Hist-Tree: Those Who Ignore It Are Doomed to Learn

Andrew Crotty
Brown University
crottyan@cs.brown.edu

ABSTRACT

Learned indexes have provided a new perspective on the well-studied problem of indexing. Despite early skepticism, recent results have shown significant improvements over traditional data structures. While some have attributed these improvements to an ability to adapt to patterns in the data, we believe that the main advantage of learned indexes comes instead from implicit assumptions (e.g., data sortedness) that are not fully leveraged by the comparison points.

In this paper, we show that, simply by incorporating these same basic assumptions, we can design a traditional data structure capable of outperforming learned alternatives. To make our case, we propose a new index called a *histogram tree* (Hist-Tree), which takes advantage of the sortedness and range of the data. We also present an optimized compact layout based on the read-only assumption made by many learned indexes. Our experiments demonstrate that, in terms of lookup latency, Hist-Tree can beat three state-of-the-art learned indexes, including RMI, PGM-index, and RadixSpline, by as much as 1.8–2.7×.

1. INTRODUCTION

Learned indexes are based on the simple premise that all indexes can be viewed as models. This new way of looking at the well-studied problem of indexing has created significant buzz and sparked an entire line of follow-up work [8, 4, 7, 16, 11, 5], as well as investigations into other learned system components [12]. While early critiques [17, 2] attempted to demonstrate that learned indexes could not outperform carefully tuned versions of traditional data structures, recent results [15] definitively show that learned approaches can beat many well-known indexes (e.g., B-trees, ART [14]).

However, we believe that all of these studies unknowingly placed traditional indexes at an unfair disadvantage. Specifically, learned indexes make several implicit assumptions that are not leveraged by any of the comparison points. For example, many learned indexes assume sorted data, whereas off-the-shelf B-trees make no such assumption; that is, since B-trees can also handle unsorted data, they will incur unnecessary overhead (e.g., redundant pointers) when the data is actually sorted.

Other assumptions commonly made by learned indexes include the minimum and maximum data values (i.e., the range) and that the data will never be updated (i.e., a read-only workload), both of which provide significant advantages over traditional data structures that do not make these assumptions. In fact, we posit that the reported improvements over traditional indexes come almost entirely from leveraging these implicit assumptions rather than any unique ability to learn and adapt to patterns in the data.

The goal of this paper is to show that a traditional data structure can actually outperform a learned index simply by taking advantage of these same basic assumptions. To make our case, we developed a new index called a *histogram tree* (Hist-Tree) that leverages two very simple properties of the data: (1) sortedness and (2) the range. Additionally, since many state-of-the-art learned indexes (e.g., RMI [13], RadixSpline [11]) assume a read-only workload, we also present a compact Hist-Tree layout optimized for this property. Our experiments show that Hist-Tree can achieve lookup latency reductions of up to 1.8–2.7× over three state-of-the-art learned indexes, including RMI, PGM-index, and RadixSpline.

2. BACKGROUND

In the following, we provide the relevant background on learned indexes, including a discussion of their precise definition, specification of the problem addressed in this work, and summary of existing approaches.

2.1 Learned Index Definition

What exactly qualifies as a learned index? The original paper [13] argues that all indexes can be viewed as models. While technically accurate, this definition is overly broad and does not reflect common usage, where invoking the term "learned" usually implies some form of black-box ML at work. Yet, simply declaring that learned indexes incorporate ML techniques is similarly vague and unhelpful.

One widely touted feature of learned indexes is their ability to adapt to patterns, such as by partitioning the data into (approximately) linear segments [8, 4, 7]. However, traditional indexes also adapt to patterns in much the same way. For example, B-trees will end up using a disproportionate number of nodes to cover denser regions of the data, effectively making them sensitive to the underlying distribution. Adaptivity alone, then, is not a sufficient criterion for classifying a data structure as learned.

Instead, we conclude that the defining characteristic of learned indexes is the use of some form of explicit curve fitting to model data distributions. More precisely, learned indexes attempt to model a curve (i.e., function) that maps an input key to an associated value, such as a position in a sorted array, and then materialize the result as a concrete data structure. For instance, the previously mentioned approaches that identify linear segments might store the slope and intercept of each segment in the node of a tree.

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

Index	CDF	Updatable	Open-Source
RMI [13]	Multiple		✓
FITing-Tree [8]	PLA	✓	
ALEX [4]	PLA	✓	✓
PGM-index [7]	PLA	✓	✓
RadixSpline [11]	Spline Fitting		✓

Table 1: Summary of existing learned indexes.

2.2 Problem Specification

In a general sense, a learned index can replace any traditional data structure. This paper specifically explores the problem of search on sorted data, which is the focus of most existing learned index work [13, 8, 4, 7, 11], and we leave other topics (e.g., hashing, set membership, multidimensional indexing [16, 5]) for future investigation.

The task involves locating a value in a sorted array given a lookup key, where we must return the position corresponding to the first occurrence of a key equal to (or greater than) the specified key. Stated more formally, the goal is to identify the position i of the greatest lower bound (i.e., infimum) of an ordered subset S of a dataset D, such that all elements of S are greater than or equal to the lookup key k.

Learned indexes tackle this search problem by approximating the CDF of D so that evaluating the CDF at k will yield an estimate of i. Existing approaches employ different methods for modeling the CDF, which we discuss next.

2.3 Existing Approaches

Table 1 summarizes (in order of first publication) existing learned indexes that tackle the problem of search on sorted data. For each learned index, we note: (1) the curve fitting techniques used to model the CDF; (2) support for updates; and (3) availability of an open-source implementation.

The original learned index paper [13] proposed the recursive model index (RMI). The key idea is to organize multiple models into a hierarchical structure, where each model selects another model in the subsequent layer until reaching a leaf that predicts the final position. Since then, several formal [10, 15, 6] and informal [17, 2, 18] benchmarking studies have compared the performance of learned indexes to traditional data structures.

To address the read-only limitation of RMI, subsequent learned indexes, including FITing-Tree [8], ALEX [4], and PGM-index [7], added support for updates. All of these approaches use some variation of piecewise linear approximation (PLA), modeling a data distribution as a series of approximately linear segments.

The recently proposed RadixSpline [11] takes essentially the same approach, fitting a linear spline to the CDF. Spline knots are stored in an auxiliary radix table to accelerate searches over the segments. Like RMI, RadixSpline also assumes a read-only workload.

3. HIST-TREE

This section describes the design of Hist-Tree, our proposed traditional data structure that incorporates some of the common implicit assumptions made by learned indexes.

3.1 Overview

We begin with a high-level overview using the example that appears in Figure 1, which is based on a dataset of 200 normally distributed keys in the range $[0, 1000)$. Figure 1a shows the keys plotted against positions in the array, as well as histograms depicting the PDF and CDF of the keys.

The corresponding Hist-Tree for this dataset is shown in Figure 1b. As the name suggests, a Hist-Tree is a tree data structure where each node represents a histogram that partitions a data range into a fixed number of equal-width bins. For instance, the root node in the example partitions the full key range into four bins, with 20 keys falling into the first bin. Then, the child node of the first bin further partitions that subrange into four smaller bins. Bins with a child node are shown as white, whereas terminal bins (i.e., bins with a count below a chosen threshold) are shaded gray. In Figure 1b, for example, note that all terminal bins have counts less than the specified threshold of 16.

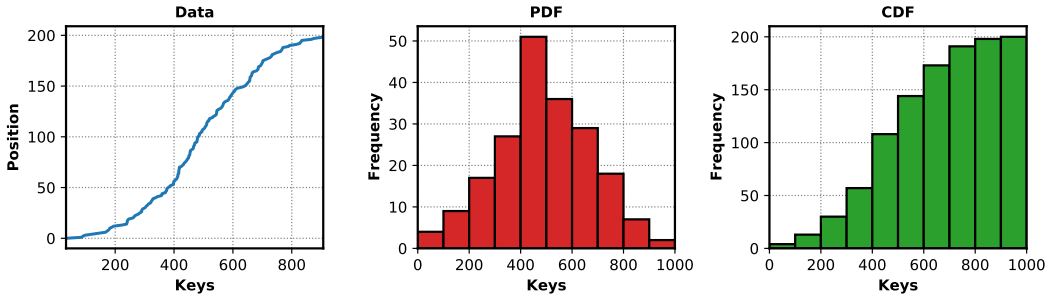
To perform a lookup, we simply descend the tree by calculating the appropriate bin at each node and following the corresponding pointer to the child node until reaching a terminal bin. Along the way, we maintain a running sum of the counts from all bins less than (i.e., to the left of) every calculated bin. The result is a range no larger than the chosen threshold, which is then supplied to any search algorithm (e.g., linear, binary) to locate the key's exact position.

Finally, Figure 1c shows a compact Hist-Tree layout that flattens the tree into an efficient lookup table for read-only workloads. In the figure, each row of the table represents a node from the Hist-Tree shown in Figure 1b. While the color scheme remains the same, notice that the values have changed: each white bin now stores the offset of the row representing its child node, and gray bins store the cumulative sum of all counts up to that bin. For example, row 0 represents the root node, storing offsets to each of its children (i.e., rows 1, 2, and 6) and the position of the smallest key that falls into the fourth bin (i.e., 187).

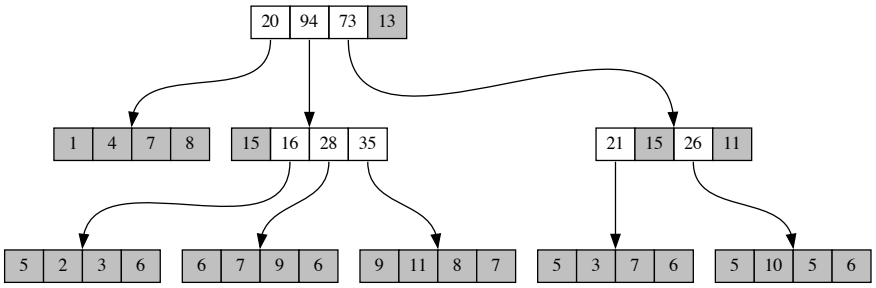
3.2 Physical Layout

While conceptually a tree data structure, the Hist-Tree nodes are physically organized into two arrays of 32-bit integers, which prevents fragmentation and reduces the overhead of child pointers. The first array contains inner nodes (i.e., nodes where at least one bin has a child), and the second contains leaf nodes (i.e., nodes with only terminal bins). For instance, the root node in Figure 1b would be stored in the inner node array as eight consecutive elements, which consist of four histogram bin counts followed by four child pointers. A flag set in the high-order bit of a child pointer denotes either a terminal bin or leaf node. In the example, we would set this flag for the first and fourth child pointers of the root node. On the other hand, the child of the first bin would consist of four elements in the leaf node array to store the histogram bin counts. Note that, although nodes in the example have only four bins for ease of illustration, using a larger number of bins in practice will limit the depth of the tree and usually improve lookup performance.

Additionally, we ensure that the range covered by each histogram is always a power of two, allowing us to use cheap bit shifts for bin calculation rather than more expensive division instructions. We must therefore round up the key range to the next largest power of two, which is 1024 in the example. While this rounding potentially leads to some wasted space in the root node, the overhead is negligible, since any bins that cover beyond the data range will remain as empty terminal bins.



(a) Example dataset



(b) Hist-Tree

0	1	2	6	187
1	0	1	5	12
2	20	3	4	5
3	35	40	42	45
4	51	57	64	73
5	79	88	99	107
6	7	135	8	176
7	114	119	122	129
8	150	155	165	170

(c) Compact Hist-Tree

Figure 1: An example dataset with the corresponding basic and compact Hist-Tree.

3.3 Lookup

A lookup involves descending a Hist-Tree to identify a bounded search range in the sorted array for the specified key. Figure 2 shows the pseudocode to perform a lookup of `key` in a given Hist-Tree `ht`.

We start with an initial check that returns immediately if `key` is outside the range covered by `ht` (lines 7–8). After initializing some variables (lines 11–14), the algorithm then proceeds by iteratively descending `ht` until reaching a terminal bin or leaf node, at which point we return `pos` (lines 24–25). We test the flag bit in the child pointer to decide whether to terminate the descent on the following iteration and assign the appropriate child (i.e., inner or leaf node) to `next` (lines 28–31). For each visited node, we compute `bin` by performing a logical right shift on `key` of `width` bits and add the counts from lower bins to `pos` (lines 19–21). Each iteration ends by updating `key` based on `bin` and reducing `width` accordingly (lines 34–35).

As a concrete example, looking up the key 567 in the Hist-Tree from Figure 1b would proceed as follows. At the root node, we first calculate the bin by dividing the key by the bin width ($567/256 = 2$). Then, we compute the sum of all smaller bins ($20+94 = 114$) before moving to the child node and updating the key ($567 - 256 * 2 = 55$). We again divide by the new bin width to choose the correct bin ($55/64 = 0$), update the running sum ($114 + 0 = 0$), and adjust the key ($55 - 64 * 0 = 55$). At the leaf node, we perform one last bin calculation ($55/16 = 3$) before returning the final sum of the counts ($114 + 5 + 3 + 7 = 129$).

As mentioned, the `lookup` function returns a bounded search range no larger than the set terminal bin threshold. In our experiments (Section 4), we vary this threshold from eight up to several thousand, which influences the choice of accompanying search algorithm.

```
1  #define BINS 4
2  #define FLAG (1 << 31)
3
4  size_t lookup(ht_t *ht, uint64_t key)
5  {
6      //ensure key is within range
7      if (key < ht->min) return 0;
8      if (key > ht->max) return ht->len;
9
10     //initialize variables
11     uint32_t next, *node = ht->node;
12     int width = ht->width, done = 0;
13     size_t i, bin, pos = 0;
14     key -= ht->min;
15
16     //descend tree
17     do {
18         //calculate bin and update running sum
19         bin = key >> width;
20         for (i = 0; i < bin; i++)
21             pos += node[i];
22
23         //return when done set
24         if (done)
25             return pos;
26
27         //set done, next, and node based on flag
28         done = node[BINS + bin] & FLAG;
29         next = node[BINS + bin] & ~FLAG;
30         node = done ? ht->leaf + next * BINS
31                     : ht->node + next * BINS * 2;
32
33         //adjust key and width
34         key -= bin << width;
35         width -= log2(BINS);
36     } while (1);
37 }
```

Figure 2: Hist-Tree `lookup` function

3.4 Insert & Delete

Both inserting and deleting a key are conceptually similar to performing a lookup. Thus, we can follow the same basic algorithm from Figure 2, with the added step of updating (i.e., incrementing or decrementing) the count of the calculated bin in each node.

Additionally, we must also ensure that the updated count satisfies the terminal bin threshold. For deletions, we simply remove the node if the count no longer exceeds the threshold, whereas insertions may require us to scan the covered subrange in order to build the histogram for a new child node. However, the subrange will only be as large as the terminal bin threshold, and we can even leverage the running sum to get the exact starting position. Finally, for both insertions and deletions, we may also need to switch nodes between the inner and leaf node arrays.

While updating a Hist-Tree is relatively efficient, we have not considered the overhead associated with updating the actual data. Since learned indexes require that the data must be stored in a densely packed sorted array, we would almost always need to shift array elements in order to insert or delete individual keys. As previously mentioned, many traditional indexes do not require sorted data, which makes updates much less expensive (e.g., new values can simply be appended to the end of the array).

3.5 Bulk Loading

Indexes are often constructed over an entire dataset rather than through individual insertions. Consequently, many traditional indexes like B-trees have efficient algorithms for bulk loading, and recent learned index work [11, 15] has also noted the importance of this functionality. Hist-Tree can similarly perform efficient bulk loading of data from a sorted array, which we describe in the following.

Based on the specified terminal bin threshold, we first calculate the maximum possible height of the tree by assuming a worst-case dense subrange and create one temporary node in each layer. Then, we construct the Hist-Tree in a single pass over the sorted array, updating the histogram bin counts of the node in the bottom layer. When we reach a key that falls outside the range covered by that node, we recursively add the counts to each node's parent and decide whether to keep the node based on the total count. If the total count is below the specified threshold, we mark the corresponding bin in the parent as terminal (e.g., the last bin of the root node in Figure 1b). Otherwise, we retain the temporary node for the final Hist-Tree.

3.6 Compact Layout

Although competitive, Hist-Tree still cannot fully match the performance of some learned indexes in all cases. As stated, we believe this performance difference comes primarily from the implicit assumption of a read-only workload, which provides a significant advantage over traditional indexes that can handle arbitrary updates after initial construction. We therefore present an optimized compact layout for Hist-Tree that offers much better lookup performance by leveraging the same read-only assumption.

Figure 1c shows the compact version of the Hist-Tree from Figure 1b. A compact Hist-Tree is essentially a lookup table, where bins from the Hist-Tree are mapped to an array of 32-bit integers via a depth-first pre-order traversal. Although each node is visually represented as a row in the figure, the physical layout is actually flattened, similar to the inner and leaf node arrays. Notice that white bins now store offsets to the rows representing child nodes, and terminal gray bins store the cumulative sum of all previous bins, again with a flag set in the high-order bit.

```
1  #define BINS 4
2  #define FLAG (1 << 31)
3
4  size_t lookup(cht_t *cht, uint64_t key)
5  {
6      //ensure key is within range
7      if (key < cht->min) return 0;
8      if (key > cht->max) return cht->len;
9
10     //initialize variables
11     uint32_t next, *ptr = cht->root;
12     int width = cht->width;
13     size_t bin;
14     key -= cht->min;
15
16     //follow pointers in table
17     do {
18         //calculate bin and set next
19         bin = key >> width;
20         next = ptr[bin];
21
22         //return when flag set
23         if (next & FLAG)
24             return next & ~FLAG;
25
26         //adjust ptr, key, and width
27         ptr += next;
28         key -= bin << width;
29         width -= log2(BINS);
30     } while (1);
31 }
```

Figure 3: Compact Hist-Tree `lookup` function

The `lookup` function for a compact Hist-Tree appears in Figure 3. The pseudocode has many similarities with Figure 2, including range checks for `key` (lines 7–8), variable initialization (lines 11–14), bin calculation (line 19), and updates at the end of each iteration (lines 28–29). The overall algorithm, though, is now much simpler, since we must only follow the pointers in the lookup table (lines 20 and 27). Moreover, we no longer need to maintain a running sum of histogram bin counts, instead returning only the final position stored in a terminal bin (lines 23–24).

Continuing the previous example, looking up the key 567 in the compact Hist-Tree of Figure 1c would proceed as follows. We again begin at the root node, stored in row 0, and index into bin 2 ($567/256 = 2$), which tells us to move to row 6. We update the key ($567 - 256 * 2 = 55$) and calculate the new bin as 0 ($55/64 = 0$), which now tells us to move to row 7. Finally, we index into bin 3 ($55/16 = 3$) and, since it is terminal, return the stored position (129).

4. EVALUATION

To evaluate Hist-Tree, we used the Search On Sorted Data (SOSD) benchmark [10], which has recently appeared in other learned index publications [11, 15]. The benchmark consists of four real-world datasets, each with a different data distribution. In our experiments, all datasets use the base size of 200M keys, and we report the average lookup latency over 1M keys chosen uniformly at random.

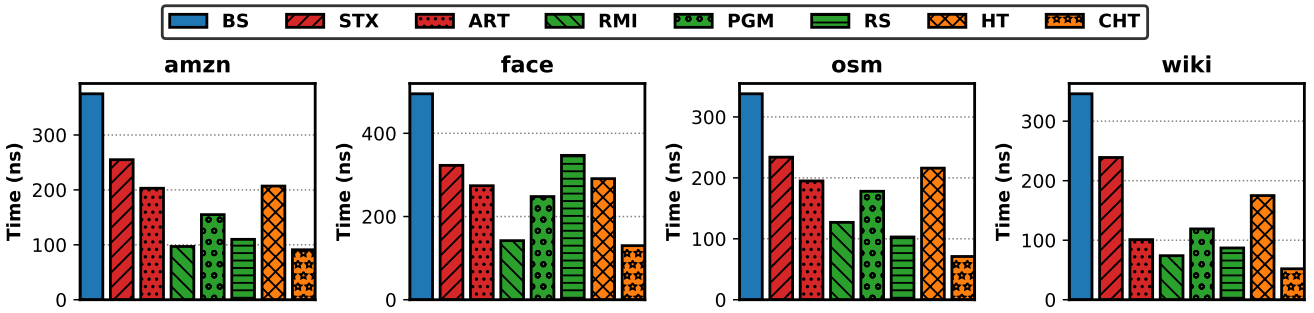


Figure 4: Lowest lookup latency achieved by each comparison point.

We implemented Hist-Tree in C and our benchmark suite in C++, borrowing several index implementations and tuning suggestions from the open-source SOSD codebase [1]. We compiled the code with `gcc/g++-9.3.0` and ran on the same `c5.4xlarge` EC2 instance, which has an Intel Xeon Platinum 8124M CPU (3.0GHz, 26MB LLC), used for previously reported SOSD results [10, 11]. Our comparison points fall into three index categories:

- **None:** As a baseline, we used binary search with no index. However, in future work, we plan to add comparisons to other methods, such as interpolation search and the more recent SIP/TIP [21].
- **Traditional:** For existing traditional indexes, we included a widely used in-memory B-tree (`stx-btree` [3]) and ART [14]. As in other work [15], we vary index size by only indexing a subset of keys. In the future, we also plan to compare against other indexes, including CSS-Tree [19], CSB$^+$-Tree [20], and FAST [9].
- **Learned:** We chose three state-of-the-art learned indexes: RMI [13], PGM-index [7], and RadixSpline [11]. All of these indexes, which appeared in another recent study [15], have publicly available implementations tuned by the original authors.

4.1 Lookup Latency

Since many learned indexes advertise superior lookup performance as their primary selling point, we begin by considering only the results for the configuration of each index that yielded the lowest lookup latency. Later, we discuss the influence of other factors (i.e., index size and build time) on lookup performance.

Figure 4 shows the lowest lookup latency achieved by each comparison point. As expected, binary search (BS) exhibits the worst performance, with the B-tree (STX) improving on BS by about 1.5× in all cases. ART outperforms STX by roughly another 1.2× on all datasets except `wiki`, where it is over 2.3× faster than STX.

Overall, Hist-Tree (HT) performs similarly to ART and between 17–47% slower than PGM-index (PGM). However, on the `face` dataset, the updatable HT outperforms the read-only RadixSpline (RS).

Across the board, we observe a 2–3× improvement when transitioning from the updatable HT to the optimized compact layout (CHT), suggesting that the assumption of a read-only workload is one of the most important factors that explains the performance of state-of-the-art learned indexes. In fact, CHT beats RMI by 1.1–1.8×, PGM by 1.7–2.5×, and RS by 1.2–2.7×. This result clearly substantiates our assertion that, by leveraging all of the same assumptions as learned indexes, a traditional data structure can offer equivalent—or better—performance.

4.2 Index Size

Another important consideration is the total size of an index, as they can often grow quite large. Figure 5 shows the relationship between index size and lookup latency. Note that BS appears as a flat horizontal line at the reported lookup latency, as it does not use an index.

Except for the smallest RMI sizes, every index outperforms BS on all datasets. STX, ART, and HT all achieve the best lookup performance at intermediate sizes. For example, STX always has the lowest lookup latency at a size of 58MB paired with a binary search on a bounded search range of 64 elements. The one exception is `wiki`, where ART performs very well at the largest size.

On the other hand, CHT almost always performs best at the largest index size, with performance degrading as index size decreases. Intuitively, smaller terminal bin thresholds will require more nodes and thus increase index size, which in turn impacts lookup latency. However, as the results show, the relationship between index size and lookup latency is highly dependent on the underlying data distribution. In many cases, we can even reduce HT and CHT sizes by multiple orders of magnitude while observing only minor increases in lookup latency.

In most cases, the learned indexes follow a similar pattern to CHT, exhibiting the best performance at or near the largest index size. Interestingly, their performance converges as index size decreases, suggesting that model differences become less important as the returned search ranges increase.

4.3 Build Time

Finally, Figure 6 shows lookup performance relative to index construction time. Again, BS does not use an index and appears as a flat horizontal line.

The STX and ART configurations with the best performance require significantly less time to build than the other indexes, since they can skip keys during construction. RMI almost always has the longest build time, taking significantly longer than the comparison points on `face` and `wiki`.

HT and CHT have similar build times (i.e., same order of magnitude) to RMI for `amzn` and `osm`. However, on `face` and `wiki`, the build times are more than 3.7× and 19× faster than RMI, respectively, which is similar to PGM and RS. Additionally, we note that converting from HT to CHT incurs only negligible overhead.

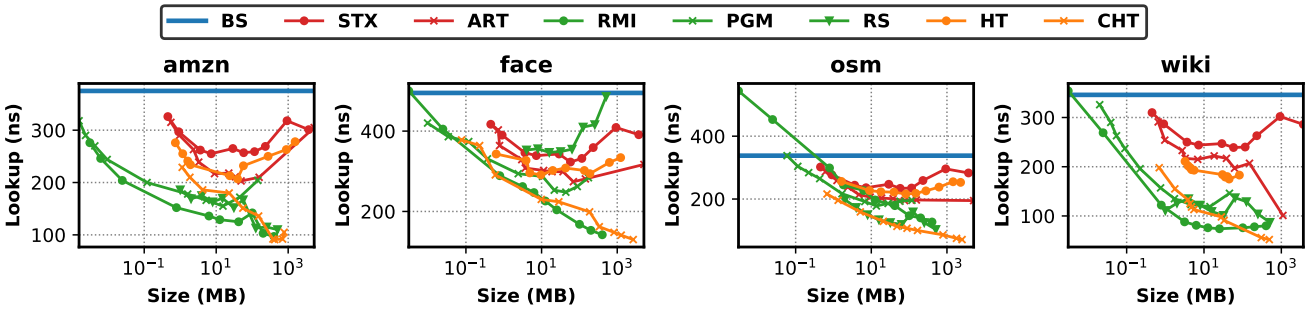


Figure 5: Index size vs. lookup latency

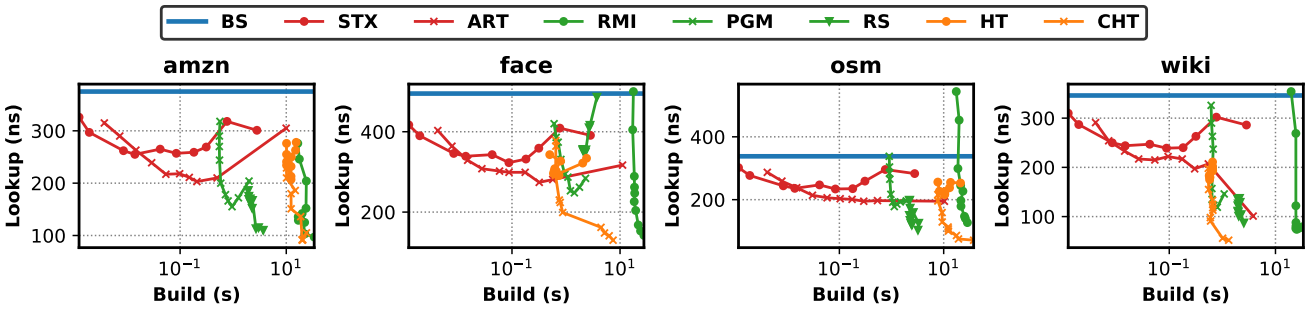


Figure 6: Build time vs. lookup latency

5. CONCLUSION & FUTURE WORK

This paper made the case that the main advantage of learned indexes comes from leveraging certain implicit assumptions, including the sortedness and range of the data. We incorporated these same assumptions into the design of our proposed traditional index, Hist-Tree, and also presented an optimized compact layout to match the read-only nature of some learned indexes. Our results show that, in terms of lookup latency, Hist-Tree can outperform three state-of-the-art learned indexes by as much as 1.8–2.7×.

While we have demonstrated the importance of these implicit assumptions, the performance of Hist-Tree requires further investigation. As an immediate next step, we plan to extend our experimental analysis to test larger data sizes, skewed workloads, and concurrent lookups. We also plan to examine how other indexes (e.g., B-trees) could be adapted to take advantage of the same assumptions built into Hist-Tree, as well how data structures for other use cases referenced in the original learned index paper [13] (e.g., hashing, set membership) might similarly benefit.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This work received support from the AWS Cloud Credits for Research Program.

7. REFERENCES

[1] SOSD. https://github.com/learnedsystems/SOSD.
[2] P. Bailis, K. S. Tai, P. Thaker, and M. Zaharia. Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes. https://dawn.cs.stanford.edu/2018/01/11/index-baselines/, 2018.
[3] T. Bingmann. STX B-tree. https://panthema.net/2007/stx-btree/.
[4] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*, pages 969–984, 2020.
[5] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *PVLDB*, 14(2):74–86, 2020.
[6] P. Ferragina, F. Lillo, and G. Vinciguerra. Why Are Learned Indexes So Effective? In *ICML*, volume 119, pages 3123–3132, 2020.
[7] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.
[8] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*, pages 1189–1206, 2019.
[9] 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 *SIGMOD*, pages 339–350, 2010.
[10] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.
[11] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: A Single-Pass Learned Index. In *aiDM@SIGMOD*, pages 5:1–5:5, 2020.
[12] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A Learned Database System. In *CIDR*, 2019.
[13] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.
[14] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, pages 38–49, 2013.
[15] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking Learned Indexes. *PVLDB*, 14(1):1–13, 2020.
[16] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning Multi-Dimensional Indexes. In *SIGMOD*, pages 985–1000, 2020.
[17] T. Neumann. The Case for B-Tree Index Structures. http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html, 2017.
[18] T. Neumann. Why use learning when you can fit? http://databasearchitects.blogspot.com/2019/05/why-use-learning-when-you-can-fit.html, 2019.
[19] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, pages 78–89, 1999.
[20] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.
[21] P. V. Sandt, Y. Chronis, and J. M. Patel. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search? In *SIGMOD*, pages 36–53, 2019.