Analysis of Encoding Schemes for String Indexing

by

Adela Yang

S.B. in Computer Science and Engineering and Mathematics,
Massachusetts Institute of Technology, 2020

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

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

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

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

Analysis of Encoding Schemes for String Indexing

by

Adela Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Lookup of strings into in-memory database indexes is a problem with different considerations from those using integer keys. With their variable sizes, efficiently inserting strings into indexes should account for properties specific to strings. We investigate learning alternate schemes for encoding and inserting strings into index structures such as the adaptive radix tree (ART) and their impact on memory and lookup performance. In this thesis, we examine three different properties of string datasets and perform three experiments aimed at taking advantage of these properties. While using a character frequency based encoding was successful in increasing throughput on a theoretical read-heavy workload, it did not preserve lexicographical order and is unlikely to be useful in most workloads. Meanwhile, the experiments that did preserve lexicographical order were unsuccessful in demonstrating space or throughput improvements. We suggest improvements on these approaches for further experimentation.

Thesis Supervisor: Tim Kraska
Title: Associate Professor

Acknowledgments

This thesis could not have been written without support and advice from many people.

First, many thanks to my advisor, Tim Kraska, who provided a great deal of advice and feedback on this project as it evolved through the last year. I am also grateful for support and guidance I received from Kapil Vaidya. The mentorship I received throughout this project has taught me a lot about research, and it has been an invaluable experience.

This thesis projected was conducted and written during a pandemic, and I am thankful for the friends who have helped support me and each other through the last year: Xiaolu Guo, Stephanie Hu, Michelle Tan, Jessica Chen, Emily Cai, my friends in Spaghetti raiding, and the entirety of the Beam Team. Last and certainly not least, thank you Mom, Dad, and Teresa.

Contents

List of Figures

Chapter 1

Introduction

In recent decades, the increased presence of online business has dramatically affected the need for fast and efficient OLTP applications. The execution of such workloads often deals with in-memory search trees and indexes on string keys. However, fast and efficient lookup of strings into in-memory database indexes is a problem with different considerations from those using integer keys. With their variable sizes, efficiently inserting strings must account for different properties as compared to indexing integer keys.

Current state-of-the art structures used for string indexing generally use a trie structure. A trie is a type of search tree where the character information is stored in branches between nodes. Keys with the same node in their parents will have the same prefix. Tries generally deal well with variable length data, as its height does not scale with the number of keys inserted; searching a trie takes time proportional to the length of the search key. Unlike other tree structures, insertions and deletions do not require balancing operations. Consequently, they are a common structure used for storing string data.

One such trie structure with good performance on insertion and lookup is the adaptive radix tree (ART) [11]. ART supports efficient insertions and deletions and addresses space inefficiency of tries by having hard-tuned node sizes. It adaptively chooses from four different node sizes as data is inserted and deleted. Furthermore, ART uses path compression and lazy expansion to further reduce space used. The performance is comparable to hash tables on dense datasets. ART's efficiency and low memory footprint make it a state-of-the-art index structure for in-memory string indexing.

The goal of this thesis is to explore methods for increasing the efficiency of trie structures such as ART by utilizing methods to learn the distribution of the datasets used on them. An area we have chosen to focus on is encoding strings prior to insertion into indexes.

String compression is a simple method for reducing the size of the data traveling through index structures. Depending on whether or not the original data must be easily recoverable, this is a useful option for achieving insertion and lookup efficiency. Regarding string compression for insertion into index structure, range queries must be supported, so a good compression algorithm must also preserve lexicographical ordering of the data.

The High-speed Order Preserving Encoder (HOPE) is one such encoder that is optimized for compressing database index keys [13]. HOPE is a dictionary-based compressor that preserves order of keys. The paper defines a string axis model to characterize dictionary encoding schemes. This model's most important properties are that it is complete, uniquely decodable, and order-preserving. The model lays out strings on a single axis in order, and defines intervals between which codes are assigned. Intervals can be fixed or variable length, and the assigned codes can also be fixed or variable length. Different combinations of interval and code type categorize four different types of dictionary encoders:

1. Fixed-Length Interval Fixed-Length Code (FIFC): ASCII
2. Fixed-Length Interval, Variable-Length Code (FIVC): Hu-Tucker encoding (order-preserving) [7], Huffman encoding (non order-preserving) [8]
3. Variable-Length Interval, Fixed-Length Code (VIFC): Antoshenkov et al.'s string compression algorithm [2]
4. Variable-Length Interval, Variable-Length Code (VIVC): HOPE's N-Grams encoding scheme

Encoders take advantage of the entropy in string distribution and assign shorter codes to more frequently-accessed intervals. While fixed-length codes are usually easier to decode, in our experiments we assume heavy read-only workloads, thus removing the need for efficient decoding. Thus, we focus on the VIVC scheme described in HOPE, as it provides higher compression rates.

HOPE's uses 3-Grams and 4-Grams as VIVC encoding schemes, which we will refer to in general as N-Grams. For a given value of n, the algorithm uses n-character strings as interval boundaries. Given a dictionary size d, the algorithm samples the data to select the top $d/2$ occurring n-character strings. Each of these strings receives an entry in the dictionary. Additionally, each gap between two lexicographical consecutive strings receives an entry in the dictionary to cover the gap. These use Hu-Tucker codes to assign codes.

In this thesis, we explore and analyze modifications to both ART and N-Grams, and we observe their effect on the efficiency of inserting keys into an index. We perform three experiments with the goal of improving lookup throughput in an ART index. Our approaches are as follows: (i) using a character frequency based encoding and modifying ART to take advantage of this encoding, (ii) encoding using longer substrings, and (iii) weighting suffixes more heavily within the compression algorithm. In the following section, we will describe in more detail the setup and methods used by these experiments.

Chapter 2

Related Work and Discussion

There exists a wide body of research relating to improving index performance for string keys.

2.1 Traditional index structures

The Height-Optimized Tree (HOT) [3] optimizes its internal structure around the height of the tree. It adaptive adjusts for high fan-out consistently throughout the tree by varying the size of the prefix considered at each node. Low height of the tree indicates fewer nodes traveled.

Masstree [12] uses B+ trees in a trie-like structure. Each B+ tree deals with a fixed length prefix, and the structure overall is capable of handling arbitrary-length keys.

Both HOT and Masstree focused on reducing tree depth via algorithmic means. In contrast, our experiments looked to key compression combined with ART to reduce key length and thus tree depth.

2.2 Learned indexes

Recent research has explored the use of machine learning to help predict and identify where data may be located in an index structure. Such learned indexes often have reduced sizes and faster lookup times. However, updates to the data often require retraining the model, so they best support a read-heavy workload.

Kraska et al.'s Learned Index [10] uses recursive model indexes (RMI) with staged models to predict where data is located in a sorted array. RMIs model the CDF of a dataset, narrowing down the subrange a key is predicted to be in and performing a local search within an error bound. These structures feature faster lookups than traditional indexes, but are suspect to requiring retraining upon inserts or updates.

ALEX [5] is a learned index based on B+ Tree. Like Learned Index, it uses RMIs to predict key location. However, it dynamically adjusts the number of levels and number of models at each level in response to the workload, allowing it more flexibility in modeling the data distribution. By using a cost model to adjust its RMI structure, it avoids the need for re-tuning for different datasets or workloads.

Self-Tuning ART (START) [6] is a modification on top of traditional ART that uses multilevel nodes to reduce tree height. START develops a cost model based on ART metrics collected on the target machine, and uses that to determine how to place multilevel nodes. It consistently performs better than ART on read-only workloads.

RadixSpline (RS) [9] is a learned index that, like RMIs, models the CDF of a dataset. RS builds a spline to fit the underlying data, then uses the spline points as entries in a radix table. RS can be constructed in a single pass and has few parameters, thus making it easier to tune.

2.3 Key compression

Orthogonal to improvements on the efficiency of the index structure, another approach is to compress input keys before they are inserted into a search structure. Compression of keys reduces the overall size of the amount of data inserted into the index structure. Optimizations can also be made around how frequently data is accessed and by how much it is compressed.

Column-store DBMSs often use string compression to replace variable-length strings with fixed-length integers, which undergo further compression to reduce size. These use dictionary compression to handle encoding. Abadi et al. presented the effect of various compression schemes on columnar databases for improving query performance [1]. Processing on top of already compressed data led to significant improvements in the system. Furthermore, order-preserving dictionary encoding, while having a worse compression rate, can still provide benefits to query performance, even when domain size is unknown or variable [4].

Chapter 3

Methods

To investigate improvements to ART, we pursued three avenues of investigation. The first focused on an alternate character frequency based encoding, the second focused on using longer substrings for N-Gram encoding, and the third focused on the impact of changing N-Gram encoding weights.

Each of the following experiments were performed with two datasets:

1. Email: 144 million keys, average length 21 bytes, max length 30 bytes
2. URL: 72 million keys, average length 55 bytes, max length 254 bytes

The benchmarks collected for each experiment were collected by running 10 trials for each variant, and averaging the metric collected. Metrics collected included compression rate, keys looked up per second, and memory usage of the index structure.

3.1 Character frequency based encoding

The first experiment investigated a frequency based encoding for the string data to insert into the index structure. We re-encoded the dataset prior to insertion, and measured the lookup speed. We altered ART to take advantage of the frequency based encoding by taking faster paths when encountering more frequently occurring characters while traversing the tree. We expected that this would result in faster lookup times, at the cost of increased memory usage.

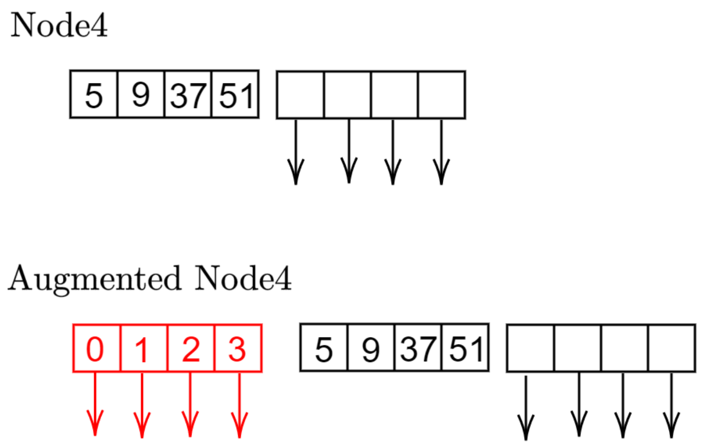

Figure 3-1: Comparison of the inner structure of original ART Node4 and our augmented Node4

3.1.1 Implementation

The data was processed and re-encoded according to the frequency of characters across the dataset. Supposing an ordering of the most to least frequent characters as $c_0, c_1, \ldots$, each character was encoded as $\pi(c_i) = i + 1$. To allow ART to take advantage of this encoding, we augmented each of the original ART nodes with an additional size n array containing pointers to children. Traversing the tree would directly follow one of these pointers if the character being looked up at that node was less than n.

Te original ART nodes each have some internal structure determining how to find the next child. Figure 3-1 demonstrates the augmented Node4 structure with $n = 4$. The additional array would be used to travel to the next child if the next character was less than n.

The experiment was run on our two previously mentioned datasets, with the size of the augmented array taking the values $n = 4, 8, 16, 24, 32$. We measured the time it took to lookup the dataset once fully inserted into the augmented ART structure, as well as the amount of space used by the new structure.

3.2 N-Gram analysis

The second experiment examined the N-Gram encodings described in the HOPE paper. The HOPE paper used 3-Grams and 4-Grams as VIVC encoding schemes. We decided to additionally implement 5-Grams. We expected that using longer sequences as intervals would provide more compression, thus leading to better performance within an index structure.

The HOPE paper indicated that 4-Grams encoding performance suffered compared to 3-Grams encoding performance. However, it was not clear whether or not the longer sampling length would take advantage of a larger dictionary size to become more efficient for index insertion and lookups as compared to a shorter sampling length. As a result, our experiment also varied dictionary size. We expected that using a larger dictionary size would result in more efficient codes, thus improving lookup time.

3.2.1 Implementation

Like 3-Grams and 4-Grams, given a dictionary size d, our implementation of 5-Grams sampled keys from the dataset and selected the $d/2$ most frequently occurring length 5 substrings. We built a dictionary encoder around these substrings as previously described, assigning codes to the $d/2$ sampled keys and the intervals between them.

We measured the performance comparison between 3-Grams, 4-Grams, and 5-Grams across encoding size, ART lookup performance, and ART memory usage. The performance metrics were gathered for dictionary values of $d = 5000, 10000, 20000, 40000$ and compared the differences in performance.

3.3 N-Grams frequency weighting

The final experiment investigated modifying the N-Grams encoding algorithm to account for n-gram sequence location within the original string. The motivation for this originated from the observation that in ART, prefixes of strings have lower impact on the size of ART as compared to suffixes. The latter half of the strings, because of exponential fanout, have a heavy impact on the size of ART. We altered the N-Grams encoding algorithm to weight n-grams appearing at the end of keys by different factors, and examined how they altered the performance of encoding and insertion into ART.

Our hypothesis was that by weighting the latter half of strings, the encoding algorithm would assign codes with shorter length more frequently to sequences at the end of strings, which would affect ART memory usage. We expected that lookup throughput would improve, at the cost of increased ART memory usage. Since the encoding algorithm was altered into a less than optimal encoding function, we expected that the compression rate of the new algorithm would suffer as compared to the original.

3.3.1 Implementation

As previously described, the N-Grams encoding algorithm samples keys from the dataset and selects the $n/2$ most frequently occurring length N substrings. We altered the method of counting the substrings to weight n-grams appearing at the end of keys more heavily by a factor f. Given a length n string $s_0 s_1 \dots s_n$, for any n-gram beginning with the character s_i with $i > n/2$, we increase its weight in the n-grams by counting it as occurring f times.

For this experiment, we used 3-Gram encoding and a dictionary size of 5000. In our experiments, the range of values for f were $f = 5, 10, 20, 50, 100$. Performance metrics measured were encoder size, lookup throughput, and ART size.

Chapter 4

Evaluation

4.1 Character-based frequency encoding

This experiment re-encoded the data according to the frequency of each character. We modified the ART nodes to short-circuit if the next character found was a highly frequent character. We expected that the throughput of lookups would increase under this scheme, alongside a slight increase in memory usage.

4.1.1 Memory usage

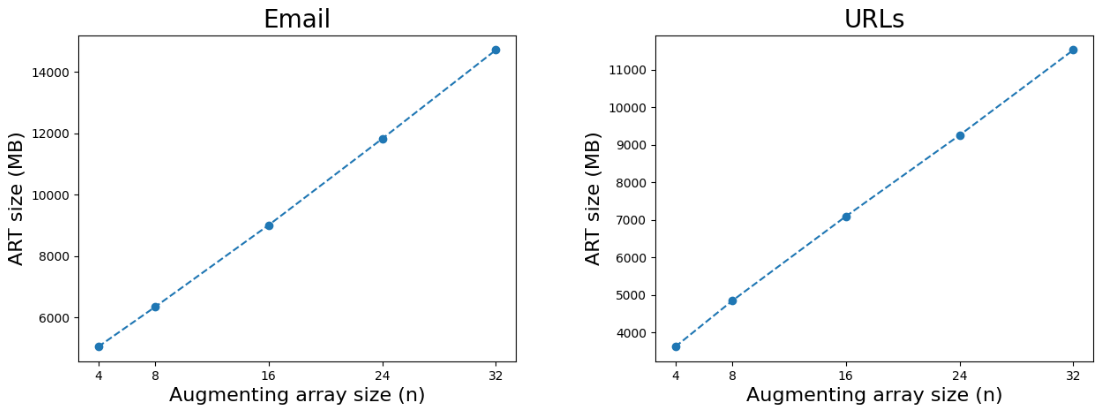

Figure 4-1: Dictionary encoding sizes across augmenting array sizes

Figure 4-1 demonstrates that as the number of characters used for short-circuiting

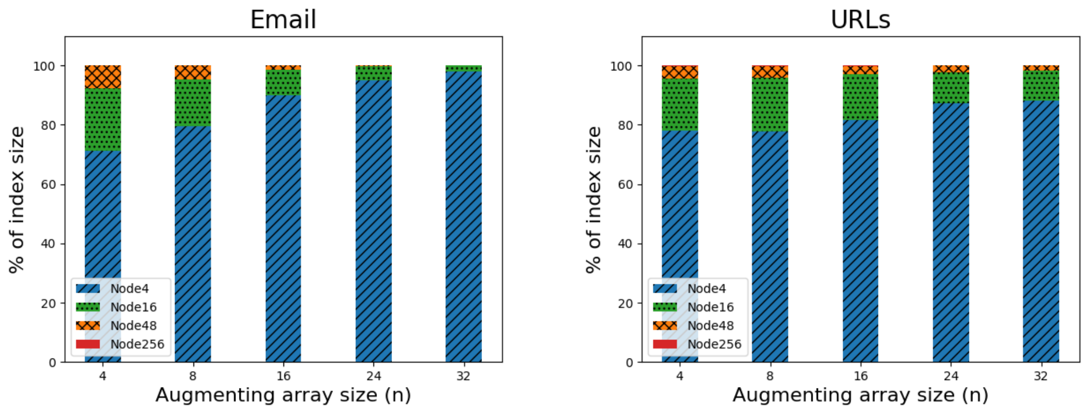

Figure 4-2: Space used by each type of augmented node

increased, the memory used by ART increased in a roughly linear manner. Since we augmented each node with further data structures, this was expected. Investigating the distribution of memory among node types, Figure 4-2 demonstrates that as n increased, data concentrated in the augmented Node4 types.

4.1.2 Throughput

4.1.3 Memory usage

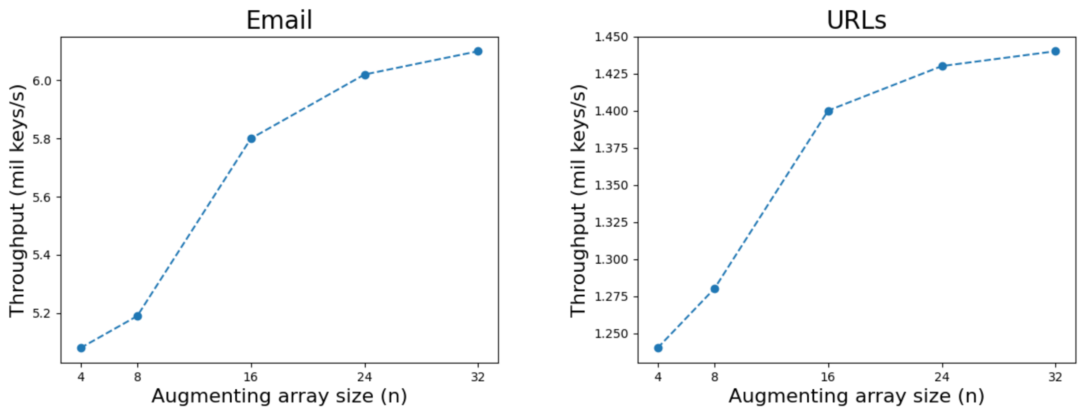

Figure 4-3: Throughput across augmenting array sizes

As seen in Figure 4-3, short-circuiting by the top n characters provides an improvement to throughput that plateaus near $n = 16$. The augmenting array size describes the number of the more frequent characters that were skipped. For the email dataset, increasing the size of the augmenting array from 4 to 32 led to a 15% increase in throughput. For the URL dataset, increasing the size of the augmenting array from 4 to 32 led to a 16% increase in throughput.

4.1.4 Discussion

We expected the throughput to improve and the memory usage to worsen. Our results indicate that throughput did increase as the number of characters directly indexed increased. However, the increase in memory usage with size of the augmenting array was far more than expected.

We found that the memory usage linearly increased with n, and tended to not utilize the higher levels of ART nodes efficiently. These observations indicate that augmenting ART in this manner had the effect of giving already tuned node sizes additional children, which had the result of concentrating children at lower levels of nodes without reaching the threshold to upgrade into the next node type. Further experimentation with these augmented ART nodes could consider tuning the upgrade thresholds for the nodes, or constructing a model from the data for when to upgrade.

Additionally, as mentioned previously, this type of encoding does not preserve lexicographical order, so this model will not support range queries. Consequently, this augmented ART structure will support a read-heavy workload that only requires point lookups and does not have a need for space efficiency.

4.2 N-Gram analysis

In this experiment, we implemented 5-Grams, and measured the performance of 3-Grams, 4-Grams, and 5-Grams with varying dictionary size. We expected that 5-Grams, by encoding longer substrings, would result in a higher compression rate and thus increased lookup throughput and lower memory usage as compared to 3-Grams and 4-Grams.

We found that on the same dictionary size, 3-Grams used the least memory for the encoding dictionary. Once the encoded keys were inserted into ART, 3-Grams also had the fastest lookup speed and used the least memory. The total memory used by the index structure consistently spanned a range of approximately 100MB in all trials across the same dictionary size, and indicated roughly linear growth.

4.2.1 Memory usage

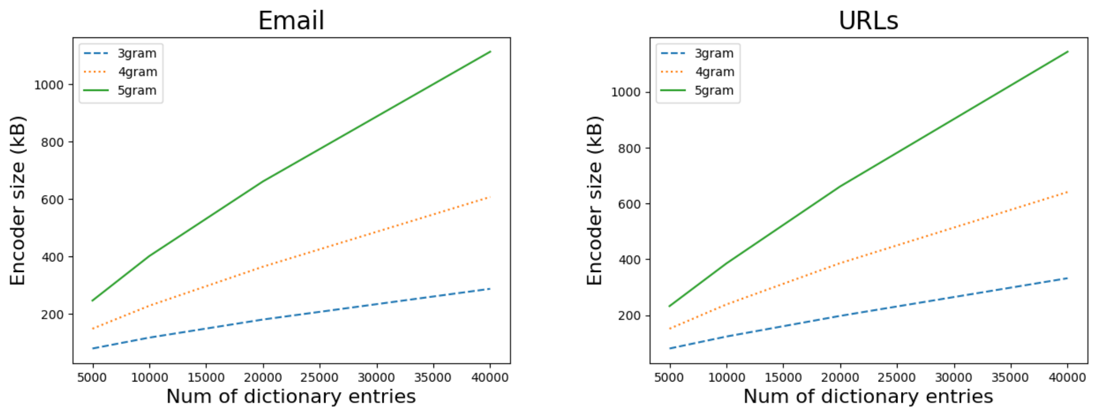

Figure 4-4: Dictionary encoding sizes across N-Gram encodings

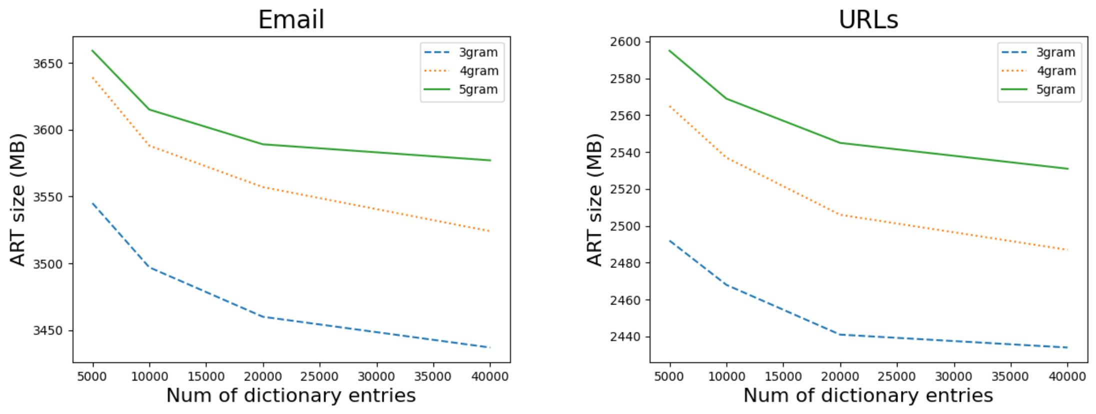

Figure 4-5: Index sizes across N-Gram encodings

Reflecting the conclusions of the HOPE paper, the dictionary encoding size grew roughly linearly with increasing dictionary size, as can be seen in Figure 4-4. Within the trials with fixed dictionary size and different N-Gram encoding choices, the dic-

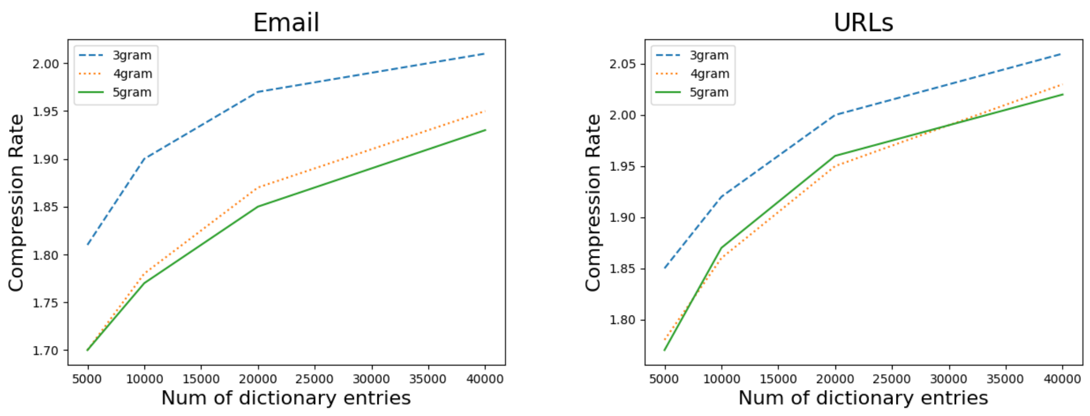

Figure 4-6: Compression rate across N-Gram encodings

tionary space usage also grew nonlinearly. However, the differences in encoding dictionary size were comparatively negligible to the range of memory usage of the index structure. The maximum dictionary encoding size found in these experiments was less than 1.5 MB.

The index sizes, as seen in Figure 4-5, decreased with increasing dictionary size, reflecting the increased compression rate that larger dictionaries can perform. Increasing the dictionary size from 5000 to 40000 reduced ART size by 2.5-3% in both datasets. However, we observed that on the same dictionary size, 5-Grams did not cause ART to use less memory in comparison to 3-Grams, indicating that using longer ngrams for compression did not aid in reducing memory usage.

This was further reflected in the compression rate metrics. As seen in Figure 4-6, increasing the number of dictionary entires was linked to a higher compression rate. However, 3-Grams performed markedly better than 4-Grams and 5-Grams, both of which have near similar compression rates at all dictionary sizes tested.

4.2.2 Throughput

The metrics gathered under using 5-Gram encoding show that increasing dictionary size allows for higher throughput. With the email dataset, the throughput was best under the 3-Gram encoding, and across all N-Grams, improved as the number of

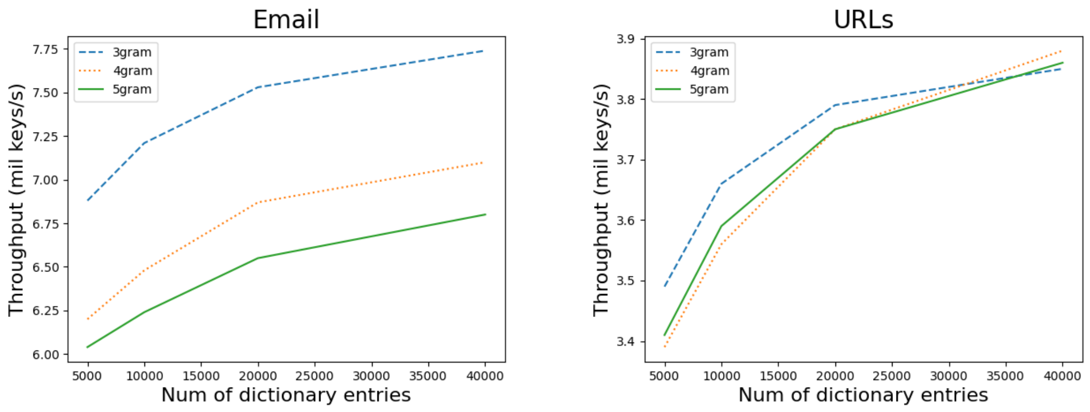

Figure 4-7: Throughput across N-Gram encodings

dictionary entries. We can likely link these results to the impact of the compression rate – higher compression rates result in shorter keys, which take less time to lookup in the search tree.

For the email dataset, the throughput differences between different N-Grams are quite stark, with 3-Grams performing noticeably better than 4-Grams or 5-Grams. However, for the URL dataset, the difference is not quite as severe, with 5-Grams and 4-Grams having just slightly better performance than 3-Gram, as can be seen in Figure 4-7.

4.2.3 Discussion

We expected that 5-Grams would have better compression rates than 3-Grams or 4-Grams, and thus lead to better performance from ART. However, the metrics we gathered indicate that this was not the case. 3-Grams had the best compression rate out of the N-Gram encodings used. In fact, the compression rate for 4-Grams and 5-Grams were generally quite close regardless of the number of dictionary entries. It is possible that the dictionary encoding algorithm had too many unique 4-grams or 5-grams to achieve a good encoding. However, further experimentation using a larger dataset or different sampling methods would be required to confirm this.

The experiments also indicated that compression alone may have reduced effects on datasets with longer keys – suggesting that further efficiency improvements through key compression alone have a limit to them. Further algorithmic improvements involving compression are likely to require consideration of the index structure being used.

4.3 N-Grams frequency weighting

In this experiment, we adjusted the N-Grams encoding algorithm to weight suffixes more frequently when counting ngrams. We expected that increasing suffix weight would result in higher lookup throughput with decreased memory usage, while having a worse compression rate.

4.3.1 Memory usage

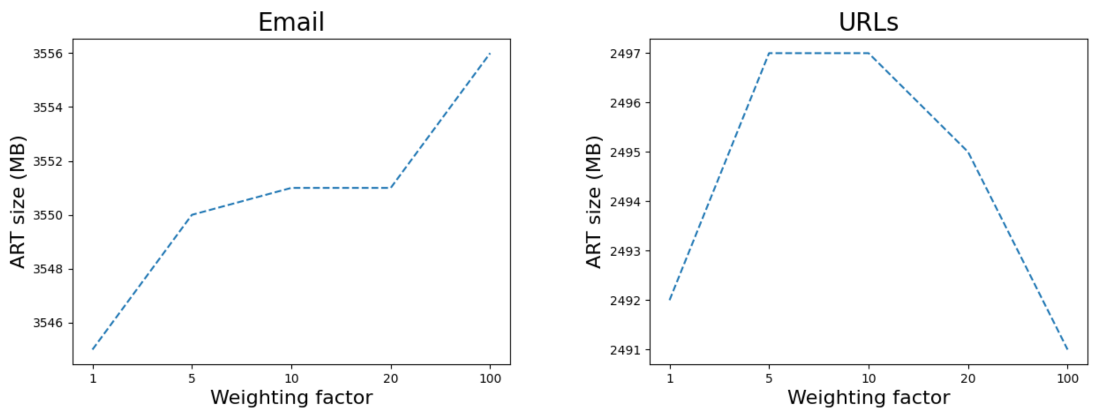

Figure 4-8: Index sizes across weighting factors

In Figure 4-8, we see that increasing the weighting factor had little effect on the size of ART. In both datasets, the index sizes fluctuated over a range of approximately 10MB, which was less than 1% of the total index size.

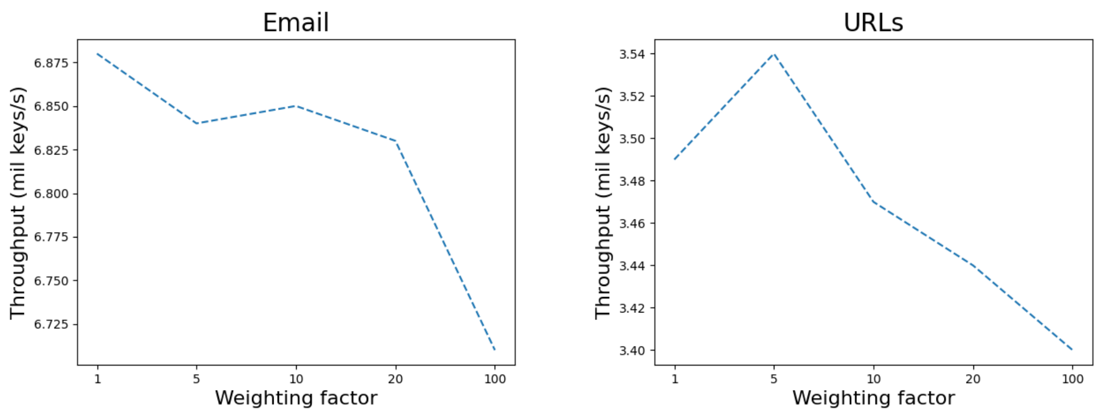

Figure 4-9: Throughput across weighting factors

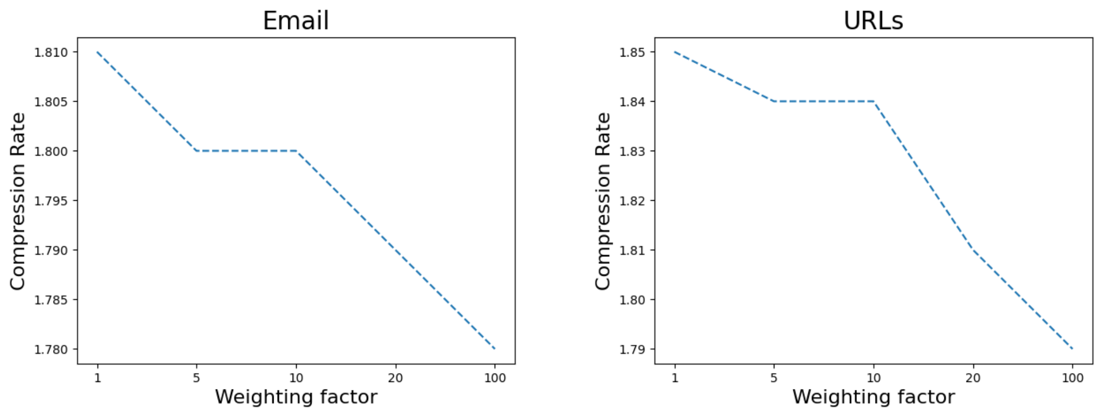

Figure 4-10: Compression rate across weighting factors

4.3.2 Throughput

Figure 4-9 shows that the throughput of lookups decreased as weighting factor increased. On both datasets, increasing the weighting factor from 1 to 100 led to a throughput decrease of approximately 2.5%. The change in throughput roughly matches the difference in compression rate across weighting factors, seen in 4-10. Increasing the weighting factor from 1 to 100 led to compression rate decreasing by 1.6% and 3.2% in the email and URL datasets, respectively.

4.3.3 Discussion

We expected that weighting the suffixes would lead to improved throughput at the cost of compression rate. As expected, giving the suffixes of keys more weight in the N-Grams encoding algorithm led a worse compression rate. However, the change in algorithm did not lead to increased throughput with increased weighting.

One possible explanation for why weighting suffixes had little impact may be that the datasets already use strings with common ngrams in the suffixes. Emails and URLs will tend to have similar endings due to sharing domain names. Consequently, it is likely that the suffixes already are weighted more frequently when using the unmodified N-Grams encoding algorithm. Examining this further would likely require experimentation with different datasets without this sort of suffix skew. Additionally, observing the effect of weighting prefixes as well as suffixes may prove insightful.

Another possible coinciding explanation for why weighting suffixes had little impact may be that more efficient compression for suffixes may have been counterbalanced by less efficient compression for the prefixes. As a result, the two may have balanced out and caused little change. Further experiments to verify this would likely require observing the compression rates of prefixes and suffixes separately. An extension of this would likely involve learning the best point within a string to split the weighting of prefix or suffix.

Chapter 5

Conclusion

In this thesis, we presented three experiments that aimed to improve throughput of lookups in ART by changing the encoding schemes used for the data. While using character frequency based encoding helped to improve throughput, the vastly increased memory usage and non-order-preserving nature of the encoding makes it impracticable for common use. While we expected that using 5-Grams for encoding would lead to better performance, we saw that 5-Grams generally performed worse than 3-Grams in both throughput and memory usage. Finally, modifying N-Gram encoding to weight suffixes more frequently did not lead to performance improvements as we expected, instead leading to slightly worse performance results. We suggest some further lines of query stemming from the observations of these experiments, including examining the effect of separately weighting prefixes and suffixes of the data, possibly via learning a model.

Bibliography

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682, New York, NY, USA, 2006. Association for Computing Machinery.

[2] Gennady Antoshenkov, David B. Lomet, and James Murray. Order preserving compression. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, pages 655–663. IEEE Computer Society, 1996.

[3] 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*, SIGMOD '18, page 521–534, New York, NY, USA, 2018. Association for Computing Machinery.

[4] 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*, SIGMOD '09, page 283–296, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David B. Lomet. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.

[6] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. Start — self-tuning adaptive radix tree. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 147–153, 2020.

[7] Te C Hu and Alan C Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[8] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[9] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: A single-pass learned index. *CoRR*, abs/2004.14541, 2020.

[10] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[11] 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, 2013.

[12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. *CoRR*, abs/2003.02391, 2020.