Learned String Index Structures for In-Memory Databases

by

Benjamin Spector

S.B. Mathematics, Computer Science and Engineering,
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Tim Kraska
Associate Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Learned String Index Structures for In-Memory Databases

by

Benjamin Spector

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Within the field of machine learning for systems, learning-based methods have brought new perspective to indexing by reframing it as a cumulative distribution function (CDF) modeling problem. The burgeoning field, despite its nascence, has brought with it many opportunities and efficiencies. However, most work in this area has focused on efficiently indexing numerical keys, as the additional challenges posed by indexing strings have prevented the effective application of these techniques to string domains. We hypothesize that the machine learning approaches which have, in recent years, made significant strides in scalar indexing applications can also be effectively adapted to string applications. First, we introduce the RadixStringSpline (RSS) learned index structure for efficiently indexing strings. RSS is a tree of learned radix splines each indexing a fixed number of bytes. RSS achieves better performance than other structures by first using the minimal string prefix to sufficiently distinguish the data, followed by a contextual learned model to predict its location. Additionally, the bounded-error nature of RSS accelerates the last mile search and also enables a memory-efficient hash-table lookup accelerator. Second, we benchmark RSS against existing algorithms on several real-world string datasets and study its performance in-depth. RSS approaches or exceeds the performance of traditional string indexes while using up to 300× less memory, suggesting this line of research may be promising for future memory-intensive database applications.

Thesis Supervisor: Tim Kraska
Title: Associate Professor

Acknowledgments

Above all else, I would like to thank Tim Kraska for his mentorship and wisdom throughout my time at MIT, both throughout the course of this research and also my general studies. I could never have hoped for a better advisor. I'd also like to especially thank Andreas Kipf, whose guidance, advice, and teaching has been truly invaluable in the completion of this work. I'd like to thank my collaborators Kapil Vaidya, Chi Wang, and Umar Minhas, especially as much of this text is adapted from [20]. Finally, I'd like to thank my family and friends who have supported me throughout the last four years.

This research is supported by Google, Intel, and Microsoft as part of DSAIL at MIT, and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Databases, as perhaps *the* key commercial application of computing, underlay the systems powering governments, financial institutions, supply chains, academic research, and more; it would not be an exaggeration to suggest that their reliability and efficiency forms the bedrock of the modern global economy [3]. Among the key components of a modern database are its indexes, which serve as librarians by directing queries to the location of requested information. One important attribute of database indexes which distinguishes them from many other associative data structures is that they need to efficiently handle both lookup queries as well as lower-bound queries – the location of the greatest data less than the query – for the application of range queries. As crucial intermediaries, the performance of these indexes is essential to the performance of the underlying systems.

In modern database indexes, it is also necessary to distinguish *primary* and *secondary* indexes. The primary index produces the mapping along the axis in which the data is already sorted in memory, and is usually the most commonly queried index; it is this index which is the focus of this work. Secondary indexes consider mappings along other unsorted and potentially non-unique attributes of the data, and the learned approaches described in this work are usually less applicable in this latter domain.

1.1 String Indexes

The simplest possible index is no index: one could simply search the database directly for the requested key. While this approach is extremely slow for secondary index scenarios, it is not unreasonable for primary indexes, as one can conduct the binary search on the data in $O(\log N)$ time. The no-index approach also has the advantage of consuming zero memory. However, its main drawback is its usually poor latency. For very large distributed systems the cost of retrieving remote data for this search is extremely high; even in in-memory settings as we will explore here it has poor data locality properties. Despite these issues, it remains an important baseline to substantiate the performance of indexes which would aspire to do better.

The standard approach to indexing strings has been, at least for the last few decades, B-trees and their variants. B-trees are a highly optimized generalization of a binary tree, in which there is increased fan-out at each node, and the data structure is additionally usually optimized to fit neatly into a memory page to improve access properties. B-trees are relatively simple, reliable, and efficient, and these properties have helped them endure in production databases [7]. However, B-trees are somewhat less useful for string-valued data than numerical data [10]. First, because string data is large and variable length, strings must be stored either outside the B-tree nodes, which negates much of the memory locality benefits for which B-trees were designed, or else they must be stored within the B-tree, dramatically raising memory consumption and lowering node fan-out (and therefore increasing the depth of the tree).

A more recent string-keyed data structure is the Adaptive Radix Tree (ART). ART takes advantage of the lexicographic ordering of string data by constructing a compressive trie. Additionally, in order to prevent the significant memory wastage present in radix tree structures, ART dynamically resizes its nodes to their data in order to achieve good lookup speeds at reasonable memory usage [16].

A third, recently developed structure is the Height-Optimized Trie (HOT). HOT achieves excellent performance on large strings containing common substrings by only storing and indexing on the so-called "discriminative bits" within the strings which

can be used to tell them apart. HOT targets a fixed fan-out per node and is highly optimized to make use of modern processor functionalities with vector instruction sets [5].

A common application of string indexes such as these is for a global dictionary encoding [6], so that other data structures can work with numerical data instead of string data and therefore increase their performance.

1.2 Learned Indexes

One of the more exciting developments to come to the field of indexing is the advent of learned indexing [15]. The idea of the learned index approach is to change one's perspective on the problem: rather than indexing being a data structures problem, it is instead a modeling problem, in which the model is provided a key and must output the location in memory. Of course, the model may not always be quite right, so the second step is to actually go to the predicted location and then look around until the desired data is either found or determined not to exist. The overall process is illustrated in Figure 1-1. From an information theoretic perspective, this approach has the benefit of relying more strongly on the entropy of the distribution than the size of the data. For example, a million points all on the same line can be stored as just a slope and intercept in the index, rather than the millions of values needed under a classical data structure. This example also serves well to illustrate that learned index structures benefit tremendously from well-behaved distributions.

Scholars have now proposed many learned indexes which operate on scalar data. The first of these and still most-used is the Recursive Model Index, as proposed in [15]. Many others have emerged since then; these include the PGMindex [11], which provides provable worst-case bounds, ALEX [9], which provides support for updates and adaptation, Compact Hist-Trees [8] which provide both excellent lookup latency as well as compact index size, Shift-Table [12] which relies on model correction, and Radix Spline [14], a compact and error-bounded learned index on which this proposal is based, as well as its successor PLEX [21] which also includes automatic tuning.

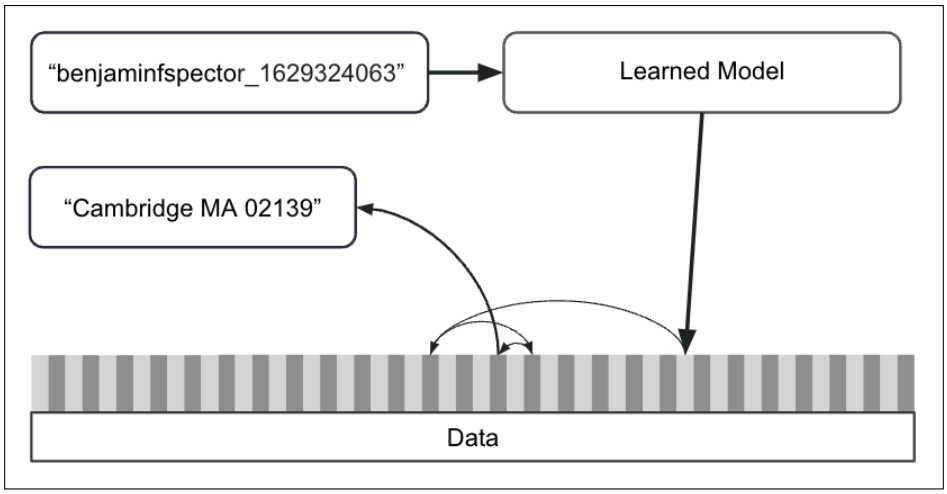

Figure 1-1: An illustration of the overall process of a lookup for a string-keyed learned index.

However, all of these structures operate on fixed-width numerical data, making them inapplicable to string-keyed data.

1.3 Learned String Indexes

There are several reasons why building learned indexes for strings is usually harder than for numerical keys:

- Strings are variable-length, complicating both model inference and memory layouts.
- Strings are much larger objects which are relatively expensive to store, compare, and manipulate.
- Strings tend to have significantly worse distributional properties than numerical keys due to the prevalence of common prefixes and substrings. As a result, strings tend to have low discriminative content per byte, which makes them exceedingly hard to model accurately and compactly.

Regarding the distributional properties of strings, we illustrate the CDF of the Wiki dataset in Figure 1-2. Unlike the relatively well-behaved distributions numerical

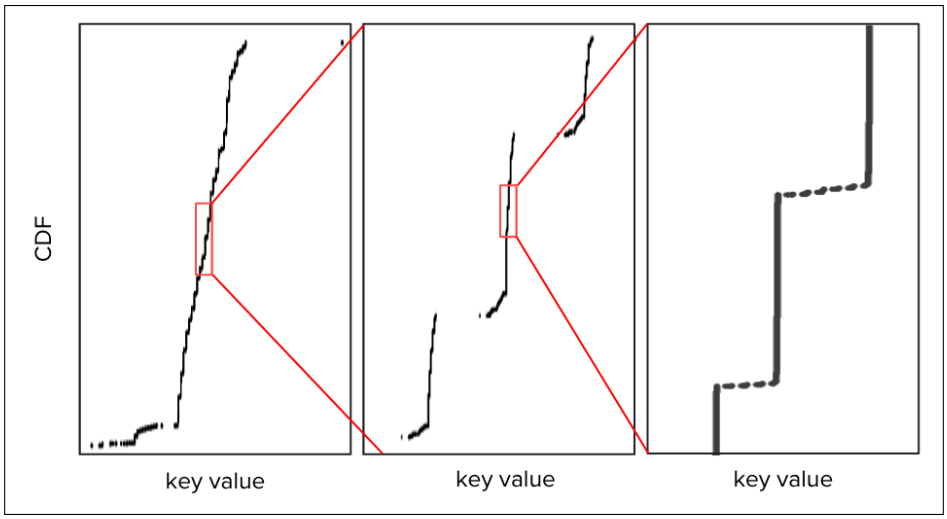

Figure 1-2: Repeated zooming on the CDF of the Wiki dataset reveals its fractally stepwise nature.

data tends to follow, the CDF appears stepwise at essentially every level of zoom. While this might seem like an arbitrary metric, zooming in on the dataset is actually a reasonable approximation of the organization of hierarchical index structures (in which category every method we consider will fall), and these stepwise distributions are by their nature very difficult to model accurately and compactly.

There is only one clear prior example of a learned string index, which is SIndex [22]. SIndex is a two-layered structure divides strings into two components, a common prefix and a unique partial key. First, it greedily groups strings according to their common prefix. It then trains models to predict the partial key of each group. The authors find that SIndex works reasonably well. However, there are two key limitations of the work. First, the authors mostly measure performance on randomized datasets. As described in the introduction, real string distributions have **terrible** distributional properties compared to uniform random data. The authors do evaluate on the URL real-world dataset and the performance of SIndex is significantly lower. Second, the baselines of the paper are somewhat weaker compared to the current state of the art; the strongest competitor, Wormhole, is still somewhat slower than ART and HOT. To conclude, while there has been some promising prior work in this field, the practicality of learned string indexing has still not been thoroughly established.

In our view, the most significant problem with learned indexes for strings is the last-mile search which corrects the learned model's prediction. Last-mile search is usually implemented as an exponential search around the prediction which expands out until a bound is found, followed by a binary search inwards to find the true index. This search is especially expensive in string scenarios for two reasons. First, average model error in these scenarios is often quite high, due to the aforementioned difficulty in modeling real-world datasets, and furthermore because the cost of the last-mile search is logarithmic with the error of the model, it is quite difficult to significantly improve this. Second, actually conducting the last mile search is slow. Each comparison is expensive and requires potentially many sequential operations, and the strings' large size decreases the number of keys which fit in cache. Modern column stores typically store strings in fixed length sections (first few bytes) and variable length sections (remainder of the string) [18]. Increasing the size of the fixed-length section can waste memory, and searching the variable length section will incur expensive cache misses; there are no easy answers.

One clear improvement, which forms part of the basis of this research, comes from having bounded error (*i.e.* the model outputs a bounded interval in which the sought key will lie, if present). Then, one can replace the exponential search with binary search and skip many string comparisons. We use this to ameliorate the cost of the last-mile search for lower bound lookups (*i.e.* finding the key that is equal to or larger than the lookup key), and then, for equality lookups, we bypass it (more later).

In this work, we emphasize increasing the lookup performance and decreasing the memory footprint for the primary-index scenario. We introduce the RadixStringSpline (RSS), a learned string index consisting of a tree of the learned index structure RadixSpline [14, 13, 17]. The RadixSpline, on which RSS is based, is a learned index that consists of an error-bounded spline which is in turn indexed in a radix lookup table. Similar to nodes in ART, each RSS node indexes a fixed partial key (*e.g.* 8 bytes). But unlike ART, our nodes may have extremely high fan-out due to the incorporation of a learned-model at each node. To increase the fan-out of a given byte-prefix, we encode the input data using the Hu-Tucker-based two-gram order-

preserving encoding from HOPE [23], which shortens common prefixes and increases information density. For example, on a popular URL dataset, two-gram encoded-data is on average 1.6× smaller than the original string data. Like the original string data, our encoded data has variable length. Often, the first few bytes of the two-gram encoding are sufficient to distinguish between most of the string keys, in which case RSS only requires a single node to index these keys. However, typically there also are many collisions among these prefixes, which requires RSS to recurse on the next section of the colliding strings. While we do not discuss updates, techniques as proposed in ALEX [9] are generally applicable to our approach.

Compared to ART and HOT, RSS indexes are faster to build, similarly fast or faster to query, and consume much less memory for equivalent latency. RSS particularly benefits from a high information density in the most significant bytes. Some data distributions, like the Wiki dataset, satisfy this reasonably well, and correspondingly RSS has high performance. For other data distributions (*e.g.* the URL dataset, which is essentially a worst-case practical dataset) that require many low-information bytes to distinguish keys, RSS needs many levels and hence may have low performance.

Chapter 2

Methods

Given an immutable, lexicographically sorted array of strings, we want to build an index on top which supports two key operations. First, the index should support equality lookups: if the string is present, return its index, and if not, return NULL. Second, the index must support lower bound queries: given a string, find the index of the greatest string greater than or equal to the provided one. These two operations are important in column stores that use dictionary encoding. For equality queries (*i.e.* WHERE str = X) one needs to support equality lookups on the dictionary, and for prefix queries (*i.e.* WHERE str LIKE 'A%') lower bound lookups are required (to find the first string in the dictionary that is lexicographically greater than or equal to 'A').

Our method consists of two parts. First, we describe the RadixStringSpline (RSS), a compact and efficient adaptation of RadixSpline to the string domain [14]. This approach provides both fast queries as well as bounded error. Second, we provide an optional add-on hash corrector which makes use of this bounded error to, at the cost of 12 bits per element, further improve equality lookup performance.

2.1 Radix String Spline

One useful perspective of the order-preserving indexing problem is that of a compressive mapping. If a million keys use a 32 bit integer type, the overall density of

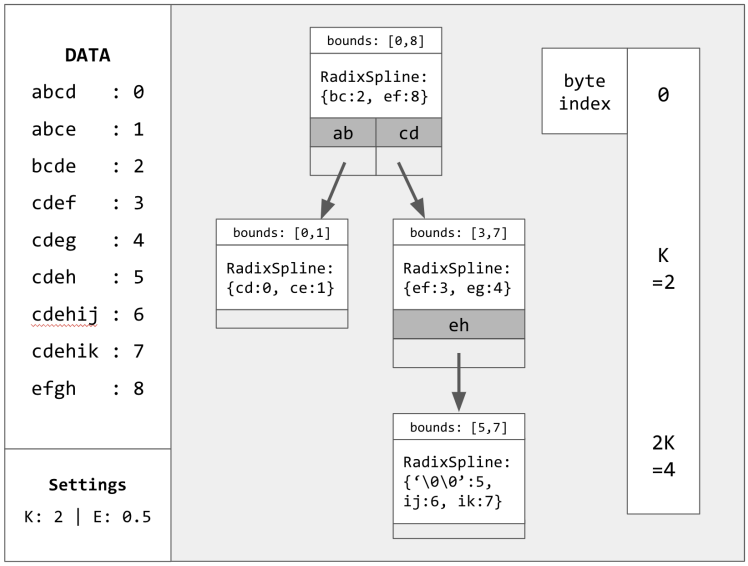

Figure 2-1: A toy example RSS tree structure, for the data and settings indicated. The root node corresponds to the entire data (bounds [0,8]) and indexes the first two bytes of the data (K=2). The RadixSpline in the root indexes the only two keys which do not have collisions in the first two bytes (bc and ef). The remaining two-byte prefixes (ab and cd) have collisions and are redirected to child nodes.

information is somewhat low relative to the capacity of the type ($10^6 < 2^{32}$). Indexing transforms these million keys into a continuous range, effectively compacting the data into the last consecutive bits.

The core difference between string indexing and integer indexing is that string distributions generally have significantly lower entropy *still* ($10^6 \ll 2^{320}$ or so) because integer keys must be entirely distinct in their first few bytes but string distributions do not have this constraint. By contrast, string keys can (and in most practical applications do frequently) share long prefixes, requiring examination of more data to fully distinguish them, and more compaction required to move all of this information to the last few bits. There are several (not necessarily exclusive) approaches one might take to ameliorate this. For example, one can actually directly compact the data, as per [23], and we find that this certainly does help. In our approach, our goal is to have the model quickly operate on the minimum amount of data to get the job done. Because the local distribution of the data is highly contextual on where one is in the global distribution, it also makes sense for the model to be similarly

contextual in nature too. It follows that our approach should be to quickly consume as many bytes as possible until context is established and the strings can be sufficiently distinguished, and then return the prediction of a contextual model.

An RSS is configured by two key parameters. First, there is the span of each node, K, which controls how many bytes are consumed at a time. Second, there is the error-bound E, which controls the maximal allowable error for predictions. The structure of an RSS is a relatively small tree, for which each node contains both a learned model – to provide error-bounded predictions for any query in that context – and also a redirector structure, which determines if it is necessary to traverse further down the tree in order to achieve a prediction with the desired error bound. Each node in the RSS consumes a fixed number of bytes of the string, usually with either $K = 8$ or $K = 16$, and in this way an RSS can also be seen as a trie with this longer span per node and correspondingly higher fan-out.

We illustrate a sample RSS in Figure 2-1. The RSS indexes the indicated data in the bottom left from 0 to 8, with each node operating on two-byte chunks and a maximum allowable error of 0.5, which (being within rounding) is to say it must predict everything perfectly. (We note that practical RSS indexes usually have a greater allowable error but these are harder to intuitively illustrate.) All searches begin at the root node (top) and simultaneously traverse the tree and the string until the partial key can no longer be found in the redirector array. At that point, the local RadixSpline at that node is guaranteed to provide a bounded-error prediction.

To build an RSS index, we begin by building a RadixSpline on the first K bytes of every string in the dataset (this is the root node), including all duplicates. Then, we iterate through the unique K-byte prefixes, and check if the estimated position is strictly within the prescribed error bounds for both the first and last appearance of the prefix. This will always be the case for unique prefixes but might not be for prefixes which have duplicates, and cannot be the case for prefixes which have $\geq 2E$ duplicates – then, even predicting the median instance can't satisfy the extrema. For each prefix which fails the test, we add it to the redirector table, and build a new RSS over just the range of the problematic prefix (and starting at byte K instead of

byte 0). This process continues recursively until every key is satisfied. RSS can be trained in either a depth-first or a breadth-first fashion; in practice we build it mostly depth-first but store a small window forwards at the uppermost layer so that we use minimal memory while building the index.

We also note that the size of the radix table in the RadixSpline should be adjusted depending on where in the tree one is. Near the root, the radix table should be large; near the leaves we often use just 6 bits to save memory.

Practically we have found $K = 8$ or $K = 16$ and $E = 63$ or $E = 127$ to be broadly good settings over the evaluated datasets; in our experiments we default to $K = 8$ since it performs better on more common distributions, although it performs worse for extremely sparse strings. Note that a fanout spanning 8 or 16 bytes is generally far higher than the maximum fanout of ART (1 byte, *i.e.* 256 keys, per level) or HOT (32 keys per level), although this is of course data dependent.

Querying an RSS index is simple. One begins by extracting the first K bytes of the string, and conducting a binary search of the redirector to try to find the prefix. If it is found, then one follows the redirect to the new RSS node and the process begins again, only operating on the next K bytes. If it is not found in the redirector, then that means the key is guaranteed to be within acceptable bounds for the RadixSpline, so one queries the RadixSpline at the current node with the appropriate substring and returns the result.

As an example, suppose we want to query the string "cdeg" from the RSS index in Figure 2-1. We begin by extracting the first two bytes of the string, in this case "cd", which are packed into a 16-bit integer. We then search the redirector of the root node for this key. Since we find it in the second slot, we follow the pointer at that slot to the next node of the RSS. Now, we extract the next two bytes, "eg" and check the node's redirector. This time, we fail to find it, so we know it is correctly indexed by the local RadixSpline. So, we query the RadixSpline and return the result as our prediction. Finally, we execute a local binary search in the data to either find the string and its index or to validate its non-existence.

Analogously, suppose we wanted to conduct a lower bound query for the string

"defg", not found in the data. We again search the root node's redirector for "de", and, failing to find it, execute the root's RadixSpline on that prefix. We again perform a binary search, only this time after failing to find it we return the left bound location.

There are a number of additional optimizations we make which meaningfully improve RSS's performance. First, RSS is very sensitive to the performance of its internal data structures, especially its redirector for long strings. As a result, when the redirector is below a fixed size, we store it in a dense array; otherwise, we use Google's dense hash map. One modification which we also implemented was a novel redirector which constructed special hashes on data so that the hashes of the prefixes of strings could be compared with a simple bitmask operation; this allows us to potentially skip many layers of redirection at once. We found that this could provide a marginal benefit for extremely long strings but was otherwise either counterproductive or not worth the added complexity; we therefore omit it from our continuing discussion.

Additionally, while in our analysis we benchmark on uncompressed and compressed data separately to enable better comparison with competitor methods, RSS does not require this. Instead, one can straightforwardly integrate into RSS an in-line Hu-Tucker encoder, so that even when operating on un-compressed data, compression may be done on the fly. Then, only the compressed data is stored within the index. This does significantly increase build times, primarily due to the cost of the Hu-Tucker algorithm to generate the encoding table, but it is largely a fixed cost because the two-gram dictionary is fixed-size; the additional cost of collecting statistics about two-gram frequencies is almost negligible. During lookups, we find that the cost of this encoder is very minimal; usually we found the encoder to be very significantly faster than the 12 ns/char reported in [23].

Finally, we also can often return tighter error bounds than the specified error bounds based on the precise data in the local RadixSpline; we do this whenever it is possible.

One additional benefit of our approach is that if the RSS index is appropriately and carefully constructed (requiring a synchronization of the predictions of different layers of the tree, which we do in our implementation) it can also be made perfectly

monotonic, which may prove useful in future work on accelerating or even completely bypassing the last mile search.

We believe that one should not undervalue the importance of the model being error-bounded for two reasons. First, in a string setting, even with relatively low errors and an optimized last-mile search, the last mile search still turns out to be the dominant cost. A bounded error means one only needs to conduct a binary search rather than an exponential search. Second, it enables a memory-efficient hash corrector, to be described below.

2.2 Hash Corrector

Often, while index structures certainly need to support lower bound queries as described above, the optimization of the actual direct lookup of known string keys is equally important. To this end, we provide an auxiliary data structure which can improve performance for this problem at the cost of a small amount of additional memory. Essentially, one stores a contiguous array of signed int8 offsets, with -128 reserved as empty. To build the Hash Corrector (HC), for each string in the dataset one runs the RSS index and computes the difference between the predicted and true values, which is guaranteed to fit in the range -127 to 127. Then, one hashes the string into these slots several times (up to a predetermined number) to find an empty slot. If one is found, we insert the offset at that slot. Compared to traditional Cuckoo hashing, this technique trades false positives (*i.e.* when the string at the offset does not match the lookup key) for memory efficiency.

When one queries a string, one again hashes the string to a few of these slots and tries each offset. If it's a match, the expensive binary search is avoided. Otherwise, the bounded binary search is a reasonable fall-back. This is also required for negative lookups, although we deem these to be less important in practice (*e.g.* in the dictionary encoding scenario). To have some benefit from false positive lookups, our implementation uses the keys it finds at these offsets to at least reduce the bounds of the binary search, and these reduced bounds can also help us rapidly reject wrong

offsets (which are out of the current bounds). So, each query to the underlying data is guaranteed to provide at least some benefit. Additionally, if a string is ever hashed to an empty slot on the table, we can immediately terminate the lookup as negative, reducing the hash corrector into a sort of approximate bloom filter; under the described settings this happens with about 80% probability.

In our implementation, we use a 128-bit MurmurHash3 hash [4], [19] giving us 4 attempts, and we set the load factor to be 2/3. This then provides a speed boost to >95% of lookup queries at the cost of 12 bits per key. Further tuning this time-space trade-off might lead to additional gains.

For example, suppose we want to look up a string S in a database of N strings. We first execute the RSS index to get an error-bounded index prediction p. We then hash S into 4 positions in range $[0, 3N/2)$ – h_1, h_2, h_3, h_4. For each position h_i, if offsets$[h_i] = -128$, we can immediately return that the queried key is not in the database. If it instead exceeds the bounds, we can simply ignore and move on to h_{i+1}. Otherwise, we compare S to the string at offsets$[h_i]$. If they match, we return; if S is larger, we set the location as a left bound, and if S is smaller we set the location as a right bound. Finally, if we try this process four times and still have not found it, we execute a binary search between the left and right bounds. The data structure can and should be entirely ignored for lower-bound queries; it does not accelerate them, and the lower-bound query reduces to the RSS lookup described previously.

Chapter 3

Evaluation

3.1 Datasets

We evaluated RSS and its Hash Corrector (RSS+HC), along with B-tree, ART, and HOT, on four datasets meant to capture a broad range of practical applications, and in both uncompressed and two-gram Hu-Tucker compressed forms.

1. Random: 10,000,000 32 character alphanumeric strings which represent a distributional baseline. When compressed, the average string falls to 24.5 bytes, which is roughly consistent with the entropy increase per character by increasing the alphabet size from 52 to 256, although slightly inferior due to the necessity of both an order-preserving encoding and also maintaining additional symbols for characters not found in the data; otherwise we would expect a compression ratio closer to 29%.

2. Email: 144,770,7422 real email addresses which average 21.2 bytes in length. We perform domain-reversal as is consistent with how emails are usually stored. Upon two-gram compression these shrink to 11.8 bytes on average.

3. URL: 103,476,164 URLs averaging 86.4 characters in length. Compression reduces these to 53.8 characters on average [2].

4. Wiki: 13,441,485 URL suffixes from the English Wikipedia, which somewhat

counter-intuitively makes this dataset more representative of natural language. Strings average 20.6 bytes before compression and 13.0 bytes after compression [1].

For every dataset other than URL, we fix RSS with $K = 8$ – that is, it works in 8-byte chunks. For URL, to ameliorate the deleterious effects of its longer, lower-density strings we set $K = 16$ instead.

3.2 Lookups and Memory

One challenge in benchmarking learned indexes against classical indexes is that their nature is somewhat difference in essence; classical indexes are not designed to produce error-bounded predictions, and learned indexes are not designed to capture the noise around individual elements but instead just the broad shape of the distribution. Yet, only considering the classical indexes with full insertion unfairly penalizes their memory consumption relative to learned indexes. The solution we consider here is to insert every $2E$ elements into classical indexes and run lower-bound queries, so that we can evaluate their performance more fairly across the board. A complete chart of our findings for the four datasets, both without and with compression, can be found in Figure 3-1. All experiments were conducted on a Intel(R) Xeon(R) Gold 6230 CPU.

One caveat we would like to express is that many of the results presented here are somewhat sensitive to choices made in performance engineering each method to show it in its best light. Where possible, we have used the original implementation of the authors, as with HOT. For others, we tuned primarily for lookup speed; as a result ART uses the same block memory layout for its last-mile search (where applicable) as RSS, which is not internally compatible with its lookup structures and thus ART is shown to use significantly more memory than if we had tuned it for memory consumption.

Overall, we find that RSS and its derivative RSS+HC are highly competitive with the current state of the art. On compressed data, RSS-based methods were strictly

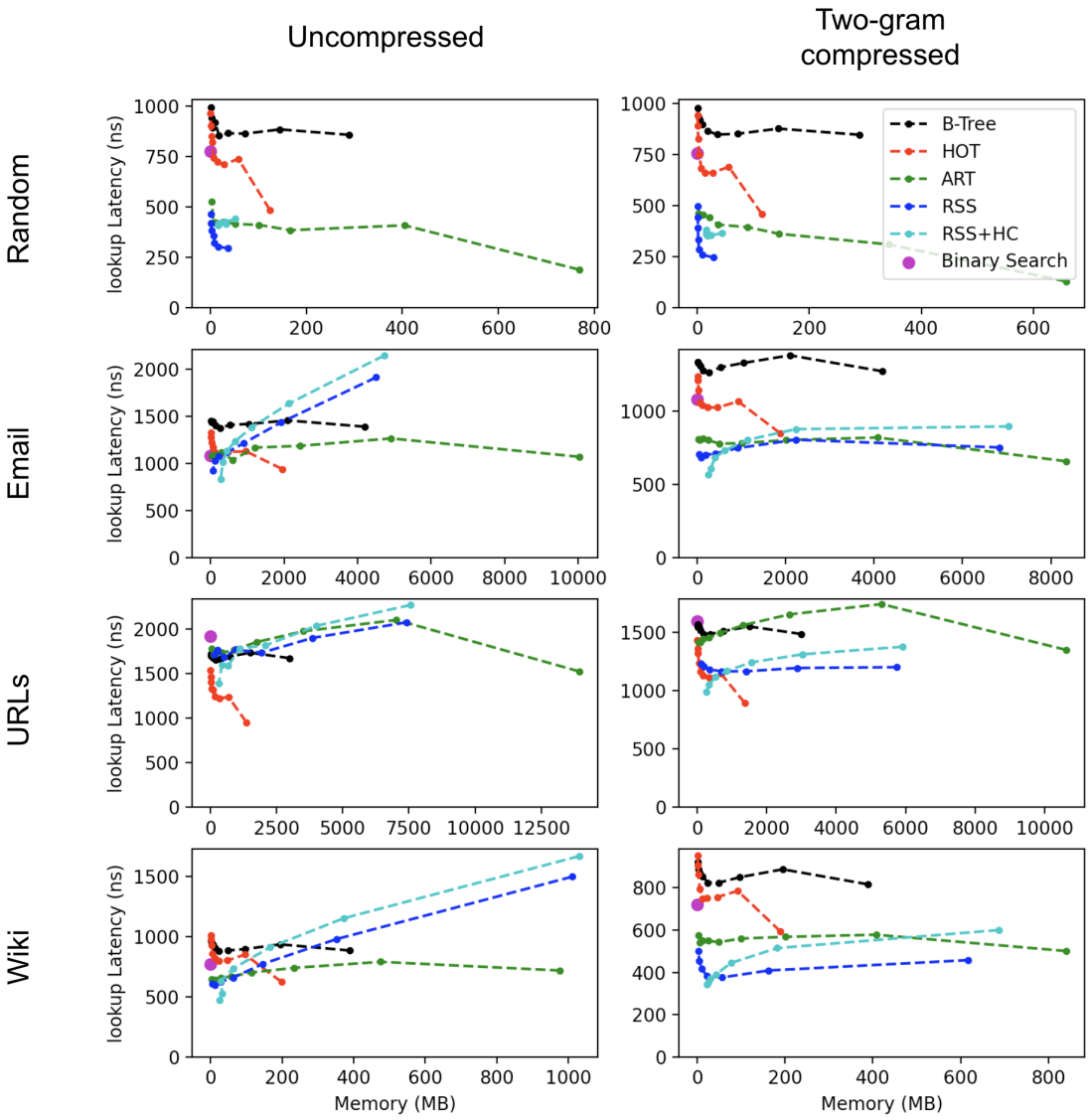

Figure 3-1: Memory consumption versus lookup latency for B-trees, HOT, ART, RSS, RSS with its Hash Corrector, and a simple binary search. For baselines, insertion periods are distributed exponentially from 1 to 256, with the former on the right and the latter on the left in each diagram. RSS-based methods use error bounds ranging exponentially from 2 to 128.

Lookup Latency (ns)	Uncompressed				Compressed			
Method	Rand	Email	URL	Wiki	Rand	Email	URL	Wiki
B-tree	858	1,389	1,674	884	847	1,271	1,483	816
HOT	485	939	**954**	623	457	849	**891**	593
ART	**189**	1,035	1,525	616	**129**	657	1,347	501
RSS	295	923	1,682	599	246	684	1,164	376
RSS+HC (+)	409	834	1,389	471	353	564	986	344
RSS+HC (-)	287	**743**	1,493	**429**	266	**504**	1,029	**327**
Binary Search	778	1,158	1,920	769	758	1,081	1,590	719

Table 3.1: Best achieved lookup query speeds for each method and dataset, including the last-mile search, with error bounds ranging exponentially from 0 to 128 for every method other than RSS, which ranges from 2 to 128 instead. We include positive lookups and negative lookups for RSS+HC separately since they follow somewhat different paths and therefore have appreciably different performance. Finally, we include an optimized binary search as the zero-index baseline.

Lower-bound Latency (ns)	Uncompressed				Compressed			
Method	Rand	Email	URL	Wiki	Rand	Email	URL	Wiki
B-tree	839	1,364	1,640	867	822	1,255	1,453	805
HOT	710	1,126	**1,121**	809	644	1,014	**1,132**	754
ART	**228**	1,035	1,711	616	**163**	**664**	1,411	518
RSS	295	**923**	1,682	**599**	246	684	1,164	**376**
Binary Search	778	1,158	1,920	769	758	1,081	1,590	719

Table 3.2: Best achieved lower-bound query speeds for each method and dataset, including the last-mile search, with error bounds ranging exponentially from 0 to 128 for every method other than RSS, which ranges from 2 to 128 instead. RSS+HC is omitted as it reduces to a simple RSS for this query, and both RSS and Binary Search perform identically on this as on lookup queries as in table 3.1

fastest on the Email and Wiki datasets, and were very close to the fastest on the Random and URLS. These latter two datasets seem require different trade-offs, as the best on one was much worse on the other, and vice versa. Most importantly, though, learned string indexes like RSS unlock new dynamics in the trade-off space. Whereas most data structures shown perform their best when they are allowed to consume maximal memory, RSS bucks this trend, and instead usually performs its best when using relatively little memory. For example, on the compressed Wiki dataset the fastest competitor to RSS is ART, with an average lookup latency of 501ns relative to RSS's best latency of 376ns and RSS+HC's best latency of 344ns. RSS is actually much better than even this, though, because under these conditions ART requires 840MB whereas RSS requires 58MB and RSS+HC 23MB. In fact, RSS achieves the exact same latency as ART of 501ns when using just 2.57MB, which is over 300x less memory! The reason for this is that RSS performs its best when it is able to ignore the high-frequency noise which makes up the local distributions of these datasets, and just fit the lower-frequency signal when increasing the error bound so that it can zoom out a bit more. Conveniently, this also represents decreased memory consumption and lookup speeds.

There are a few additional attributes of these results we wish to discuss. The first, and perhaps the most striking, is the significant impact of compression on the performance of RSS and ART, relative to the very little effect it has on HOT and B-tree. Across the four datasets, compression improved RSS's lookup speed by an average of 40% and ART's by 41%. By contrast, the impact on HOT was just 7% and on B-tree was 8%. Interestingly, the distribution of these benefits was not at all uniform; ART benefited much more than RSS on the Email and Random datasets, whereas RSS benefited more on Wiki and URL.

Under close examination of the results, one finds a clear similarity pairing of performance characteristics between the string indexes. On one side is ART and RSS, which both significantly benefit from string compression and tend to perform well on the same datasets. On another side is B-tree and HOT whose performances are also correlated. In consideration of the structures of these indexes, this makes

Internal Latency (ns)	Uncompressed				Compressed			
Method	Rand	Email	URL	Wiki	Rand	Email	URL	Wiki
B-tree	285	644	782	297	291	586	776	276
HOT	315	533	**606**	317	307	424	569	279
ART	54	337	643	139	46	203	479	93
RSS	**30**	**318**	691	**126**	**20**	**164**	**372**	**57**

Table 3.3: Internal latency – that is, the cost of a lower-bound query without the last-mile search – with an error bound of 128**, equivalent to an insertion period of 256. RSS is almost always the fastest, and often by a large margin. HOT is always the smallest (memory consumption not shown), although all data structures surveyed use relatively little memory under these conditions.**

sense: B-tree and HOT are both fixed fan-out indexes which are expressly designed to care more about the size of the dataset than their distribution. By contrast, RSS and ART are trie-based methods which maintain guaranteed ordering at all stages, and as such are more sensitive to the span of the data. This explains their additional sensitivity to compression, and also their much better performance relative to HOT and B-Tree on random data, as random data contains sufficient discriminative power in just the first few bytes to index the entire data.

One additional question one might wish to answer is how long does it take for the various data structures to get close to the desired data. This is an important question when the data desired is very distant or expensive, in which case one overall structure might be to query a local index to decide which machine to ask or compressed block of data to uncompress, and then let it do the final last-mile search. To test this, we ran lower-bound queries on each dataset without conducting the last-mile search, so that all data structures are now producing predictions with the same error-bound. The results are presented in table 3.3. Overall, we find that RSS is almost always the fastest by a significant margin, and does especially well on both compressed data and reasonably well on string distributions without enormous common prefixes.

Build Speed (ns / item)	Uncompressed				Compressed			
Method	Rand	Email	URL	Wiki	Rand	Email	URL	Wiki
B-tree	15	16	16	16	16	15	16	15
HOT	164	227	283	249	153	199	254	217
ART	100	174	223	156	90	115	182	129
RSS	84	133	255	85	82	83	186	86
RSS+HC	175	423	633	236	184	297	465	187

Table 3.4: Build times for lookup speed-optimized index structures.

3.3 Build Time

In addition to unlocking different tradeoffs in the lookup and memory space, RSS has different properties than traditional indexes in its build speed. While this characteristic is usually less important than the raw performance of the index, as most indexes are built infrequently and used frequently, it is nonetheless helpful to see the overhead incurred. A summary of the build speed of the various for the presented datasets, each being optimized for lookup speed, is presented in table 3.4.

The B-tree is by far the fastest, as it has a highly efficient bulk loading structure on sorted data that allows it to avoid actually doing any comparisons whatsoever. Among the more adaptive index structures, RSS is fastest on every dataset other than the long-span URL dataset, and even then it is less than 15% slower to build of the fastest to build, ART, which, under these settings, RSS trounces with 14% faster lookups while using less than one-fifteenth the memory.

One note about RSS's performance we may also wish to consider is that if lookup speeds are the absolute priority, and its add-on hash-corrector is therefore to be used, build speeds will decrease significantly beyond any other index. RSS+HC is the slowest index to build on every dataset other than Wiki, on which it is second slowest. The reason for this is that to construct the whole data structure, one must first build the RSS, and then also run it on the entire dataset in order to determine errors so that offsets may be added into the corrector, and then finally hash the entire dataset and insert accordingly. It is therefore not too surprising that this incurs significant additional costs.

Build Speed (ns / item)	Uncompressed				Compressed			
Error Bound	Rand	Email	URL	Wiki	Rand	Email	URL	Wiki
2	84	181	256	210	82	134	207	120
4	79	157	226	128	81	109	183	94
8	78	152	206	106	78	99	172	86
16	76	146	199	95	76	92	186	82
32	73	143	217	88	75	87	190	80
64	72	139	231	85	74	83	208	79
128	72	133	255	81	74	77	209	78

Table 3.5: Build times for RSS for various error bounds

Finally, a key difference between RSS and other index structures is that its build speed is not too sensitive to the error bound E imposed, as it still must consider every key and ensure that its splines handle them correctly, whereas other index structures build times tend to decrease approximately linearly with the allowable error bounds, as they simply insert a fraction of $1/E$ of the total data. It seems likely that there is an inherent tradeoff here: RSS derives much of its advantage precisely because it is able to consider the whole data and choose the elements to add to its model which best capture the full distribution of the CDF, so it is this additional computation during the build process – even for large error bounds – which increases the performance and decreases the memory consumption of RSS relative to traditional indexes.

Chapter 4

Conclusion

In this work, we have introduced a novel learned string index for primary indexing which is a marked improvement over existing string index structures in speed and size. We find RSS-based methods are fastest on half of evaluated datasets while consuming up to 300x less memory. RSS also introduces new positive trade-offs in build speed, lookup speed, and memory footprint. We evaluated the learned index on datasets varied in both distribution and size, and found that our method works especially well in conjunction with string compression schemes.

There exist many interesting future directions: First, better compression techniques could further improve the performance of both RSS and other index structures. Second, the internal redirector of RSS could be improved to be more efficient for large datasets with common prefixes (like URL). Third, we also believe there is likely considerable further tuning (and auto-tuning) which could be done on RSS to further improve performance. One optimization which might be particularly fruitful would be automatically deciding the correct number of bytes K to operate on based on the local distribution being considered. In this way, one might imagine constructing an adaptive RSS, analogous to ART. Finally, RSS currently only uses splines as its internal fixed-width model. However, other types of models could provide significant benefits. In fact, one might see the tree and redirector structure of RSS as a new model for generating error-bounded models out of unbounded components. Consequently, this general approach might also be applied to numerical keys to achieve

greater space-efficiency there, too.

Bibliography

[1] English Wikipedia Article Title. `https://dumps.wikimedia.org/enwiki/20190701/enwiki-20190701-all-titles-in-ns0.gz`.

[2] 2007. URL Dataset. `http://law.di.unimi.it/webdata/uk-2007-05/uk2007-05.urls.gz`.

[3] Global database management system (dbms) market report and forecast 2021-2026. 2021.

[4] Austin Appleby. Murmurhash3, 2012. *URL: https://github. com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp*, 2012.

[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534, 2018.

[6] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296, 2009.

[7] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[8] Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. 2021.

[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[10] Paolo Ferragina and Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280, 1999.

[11] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(10):1162–1175, 2020.

[12] Ali Hadian and Thomas Heinis. Shift-table: A low-latency learned index for range queries using model correction. *arXiv preprint arXiv:2101.10457*, 2021.

[13] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: a single-pass learned index. In Rajesh Bordawekar, Oded Shmueli, Nesime Tatbul, and Tin Kam Ho, editors, *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 5:1–5:5. ACM, 2020.

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

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

[17] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[18] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[19] Peter Scott. Murmurhash3.

[20] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the last mile: Efficient learned string indexing, 2021.

[21] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. PLEX: towards practical learned indexing. *CoRR*, abs/2108.05117, 2021.

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

[23] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. pages 1601–1615, 2020.