associates a versioned lock per node. The lock word carries a version number, such that readers only need to verify the version did not change before and after their accesses; writers would acquire locks as usual but also increment the version number. This extracts more concurrency by allowing readers to proceed (especially, to traverse inner nodes) without holding locks.

Table 1: Configurations of learned indexes. Indexes named with a "+" sign are our concurrent implementations.

Index	Parameters
ALEX	Max inner/data node size: 16MB
	Min/avg/max node density: 0.6/0.7/0.8
ALEX+	Max inner/data node size: 16MB/512KB
	Min/avg/max node density: 0.6/0.7/0.8
LIPP(+)	Node density: 0.5; max node size: 16MB
	Subtree inserted/conflict ratio: 2/0.1
PGM-Index	Error bound: 64
XIndex	Error bound: 32; delta size: 256
	Error tolerance: 1/4; max number of models per group: 4
FINEdex	Error bound: 32

The use of optimistic concurrency has impact on memory reclamation and delta merging procedures. For example, in XIndex, before merging a node's delta with its main data array, it uses RCU [17] to ensure the existing readers of the delta have all finished. During the merge, writers follow the normal write path but inserts would work on yet another temporary per-node delta (so readers also need to read this temporary delta in additional to the index before merge; XIndex's current implementation uses Masstree [37] to construct the delta). Finally, the temporary delta would be promoted as the official delta after the merge is done. Using this approach, both readers and writers are non-blocking.

3 BENCHMARKING SETUP

We conduct experiments using various datasets and workloads, and compare them against state-of-the-art traditional indexes. To set the stage, we describe the necessary changes done to each index, datasets and workloads, and our environment.

3.1 Index Implementations

Since the implementation of a learned index can highly influence experimental results [21], we follow previous work [27, 36, 38, 59] to use the original authors' or widely-used open-source implementations. Unless otherwise stated, we use the recommended parameters that can be found in each index's original implementation. All the evaluated learned indexes along with their configurations are listed in Table 1. Among the included learned indexes, LIPP and ALEX do not support concurrency out-of-the-box, but our evaluation shows that they are the most competitive under a single thread. So we implemented concurrent versions of them (LIPP+ and ALEX+) to make comprehensive comparisons and reason about their design decisions in realistic multi-core environments.

LIPP recommended lock coupling [1] for concurrency control [57]. However, coupling using traditional locks may severely limit concurrency [26]. Although using optimistic lock coupling [26] can mitigate the impact, we observe that LIPP actually requires no coupling because when following a pointer from a parent node to a child node, the pointer would not be invalidated by another concurrent thread. Yet, LIPP does not differentiate inner node and leaf node but uses a unified node layout: a node can store both data and pointers to child nodes. Unfortunately, this design choice limits scalability because (i) inner nodes now contain keys and must also maintain statistics for SMO decision making (where ALEX only needs to maintain statistics for

leaf nodes) and (ii) even a simple key insert that triggers no SMOs could lock a node at any level (e.g., the root). We mitigate the latter by using item-level optimistic locks (without coupling) for LIPP (denoted as LIPP+), where reads proceed without taking the lock but only need to verify that the read item (i.e. data or child pointer) did not change; only writers are required to take the lock in exclusive mode. Furthermore, we uncovered two bugs in LIPP's original implementation, fixing which led to slightly lower performance compared to what was originally reported [57].[2]

The original implementation of ALEX did not support concurrency. We implemented a concurrent version (ALEX+) by adapting the concurrency protocol of ALEX's persistent memory variant, APEX [33]. APEX employs optimistic locks at data (leaf) node level for every 256 records. By employing out-of-place-based SMOs (i.e., always allocating new nodes), APEX allows tree traversals to proceed without holding locks. Inner nodes are synchronized using per-node shared-exclusive locks. ALEX+ adopts all the concurrency design from APEX except it uses a single optimistic lock per data node, which exhibits better performance in DRAM environments. We excluded FITing-tree [18] in our experiments since it is not open-source.

For traditional indexes, we include STX B+-tree [3], ART [25] and HOT [4] for single-threaded experiments. For multi-threaded experiments, we include B+-TreeOLC [61], ART-OLC (ART with optimistic lock coupling) [26], HOT-ROWEX (HOT with ROWEX [26]), Masstree [37] and Wormhole [58]. Due to space limitation, we omit their details and only cover the necessary changes made by us. We added side-links in leaf nodes for B+-TreeOLC for better range scan performance. Two ART implementations support concurrency; we use the best performing one based on OLC. We ported HOT's epoch-based memory reclamation (EBMR) to ART for better scalability. Although HIST-tree [10] shows that as a traditional index, it can achieve promising speedups by leveraging certain implicit assumptions made by learned indexes (e.g., sortedness), its open-source implementation does not support dynamic workloads and we exclude it from our study.

3.2 Datasets

Table 2 shows the real datasets used in our benchmarks. The first four datasets are from SOSD [21]. osm from SOSD is known to be a one-dimensional projection of multi-dimensional spatial data. Although we focus on single-dimensional learned indexes, we include it for stress testing under the worst case scenarios. Except wiki, each dataset consists of 200M unique 8-byte unsigned integer keys, and we pair each key with an 8-byte payload. wiki has duplicated keys.

These datasets have been used by prior studies but it remains hard to quantify their potential impact on index performance. For example, some studies have shown that learned indexes cannot outperform traditional indexes on the osm dataset [38]. Other results have shown that learned indexes can outperform traditional indexes under wiki and books in read-only workloads [36, 38]. This leads to questions such as *how "hard" is* osm *actually?* and *is it much harder or just a bit harder than* books*?* We thus observe the need

Table 2: Datasets used in experiments.

| Dataset	Description	Source
books	Amazon book sales popularity	[21]
fb	Upsampled Facebook user ID	[21]
osm	Uniformly sampled OpenStreetMap locations	[21]
wiki	Wikipedia article edit timestamps	[21]
covid	Uniformly sampled Tweet ID with tag COVID-19	[32]
genome	Loci pairs in human chromosomes	[47]
stack	Vote ID from Stackoverflow	[51]
wise	Partition key from the WISE data	[56]
libio	Repository ID from libraries.io	[31]
history	History node ID in OpenStreetMap	[7]
planet	Planet ID in OpenStreetMap	[7]
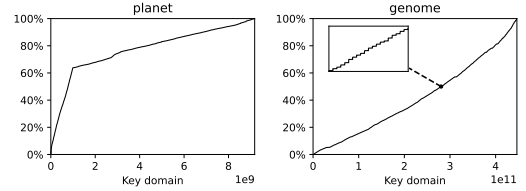
Figure 1: CDFs of planet and genome

to quantify the "hardness" of a dataset. As we detail next, we find that piecewise linear approximation (PLA) [16] is a good start.

All 1D learned indexes regard indexing as mappings from keys to positions. If the data distribution is more non-linear, learned indexes will use more linear models to approximate the distribution. Hence, we could use the minimum number of linear models required to fit the distribution to approximate the data hardness:

DEFINITION (ϵ-APPROXIMATE [16]). *Given an array* $D = [k_1, k_2, \ldots, k_n]$*, where* k_i *is a key with rank* r_i *in the array, if a model* F *for array* D *is* ϵ-approximate, *then* $|F(k_i) - r_i| \leq \epsilon, \forall i \in [1, n]$.

A given array D may not be perfectly fitted by a single ϵ-approximate linear model. But one could split D into multiple small segments $D_1, D_2, \ldots, D_m$ such that for each D_i there exists an ϵ-approximate linear model $F_i, i \in [1, m]$. The minimal collection of those models is then the PLA of D. Computing the optimal PLA model of D can be done in linear time using the algorithm in [16]. For a given array D and an error bound ϵ, we define the data hardness H as the number of segments in D's optimal PLA model, which includes the least number of segments. For the same dataset, H should increase as ϵ decreases. In the rest of this paper, we use $\epsilon = 4096$ and $\epsilon = 32$ to quantify the hardness of a dataset in two dimensions.

When ϵ is large (4096), the PLA is more coarse-grained and can capture the dataset's global non-linearity, which mainly challenges the structural aspect of a learned index. Specifically, since learned indexes conceptually break down a CDF as a tree of models, its structure (e.g., fanout and height) is strongly influenced by the global non-linearity of the CDF. For example, our experimental results (Section 3) indicate that many learned indexes cannot outperform the traditional ones on the planet dataset. By looking at its CDF (Figure 1a), we see that its hardness comes from its sharp deflection of distribution at key value around 1M, giving it high global non-linearity. On datasets like this, learned indexes that

insist on a balanced tree structure (e.g., PGM-Index) would give the dense key region (keys <1M) and the sparse key region (keys >1M) the same height, requiring more layers to be traversed in the sparse key region than needed. ALEX and LIPP combat the issue by adopting unbalanced trees (hence the dense region, if hard-to-fit, can be approximated with more models). Nonetheless, a CDF like the one in Figure 1a is still challenging since it can increase their path length variance (affecting latency) and stress their decision components (e.g., the cost model in ALEX).

Global non-linearity alone is insufficient to fully characterize the hardness of a dataset to learned indexes. For example, we found that the genome dataset also gives a hard time to many learned indexes. However, if looking at its CDF (Figure 1b), its CDF looks smooth. But if we zoom into its CDF, it is found that genome has a very bumpy distribution locally. Non-linearities in local regions challenge the individual machine learning models of a learned index. Hence, we use PLA with a small ϵ (32) to capture local non-linearity.

Global non-linearity (PLA ϵ-4096) and local non-linearity (PLA ϵ-32) together form our *data space* that quantifies the hardness of a dataset. The ϵ values 4096 and 32 are empirically decided. We have also tried other metrics to approximate a dataset's hardness, e.g., by measuring the mean square error of fitting only one linear regression line. But we found that PLA ϵ-4096 and PLA ϵ-32 aligned the best with the actual index performance when there are no other factors (e.g., NUMA, concurrency control) in play.

3.3 Workloads

We devise synthetic workloads that issue requests using the afore-mentioned datasets. For each dataset, we first randomly shuffle all the 200 million keys, and then issue lookup/insert requests according the specified ratios below.

- `Read-Only (0% Write)`: Bulk load all the 200M keys and randomly lookup for 800M keys.
- `Read-Intensive (20% Write)`: Bulk load 100M random keys, then issue requests where 80% are lookup operations and 20% are insert operations that insert all the remaining keys.
- `Balanced (50% Write)`: Same as `Read-Intensive` but 50% are lookup operations and 50% are insert operations.
- `Write-Heavy (80% Write)`: Same as `Balanced` but the lookup/insert ratio is 20%:80%.
- `Write-Only (100% Write)`: Issue 100 million insertion after bulk loading 100 million keys.

For each workload and index, we repeat the experiment three times and report the average throughput (operations per second) and the average latency after bulk loading.

3.4 Hardware and Platform

All the experiments are conducted on a quad-socket machine with four 24-core Intel Xeon Platinum 8268 CPUs clocked at 2.9 GHz. The machine in total has 96 cores and 768GB of main memory. By default, hyper-threading is disabled. We use Ubuntu 18.04 with a 4.15.0-151-generic kernel. All the code is compiled using gcc 8.3.0 under the O3 optimization level.
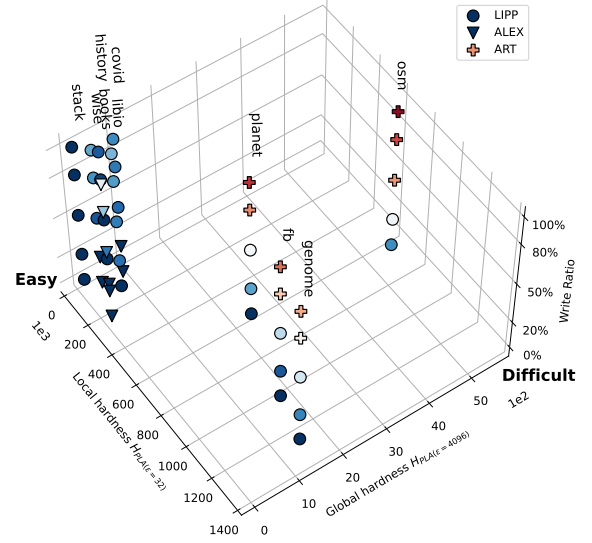

Figure 2: Throughput heatmap (single-threaded) that shows the throughput ratio between the best performing learned index and the best performing traditional index. A positive ratio (in red) means a traditional index is the winner under that particular workload and dataset; a negative ratio (in blue) indicates that a learned index is the winner.

4 PERFORMANCE

In this section, we study on the performance of updatable learned indexes in single-core and multi-core settings.

4.1 Single-Threaded Experiments

We start with single-threaded settings to compare state-of-the-art updatable learned indexes with the traditional ones. Figure 2 shows a heatmap of *throughput ratios* between learned indexes and traditional indexes. The color of each point in the heatmap indicates the throughput ratio between the *best* learned index and the *best* traditional index under that specific data-workload. A blue point indicates a positive ratio, i.e., there exists a learned index that outperforms all the state-of-the-art traditional indexes. A red point (negative ratio) indicates otherwise, i.e., there exists a traditional index that outperforms all the state-of-the-art learned indexes. The darkness of a point represents the "winning" ratio with a darker color indicating the winner outperforms the other indexes by a higher margin. Figure 2 shows that 80% of the data points are in blue color, leading to our first message:

Message 1. In a single-core environment, updatable learned indexes outperform traditional indexes over 80% of our data-workload combinations.

In fact, we have another four datasets but they are also easy. To avoid cluttering the easy region, we include only 10 out of the 14 real datasets.

Message 2. Most real datasets are easy.

In Figure 2, ALEX, LIPP and ART are the overall winners. Strictly speaking, PGM-Index could be the winner on 100% write workloads
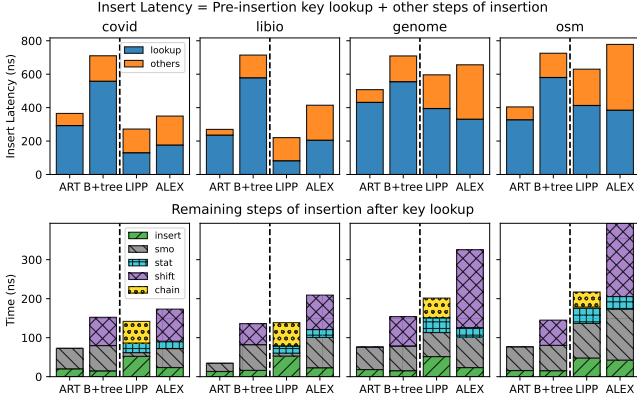
Figure 3: Time breakdown of insert operations.

Table 3: Statistics of an insert operation in ALEX and LIPP.

Dataset	ALEX		LIPP	
	Node traversed	Keys shifted	Node traversed	Node created
covid	1.02	8.07	1.23	0.4
libio	1.04	19.92	1.09	0.4
genome	1.01	42.62	2.12	0.32
osm	1.62	35.84	2.33	0.28

Figure 3(bottom) details the remaining insert steps, where a large part of the latency in learned indexes is due to collision resolution. For ALEX, that is the time spent on shifting the elements in data nodes and carrying out SMOs (e.g., node resizing); for LIPP, that is the time spent on creating and chaining new nodes and its SMO-like adjustment procedure to bound the tree height. Although the shifting and SMO costs in ALEX are not specific to learned indexes (also seen in B-tree variants), they are more expensive than in B-trees and worsen in harder datasets. In contrast, although LIPP also exhibits higher collision resolution costs on harder datasets, it is still smaller than that of ALEX.

Table 3 further shows the detailed statistics per insert in ALEX and LIPP. As the table lists, a harder dataset does not particularly increase the number of chaining operations (node creation) in LIPP but only slightly increase tree traversal time to reach the designated node. That is because LIPP creates at most one new node on collision, which successfully bounds the write amplification to be one node allocation per collision. In contrast, ALEX has a number of key shifts and the number of shifts increases with the data hardness, because a harder dataset challenges ALEX in multiple aspects including its cost models, fill factor, and model accuracy. ALEX's write amplification (the number of key shifts due to a collision) is large because it is only bound by its huge node size (maximum 16MB). Figure 3 also reveals a unique component in learned indexes insertion: the update of the various statistics on insertions. The cost is non-negligible and is particularly pronounced in LIPP because it updates the statistics in every node on the insertion path.

Message 5. In a single-core environment, LIPP's node chaining collision resolution has a lower write amplification than ALEX's key shifting collision resolution.

With the understanding of where the time goes in insertions, it seems that LIPP as a learned index is "ready" by having competitive insert performance and excellent lookup on a majority of real datasets. However, we observe its design is mainly optimized for single-threaded execution and is often at odds with other important aspects, including multi-core scalability as we discuss next.

4.2 Multi-Threaded Experiments

We now move on to the multi-threading environment and begin with the heatmap that shows the best indexes in our data-workload space under 24 threads without hyperthreading in Figure 4. LIPP+, ALEX+ and ART-OLC are the only winners. However, two notable changes are that LIPP+ has lost its leading position to ALEX+ except on read-only workloads; and ART-OLC also has taken over some easy datasets on write-intensive workloads. The latter is because ART has been very competitive — sometimes its performance is close to LIPP and is better than ALEX in the single-threaded setting.
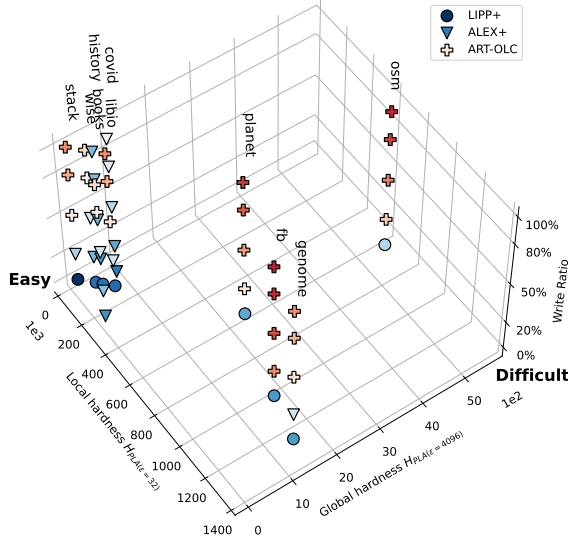
as it exhibits the highest insert throughput when there is no lookup. Nonetheless, its lookup performance is the worst among all the learned indexes and its good insert performance is not attributed to its core design but its LSM-styled approach to handling inserts. We therefore do not show PGM-Index in the heatmap. The main factors that lead ALEX and LIPP perform better than the other learned indexes are (1) the use of ML for subspace lookup in their inner nodes that eliminate search in inner nodes and (2) the sparse-node design avoids visiting multiple trees or delta trees as in PGM-Index, XIndex, and FINEdex. ART as a traditional index outperforms the other traditional indexes because of its cache friendliness [25] and performs especially well on integer keys (whereas the more recent ones like Wormhole and HOT are specialized for long string keys). Yet, ALEX and LIPP as learned indexes can outperform ART except on very hard data because they are instance-optimized. Focusing on the write-intensive workloads leads to the third message:

Message 3. In a single-core environment, updatable learned indexes only cannot outperform traditional indexes on hard datasets with ≥50% writes.

While it is known that hard datasets could give a hard time to learned indexes, Figure 2 shows that recent learned indexes like ALEX and LIPP have already overcome that on *lookup* operations:

Message 4. In a single-core environment, updatable learned indexes outperform traditional indexes from read-only to read-intensive workloads, regardless of the data hardness.

Hence, with the fact that *lookup is the first step of insert* (an insert of key k would first lookup k to locate the slot to be inserted), what **else** inside a learned index's insert operation can out-bleed the speed gain from its first step?

To answer this question, we break down the average latency of the insertions of ALEX and LIPP in the write-only workload. Figure 3 shows the result. We include ART and B+-tree as references. We show the results for two easy datasets (covid) and (libio), the locally hardest dataset (genome), and the globally hardest dataset (osm). Figure 3(top) confirms that learned indexes generally get a more efficient first-step (lookup) in an insertion (except osm), but the remaining steps of an insert perform much worse than the best traditional index (ART) and are no better than B+-trees.

Figure 4: Throughput heatmap under 24 threads (one socket).

When LIPP+ loses its edge in the multi-threading setting, ART-OLC takes over as the best performing index, followed by ALEX+.

Figure 5 (white area) shows the scalability from 2 to 24 cores of the read-only, balanced, and write-only workloads. All the learned indexes scale well on read-only workloads. However, once the workloads include writes, LIPP+ can no longer sustain its scalability despite its use of optimistic concurrency because every insert thread has to update the per-node statistic on its insertion path. That induces high contention and cacheline ping-pong, especially at the root node. This is a drawback of using a unified node layout where statistics have to be maintain in every node rather than in only the leaf nodes like ALEX+. In contrast, ALEX+ scales well by taking the advantage of lock-free lookup and optimistic locking in leaf nodes, until its write amplification becomes severe under hard data.

Message 6. After parallelization, some single-threaded updatable learned indexes (e.g., ALEX) can scale and perform better than native concurrent learned indexes (e.g., XIndex), but some (e.g., LIPP) cannot and perform worse.

4.3 Impact of Hyper-threading/NUMA

The grey area of Figure 5 shows the scalability with 36 and 48 threads with hyper-threading (48 hardware threads per socket). When writes are involved, LIPP+ cannot scale and Wormhole's single lock for the inner layer severely limits concurrency. Except them, all indexes benefit from hyper-threading but show different degree of performance saturation. For example, ALEX+ exhibit slow down from memory bandwidth exhaustion due to its high write amplification and long last-mile search, especially on harder data. In fact, our profiling results indicate that ALEX+ has already saturated the memory bandwidth with 24 threads in one socket.

Moving on to NUMA, Figure 6 shows the scalability of each index when scaling from 1 to 4 sockets. We use the `Interleave` NUMA memory allocation policy that allocates memory pages from different sockets in a round-robin fashion. This is also the setting which yields the best performance for all indexes.

All indexes show diminishing return once we use more than one socket due to cross-socket bandwidth is lower than the intra-socket (24 cores) bandwidth. ART-OLC has difficulty scaling on easier datasets (`libio` and `covid`). That is because those datasets have dense keys that result in dense nodes and hence, greatly increase contention on high thread counts. Masstree crumbles when there are writes because its write amplifications and concurrency control together have exhausted the cross-socket bandwidth [42]. ALEX+, performs worse either with insertion or harder data on two sockets because distributing the memory accesses over two sockets would experience the tighter cross-socket bandwidth bottleneck despite the increased bandwidth in aggregation. ALEX+ however can scale again with more sockets as by then there are more bandwidth channels. Overall, this experiment shows that:

Message 7. Hyper-threading and NUMA have influence on the scalability of learned indexes to various degree, and their core designs play a role in their scalability under hyper-threading/NUMA.

4.4 Deletion Performance

Before we move on to the next section, we complement this section using deletion workloads. LIPP, Masstree, Wormhole, B+TreeOLC, and HOT-ROWEX do not cover deletions. We exclude them from our study except for LIPP (and LIPP+), which we implemented deletion for it. We also extended ALEX's deletion for ALEX+.

In this experiment, for each dataset, we bulk load all the 200M keys, and then randomly issue lookup/delete requests until 100M keys are deleted. Figure 7 shows the single-threaded throughput heatmap under five different deletion workloads, from read-only (0% delete) to delete-only (100% delete). The results of multi-threading are similar and are omitted for space reason. From the results, we see that ALEX, LIPP, and ART (as well as their concurrent versions) are still the most competitive ones. Note that when compared with the insertion workloads (Figure 2), the learned indexes take over more territory from ART even on hard data. Although the deletion path of a learned index is analogous to its insertion path, where deleting a key may also incur write amplification (filling up the gaps) and trigger SMOs (node resizing), one crucial difference is that deleting a key from a data node would not "pollute" a node's ML model. Consequently, deletion and lookup could continue to enjoy high quality model-based search without model pollution or retraining overhead. This makes learned indexes perform even better and outperforms traditional indexes on more datasets and workloads.

Message 8. Deletions in learned indexes are lightweight because there is no model pollution.

5 MEMORY SPACE EFFICIENCY

Most previous work on learned indexes excluded the size of the leaf layer when evaluating memory space efficiency. We aim to study the end-to-end space consumption (i.e., the size of the whole index including both non-leaf and leaf layers) of the indexes. We report the size of the indexes after running the write-only workload
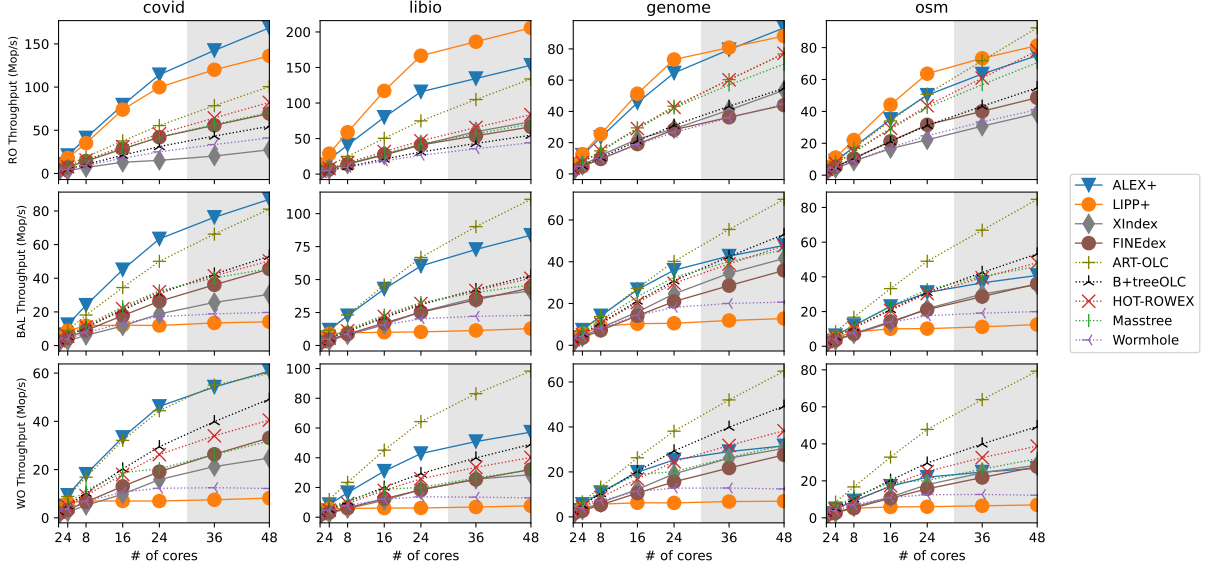
Figure 5: **Throughput of the read-only** (top), **balanced** (middle) and **write-only** (bottom) workloads using **one socket**. The grey area indicates hyper-threading is enabled.
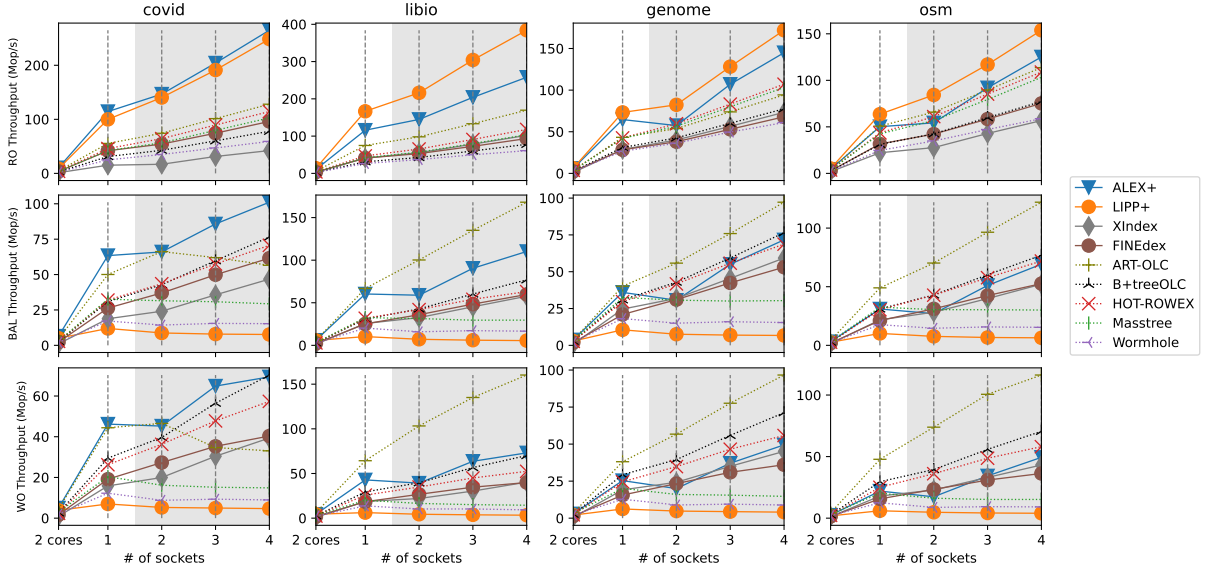

Figure 6: Throughput under varying socket counts.

where the first 100M keys are bulk loaded and the rest of them are individually inserted into the index.

Figure 8 shows the end-to-end space consumption of the indexes. For clarity, we present only indexes that are the best either in terms of throughput or space. From the figure, we observe that:

Message 9. When considering the index size end-to-end, updatable learned indexes only have at most 3.2× space saving over state-of-the-art traditional indexes, and all of them use even more space than HOT.

The factor 3.2× comes from measuring the difference between the sizes of the most space-efficient learned index (PGM-Index) and the least space-efficient traditional index (ART). Although learned indexes do have small non-leaf layers, memory-optimized traditional index like HOT can also be very space-efficient.

Although PGM-Index and HOT are space-efficient, they did not show outstanding performance. Concerning the three winners in the performance heatmaps, ALEX is the most space-efficient because its inner layers store no keys. Yet it is slightly larger than
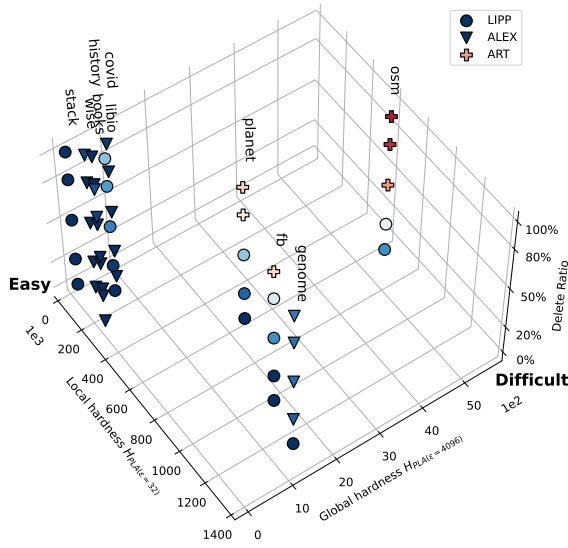
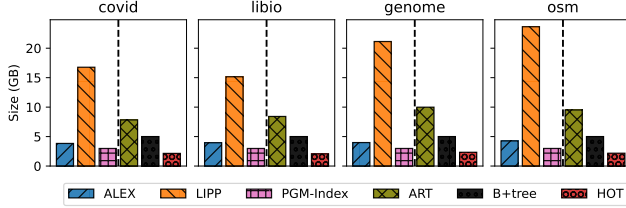**Figure 7: Throughput heatmap (single-threaded) under deletion workloads.**

Figure 8: Memory space efficiency of the best indexes.

PGM-Index because it leaves gaps for insertions. ART as a trie, has low space utilization in its nodes (many null child pointers) when the key space is not dense enough [4]. Despite its excellent single-threaded performance, LIPP's memory consumption is 4–5× larger than ALEX's, and is the highest in this experiment. Having this observation, one might wonder (1) *what would the performance of LIPP and ALEX be if they were given the same memory space?* and (2) *is LIPP's single-thread advantage a fundamental achievement or more a space-performance tradeoff?*

To answer these questions, we carried out another experiment which tuned the fill factor of ALEX data nodes such that the resulting index (ALEX-M) uses roughly the same amount of memory space as LIPP.[3] The resulting fill factor of ALEX-M is 0.2–0.25, whereas ALEX has an original fill factor of 0.7. Figure 9 shows the throughput of ALEX-M and LIPP under this new setting. ALEX's lookup performance has improved significantly and dominated LIPP on both easy and hard datasets. The reason is that with lower density data nodes, an insert in ALEX can often find a gap, maintaining high model accuracy and incurring fewer key shifting. These

---

[3]We cannot do the other way round because of LIPP's implementation sets its fill factor as an integer; hence we could not set any fill factor smaller than 2, and a fill factor of 1 means no gap is allowed.
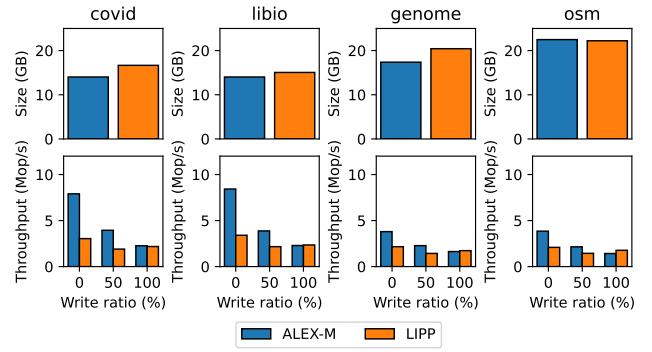

Figure 9: ALEX-M vs. LIPP (when ALEX is tuned to use roughly the same amount of memory as LIPP).

results indicate that LIPP's collision-driven design is trading space for speed.

6 ROBUSTNESS

A robust index shall have low tail latency despite any concurrency degree or heavy-lifting part (e.g., SMO). A reliable index shall also have a robust performance as the underlying data distribution changes. Furthermore, it is important for the indexes to perform robustly under a variety of range scan sizes.

6.1 Tail Latency

In this experiment, we report the tail latencies (variance and 99.9 percentile) of the indexes under both single-threaded and multi-threaded (24 cores) settings. The lookup and insert latencies are sampled from 1% of the operations from the read-only and write-only workloads, respectively. As Figure 10a shows, the updatable learned indexes (except XIndex) exhibit comparable tail latency with the traditional indexes under single-threaded lookup operations. XIndex's latency variance is especially high due to the expensive context switching between its foreground and background threads. Specifically, unlike other indexes, XIndex requires an extra background thread to merge deltas. For fair comparison, we pinned its operational and background threads to the same core so that all the indexes are evaluated using the same CPU budget. This experiment reveals that using background threads to handle dynamic workloads would hurt latency variance. We have confirmed this reason by pinning the background thread to an extra physical core, after which its tail latency is back to normal. The results using multiple threads (Figure 10b) are similar. It is also worth-noting that LIPP+'s tail latencies remain low even under multi-threading, although it did not scale well on throughput. The reason is that LIPP+ uses atomic instructions to update statistics, affecting average latency rather than tail latency.

Figure 11 shows the tail latency for inserts. With a single thread (Figure 11a), updatable learned indexes except XIndex generally have similar tail latencies as the traditional ones. ALEX and LIPP are sensitive to data hardness. They have high 99.9% tail insert latency on the hard osm and genome datasets due to the increased number of SMOs. Similar to its lookup operations, XIndex's insert
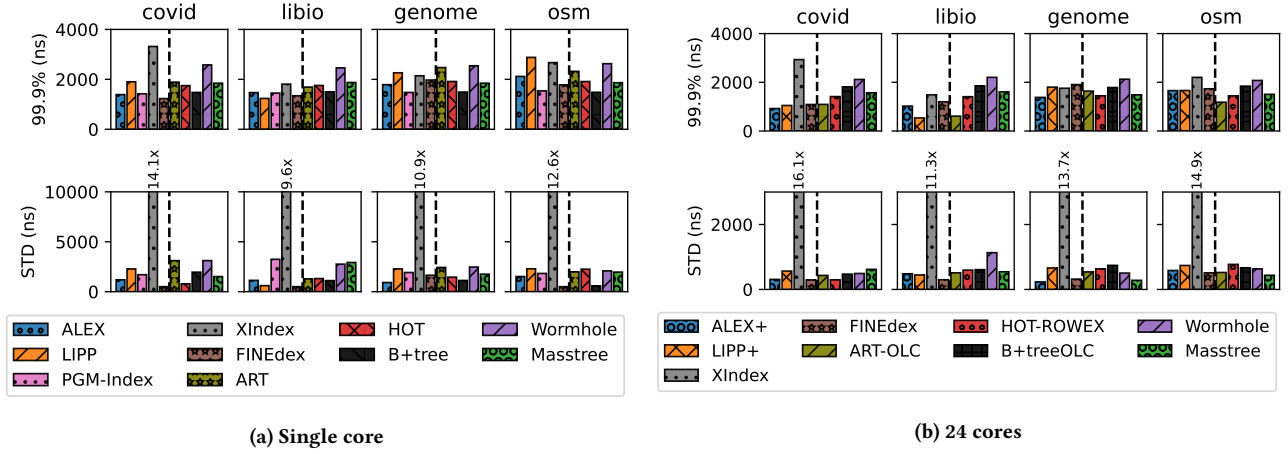
(a) Single core

(b) 24 cores

Figure 10: Tail latency of lookup operations.

(a) Single core
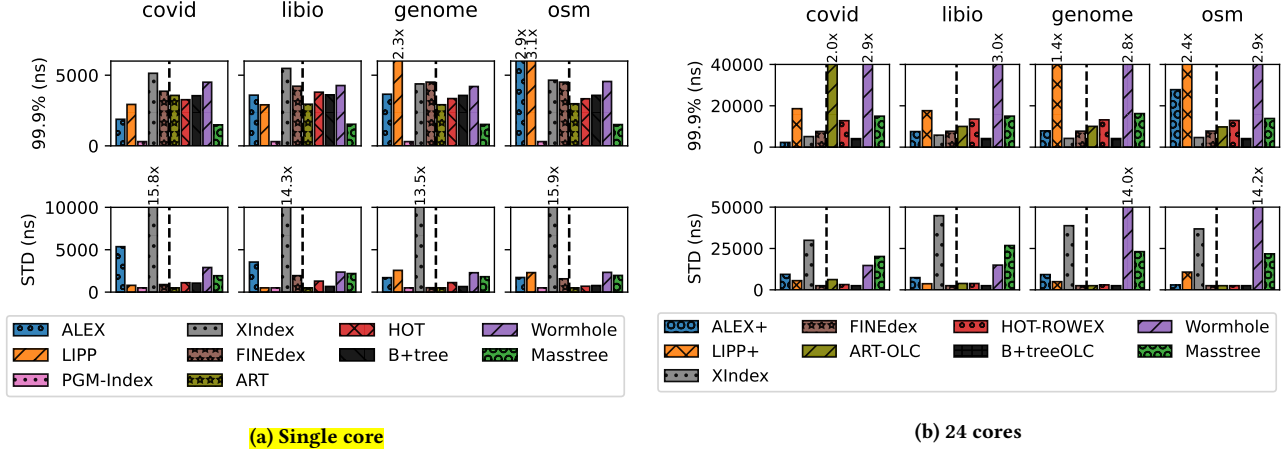

(b) 24 cores

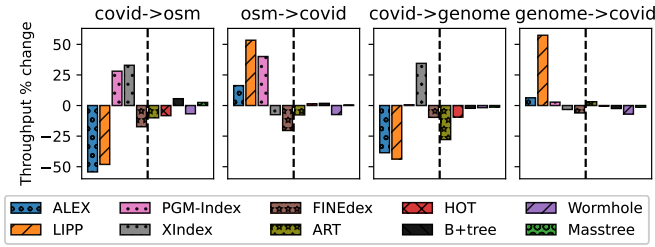**Figure 11: Tail latency of insert operations.**

Figure 12: Throughput changes as data distributions change.

operations exhibit very high variance in tail latencies regardless of the data hardness because of the context switching overhead between its 24 foreground threads and 3 background threads using 24 physical cores.[4] Overall, the multi-threaded results are similar

---

[4]The foreground and background thread ratio follows the recommendation from XIndex's original paper [52].

to the single-threaded ones, except that Wormhole has a higher tail latency for inserts due to its use of a single exclusive lock for the whole inner layer.

Message 10. Except XIndex, updatable learned indexes exhibit low tail latency on both single- and multi-threaded settings. Yet, some traditional indexes (ART, HOT and B+tree) exhibit impeccable robustness in this aspect.

6.2 Shifting Data Distributions

The goal of our next experiment is to study how learned indexes behave and adapt to changes of data distribution after the index has been deployed. We follow earlier work [14] to (1) bulk load an index using 100 million keys in one dataset X, and (2) start a read-write balanced workload using 100 million keys of another dataset Y for insertions and lookups for the keys in X. The keys of both datasets are scaled to the same domain. With the ability to quantify data hardness, we shift from easy data (covid) to two different kinds of hard data (genome and osm) and vice versa.

Figure 13: Range query throughput under varying scan sizes.

Figure 14: Throughput heatmap (single-thread) from synthetic datasets.

Figure 15: Synthetic datasets from heatmap corners.

Figure 12 shows the change of the throughput on the balanced workload with respect to the original workload with no change of the dataset. The result shows that learned indexes are sensitive to data distribution changes while traditional indexes are not. The changes, however, can be both positive and negative. For example, ALEX's throughput can drop by up to 52% when it is bulk-loaded with easy data `covid`, followed by inserts of hard data (`osm`). However, its performance can improve by up to 15% when the hard data `osm` is first bulk-loaded, followed by the easier `covid`. The result aligns with the observations made by prior work [14], where the index starts with easy data can incur significant overhead to adapt to the new, harder distribution. Yet, an index started with harder data requires less/no overhead to adapt to easy data. LIPP behaves similarly to ALEX. PGM-Index and XIndex are more resilient to distribution changes. In PGM-Index, the different distributions are likely to be stored in different trees in its LSM structure. XIndex's throughput is less sensitive to the increased SMOs as they are handled by background threads, which impact tail latency.

Message 11. Learned indexes are sensitive to data distribution changes, while traditional indexes are not. Corroborating with prior work, it is harder (easier) for a learned index that is pre-filled with easy (hard) data to adapt to a harder (easier) dataset.

6.3 Range Queries

This experiment evaluates range query performance. We bulk load each index using the whole dataset of 200M keys and start a read-only scan workload. Each query picks a random start key K and fetches a fixed number of keys starting from K. Each workload issues 10 million range queries in total and we measure the throughput in number of keys accessed per second. Figure 13 shows the results (for indexes that implemented range scan only) under a varying range query size from 10 to 10,000 under a single thread. As shown by the figure, all indexes exhibit higher throughput as the query size increases because a larger scan size involves less tree traversal and more efficient in-node scan. However, this experiment also reveals yet another drawback of LIPP's unified node design: with a node layout that interleaves child pointers and data in the node array, a range scan on the array would inevitably encounter a lot of branches. Specifically, to continue to the next entry in the data array, LIPP needs a branching instruction to decide whether the entry is a child entry (hence recursively visiting the subtree) or a data entry. This largely cancels out the potential improvement brought by using fewer traversals under larger scan sizes.

Message 12. Learned indexes are good on range queries. Yet, the unified node design is not range-scan friendly.

7 COMPLEMENTING REAL DATA WITH SYNTHETIC DATA

The global and local hardness defined based on PLA are mildly correlated because they are extracted from the same CDF after all, just in different granularity. This also explains why we seldom find real datasets that are positioned near the "hard" corners (e.g., globally-hard-locally-easy) in our heatmaps. Although we find no real dataset that exhibits those extreme hardness, we have built a synthetic data generator that generates data using local and global
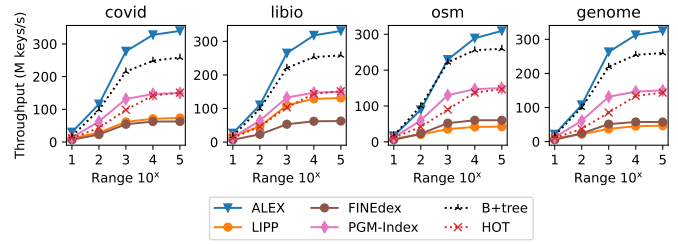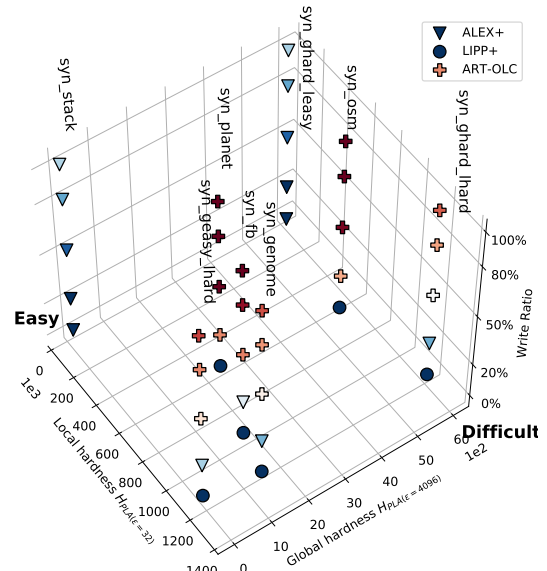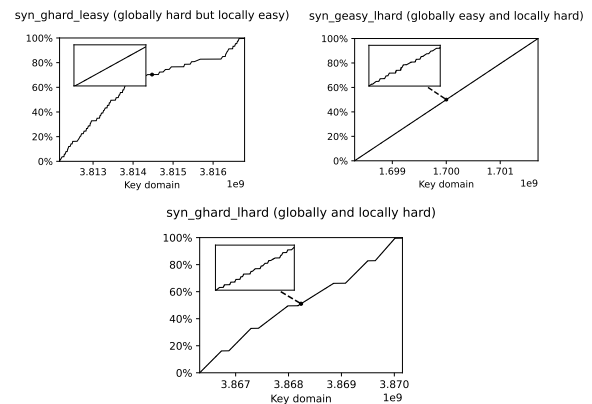
hardness as inputs. Figure 14 is the corresponding single-thread throughput heatmap based on the generated data. We selected local/global hardness values H from the "hard" corners of the heatmap. Figure 15 illustrates their CDFs.

Our synthetic data generator samples data from a set of random linear models. Specifically, we first randomly generate a positive slope m and an intercept b for a segment's linear model. Given that a linear model maps keys to positions, we have $y = mk + b$, where k is a key and y is its rank. For a given rank and ϵ, a key is then uniformly sampled from $[max(\frac{y-\epsilon-b}{m}, prev + 1), \frac{y+\epsilon-b}{m}]$, where $prev$ is the key from rank $y - 1$. Keys are incrementally generated from rank 1 to rank 200M. A segment size is controllable but currently we simply give each segment an equal number of keys. Following the algorithm we described in Appendix C for computing data hardness, we create a convex hull for all keys of a generated segment. Then, we generate the first key k_i of the next segment by incrementing the value of k_i until (k_i, r_i) goes beyond the bounding box of the convex hull of the previous segment. The process is iterative and recursive. We first generate a global segment and then its local segments, and repeat the process until all segments are generated.

Although primitive, we can see that the resulting heatmap from the synthetic data is similar to the one using real data. From the heatmap of synthetic data, learned indexes can also do well when only one dimension is hard while the other one remains easy. In other words, learned indexes lose their edge only when both dimensions are hard and with intensive-writes, corroborating Message 3.

8 LESSONS LEARNED

In this section, we summarize six lessons learned from the study:
1. Using ML for subspace lookup in internal layers and using sparse-node as node design generally well balances performance, space, and robustness. This observation inspires future work to continue in this direction to refine cost models and address limitations (e.g., write amplification).
2. Use hard datasets judiciously. During the experimentation, we found that most real datasets are easy. In contrast, fb has been upsampled and osm is not one-dimensional in nature [7]. Therefore, while we can use those hard datasets to stress test the corner cases, index designers may not need to put too much weights on those.
3. Concurrency control and robustness should be first-class citizens when designing a learned index, i.e., designs should be holistic. Otherwise, different design choices may inherently contradict each other later. For example, LIPP's unified node layout which mainly considered single-threaded scenarios can significantly hurt scalability and add many branches to range scans.
4. Future learned indexes can benefit from cache-friendly and NUMA-aware designs. Since we found that good learned indexes like ALEX+ can saturate memory bandwidth, reducing cache misses and NUMA-aware data placement are two promising optimization directions.
5. Memory efficiency is *not* a clear advantage of updatable learned indexes. Simply replacing traditional indexes with learned indexes may not help much in space saving. We recommend future learned indexes must report the end-to-end memory consumption and explore alternatives (e.g., compression, persistent memory) to reduce DRAM pressure.
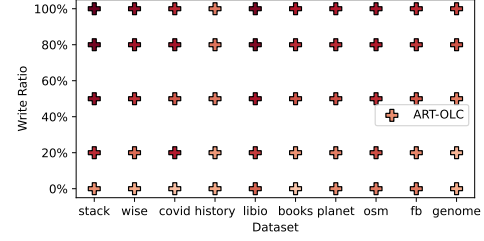

Figure 16: The world without this benchmark study. Heatmap on 24 cores with all workloads. The datasets can't be ordered.

6. Traditional indexes are not good-for-nothing. Traditional indexes should not be completely abandoned because of their efficiency in write-intensive workload and hard datasets, rich functionality (e.g., support for variable-length keys) and extreme robustness.

9 SO, ARE UPDATABLE LEARNED INDEXES READY?

In this section, we try to answer this question from three different angles: *yesterday*, *today*, and *tomorrow*.

Yesterday. If the question were asked before this benchmark study, the answer would have been **no**. Specifically, a modern index must support concurrency, making XIndex [52] and FINEdex [29] the only choices. Figure 16 shows what the 24-core performance heatmap would have been without our study — ART-OLC dominates the heatmap. Furthermore, without a quantitative metric on data hardness, practitioners would have no way to make informed decisions on when to use a learned index or not. In addition to issues with concurrency, we observed that the supposedly fastest (single-threaded) learned index LIPP actually trades space for speed.

Today. Despite being a performance study, we have made contributions to parallelizing two state-of-the-art updatable learned indexes (ALEX+ and LIPP+). We show that LIPP by design is not concurrency-friendly nor space-efficient, yet ALEX+ is promising in terms of performance, scalability, space, and robustness. Although ALEX+ has a harder time on hard datasets when the workloads are write-intensive, most real datasets are easy and most real-world OLTP workloads are actually read-intensive [9, 24, 34, 48]. Hence, we conclude that ALEX+ as an updatable learned index that is **almost ready**. For write-intensive workloads, we recommend LSM-style indexing. Some early efforts have already started to explore this direction [11].

Tomorrow. Despite the promising results, ALEX+ has limitations and we recommend using it in a hardness-conscious manner. For example, the hardness of a dataset can be added as a new feature/dimension in index selection tools [5, 6, 22, 35, 53] and query optimizers [12, 39–41, 49]. When those components are ready, ALEX+ would also be ready.

REFERENCES

[1] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Inf.* (1977).
[2] Laurent Bindschaedler, Andreas Kipf, Tim Kraska, Ryan Marcus, and Umar Farooq Minhas. 2021. Towards a Benchmark for Learned Systems. In *2021 IEEE 37th*

International Conference on Data Engineering Workshops (ICDEW). 127–133.

[3] Timo Bingmann. 2013. STX B+ Tree 0.9. https://panthema.net/2007/stx-btree/, retrieved Sep. 1, 2021.

[4] Robert Binna and et al. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*.

[5] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. 2004. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1313–1323.

[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[7] Google Cloud. 2017. OpenStreetMap. (2017). https://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[9] James C. Corbett and et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*, Chandu Thekkath and Amin Vahdat (Eds.).

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

[11] 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)*.

[12] Dinesh Das, Jiaqi Yan, Mohamed Zaït, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query Optimization in Oracle 12c Database In-Memory. *Proc. VLDB Endow.* 8, 12 (2015), 1770–1781.

[13] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*.

[14] Jialin Ding and et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

[15] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *PVLDB* (2020).

[16] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.* (2020).

[17] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, UK.

[18] Alex Galakatos and et al. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*.

[19] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction. *arXiv preprint arXiv:2101.10457* (2021).

[20] Andreas Kipf and et al. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '20)*.

[21] 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).

[22] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[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 (SIGMOD '18)*.

[24] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB Endow.* 5, 1 (sep 2011), 61–72.

[25] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*.

[26] 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)*.

[27] Lucas Lersch and et al. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* (2019).

[28] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *Proc. VLDB Endow.* (2020).

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

[30] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.

[31] Libraries.io. 2017. Repository ID. (2017). https://libraries.io/data.

[32] Christian E Lopez and Caleb Gallemore. 2021. An augmented multilingual Twitter dataset for studying the COVID-19 infodemic. *Social Network Analysis and Mining* 11, 1 (2021), 1–14.

[33] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610.

[34] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 333–349.

[35] Vincent Y. Lum and Huei Ling. 1971. An optimization problem on the selection of secondary keys. In *Proceedings of the 26th ACM annual conference*. ACM, 349–356.

[36] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *Proc. VLDB Endow.* (2022).

[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 183–196.

[38] Ryan Marcus and et al. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* (2020).

[39] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288.

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

[41] Volker Markl, Guy M. Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Syst. J.* (2003).

[42] Ajit Mathew and Changwoo Min. 2020. HydraList: A Scalable in-Memory Index Using Asynchronous Updates and Partial Replication. *Proc. VLDB Endow.* (2020).

[43] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[44] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* (1996).

[45] Joseph O'Rourke. 1981. An On-Line Algorithm for Fitting Straight Lines between Data Ranges. *Commun. ACM* (1981).

[46] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 12 (2020), 2341–2354.

[47] Suhas S.P. Rao and et al. 2014. A 3D Map of the Human Genome at Kilobase Resolution Reveals Principles of Chromatin Looping. *Cell* (2014).

[48] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. 2018. OLTPshare: The case for sharing in OLTP workloads. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1769–1780.

[49] Srinath Shankar and et al. 2012. Query optimization in microsoft SQL server PDW. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*. ACM, 767–776.

[50] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing (Extended Abstracts). In *3rd International Workshop on Applied AI for Database Systems and Applications, AIDB Workshops*.

[51] Stackoverflow. 2021. Vote ID. (2021). https://archive.org/download/stackexchange.

[52] Chuzhe Tang and et al. 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 (PPoPP '20)*.

[53] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering*. 101–110.

[54] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. 569–574.

[55] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 17–24.

[56] Edward L Wright, Peter RM Eisenhardt, Amy K Mainzer, Michael E Ressler, Roc M Cutri, Thomas Jarrett, J Davy Kirkpatrick, Deborah Padgett, Robert S McMillan, Michael Skrutskie, et al. 2010. The Wide-field Infrared Survey Explorer (WISE): mission description and initial on-orbit performance. *The Astronomical Journal* 140, 6 (2010), 1868.

[57] Jiacheng Wu and et al. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.*

[58] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *EuroSys*. 18:1–18:16.

[59] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 641–652.

[60] Zongheng Yang and et al. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *ACM SIGMOD International Conference on Management of Data*.

[61] Huanchen Zhang and et al. 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 '16)*.

[62] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A Learned Spatial Index for Range and kNN Queries. In *17th International Symposium on Spatial and Temporal Databases*.

APPENDIX

A LOCK GRANULARITY IN ALEX+

We implemented two versions of ALEX+ using different lock granularities. We finally adopted the version that uses a single optimistic lock per data node over the one that uses one optimistic lock per 256 records in a data node.

Figure A shows the throughput of the two versions under the `Balanced` workload on four real datasets. It shows that using a single optimistic lock per data node is consistently better regardless of the data hardness. The overhead of using a lock per 256 records is higher although it admits more concurrency. Specifically, ALEX+ inherits ALEX's design to use exponential search within a data node and when using one lock per 256 records, locks could be acquired in different orders as a search can go either direction within a data array. This can lead to deadlocks. To prevent deadlock, per-256-record locking requires ALEX+ to release all the acquired locks once it fails to acquire a lock and restarts. In contrast, per-node locking is deadlock-free. Moreover, the lock can be inlined within a data node's header, accessing which incurs no extra cost if the header is smaller than a cacheline because the node header contains essential metadata (e.g., the ML model) that must be accessed anyway.

B SUPPORT FOR NON-UNIQUE KEYS

Our evaluation has been mainly using workloads with unique keys only. But one may also build indexes on non-primary attributes with duplicated keys. In this section, we explore the impact of duplicate keys on learned indexes.

There are two main approaches for an index to handle duplicated keys: (1) using a linked list and (2) inlining. For (1), values of the same key form a linked list, and only the first occurrence of key would be presented in the nodes. For (2), all occurrences of the values of the same key would be stored (inlined) in the index node.

Generally, the use of inlining or linked list is a tradeoff between lookup and insert. Inlining favors lookup because the values are co-located in memory, traversing which does not require additional pointer chasing. However, inlining is less friendly to insertion and may require sophisticated space management (e.g., for variable-size payload) in index nodes, increasing the write amplification as new inserts may cause the payloads and keys to be moved around in the node (e.g., to keep key-value pairs sorted). In contrast, the linked list approach favors insertion because the values are stored out-of-place; the index node only needs to store a pointer to the list of values per key. Yet, the linked list approach may reduce lookup throughput because of pointer chasing.

Among all the learned indexes surveyed, none but ALEX (hence ALEX+) supports duplicated keys using inlining. Hence, to study whether learned indexes would follow the tradeoff discussed above when facing duplicated keys, we implemented a version of ALEX+ that uses linked list to handle duplicates (denoted as ALEX+LL). Figure B shows the performance of ALEX+ (that uses inlining) and ALEX+LL on SOSD's `wiki` dataset, which contains duplicates. The results show that ALEX+ as the overall best concurrent learned index also exhibits the aforementioned tradeoff, where the use of linked list makes it better in insert but the use of inlining makes in better it lookup.
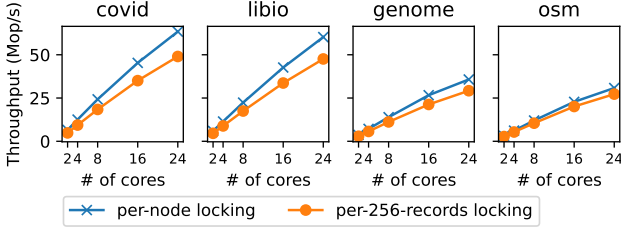
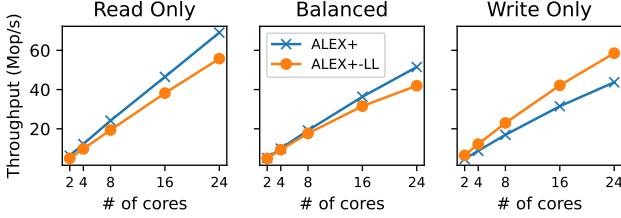Figure A: Throughput of ALEX+ using different lock granularities under the Balanced workload.

Figure B: Throughput of ALEX+ on the `wiki` dataset with non-unique keys under different workloads.

Figure C: Throughput of the Balanced workload under varying PLA local hardness with a small ϵ value (32) .

Figure D: Throughput of the Balanced workload under vaying PLA ($\epsilon = 4096$) global hardness.

C COMPUTING THE OPTIMAL PLA-MODEL

We use the algorithm in [16] to compute the optimal PLA-model of a given dataset. It belongs to a family of online algorithms [45] that aim to fit straight lines for time-series data, where each data point (t, v) has a timestamp t with an error range ϵ on the value v. Their goal is to maintain a set of straight lines that can fit all the data points within their error ranges. Since the data is time-series, t_i is smaller than t_j if $i < j$. In [16], it views each value in a data array as a tuple (k, r), where k is the key and r is the position in the data array. In range indexes, it is natural that k_i is smaller than k_j if $i < j$. Hence, the algorithm in [16] leverages the time-series-like total order to achieve linear time and space complexity. Briefly, the idea is to incrementally construct a convex hull for a set of points (k_i, r_i) for $i = 1...n$. The rank r_i would be within $\pm\epsilon$ as long as their convex hull can be enclosed by a bounding box with height 2ϵ. Incrementally updating the convex hull admits linear time and space complexity. If a point (k_i, r_i) goes beyond the bounding box, that means the current set of data cannot be fitted by any straight line, then the algorithm increments the number of segments by one and moves on to construct the next segment starting with k_i. For more details, interested readers may refer to [16, 45].

D APPROXIMATING DATA HARDNESS

All single-dimensional indexes, to our best knowledge, are designed as a tree of linear models. Therefore, the PLA of a single-dimensional dataset that captures the minimal number of linear models H required to fit the data distribution is a natural candidate to approximate the data hardness. Yet, PLA has a subtle parameter ϵ that governs the approximation quality. Intuitively, a good approximation (i.e., a good choice of ϵ) shall align best with any learned index performance. In other words, if a data hardness approximation metric determines that dataset A is easier than dataset B (i.e.,
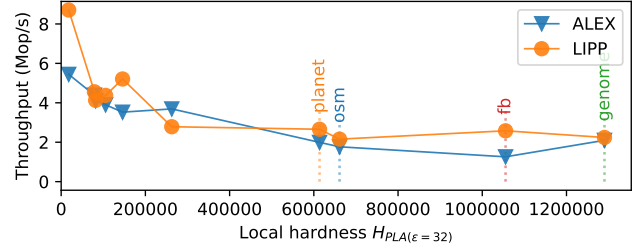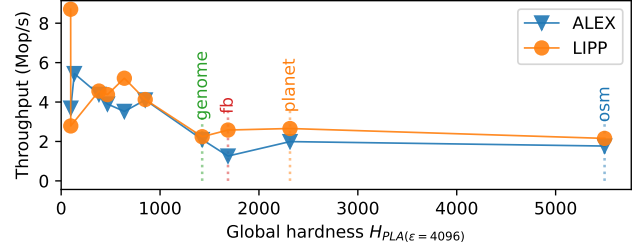
$H_A < H_B$), then it would be a good approximation if all learned indexes perform better on A than on B. That forms our basis for choosing the ϵ values.

From a high-level, our goal is to choose a pair of ϵ values for local and hardness approximation, respectively. The best choice should lead to data hardness (both local and global) that aligns well with the index performance expectations (i.e., harder leads to lower performance). In general, a small/large ϵ in PLA is a fine-grained approximation of the CDF that captures the local/global non-linearity better. Therefore, to pick a pair of suitable ϵ values, we first fix the small value, and try different large values to observe whether the resulting approximation would align with index performance behaviors. We repeat this process with different small values (for each of which we subsequently test different large values) to find a desirable pair eventually. We empirically experimented various values and found that many choices would work and provide similar results (e.g., 32/4096, 64/2048, etc.). For brevity, below we take 32/4096 as the small/large ϵ values (which led to the best results among our tests) to explain the rationale and process in more detail.

Figure C shows the throughput of ALEX and LIPP under different data hardness based on a small $\epsilon = 32$ value (other learned indexes behave similarly; omitted for clarity). It is a fairly good approximation because the performance of the surveyed learned indexes align with the data hardness pretty well — the index throughput generally degrades when H increases. However, we observe that the use of PLA ($\epsilon = 32$) alone is insufficient because the indexes show similar throughput even though H_{planet} and H_{osm} are much smaller than H_{fb} and H_{genome} under that metric. That essentially implies that `planet` and `osm` possess some other hardness that is not captured by PLA ($\epsilon = 32$).

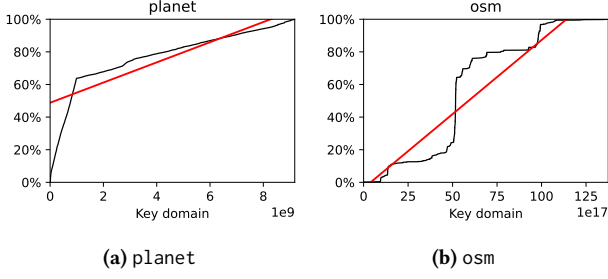**(a)** `planet`

(b) `osm`

Figure E: Fitting one linear regression model (red line) and using the MSE as the global hardness.
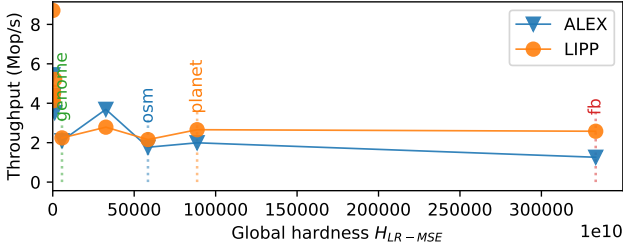

Figure F: Throughput of the Balanced workload under varying global hardness based on MSEs.

With a small ϵ capturing local hardness, now we experiment with different large values for global non-linearity. Figure D shows the throughput of ALEX and LIPP by arranging a dataset's global hardness using PLA with $\epsilon = 4096$. A high ϵ value means a large step size when segmenting a CDF, which can capture a dataset's global non-linearity. Furthermore, PLA is less sensitive to outliers (e.g., the few outliers in `fb` can all be represented by a few new models, hence adding the hardness value only by a bit). By using PLA with a large ϵ, the approximation correctly ranks `planet` and `osm` as globally harder than `fb` and `genome`, complementing the local non-linearity approximation obtained using a small ϵ value.

One may notice that there are other alternatives to complement local non-linearity. For example, we can fit the whole CDF using one linear regression model and use its mean-square-error (MSE) to measure its global non-linearity, as shown in Figure E. However, we note using a large ϵ value in PLA can capture the global non-linearity better than using MSEs. Figure F shows the throughput of ALEX and LIPP by arranging the datasets' global hardness using their MSEs. In the figure, `planet` and `osm` are harder than `genome` in global non-linearity, indicating the approach can complement the local non-linearity dimension. However, it still cannot explain the behavior of the indexes on `fb`. Specifically, `fb` is regarded as much harder than `planet` and `osm` both locally (Figure C) and "globally" (Figure F). One would therefore expect an index to perform worse on `fb` than on `osm` and `planet`. However, we observe that for the same index (e.g., ALEX), it performs more or less the same across `planet`, `osm` and `fb`. In other words, `fb`'s overall hardness (considering both local and global) should be similar to the other two datasets, otherwise the index should not perform similarly across all the

three datasets. This shows the limitation of MSE: it is too sensitive to outliers, giving `fb` (which has a few outliers with very large keys) an overly-high global non-linearity. Yet using PLA as we explained above can model the data hardness better.

E YCSB RESULTS

Our evaluation has mainly focused on index performance under workloads whose keys are uniformly sampled from real data. In this section, we study the index performance using workloads with non-uniform key distributions. Specifically, we use three workloads from the industrial-strength YCSB benchmark [8]:

- YCSB-A: an update-heavy workload that contains 50% lookup requests and 50% update requests where keys are chosen under a Zipfian distribution.
- YCSB-B: similar to YCSB-A except it is read-heavy with 95% of requests are lookups and 5% are updates.
- YCSB-C: similar to the other two except it is a read-only workload with 100% lookup.

The default Zipfian constant in YCSB is 0.99. Figure G shows that our findings in the main discussion can be extended to non-uniform workloads. For example, ALEX and LIPP are still the leaders in most cases while ART comes close on hard data. It is worth noting that LIPP+ remains competitive in YCSB workloads even using multiple cores (Figure Gb). YCSB workloads have no key insertions but only updates the payloads of existing keys. Hence, it can scale because there are no updates on the per-node statistics along its update path and triggering no atomic instructions. Yet, that does not change the fact that LIPP consumes huge space, cannot scale with inserts, and is not range-scan friendly.
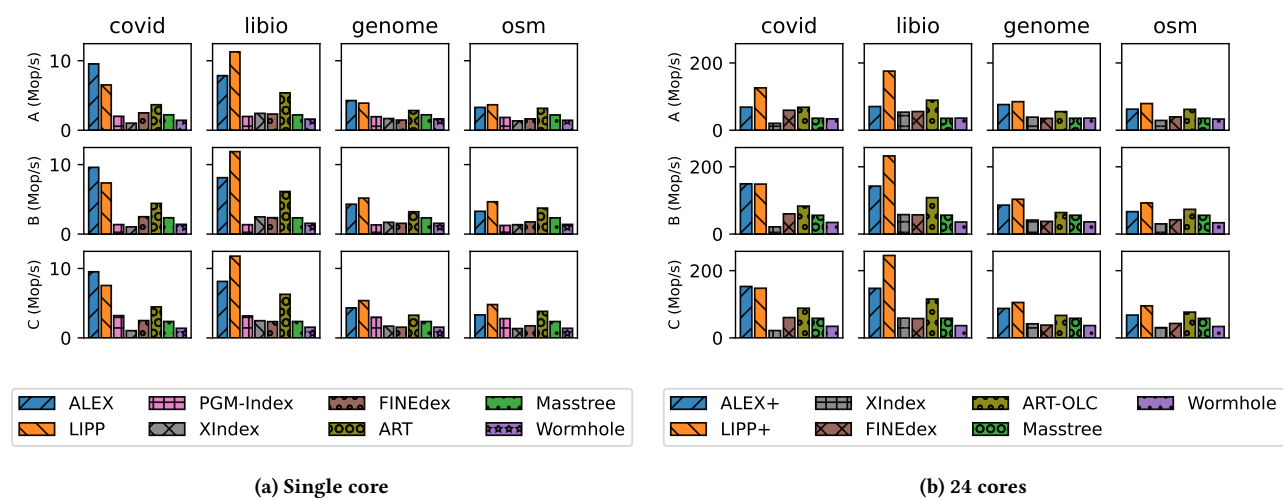
(a) Single core

(b) 24 cores

Figure G: Throughput of indexes under YCSB workload with Zipfian distribution.