Benchmarking Learned Indexes

Ryan Marcus
MIT CSAIL / Intel Labs
ryanmarcus@csail.mit.edu

Andreas Kipf
MIT CSAIL
kipf@csail.mit.edu

Alexander van Renen
TUM
renen@in.tum.de

Mihail Stoian
TUM
stoian@in.tum.de

Sanchit Misra
Intel Labs
sanchit.misra@intel.com

Alfons Kemper
TUM
kemper@in.tum.de

Thomas Neumann
TUM
neumann@in.tum.de

Tim Kraska
MIT CSAIL
kraska@csail.mit.edu

ABSTRACT

Recent advancements in learned index structures propose replacing existing index structures, like B-Trees, with approximate learned models. In this work, we present a unified benchmark that compares well-tuned implementations of three learned index structures against several state-of-the-art "traditional" baselines. Using four real-world datasets, we demonstrate that learned index structures can indeed outperform non-learned indexes in read-only in-memory workloads over a dense array. We investigate the impact of caching, pipelining, dataset size, and key size. We study the performance profile of learned index structures, and build an explanation for why learned models achieve such good performance. Finally, we investigate other important properties of learned index structures, such as their performance in multi-threaded systems and their build times.

PVLDB Reference Format:
Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. PVLDB, 14(1): 1 - 13, 2021.
doi:10.14778/3421424.3421425

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://learned.systems/sosd.

1 INTRODUCTION

While index structures are one of the most well-studied components of database management systems, recent work [11, 18] provided a new perspective on this decades-old topic, showing how machine learning techniques can be used to develop so-called *learned* index structures. Unlike their traditional counterparts (e.g., [9, 14, 15, 19, 30, 32]), learned index structures build an explicit model of the underlying data to provide effective indexing.

Since learned index structures have been proposed, they have been criticized [25, 26]. These criticisms were motivated by the lack of an efficient open-source implementation of the learned index structure, inadequate data-sets, and the lack of a standardized benchmark suite to ensure a fair comparison between the different approaches.

Even worse, the lack of an open-source implementation forced researchers to re-implement the techniques of [18], or use back-of-the-envelop calculations, to compare against learned index structures. While not a bad thing *per se*, it is easy to leave the baseline unoptimized, or make other unrealistic assumptions, even with the best of intentions, potentially rendering the main takeaways void.

For example, recently Ferragina and Vinciguerra proposed the PGM index [12], a learned index structure with interesting theoretical properties, which is recursively built bottom-up. Their experimental evaluation showed that the PGM index was strictly superior to traditional indexes as well as their own implementation of the original learned index [18]. This strong result surprised the authors of [18], who had experimented with bottom-up approaches and found them to be slower (see Section 3.4 for a discussion why this may be case). This motivated us to investigate if the results of [12] would hold against tuned implementations of [18] and other structures.

Further complicating matters, learned structures have an "unfair" advantage on synthetic datasets, as synthetic datasets are often surprisingly easy to learn. Hence, it is often easy to show that a learned structure outperforms the traditional approaches just by using the right data. While this is true for almost any benchmark, it is more pronounced for learned algorithms and data structures, as their entire goal is to automatically adjust to the data distribution / workload.

In this paper, we try to address these problems on three fronts: (1) we provide a first open-source implementation of RMIs for researchers to compare against and improve upon, (2) we created a repository of several real-world datasets and workloads for testing, and (3) we created a benchmarking suite, which makes it easy to compare against learned and traditional index structures. To avoid comparing against weak baselines, our open-source benchmarking suite [4] contains implementations of index structures that are either widespread, tuned by their original authors, or both.

Understanding learned indexes. In addition to providing an open source benchmark for use in future research, we also tried to achieve a deeper understanding of learned index structures, extending the work of [16]. First, we present a Pareto analysis of three recent learned index structures (RMIs [18], PGM [12], and RS [17]) and

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:10.14778/3421424.3421425

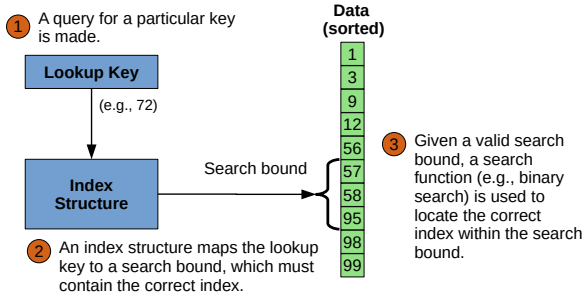


Figure 1: Index structures map each lookup key to a *search bound*. This search bound must contain the "lower bound" of the key (i.e., the smallest key ≥ the lookup key). The depicted search bound is valid for the lookup key 72 because the key 95 is in the bound. A search function, such as binary search, is used to locate the correct index within the search bound.

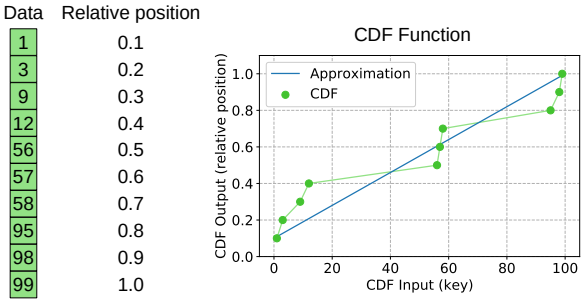


Figure 2: The cumulative distribution function (CDF) view of a sorted array.

several traditional index structures, including trees, tries, and hash tables. We show that, in a warm-cache, tight-loop setting, all three variants of learned index structures can provide better performance/size tradeoffs than several traditional index structures. We extend this analysis to multiple dataset sizes, 32 and 64-bit integers, and different search techniques (i.e., binary, linear, interpolation).

Second, we analyze *why* learned index structures achieve such good performance. While we were unable to find a single metric that fully explains the performance of an index structure (it seems intuitive that such a metric does not exist), we offer a statistical analysis of performance counters and other properties. The single most important explanatory variable was cache misses, although cache misses alone are not enough for a statistically significant explanation. Surprisingly, we found that branch misses do *not* explain why learned index structures perform better than traditional structures, as originally claimed in [18]. In fact, we found that both learned index structures and traditional index structures use branching efficiently.

Third, we analyze the performance of index structures in the presence of memory fences, cold caches, and multi-threaded environments, to test behavior under more realistic settings. In all scenarios, we found that learned approaches perform well.

However, our study is not without its limitations. We focused only on read-only workloads, and we tested each index structure in isolation (e.g., a lookup loop, not with integration into any broader application). While this certainly does not cover all potential use cases, in-memory performance is increasingly important, and many write-heavy DBMS architectures are also moving towards immutable read-only data-structures (for example, see LSM-trees in RocksDB [3, 20]). Hence, we believe our benchmark can serve as a foundation for mixed read/write workloads and the next generation of learned index structures which supports writes [10, 12, 13].

2 FORMULATION & DEFINITIONS

As depicted in Figure 1, we define an index structure I over a zero-indexed sorted array D as a mapping between an integer lookup key $x \in \mathbb{Z}$ and a *search bound* $(lo, hi) \in (\mathbb{Z}^+ \times \mathbb{Z}^+)$, where $\mathbb{Z}^+$ is the positive integers and zero:

$$I : \mathbb{Z} \rightarrow (\mathbb{Z}^+ \times \mathbb{Z}^+)$$

We consider only sorted data and integer keys. We assume that data is stored in a way supporting fast random access (e.g., an array).

Search bounds are indexes into D. A valid index structure maps any possible lookup key x to a bound that contains the "lower bound" of x: the smallest key in D that is greater than or equal to x. Formally, we define the lower bound of a key x, $LB(x)$, as:

$$LB(x) = i \leftrightarrow \left[D_i \geq x \wedge \neg\exists j (j < i \wedge D_j \geq x)\right]$$

As a special case, we define the lower bound of any key greater than or equal to the largest key in D as one more than the size of D: $LB(\max D) = |D|$. Our definition of "lower bound" corresponds to the C++ standard [1]. We say that an index structure is *valid* if and only if it produces search bounds that contain the lower bound for every possible lookup key.

$$\forall x \in \mathbb{Z}\, [I(x) = (lo, hi) \rightarrow D_{lo} \leq LB(x) \leq D_{hi}]$$

Intuitively, this view of index structures corresponds to an *approximate index*, an index that returns a search range instead of the exact position of a key [7, 18]. Given a valid index, the actual index of the lower bound for a lookup key is located via a "last mile" search (e.g., binary search). This last mile search only needs to examine the keys within the provided search bound (e.g., Figure 1).

2.1 Approximating the CDF

Learned index structures use machine learning techniques ranging from deep neural networks to simple regression in order to model the *cumulative distribution function*, or CDF, of a sorted array [18]. Here, we use the term CDF to mean the function mapping keys to their relative position in an array. This is strongly connected to the traditional interpretation of the CDF from statistics: the CDF of a particular key x is the proportion of keys less than x. Figure 2 shows the CDF for some example data.

Given the CDF of a dataset, finding the lower bound of a lookup key x in a dataset D with a CDF CDF_D is trivial: one simply computes $CDF_D(x) \times |D|$. Learned index structures function by *approximating* the CDF of the dataset using learned models (e.g., linear regressions). Of course, such learned models are never entirely accurate. For example, the blue line in Figure 2 represents one possible imperfect approximation of the CDF. While imperfect, this approximation has a bounded error: the largest deviation from the blue line to the actual CDF occurs at key 12, which has a true CDF value

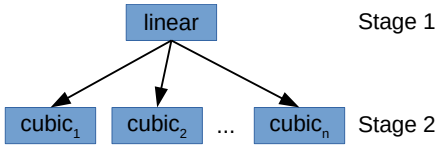


Figure 3: A recursive model index (RMI). The linear model (stage 1) makes a coarse-grained prediction. Based on this, one of the cubic models (stage 2) makes a refined prediction.

of 0.4 but an approximated value of 0.24. The maximum error of this approximation is thus $0.4 - 0.24 = 0.16$ (some adjustments may be required for lookups of absent keys). Given this approximation function A and the maximum error of A, we can define an index structure I_A as such:

$$I_A(x) = (A(x) - |D| \times 0.16, A(x) + |D| \times 0.16)$$

Note that this technique, while utilizing approximate machine learning techniques, *never produces an incorrect search bound*.

One can view a B-Tree as a way of memorizing the CDF function for a given dataset: a B-Tree in which every nth key is inserted can be viewed as an approximate index with an error bound of $n - 1$. At one extreme, if every key is inserted into the B-Tree, the B-Tree perfectly maps any possible lookup key to its position in the underlying data (an error bound of zero). Instead, one can insert every other key into a B-Tree in order to reduce the size of the index. This results in a B-Tree with an error bound of one: any location given by the B-Tree can be off by at most one position.

3 LEARNED INDEX STRUCTURES

In this work, we evaluate the performance of three different learned index structures: recursive model indexes (RMI), radix spline indexes (RS), and piecewise geometric model indexes (PGM). We do not compare with a number of other learned index structures [10, 13, 23] because tuned implementations could not be made publicly available. While all three of these techniques approximate the CDF of the underlying data, the way these approximations are constructed vary. We next give a high-level overview of each technique, followed by a discussion of their differences.

3.1 Recursive model indexes (RMI)

Originally presented by Kraska et al. [18], RMIs use a multi-stage model, combining simpler machine learning models together. For example, as depicted in Figure 3, an RMI with two stages, a linear stage and a cubic stage, would first use a linear model to make an initial prediction of the CDF for a specific key (stage 1). Then, based on that prediction, the RMI would select one of several cubic models to refine this initial prediction (stage 2).

Structure. When all keys can fit in memory, RMIs with more than two stages are almost never required [21]. Thus, here we explain only two-stage RMIs for simplicity. See [18] for a generalization to n stages. A two-stage RMI is a CDF approximator A trained on $|D|$ data points (key / index pairs). The RMI approximator A is composed of a single first stage model f_1, and B second-stage models f_2^i. The value B is referred to as the "branching factor" of the RMI.

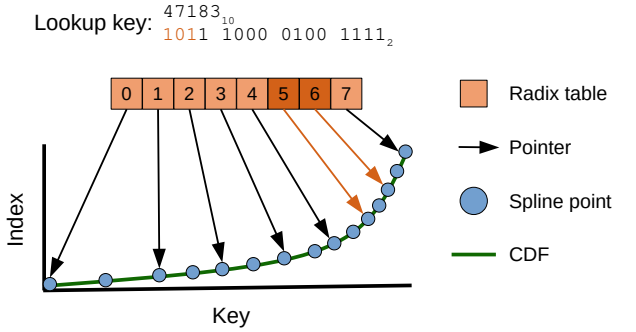


Figure 4: A radix spline index. A linear spline is used to approximate the CDF of the data. Prefixes of resulting spline points are indexed in a radix table to accelerate the search on the spline.

Formally, the RMI is defined as:

$$A(x) = f_2^{\lfloor B \times f_1(x)/|D| \rfloor}(x) \quad (1)$$

Intuitively, the RMI first uses the stage-one model $f_1(x)$ to compute a rough approximation of the CDF of the input key x. This coarse-grained approximation is then scaled between 0 and the branching factor B, and this scaled value is used to select a model from the second stage, $f_2^i(x)$. The selected second-stage model is used to produce the final approximation. The stage-one model $f_1(x)$ can be thought of as partitioning the data into B buckets, and each second-stage model $f_2^i(x)$ is responsible for approximating the CDF of only the keys that fall into the ith bucket.

Choosing the correct models for both stages (f_1 and f_2) and selecting the best branching factor for a particular dataset depends on the desired memory footprint of the RMI as well as the underlying data. In this work, we the CDFShop RMI auto-tuner [21].

Training. Let $(x, y) \in D$ be the set of key / index pairs in the underlying data. Then, an RMI is trained by adjusting the parameters contained in $f_1(x)$ and $f_2^i(x)$ to minimize:

$$\sum_{(x,y) \in D} (F(x) - y)^2 \quad (2)$$

Intuitively, minimizing Equation 2 is done by training "top down": first, the stage one model is trained, and then each stage 2 model is trained to fine-tune the prediction. Details can be found in [18].

3.2 Radix spline indexes (RS)

An RS index [17] consists of a linear spline [24] that approximates the CDF of the data and a radix table that indexes spline points (Figure 4). RS is built in a bottom-up fashion. Uniquely, RS can be built in a single pass with a constant worst-case cost per element (PGM provides a constant amortized cost).

Structure. As depicted in Figure 4, RS consists of a radix table and a set of spline points that define a linear spline over the CDF of the data. The radix table indexes r-bit prefixes of the spline points and serves as an approximate index over the spline points. Its purpose is to accelerate binary searches over the spline points. The radix table is represented as an array containing 2^r offsets into the sorted array of spline points. The spline points themselves are represented as key / index pairs. To locate a key in a spline segment, linear interpolation between the two spline points is used.

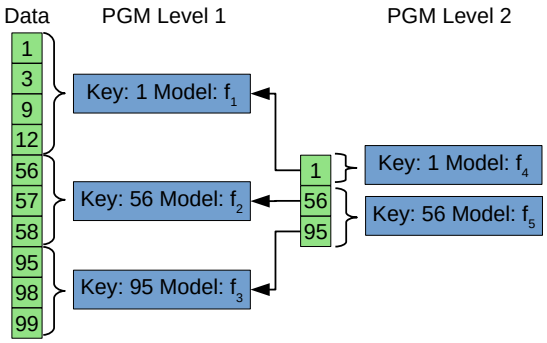


Figure 5: A piecewise geometric model (PGM) index.

Using the example in Figure 4, a lookup in RS works as follows: First, the r most significant bits b of the lookup key are extracted ($r = 3$ and $b = 101$). Then, the extracted bits b are used as an offset into the radix table to retrieve the offsets stored at the bth and the $b+1$th position (e.g., the 5th and the 6th position). Next, RS performs a binary search between the two offsets on the sorted array of spline points to locate the two spline points that encompass the lookup key. Once the relevant spline segment has been identified, it uses linear interpolation between the two spline points to estimate position of the lookup key in the underlying data.

Training. To build the spline layer, RS uses a one-pass spline fitting algorithm [24] that is similar to the shrinking cone algorithm of FITing-Tree [13]. The spline algorithm guarantees a user-defined error bound. At a high level, whenever the current error corridor exceeds the user-supplied bound, a new spline point is created. Whenever the spline algorithm encounters a new r-bit prefix, a new entry is inserted into the pre-allocated radix table.

RS has only two hyperparameters (spline error and number of radix bits), which makes it straightforward to tune. In practice, few configurations need to be tested to reach a desired performance / size tradeoff on a given dataset [17].

3.3 Piecewise geometric model indexes (PGM)

The PGM index is a multi-level structure, where each level represents an error-bounded piecewise linear regression [12]. An example PGM index is depicted in Figure 5. In the first level, the data is partitioned into three segments, each represented by a simple linear model (f_1, f_2, f_3). By construction, each of these linear models predicts the CDF of keys in their corresponding segments to within a preset error bound. The partition boundaries of this first level are then treated as their own sorted dataset, and another error-bounded piecewise linear regression is computed. This is repeated until the top level of the PGM is sufficiently small.

Structure. A piecewise linear regression partitions the data into $n+1$ segments with a set of points $p_0, p_1, \ldots, p_n$. The entire piecewise linear regression is expressed as a piecewise function:

$$F(x) = \begin{cases} a_0 \times x + b_0 & \text{if } x < p_0 \\ a_1 \times x + b_1 & \text{if } x \geq p_0 \text{ and } x < p_1 \\ a_2 \times x + b_2 & \text{if } x \geq p_1 \text{ and } x < p_2 \\ \ldots & \\ a_n \times x + b_n & \text{if } x \geq p_n \text{ and } x < p_n \end{cases}$$

Each regression in the PGM index is constructed with a fixed error bound ϵ. Such a regression can trivially be used as an approximate index. PGM indexes apply this trick recursively, first building an error-bounded piecewise regression model over the underlying data, then building another error-bounded piecewise regression model over the partitioning points of the first regression. Key lookups are performed by searching each index layer until the regression over the underlying data is reached.

Training. Each regression is constructed optimally, in the sense that the fewest pieces are used to achieve a preset maximum error. This can be done quickly using the approach of [31]. The first regression is performed on the underlying data, resulting in a set of split points (the boundaries of each piece of the regression) and regression coefficients. These split points are then treated as if they were a new dataset, and the process is repeated, resulting in fewer and fewer pieces at each level. Since each piecewise linear regression contains the fewest possible segments, the PGM index is optimal in the sense of piecewise linear models [12].

Intuitively, PGM indexes are constructed "bottom-up": first, an error bound is chosen, and then a minimal piecewise linear model is found that achieves that error bound. This process is repeated until the piecewise models become smaller than some threshold. The PGM index can also handle inserts, and can be adapted to a particular query workload. We do not evaluate either capability here.

3.4 Discussion

RMIs, RS indexes, and PGM indexes all approximate the CDF of the underlying data. However, the specifics vary.

Model types. While RS indexes and PGM indexes use only a single type of model (spline regression and piecewise linear regression, respectively), RMIs can use a wide variety of model types. This gives the RMI a greater degree of flexibility, but also increases the complexity of tuning the RMI. While both the PGM index and RS index can be tuned by adjusting just two knobs, automatically optimizing an RMI requires a more involved approach, such as [21]. Both the PGM index authors and the RS index authors mention integrating other model types as future work [12, 17].

Top-down vs. bottom-up. RMIs are trained "top down", first fitting the topmost model and training subsequent layers to correct errors. PGM and RS indexes are trained "bottom up", first fitting the bottommost layer to a fixed accuracy and then building subsequent layers to quickly search the bottommost layer for the appropriate model. Because both PGM and RS indexes require searching this bottommost layer (PGM may require searching several intermediate layers), they may require more branches or cache misses than an RMI. While an RMI uses its topmost model to directly index into the next layer, avoiding a search entirely, the bottommost layer of the RMI does not have a fixed error bound; any bottom-layer model could have a large maximum error.

RS indexes and PGM indexes also differ in how the bottommost layer is searched. PGM indexes decompose the problem recursively, essentially building a second PGM index on top of the bottommost layer. Thus, a PGM index may have many layers, each of which must be searched (within a fixed range) during inference. On the other hand, an RS index uses a radix table to narrow the search range, but there is no guarantee on the search range's size. If the

Method	Updates	Ordered	Type
PGM [12]	Yes	Yes	Learned
RS [17]	No	Yes	Learned
RMI [18]	No	Yes	Learned
BTree [6]	Yes	Yes	Tree
IBTree [14]	Yes	Yes	Tree
FAST [15]	No	Yes	Tree
ART [19]	Yes	Yes	Trie
FST [32]	Yes	Yes	Trie
Wormhole [30]	Yes	Yes	Hybrid hash/trie
CuckooMap [5]	Yes	No	Hash
RobinHash [2]	Yes	No	Hash
RBS	No	Yes	Lookup table
BS	No	Yes	Binary search

Table 1: Search techniques evaluated

radix table provides a comparable search range as the upper level of a PGM index, then an RS index locates the proper final model with a comparatively cheaper operation (a bitshift and an array lookup). If the radix table does not provide a narrow search range, significant time may be spent searching for the appropriate bottom-layer model.

4 EXPERIMENTS

Our experimental analysis is divided into six sections.

Setup (4.1): we describe the index structures, baselines, and datasets used. **Pareto analysis (4.2)**: we analyze the size and performance tradeoffs offered by each structure, including variations in dataset and key size. We find that learned index structures offer competitive performance. **Explanatory analysis (4.3)**: we analyze indexes via performance counters (e.g., cache misses) and other descriptive statistics. We find that no single metric can fully account for the performance of learned structures. **CPU interactions (4.4)**: we analyze how CPU cache and operator reordering impacts performance. We find that learned index structures benefit disproportionately from these effects. **Multithreading (4.5)**: we analyze the throughput of each index in a multithreaded environment. We find that learned structures have comparatively high throughput, possibly attributable to the fact that they incur fewer cache misses per lookup. **Build times (4.6)**: we analyze the time to build each index structure. We find that RMIs are slow to build compared to PGM and RS indexes, but that (unsurprisingly) no learned structure yet provides builds as fast as insert-optimized traditional index structures.

4.1 Setup

Experiments are conducted on a machine with 256 GB of RAM and an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz.

4.1.1 Indexes. In this section, we describe the index structures we evaluate, and how we tune their size/performance tradeoffs. Table 1 lists each technique and its capabilities.

Learned indexes. We compare RMIs, PGM indexes, and RadixSpline indexes (RS), each of which are described in Section 3. We use implementations tuned by each structure's original authors. RMI hyperparameters are tuned using CDFShop [21], an automatic RMI optimizer. RS and PGM are tuned by varying the error tolerance of the underlying models.

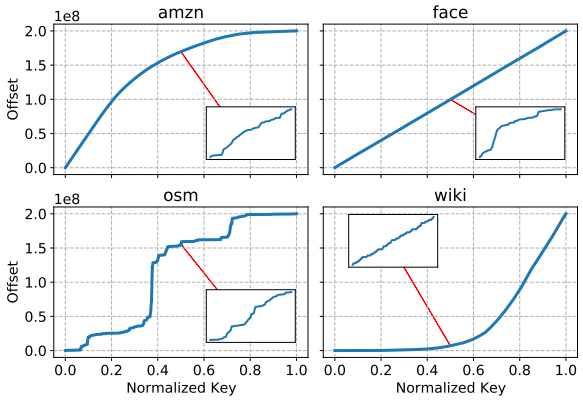


Figure 6: CDF plots of each testing dataset. The `face` dataset contains ≈ 100 large outlier keys, not plotted.

Tree structures. We compare with several tree-structured indexes: the STX B-Tree (BTree) [6], an interpolating BTree (IBTree) [14], the Adaptive Radix Trie (ART) [19], the Fast Architectural-Sensitive Tree (FAST) [15], Fast Succinct Trie (FST) [32], and Wormhole [30].

For each tree structure, we tune the size/performance tradeoff by inserting subsets of the data (Section 2.1). To build a tree of maximum size with perfect accuracy, we insert every key. To build a tree with a smaller size and decreased accuracy, we insert every other key. We note that this technique, while simple, may not be the ideal way to trade space for accuracy in each tree structure. Specifically, ART may admit a smarter method in which keys are retained or discarded based on the fill level of a node. We only evaluate the simple and universal technique of inserting fewer keys into each structure, and leave structure-specific optimizations to future work.

Hashing. While most hash tables do not support range queries, hash tables are still an interesting point of comparison due to their unmatched lookup performance. Unordered hash tables cannot be shrunk using the same technique as we use for trees.[1] Therefore, we only evaluate hash tables that contain every key. We evaluate a standard implementation of a Robinhood hash table (RobinHash) [2] and a SIMD-optimized Cuckoo map (CuckooMap) [5].

Baselines. We also include two naive baselines: binary search (BS), and a radix binary search (RBS). Radix binary search [16] stores only the radix table used by the learned RS approach. We vary the size of the radix table to achieve different size/performance tradeoffs.

4.1.2 Datasets. We use four real-world datasets for our evaluation. Each dataset consists of 200 million unsigned 64-bit integer keys. We generate random 8-byte payloads for each key. For each lookup, we compute the sum of these values.

- **amzn**: book popularity data from Amazon. Each key represents the popularity of a particular book.
- **face**: randomly sampled Facebook user IDs [29]. Each key uniquely identifies a user.
- **osm**: cell IDs from Open Street Map. Each key represents an embedded location.

[1] We evaluate Wormhole [30], a state-of-the-art ordered hashing approach.

- **wiki**: timestamps of edits from Wikipedia. Each key represents the time an edit was committed.

The CDFs of each of these datasets are plotted in Figure 6. The zoom window on each plot shows 100 keys. While the "zoomed out" plots appear smooth, each CDF function is much more complex, containing both structure and noise. We select 10M random lookup keys from each dataset. Indexes are required to return search bounds that contain the lower bound of each lookup key (see Section 2).

Why not test synthetic datasets? Synthetic datasets are often used to benchmark index structures [12, 18, 19]. However, synthetic datasets are problematic for evaluating learned index structures. Synthetic datasets are either (1) entirely random, in which case there is no possibility of learning an effective model of the underlying data (although a model may be able to overfit to the noise), or (2) drawn from a known distribution, in which case learning the distribution is trivial. Here, we focus only on datasets drawn from real world distributions, which we believe are the most important. For readers specifically interested in synthetic datasets, we refer to [16].

4.2 Pareto analysis

A primary concern of index structures is lookup performance: given a query, how quickly can the correct record be fetched? However, size is also important: with no limits, one could simply store a lookup table and retrieve the correct record with only a single cache miss. Such a lookup table would be prohibitively large in many cases, such as 64-bit keys. Thus, we consider *the performance / size tradeoff* provided by each index structure, plotted in Figure 7.

For each index structure, we selected ten configurations ranging from minimum to maximum size. While different applications may weigh performance and size differently, all applications almost surely desire a *Pareto optimal* index: an index for which no alternative has both a smaller size and improved performance. For the `amzn` and `wiki` datasets, learned structures are Pareto optimal up to a size of 100MB, at which point the RBS lookup table becomes effective. For `face`, learned structures are Pareto optimal throughout.

Poor performance on `osm`. Both traditional and learned index structures fail to outperform RBS on the `osm` dataset for nearly any size. The poor performance of learned index structures can be attributed to the `osm`'s dataset lack of local structure: even small pieces of the CDF exhibit difficult-to-model erratic behavior. This is an artifact of the technique used to project the Earth into one-dimensional space (a Hilbert curve). In Section 4.3, we confirm this intuition by analyzing the errors of the learned models; all three learned structures required significantly more storage to achieve errors comparable to those observed on the other datasets. Simply put, learned structures perform poorly on `osm` because `osm` is difficult to learn. Because `osm` is a one-dimensional projection of multi-dimensional data, a multi-dimensional learned index [23] may yield improvements.

Performance of PGM. In [12], the authors showed that "the PGM-index dominates RMI," contradicting our previous experience that the time spent on searches between the layers of the index outweighed the benefits of having a lower error. Indeed, in our experimental evaluation we found that the PGM index performs significantly worse than RMI on 3 out of the 4 datasets and slightly worse on `osm`. After contacting the authors of [12], we found that their RMI implementation was missing several key optimizations: their RMI only used linear models rather than tuning different type of models as proposed in [18, 21], and omitted some optimizations for RMIs with only linear models.[2] This highlights how implementation details can affect experimental results, and the importance of having a common benchmark with strong implementations. We stress that our results are the first to compare RMI and PGM implementations tuned by their respective authors.

Performance of RBS. Both RS and RBS exhibits substantially degraded performance on `face`. This is due to a small number (≈ 100) of outliers in the `face` dataset: most keys fall within $(0, 2^{50})$, but the outliers fall in $(2^{59}, 2^{64} - 1)$. These outliers cause the first 16 prefix bits of the radix table to be nearly useless. One could adjust RBS to handle this simple case (when all outliers are at one end of the dataset), but in general such large jumps in values represents a severe weakness of RBS. ART [19] can be viewed as a generalization of RBS to handle this type of skew. On other datasets, RBS is surprisingly competitive, often outperforming other indexes. This is partially explained by the low inference time required by RBS: getting a search bound requires only a bit shift and an array lookup. When the prefixes of keys are distributed uniformly, an RBS with a radix table of size 2^b provides equally accurate bounds as a binary search tree with b levels, but requires only a single cache miss.

Tree structures are non-monotonic. All tree structures tested (ART, BTree, IBTree, and FAST) become less effective after a certain size. For example, the largest ART index for the `amzn` data occupies nearly 1GB of space, but has worse lookup performance than an ART index occupying only 100MB of space. This is because, at a certain point, performing a binary search on a small densely-packed array becomes more efficient than traversing a tree. As a result, tree structures show non-monotonic behavior in Figure 7.

Indexes slower than binary search? At extremely small or large sizes, some index structures perform worse than binary search. In both cases, this is because some index structures are unable to provide sufficiently small search bounds to make up for the inference time required. For example, on the `osm` dataset, very small RMIs barely narrow down the search range at all. Because this small RMIs fit is so poor (analyzed later, Figure 12), the time required to execute the RMI model and produce the search bound is comparatively worse than executing a binary search on the entire dataset.

Structures for strings. Many recent works on index structures have focused on indexing keys of arbitrary length (e.g., strings) [30, 32]. We evaluated two structures designed for string keys – FST and Wormhole – in Figure 8. Unsurprisingly, neither performed as well as binary search. These indexes contain optimizations that assume comparing two keys is expensive, which is not the case when considering only integer keys. ART, an index designed for both string and integer data, does so by indexing one key-byte per radix tree level.

Hashing. Hashing provides $O(1)$ time point lookups. However, hashing generally does not support lower bound lookups, and hash tables generally have a large footprint, as they store every key. We evaluate

[2] We shared our RMI implementation with Ferragina and Vinciguerra before the publication of [12], but since [12] was already undergoing revision, they elected to continue with their own RMI implementation instead, without note. All PGM results in this paper are based on Ferragina and Vinciguerra's tuned PGM code as of May 18th, 2020.

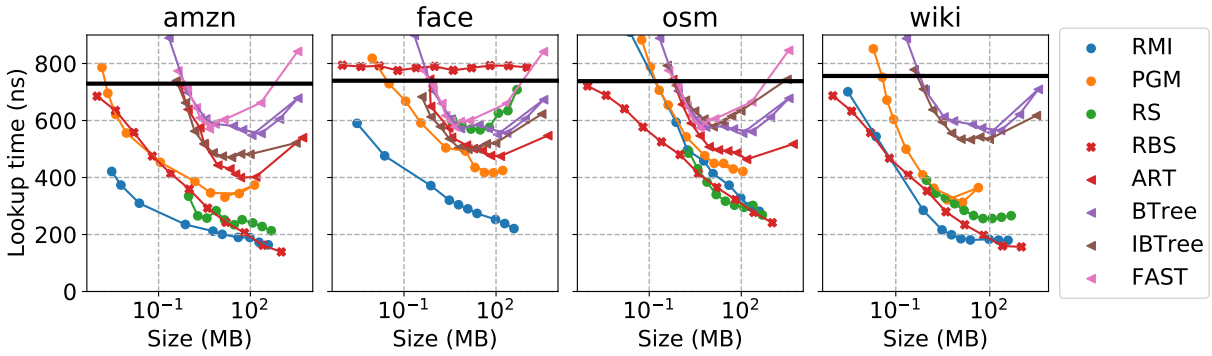


Figure 7: Performance and size tradeoffs for four different datasets. The black horizontal line represents the performance of binary search (which has a size of zero). Extended plots with all techniques are available here: https://rm.cab/lis1

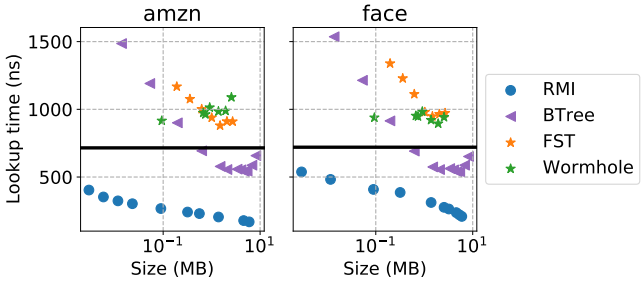


Figure 8: Performance of index structures built for strings (stars) on our integer datasets.

Method	Time	Size
PGM	326.48 ns	14.0 MB
RS	266.58 ns	4.0 MB
RMI	180.90 ns	48.0 MB
BTree	482.11 ns	166.0 MB
IBTree	446.55 ns	9.0 MB
FAST	435.33 ns	102.0 MB
BS	741.69 ns	0.0 MB
CuckooMap	114.50 ns	1541.0 MB
RobinHash	93.69 ns	6144.0 MB

Table 2: The fastest variant of each index structure compared against two hashing techniques on the `amzn` dataset.

two hashing techniques – a Cuckoo hash table [5] and a Robin-hood hash table [2]. We found that a load factor of 0.99 and 0.25 (respectively) maximized lookup performance.

Table 2 lists the size and lookup performance of the best-performing (and thus often largest) variant of each index structure and both hashing techniques for a 32-bit version[3] of the `amzn` dataset (results similar for others). Unsurprisingly, both hashing techniques offer superior point-lookup latency compared to traditional and learned index structures. This decreased latency comes at the cost of a larger in-memory footprint. For example, CuckooMap provides a 114ns lookup time compared to the 180ns provided by the RMI, but CuckooMap uses over 1GB of memory, whereas the RMI uses only 48MB. When range lookups and memory footprint are not concerns, hashing is a clear choice.

[3]The SIMD Cuckoo implementation only supports 32-bit keys.

4.2.1 Larger datasets. Figure 9 shows the performance / size tradeoff for each learned structure and a BTree for four different data sizes of the `amzn` dataset, ranging from 200M to 800M. All three learned structures are capable of scaling to larger dataset sizes, with only a logarithmic slowdown (as expected from the final binary search step). For example, consider an RMI that produces an average search bound that spans 128 keys, requiring 7 steps of binary search. If the dataset size doubles, an RMI of equal size is likely to return bounds that are twice as large: search bounds that span 256 keys. Such a bound requires only 8 total (1 additional) binary search steps. Thus, learned index structures scale to larger datasets in much the same way as BTrees. If larger datasets have more pronounced patterns, learned index structures may provide better scaling.

4.2.2 32-bit datasets. Here, we scale down the `amzn` dataset from 64 to 32 bits, and compare the performance of the three learned index structures, BTrees, and FAST. The results are plotted in Figure 10. For learned structures, the performance on 32-bit data is nearly identical to performance on 64-bit data. Our implementations of RS and RMI both transform query keys to 64-bit floats, so this is not surprising. We attempted to perform computations on 32-bit keys using 32-bit floats, but found that the decreased precision caused floating point errors. The PGM implementation uses 32-bit computations for 32-bit inputs, achieving modest performance gains.

For both tree structures, the switch from 64-bit to 32-bit keys allows twice as many keys to fit into a single cache line, improving performance. For FAST, which makes heavy use of AVX-512 operations, doubling the number of keys per cache line essentially doubles computational throughput as well, as each operator can work on 16 32-bit values simultaneously (as opposed to 8 64-bit values).

4.2.3 Search function. Normally, we use binary search to locate the correct key within the search bound provided by the index. However, other search techniques can be used. Figure 11 evaluates using binary, linear, and interpolation search for various index structures on `osm` and `amzn`. We observed that binary search (first column) was always faster than linear search (second column). This aligns with prior work that showed binary search being effective until the data size dropped below a very small threshold [28].

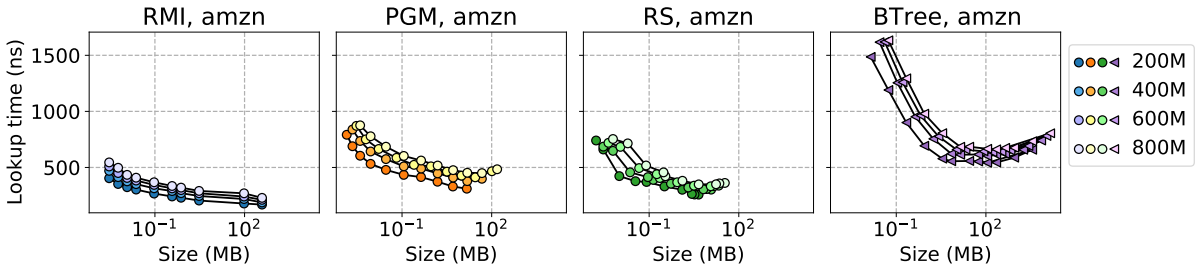


Figure 9: Performance / size tradeoffs for datasets of various sizes (200M, 400M, 600M, and 800M keys) for the `amzn` dataset. The `face` and `wiki` datasets were not sufficiently large to compare. Extended plots with all techniques and the `osm` dataset are available here: https://rm.cab/lis2

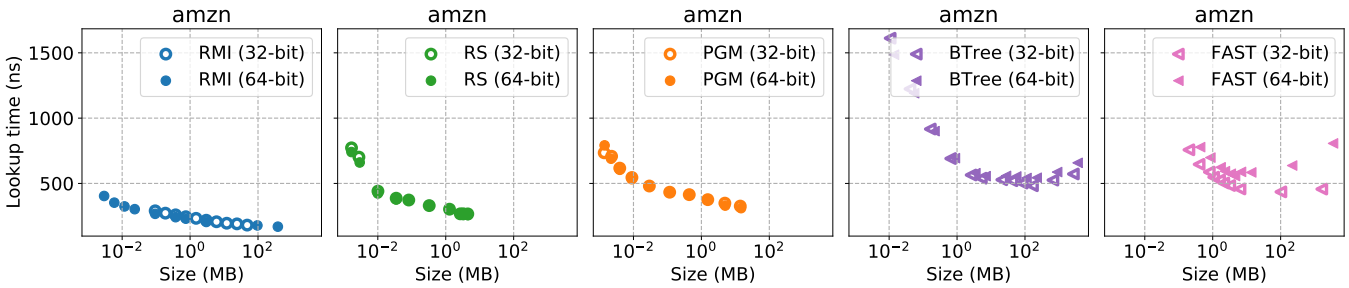


Figure 10: Performance / size tradeoff for 32 and 64 bit keys. While decreasing the key size to 32-bits has a minimal impact on learned structures, the ability to pack more values into a single cache line improves the performance of tree structures.

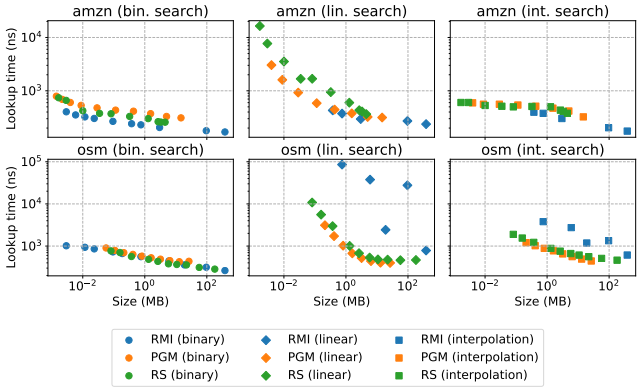


Figure 11: A comparison of "last mile" (Section 2) search techniques for the `osm` and `amzn` datasets.

Interpolation search (third column) behaves similarly to binary search on the `amzn` dataset, even offering improved performance on average ($\approx 2\%$). This was surprising, because interpolation search works by assuming that keys are uniformly distributed between two end points. If this were the case, one would expect a learned index to learn this distribution, subsuming any gains from interpolation search. However, because the learned structures have a limited size, there can be many segments of the underlying data that exhibit linear behavior that the learned structure does have the capacity to learn. For `osm`, which is relatively complex, interpolation search does not provide a benefit. This is unsurprising, since interpolation search works best on smooth datasets.

One could also integrate more complex interpolation search techniques, such as SIP [29]. One difficulty with incorporating SIP is the precomputation steps, which vary depending on the search bound used. Integrating an exponential search [8] technique could also be of interest, although it is not immediately clear how to integrate a search bound. We leave such investigations to future work.

4.3 Explaining the performance

In this section, we investigate *why* learned index structures have such strong performance and size properties. While prior work [18] attributed this to decreased branching and instruction count, we discovered that the whole story was more complex. None of model accuracy, model size (or "precision gain", the combination of the two in [18]), cache misses, instruction count, or branch misses can fully account for learned index structures' performance.

Figure 12 shows the correlation between lookup time and various performance characteristics of different index structures for the `amzn` and `osm` datasets. The first column shows the in-memory size of each model, the second column shows average $\log_2$ search bound size (i.e., expected binary search steps required), the third column shows last-level cache misses, the fourth column shows branch mispredictions, and the fifth column shows instruction counts. One can visually dismiss any single metric as explanatory: any vertical line corresponds to structures that are equal on the given metric, but exhibit different lookup times. For example, at a size of 1MB, RMIs

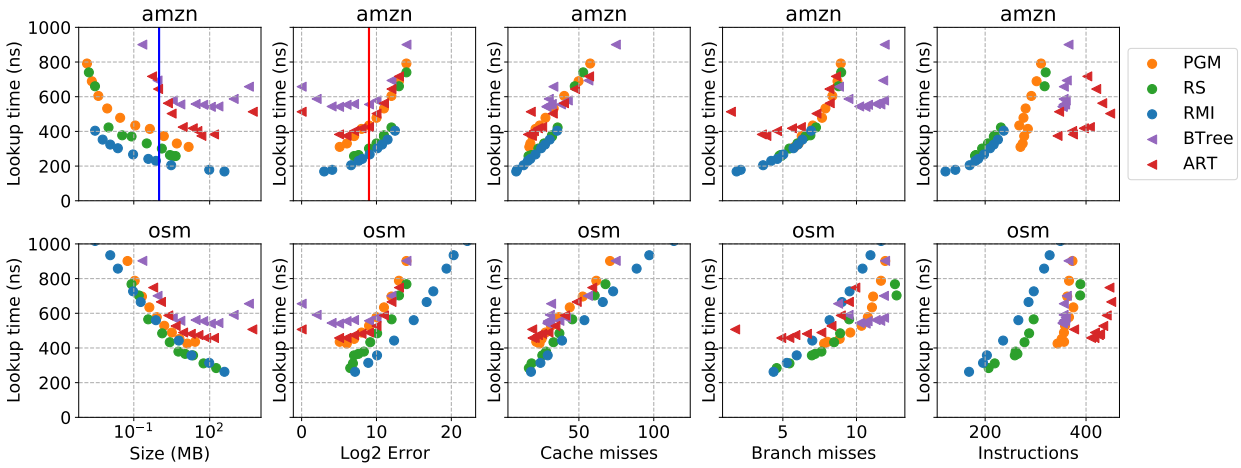


Figure 12: Various metrics compared with lookup times across index structures and datasets. No single metric can fully explain the performance of different index structures, suggesting a multi-metric analysis is required. Extended plots for all techniques and datasets are available here: https://rm.cab/lis5

achieve a latency of 220ns on `amzn`, but a BTree with the same size achieves a latency of 650ns (blue vertical line).

The second column ("$\log_2$ error"), is especially interesting. Learned indexes must balance inference time with model error [21]. For example, with a $\log_2$ error of 7, an RMI achieves a lookup time of 250ns on the `amzn` dataset, but the PGM index with the same $\log_2$ error achieves a latency of 480ns (red vertical line). This is attributable to the higher inference time of the PGM index. Of course, other factors, such as overall model size, must be taken into account.

Analysis. In order to test each potential explanatory factor, we performed a linear regression analysis using every index structure on all four datasets at 200 million 64-bit keys. The results indicated that *cache misses, branch misses, and instruction count had a statistically significant effect on lookup time* ($p < 0.001$), whereas size and $\log_2$ error did not ($p > 0.15$). To be clear, this means that *given the branch misses, cache misses, and instruction counts*, the size and $\log_2$ error do not significantly affect performance. This does not mean that the $\log_2$ error and size do not have an impact on cache misses; just that the relevant variation in lookup time explained by model size and $\log_2$ error is accounted for fully in the other measures.

Overall, a regression on cache misses, branch misses, and instruction count explained 95% of the variance ($R^2 = 0.955$). This means that 95% of the variation we observed in our experiments can be explained by a linear relationship between cache misses, branch misses, instructions, and lookup latency. The standardized regression coefficients for cache misses, branch misses, and instruction misses were 0.85, −0.28, and 0.50, respectively. Standardized regression coefficients can be interpreted as the number of standard deviations that a particular measure needs to increase by, assuming the other measures stay fixed, in order to increase the output by one standard deviation; in other words, these coefficients are descriptive of the variations within our measurements, not of the actual hardware impact of the metrics (although these are obviously related).

Interpretation: branch misses. While the magnitude of standardized coefficients are not useful on their own, their sign can provide insights. Surprisingly, the coefficient on branch misses is negative. This does not mean that an increased number of branch misses leads to increased model performance. Instead, the negative coefficient means that *for a fixed number of cache misses and instructions*, the tested indexes that incurred more branch misses performed better. In other words, indexes are getting significant value from branch misses; when an index incurs a branch miss, it does so in such a way that reduces lookup time more than an hypothetical alternative index that uses the same number of instructions and misses.

We offer two possible explanations. First, structures may be over-optimized to avoid branching, trading additional cache misses or instructions to reduce branching. Second, indexes that experience more branch misses may benefit from speculative loads on modern hardware. We leave further investigation to future work.

Interpretation: what metrics matter? If there is a single metric that explains the performance of learned index structures, we were unable to find it. Regression analysis suggests that cache misses, branch misses, and instruction counts are all significant, and account for model size and $\log_2$ error. Of the significant measures, cache misses had the largest explanatory power. This is consistent with indexes being latency-bound (limited by round-trip time to RAM).

The vast majority of cache misses for RMIs happen during the last-mile search. Two-layer RMIs require at most two cache misses for inference (potentially only one if the RMI's top layer is small enough). On the other hand, for a full BTree, no cache misses happen during the final search at all, but BTrees generally require at least one cache miss per level of the tree. Cache misses also help explain performance differences between RMI and PGM: since each additional PGM layer likely requires a cache miss at inference time, a large RMI with low $\log_2$ error will incur fewer cache misses than a large PGM index with a similar $\log_2$ error (e.g., `amzn` in Figure 12). When an RMI is not able to achieve a low $\log_2$ error, this advantage vanishes, as more cache misses are required during the last-mile search (e.g., `osm` in Figure 12).

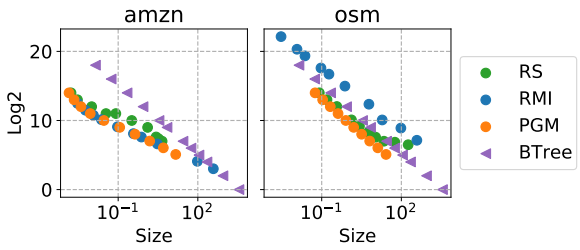


Figure 13: Size and $\log_2$ error bound of various index structures. When evaluated as a compression technique, learned index structures can be evaluated purely based on their size and $\log_2$ error. Extended plots are available here: https://rm.cab/lis7

Current implementations of learned index structures seem to prioritize fast inference time over $\log_2$ error. This makes sense, since a linear increase in $\log_2$ error only leads to a logarithmic increase in lookup time (due to binary search). However, our analysis suggests that a learned index structure could use significantly more cache misses if it could accurately pinpoint the cache line containing the lookup key. We experimented with multi-stage RMIs ($>$ 10 levels), but were unable to achieve such an accuracy. This could be an interesting direction for future work.

We encourage future development of index structures to take into account cache misses, branch misses, and instruction counts. Since all three of these metrics have a statistically significant impact on performance, ignoring one or two of them in favor of the other may lead to poor results. While we cannot suggest a single metric for evaluating index structures, if one must select a single metric, our analysis suggests that cache misses are the most significant.

Learned indexes as compression. A common view of learned index structures is to think of learned indexes as a lossy compression of the CDF function [12, 18]. In this view, the goal of a learned index is similar to lossy image compression (like JPG): come up with a representation that is smaller than the CDF with minimal information loss. The quality of a learned index can thus be judged by just two metrics: the size of the structure, and the $\log_2$ error (information loss). Figure 13 plots these two metrics for the three learned index structures and BTrees. These plots indicate that the information theoretic view, while useful, is not fully predictive of index performance. For example, for `face`, all three structures have very similar size and $\log_2$ errors after 1MB. However, some structures are substantially faster than others at a fixed size (Figure 7).

We encourage researchers and practitioners to familiarize themselves with the information theoretic view of learned index structures, but we caution against ending analysis at this stage. For example, an index structure that achieves optimal compression (i.e., an optimal size to $\log_2$ error ratio) is *not necessarily* going to outperform an index with suboptimal compression. The simplest way this could occur is because of inference time: if the index structure with superior compression takes a long time to produce a search bound, an index structure that quickly generates less accurate search bounds may be superior. However, if one assumes that storage mediums are arbitrarily slow (i.e., search time is strictly dominated by the size of search bound), then there is merit in viewing learned index structures as a pure compression problem, and investigating more advanced compression techniques for these structures [12] could be fruitful.

4.4 CPU interactions

Many prior works on both learned and non-learned index structures (including those by authors of this work) have evaluated their index structures by repeatedly performing lookups in a tight loop. While convenient and applicable to many applications, this experimental setup may exaggerate the performance of some index structures due, in part, to *caching* and *operator reordering*.

4.4.1 Caching. Executing index lookups in a tight loop, as it is often done to evaluate an index structure, will cause nearly all of the CPU cache to be filled with the index structure and underlying data. Since accessing a cached value is significantly faster (10s of nanoseconds) than accessing an uncached value ($\approx$ 100 nanoseconds), this may cause such tight-loop experiments to exaggerate the performance of an index structure.

The amount of data that will remain cached from one index lookup to another is clearly application dependent. In Figure 14, we investigate the effects of caching by evaluating the two possible extremes: the datapoints labeled "warm" correspond to a tight loop in which large portions of the index structure and underlying data can be cached between lookups. The datapoints labeled "cold" correspond to the same workload, but with additionally fully flushing the cache after each lookup. The gain from a warm cache for all five index structures ranges from 2x to 2.5x. With small index sizes ($<$ 1MB), the cold-cache variant of several learned index structures outperform the warm-cache BTree. With larger (and arguably more realistic) index structure sizes, obviously whether or not the cache is warm or cold is more important than the choice of index structure. Regardless of if the cache is warm or cold, we found that learned approaches exhibited dominant performance / size tradeoffs.

4.4.2 Memory fences. Modern CPUs and compilers may reorder instructions to overlap computation and memory access or otherwise improve pipelining. For example, consider a simple program that loads x, does a computation $f(x)$, loads y, and then does a computation $g(y)$. Assuming the load of y does not depend on x, a load of y may be reordered to occur before the computation of $f(x)$, so that the latency from loading y can be hidden within the computation of $f(x)$. When considering index structures, lookups placed in a tight loop may cause the CPU or compiler to overlap the final computation of one query with the initial memory read of the next query. In some applications, this may be realistic and desirable – in other applications, expensive computations between index lookups may prevent such overlapping. Thus, some indexes may disproportionately benefit from this reordering.

To test the impact of reordering on lookup time, we inserted a memory fence instruction into our experimental loop. Figure 15 shows that RMI and RS – two of the most competitive index structures – have the largest drop in performance when a memory fence is introduced($\approx$ 50% slowdown). The BTree, FAST and PGM are almost entirely unaffected. While the inclusion of a memory fence harms the performance of RMI and RS, learned structures still provide a better performance / size tradeoff for the `amzn` dataset (results for other datasets are similar, but omitted due to space constraints).

The impact of a memory fence was highly correlated with the number of instructions used by an index structure (Figure 12): indexes using fewer instructions, like RMI and RS, were impacted to a

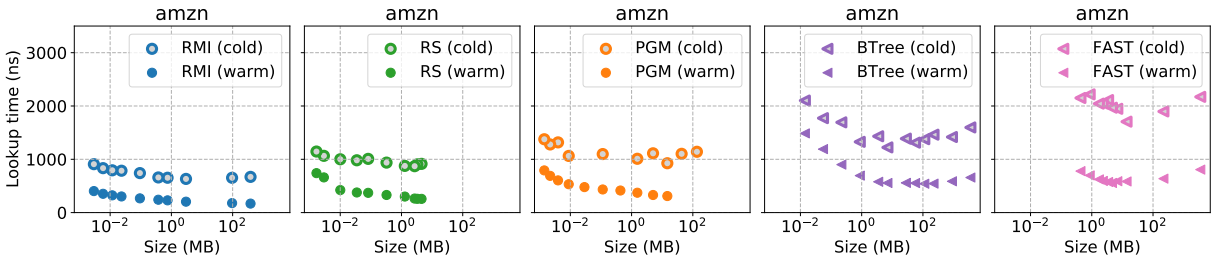


Figure 14: The performance impact of having a cold cache for various index structures. Extended plots with all techniques are available here: https://rm.cab/lis3

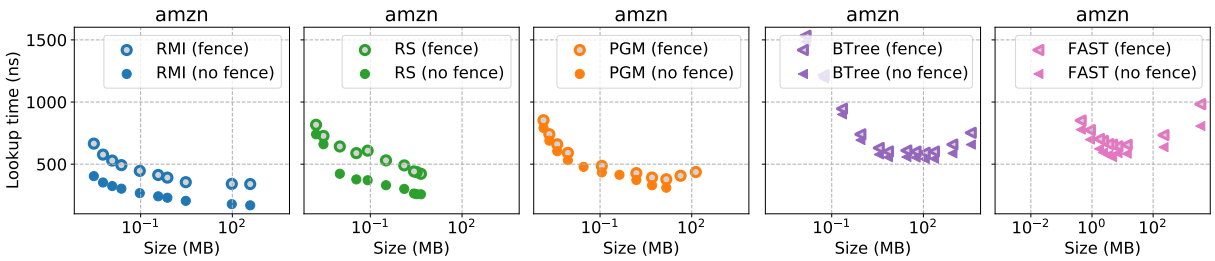


Figure 15: Performance of various index structures with and without a memory fence. Without the fence, the CPU may reorder instructions and overlap computation between lookups. With the fence, each lookup must be completed before the next lookup begins. Extended plots with all techniques and datasets are available here: https://rm.cab/lis4

greater extent than structures using more instructions. Since reordering optimizations often examine only a small window of instructions (i.e., "peephole optimizations" [22]), reordering optimizations may be more effective when instruction counts are lower.

We recommend that future researchers test index structures with memory fences to determine the benefit their structure gets from reordering. Getting a lot of benefit from reordering is not necessarily bad; plenty of applications require performing index lookups in a tight loop, with only minimal computation being performed on each result. Ideally, researchers should evaluate their index structures within a specific application, although this is much more difficult.

4.5 Multithreading

Here, we evaluate how various index structures scale when queried by concurrent threads. Our test CPU had 20 physical cores (40 with hyperthreading). Since multithreading strictly increases latency, here we evaluate throughput (lookups per second).

Varying thread count. We first vary the number of threads, fixing the model size at 50MB except for RobinHash, which is still the full size. The results are plotted in Figure 16a, with and without a memory fence. Overall, all three learned index variants scale with an increasing number of threads, although only the RMI achieved higher throughput than the RBS lookup table in this experiment.

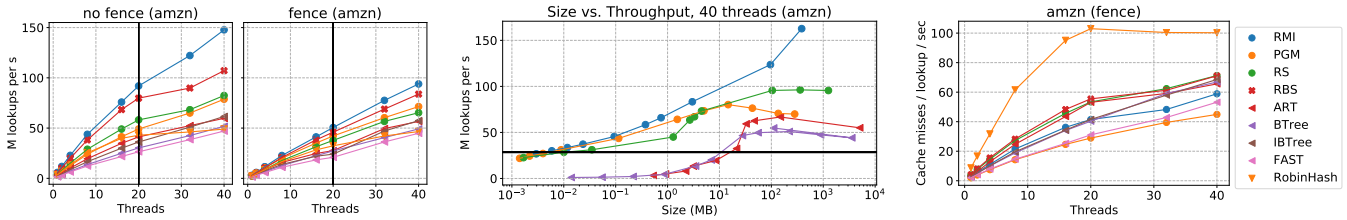
RobinHash, the technique with the lowest latency with a single thread, does not achieve the highest throughput in a concurrent environment.[4] We do not consider hash tables optimized for concurrent environments [27]; here we only demonstrate that an off-the-shelf hash table with a load factor optimized for single-threaded lookups does not scale seamlessly.

To help explain why certain indexes scaled better than others, we measured the number of cache misses incurred *per second* by each structure, plotted in Figure 16c. If a index structure incurs more cache misses per second, then the benefits of multithreading will be diminished, since threads will be latency bound waiting for access to RAM. Indeed, RobinHash incurs a much larger number of cache misses per second than any other technique. The larger size of the hash table may contribute to this, as fewer cache lines may be shared in between lookups compared with a smaller index.

PGM and FAST have the fewest cache misses per second at 40 threads, suggesting that PGM and FAST may benefit the most from multithreading. To investigate this, we tabulated the *relative* speedup factor of each technique. Due to space constraints, the plot is available online: https://rm.cab/lis8. FAST has the highest relative speedup, achieving 32x throughput with 40 threads. In addition to having few cache misses per second, FAST also takes advantage of streaming AVX-512 instructions, which allows for effective overlap of computation with memory reads. PGM, despite having the least cache misses per second, achieved only a 27x speedup at 40 threads. On the other hand, RobinHash had by far the most cache misses per second and the lowest relative speedup at 40 threads (20x). Thus, cache misses per second correlate with, but do not always determine, the speedup factor of an index structure.

Varying index size. Next, we fix the number of threads at 40, and vary the size of the index. Results are plotted in Figure 16b. One might expect smaller structures to have better throughput because of caching effects; we did not find this to be the case. In general, larger indexes had higher throughput than smaller ones. One possible explanation of this behavior is that smaller models, while more likely

[4]The SIMD Cuckoo implementation [5] only supports 32-bit keys, and was not included in this experiment.



(a) Multithreaded throughput for `amzn`, models have a fixed size of 50MB. No memory fence (left) and with memory fence (right).

(b) Model size vs. 40-thread throughput for the `amzn` dataset. An extended plot with all index techniques is available here: https://rm.cab/lis6

(c) Cache misses per lookup per second. Higher values indicates that speedup from multithreading may be negatively impacted.

Figure 16: Multithreading results

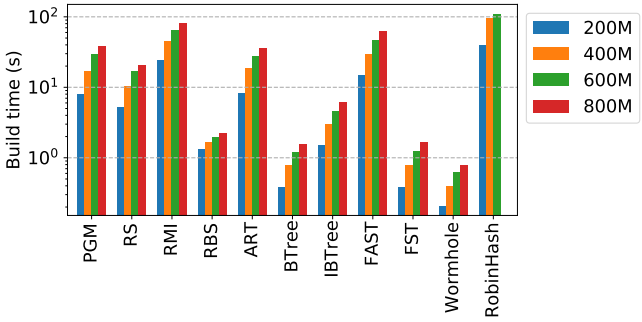


Figure 17: Build times for the fastest (in terms of query time) variant of each index type for the `amzn` dataset at four different data sizes. Note the log scale.

to remain cached, produce larger search bounds, which cause more cache misses during the last mile search.

PGM, BTree, RS, and ART indexes suffered decreased throughput at large model sizes. This suggests that the cache misses incurred from the larger model sizes are not enough to make up for the refinement in the search bound. The RMI did not suffer such a regression, possibly because each RMI inference requires at most two cache misses (one for each model level), whereas for other indexes the number of cache misses per inference could be higher.

4.6 Build times

Figure 17 shows the single-threaded build time required for the fastest (in terms of lookup time) variants of each index structure on `amzn` at different dataset sizes. We do not include the time required to tune each structure (automatically via CDFShop [21] for RMIs, manually for other structures). We note that automatically tuning an RMI may take several minutes. Unsurprisingly, BTrees, FST, and Wormhole provide the fastest build times, as these structures were designed to support fast updates.[5] Of the non-learned indexes, FAST and RobinHash have the longest build times. Maximizing the performance of Robinhood hashing requires using a high load factor (to keep the structure compact), which induces a high number of swaps. We note that many variants of Robinhood hashing support parallel operations, and thus lower build times.

[5]In particular, Wormhole and PGM can handle parallel inserts and builds respectively, which we do not evaluate here.

For the largest dataset, the build times for the fastest variants of RMI, PGM, and RS were 80 seconds, 38 seconds, and 20 seconds respectively. Of the learned index structures, RS consistently provides the fastest build times regardless of dataset size. This is explained by the fact that an RS index can be built in a single pass over the data with constant time per element [17]. In contrast, while a PGM index could theoretically be built in a single pass, the tested implementation of the PGM index builds the initial layer of the index in a single pass, and builds subsequent layers in a single pass over the previous layer (each logarithmically smaller). RMIs require one full pass over the underlying data per layer. In our experiments, no learned index takes advantage of parallelism during construction, which could provide a speedup.

5 CONCLUSION AND FUTURE WORK

In this work, we present an open source benchmark that includes several tuned implementations of learned and traditional index structures, as well as several real-world datasets. Our experiments on read-only in-memory workloads searching over dense arrays showed that learned structures provided Pareto dominant performance / size behavior. This dominance, while sometimes diminished, persists even when varying dataset sizes, key sizes, memory fences, cold caches, and multi-threading. We demonstrate that the performance of learned index structures is not attributable to any specific metric, although cache misses played the largest explanatory role. In our experiments, learned structures generally had higher build times than insert-optimized traditional structures like BTrees. Amongst learned structures, we found that RMIs provided the strongest performance / size but the longest build times, whereas both RS and PGM indexes could be constructed faster but had slightly slower lookup times.

In the future, we plan to examine the end-to-end impact of learned index structures on real applications. Opportunities to combine a simple radix table with an RMI structure (or vice versa) are also worth investigating. As more learned index structures begin to support updates [10, 12, 13], a benchmark against traditional indexes (which are often optimized for updates) could be fruitful.

ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, NSF IIS 1900933, and DARPA Award 16-43-D3M-FP040.

REFERENCES

[1] C++ lower_bound, http://cplusplus.com/reference/algorithm/lower_bound/.

[2] RobinMap, https://github.com/Tessil/robin-map.

[3] RocksDB, https://rocksdb.org/.

[4] Searching on sorted data benchmark, https://learned.systems/sosd.

[5] SIMD Cuckoo Hash, https://github.com/stanford-futuredata/index-baselines.

[6] STX B+ Tree, https://panthema.net/2007/stx-btree/.

[7] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment*, 4(8):470–481, May 2011.

[8] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, Aug. 1976.

[9] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. Association for Computing Machinery.

[10] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. Lomet. ALEX: An Updatable Adaptive Learned Index. *arXiv:1905.08898 [cs]*, May 2019.

[11] P. Ferragina and G. Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data*, volume 896 of *Studies in Computational Intelligence*. Springer, 2020.

[12] P. Ferragina and G. Vinciguerra. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, Apr. 2020.

[13] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, 2019. ACM.

[14] G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, DaMoN '06, Chicago, Illinois, June 2006. Association for Computing Machinery.

[15] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, 2010.

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. In *ML for Systems at NeurIPS*, MLForSystems @ NeurIPS '19, Dec. 2019.

[17] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: A single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM @ SIGMOD '20, pages 1–5, Portland, Oregon, June 2020. Association for Computing Machinery.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM.

[19] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 38–49, USA, 2013. IEEE Computer Society.

[20] C. Luo and M. J. Carey. LSM-based storage techniques: A survey. *PVLDB*, 29(1):393–418, Jan. 2020.

[21] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, Portland, OR, June 2020.

[22] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[23] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning Multi-dimensional Indexing. In *ML for Systems at NeurIPS*, MLForSystems @ NeurIPS '19, Dec. 2019.

[24] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases*, BNCOD '08, pages 126–138, 2008.

[25] Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia. Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes (blog post), https://dawn.cs.stanford.edu/2018/01/11/index-baselines/, 2018.

[26] Peter Boncz and Thomas Neumann. The Case for B-Tree Index Structures (blog post), http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html, 2017.

[27] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment*, 9(3):96–107, Nov. 2015.

[28] L.-C. Schulz, D. Broneske, and G. Saake. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proceedings of the VLDB Endowment*, 11(11):1550–1562, July 2018.

[29] P. Van Sandt, Y. Chronis, and J. M. Patel. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search? In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 36–53, New York, NY, USA, 2019. ACM.

[30] X. Wu, F. Ni, and S. Jiang. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–16, Dresden, Germany, Mar. 2019. Association for Computing Machinery.

[31] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *The VLDB Journal*, 23(6):915–937, Dec. 2014.

[32] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 323–336, Houston, TX, USA, May 2018. Association for Computing Machinery.