Efficient sorting of search results by string attributes

Nicholas Sherlock
Department of Computer Science
University of Otago
Otago 9054 New Zealand
nsherloc@cs.otago.ac.nz

Andrew Trotman
Department of Computer Science
University of Otago
Otago 9054 New Zealand
andrew@cs.otago.ac.nz

Abstract *It is sometimes required to order search results using textual document attributes such as titles. This is problematic for performance because of the memory required to store these long text strings at indexing and search time. We create a method for compressing strings which may be used for approximate ordering of search results on textual attributes. We create a metric for analyzing its performance. We then use this metric to show that, for document collections containing tens of millions of documents, we can sort document titles using 64-bits of storage per title to within 100 positions of error per document.*

Keywords Information Retrieval, Web Documents, Digital Libraries

1 Introduction

A document search engine typically takes a search query provided by the user and generates a list of documents which contain some or all of the terms in the query. This list of documents is sorted in a particular order, defined by a "ranking function". Common ranking functions assign an importance to each term in the search query using some metric, then for each term in the query, apply that importance to the number of occurences of the term in each document to give each document a score. The list of documents is then sorted using this score to present documents to the user with the highest scores first.

There are several such ranking functions that are of interest in search over online discussion forums. For example, the online forum software "phpBB" offers users the ability to order their results by document attributes such as post time, author name, forum name, topic title, and post title. In the case of text fields like names and titles, this is problematic, as it requires the search engine to have these fields available in text form for comparison sorting at search time. This consumes a large amount of memory. For example, in a collection of 14.8 million forum posts, simply storing the post title for each post requires 500 megabytes of memory.

Proceedings of the 16th Australasian Document Computing Symposium, Canberra, Australia, 2 December 2011.

In addition, the search engine must allocate memory to an index structure which allows it to efficiently retrieve those post titles by document index, which, using a simplistic scheme with 4 bytes required per document offset, would require an additional 50 megabytes of storage.

For search terms which occur in many documents, most of the memory allocated to storing text fields like post titles must be examined during result list sorting. As 550 megabytes is vastly larger than the cache memory available inside the CPU, sorting the list of documents by post title requires the CPU to load that data from main memory, which adds substantial latency to query processing and competes for memory bandwidth with other processes running on the same system.

In this paper, we examine several methods which can be used to reduce the memory requirements for storing document attributes in a form suitable for sorting. We examine the tradeoffs which can be made between sorting accuracy and the memory required for storage at search time. We then demonstrate the accuracy of our methods by applying them to a corpus of 14.8 million discussion posts taken from an online discussion forum called "Chicken Smoothie", and 22 million posts from the online discussion forum at "Ancestry.com".

2 Ranking

First, let us examine the way that the search engine calculates ranking scores for documents. For each search term in the query, an index over the document collection is consulted. This index maps the term onto a list of document IDs which contain that term, along with extra information about that term's appearance in each document (for example, a count of the number of times that term appears in the document). This information is passed to a "ranking function", which uses it to generate a ranking score for each document.

The scores generated by the ranking function for each term in the search query are combined together in a structure called the "accumulator". The accumulator can be represented as an array of integers of a fixed size, one integer for each document in the collection. Every element in the accumulator is initialised to zero at the beginning of the search. Then, the value returned from

the ranking function for each term in the search query is typically added (using simple arithmetic addition) to the value already present in the accumulator for that document, to give a new score for the document.

At the end of the search process, the search engine's accumulator is sorted in descending order, which brings the documents which the ranking function scored the highest (which are hoped to be the "most relevant" documents to the user's information need, based on their search query) to the front of the result list.

2.1 Pregenerated ranking

If a ranking function's value does not depend on the content of the user's query, then instead of computing it when a document is located at search time, it can be computed when the document index is first created (at "indexing time"). This pregenerated ranking can be stored as a file (a "pregen") consisting of an array of integers, one for each document in the collection. When the search engine starts up, it can read each of its pregen files into memory. Then at search time, rather than performing any computations to calculate the ranking score for a document, the search engine can simply read the precalculated value stored in the pregen and store that value directly into its accumulator to be sorted. This technique allows the pregenerated ranking to take advantage of the search engine's existing implementation of accumulator sorting. For example, the search engine may already implement special optimisations like partial sorting of accumulators for the case where only the top-K sorted documents are required[5].

If a ranking function is particularly expensive to compute (compared to reading a precomputed value from memory), speed gains can be made at search time, at the expense of the extra memory required to store the pregens. For example, one ranking function which is expensive to compute, but independent of the user's query, is Google's PageRank algorithm[4]. This ranking function essentially assigns a score to web documents based on how many incoming links they have.

If we could compute a compact pregenerated ranking for each of the document attributes that our users want to sort on, we could improve the speed at search time (by sorting the smaller pregen instead of the longer original strings) and reduce the memory required to store those text attributes. For example, if we could fit the 14.8 million post titles in the Chicken Smoothie collection into 64-bit integers, the pregenerated ranking would only require 110 megabytes of memory to store. That would allow the search engine to save 440 megabytes of memory that would otherwise be required to store complete post titles and an index structure for those titles.

3 Generating pregens by sorting

One way in which a pregenerated ranking on a textual document attribute can be generated is by sorting. At indexing time, the text attributes for each document are extracted and stored in temporary memory. When the indexing of the document collection is complete, the document attributes are sorted using a string comparison function. Each document's position in the list of sorted attributes then directly becomes its pregenerated ranking score. In this way, we generate a maximally-compact set of ranking scores for the document collection (having exactly as many distinct values as there are distinct attributes in the collection), which also perfectly encodes the relative ordering of the documents in the sorted list of attribute text. If each integer in the pregen is 32 bits large, approximately 4.3 billion distinct documents can be perfectly ranked using this method (2^{32}).

This method has several drawbacks. Because every attribute of the documents which must be sorted needs to be available at the end of indexing time, we must either allocate enough memory to store those values (e.g. 550MB for post titles in the Chicken Smoothie corpus), or avoid allocating memory by storing those values in some sort of disk structure (requiring at least 1 gigabyte of additional disk I/O, as we must both write the values to disk and read them in again).

Another drawback is that each value in the generated ranking is dependent on the attributes of every document in the collection. If a later change to the document collection results in the first post in the ranking being deleted (say, "aardvarks and apples"), every pregenerated ranking function for the collection is now invalid and must be recomputed.

This is also a problem in distributed search, where a document collection may be split into several chunks, with each chunk being indexed by a separate computer. The pregens created by each index will be incompatible, since they are based on different, incomplete fragments of the total document collection. If, during searching, a search engine consults several such distributed indexes, and retrieves a list of documents from each with corresponding scores from their pregens, it will be unable to merge those results using just the values of the pregens to create a list which is sorted by the ordering over the complete collection.

A better method for generating pregens would be able to compute a ranking score for a document attribute immediately upon reading it (which would eliminate the requirement to store those values until the end of indexing), and the computed value would be independent of all other documents in the collection (which would provide consistency between different distributed fragments of the complete document collection).

4 Approximate pregens

The immediate issue with a method that directly computes one ranking score from one attribute (examining no other attributes) is that the pigeonhole princi-

ple shows that the resulting pregen cannot, in general, perfectly rank the documents, as the previous approach could. The average post title's length in the Chicken Smoothie corpus is 35 bytes, and we hope to reduce that to an 8-byte integer for each document. Even in the best case where the resulting values are evenly distributed, each post title (in the universe of possible post titles) would alias with 1.1×10^{65} different titles (i.e. $\frac{2^{35\times8}}{2^{8\times8}}$), destroying the information about the relative ordering of those titles.

In order for this approximate pregen method to be effective, we hope for two things to be true. First, that there is significantly less information in the post titles than their raw byte counts would suggest, and secondly that the error in the resulting ranking is sufficiently small that our users will not notice the difference compared to a perfect ranking.

5 Baseline

The accuracy of an approximate pregenerated ranking can be measured against a baseline which is considered "perfectly ranked". This perfect ranking corresponds to the user's expectations for document ordering in the search results, which allows users to locate the documents that they are interested in in the results.

The Chicken Smoothie corpus consists of 14.8 million forum posts, which are overwhelmingly written in the English language. Topics in the Chicken Smoothie's "International Forum", which is intended to be a forum for posts written in non-English languages, comprise only 0.1% of the total number of topics in the corpus. Because of this language bias, it is reasonable for the perfect ranking to ignore the relative order of non-English characters, as most users are not aware of, and do not take advantage of, the "correct" ranking for these characters. We will also consider that accented characters are equivalent to non-accented characters for the purpose of sorting. The ordering rules for accented characters are different even among common languages such as French and German, so in a multilingual collection like Chicken Smoothie, there is no one collating sequence for accented characters which is useful for all users. We consider that uppercase letters are equivalent to lowercase letters, as this difference does not play a significant role in English text. For this baseline, we consider the relative ordering of punctuation characters to be significant, and require it to follow the same ordering as in ASCII. We will call this baseline the "standard baseline".

We also consider the performance of our pregen ranking against a more limited baseline. This baseline contains only the ASCII alphanumeric characters and the space character, ignoring all punctuation marks and non-English characters. The rationale of this baseline is that users are unaware of the correct ordering of ASCII punctuation characters or other special characters, and so are not able to take advantage of that ordering when reading the search results. We will call this baseline the "restricted baseline".

The baseline is defined by a comparison function which, when applied to the attributes of a pair of documents, either declares that the two documents are considered equivalent for the purposes of ranking (that is, their characters do not differ in a way that the baseline considers significant), or identifies that one of the documents must occur before the other in the final ranking. This comparison function is the collating sequence of the document collection.

6 Character encoding

The Chicken Smoothie corpus is Unicode text encoded using "UTF-8", which encodes the majority of the characters used in English text into single bytes whose encoding matches ASCII, and encodes more complex characters, such as characters with accents and characters from other languages, into multi-byte sequences. As the discussion in the Chicken Smoothie forum is primarily in the English language, Unicode characters outside the ASCII range are rare (about 0.1% of the byte total). Since ASCII is a 7-bit encoding (128 distinct codepoints), we can expect to perfectly encode 35-byte titles into about 30.6 bytes ($\log_{256}(128^{35})$ bytes). That falls far short of the 8-byte target.

6.1 Standard baseline

Since the standard baseline does not distinguish between uppercase and lowercase characters, we can reduce the number of ASCII codepoints that we encode by 26, by lowercasing the text before we process it. We also consider the fact that a title of a document may not contain any ASCII control codes, since document titles in our collection are single-lines of text (and so contain no newline control characters), and cannot contain tab characters because of web browser limitations. This allows the number of codepoints to be reduced by a further 32 characters. Since the baseline considers the relative order of non-ASCII characters to be unimportant, we merge all those characters into one encoded codepoint, which sorts after any ASCII codepoint. This character encoding has 90 codepoints in total. Using this encoding, we can encode 35-byte titles to text which users consider "perfectly ranked" (compared to the standard baseline) in $\log_{256}(90^{35}) = 28.4$ bytes. We call this simple 90-codepoint encoding the "printable ASCII" text encoding.

We developed a base-40 character encoding, which has distinct codepoints for alphanumeric characters and the space character, but merges all punctuation and non-ASCII characters into the remaining 3 codepoints. Those 3 codepoints are chosen to preserve the relative ordering between the groups of alphanumeric characters and the groups of punctuation characters. The first codepoint is allocated to those punctuation characters occurring before the character '0', the

second codepoint for those characters occurring before the character 'a', and the last codepoint is allocated to the characters which occur after the letter 'z'. The performance of this character encoding depends on punctuation characters being rare enough in the collection that the relative ordering within the groups of punctuation marks which are conflated do not significantly perturb the ranking.

6.2 Restricted baseline

The restricted baseline only contains lowercase alphanumeric characters and the space character. As with the standard baseline, text is converted to lowercase before processing, and characters with accents are converted to their unaccented equivalents. A trivial base-37 encoding can encode every character in the restricted baseline to a distinct codepoint. Base-36 encodes every character in the baseline except space (and so is useful in the extreme circumstance where the encoded string terminates before the first word of input is completed).

We also developed a base-32 encoding, which halves the number of codepoints allocated to storing numeric characters (every second digit from ASCII is conflated with the digit before it). This encoding trades off minor inaccuracy in the ordering of the (relatively rare) numeric characters in exchange for more available precision for encoding more-common characters. Bases which are powers of two are attractive for encoding text, because many of the arithmetic operations required for encoding strings can be reduced to simple bit shifts.

7 Encoded string compression

The characters encoded by our various encoding schemes must be combined together to give an integer for sorting. A simple method of achieving this is "radix encoding". Radix encoding treats the output as an integer of the same base as the number of possible symbols in an encoded character, and adds each encoded character to the output as a digit in that base, with the earliest characters in the string becoming the most significant digits of the encoded integer. If the final encoded character can not completely fit in the encoded integer, it is scaled down.

If the number of symbols used to represent encoded characters is r, and the size of the encoded integer is b bits, we have the following psuedocode for a radix encoder of a string:

$\mathit{chars\ in\ output} := \lfloor \log_r 2^b \rfloor$
$r_{\mathit{last}} := r^{\frac{2^b}{\mathit{chars\ in\ output}}}$
$\mathit{output} := 0$
for $i := 1$ **to** *chars in output* **do**
    $\mathit{output} := \mathit{output} \times r + \mathit{encode}(\mathit{read}())$
od
$\mathit{output} := \mathit{output} \times r_{\mathit{last}} + \frac{\mathit{encode}(\mathit{read}())}{r-1} \times (r_{\mathit{last}} - 1)$

The "read" operation retrieves the next character from the input string, and the "encode" operation applies the chosen character encoding (characters which have no representation in the character encoding are ignored and the next character is read instead). The radix encoder assumes that every symbol is equally likely, so assigns equal-length encodings to each symbol. When the actual probabilities of the symbols being encoded are known, it is possible to choose an encoding which uses shorter strings of bits to represent common symbols, and longer strings of bits to represent infrequent symbols. This allows space characters and vowels, which are the most common characters in our collection of English text, to have short encodings, while the uncommon letters Q and Z have longer encodings.

The most well-known variable-length coding scheme is probably Huffman coding[2]. Huffman coding selects a bit-string representation for each input symbol such that the average length of encoded strings which follow the expected symbol probability distribution is minimized. A similar coding scheme, Hu-Tucker coding[1], is additionally able to preserve the lexicographic ordering of the resulting encoded strings, and so would be suitable for pregen encoding. However, as both Huffman coding and Hu-Tucker coding assign codes for input symbols which are an integer number of bits long, they only accurately model input symbol probabilities which are negative powers of two. Input symbol distributions which deviate from that pattern are encoded slightly less efficiently than theoretically possible.

There are coding schemes capable of assigning representations to input symbols which are effectively a fractional number of bits long, while also preserving the lexicographic ordering of input strings. "Arithmetic encoding"[6] is one such scheme. The goal of the arithmetic encoder is to construct an interval which represents the string of symbols being encoded, then to arbitrarily choose an integer which lies in this interval. This integer represents the encoded string.

For example, consider an encoder which encodes strings of base-4 characters (a, b, c, d) with relative probabilities (4, 2, 1, 1) into a 6-bit integer. The initial interval is the full range of the output integer, $[0..64)$. This interval is subdivided into 4 segments, one segment for each of the possible first symbols of the string, according to the relative probabilities of the characters. The first encoded character is therefore represented by one of the intervals $[0..32)$, $[32..48)$, $[48..56)$, $[56..64)$. Imagine that the first character is an "a". The current interval now becomes $[0..32)$. To encode the next character, the interval is subdivided again, into $[0..16]$, $[16..24)$, $[24..28)$, $[28..32)$. If the next character in the string is a "c", the current interval now becomes $[24..28)$. At this point, there is not enough precision left in the output integer for each range to be distinct: $[24..26)$, $[26..27)$, $[27..27.5)$,

$[27.5..28)$. If the next character is an "a" or "b", it will be unambiguously encoded, but "c" and "d" are no longer distinct.

Each time an interval is chosen which lies entirely within the smaller half of the current interval, it is the same as choosing a zero bit as the next-most significant bit of the output integer (because the value of the midpoint of the interval, represented by a 1 bit in the output integer, did not need to be added). Conversely, choosing an interval in the upper half of the current interval outputs a 1 bit. Small subintervals, which represent less-likely strings, require more bits to specify as the interval must be bisected more times in order to accurately specify their position in the initial interval.

This encoding scheme is attractive as a pregen string compressor, as the resulting integer preserves the relative ordering between the input symbols—if the symbol "b" occurs after the symbol "a" in the character encoding being used, then all arithmetic-encoded strings which begin with "b" will be larger than those which begin with "a".

We have chosen to use a simple unigram model for English text, which assigns a relative probability to each possible symbol in the encoded characterset. We derived these probabilities from the Chicken Smoothie post title corpus. The measured symbol frequencies for the ASCII printable encoding are shown in figure 1. It is likely that a more advanced English character probability model (such as a bigram or higher order model) would be more effective at compressing text. For example, Moffat and Turpin[3] suggest that an order-2 model could encode English text at around 2.5 bits per character, which would allow strings of about 26 characters to be stored in a 64-bit pregen value. We hope to investigate this in the future.

8 Metric

In order to measure the performance of the approximate pregenerated rankings against the baselines, a metric must be defined. We have chosen to use the Kendall rank correlation coefficient (τ). This metric compares the relative rank order of every pair of documents in the two rankings, and assigns a correlation score in the range $[-1.0..1.0]$. Kendall's τ is computed for a list of n documents which has no ties with the expression:

$$\tau = \frac{n_c - n_d}{\frac{1}{2}n(n-1)}$$

Where n_c is the number of concordant pairs, and n_d is the number of discordant pairs. A concordant pair is a pair of documents n the baseline (B_a, B_b) having $B_a < B_b$, where that ordering is preserved by the pregen ($P_a < P_b$). A discordant pair is the opposite ($B_a < B_b$, but $P_a > P_b$). If $B_a = B_b$ or $P_a = P_b$, then the pair is said to be "tied", and is neither concordant or discordant. The "tau-b" variant of Kendall's Tau which we use also includes a correction factor for these tied pairs (this is not shown in the expression above).

A score of 1.0 indicates perfect agreement in the rankings, a score of -1.0 indicates perfect disagreement in the rankings (one ranking is the reverse of the other), and a score of 0.0 indicates rankings which are not correlated with each other (random ranking). We are interested in how close we can bring this ratio to 1.0.

The Tau value for simple string truncation on the standard baseline is already 0.999 for a 64-bit pregen, so simple intuition about how close Tau is to 1.0 will not be enough to understand the pregen's behaviour. For that purpose, we will compare our pregen's performance against that of a "reasonable pregen". The ordering of a "reasonable pregen" behaves the same way as a simple truncation of the input strings. More precisely, if a document D_a ranks before document D_b in the baseline ranking, the pregenerated ranking must not rank D_a after D_b, it can either rank D_a before D_b, or put D_a and D_b at the same position in the ranking. Additionally, if two documents D_a and D_b rank at an equal position in the baseline ranking, they must also rank at an equal position in the pregenerated ranking. The radix and arithmetic string coding methods can be considered to be "reasonable pregens" by this definition if the character encoding being used does not misorder input characters compared to the ordering in the baseline. This is true, for example, of the base-37 encoding on the restricted baseline, or the printable ASCII encoding on the standard baseline. The base-37 encoding is not a reasonable encoding on the standard baseline, since it entirely discards characters that the baseline considers significant.

Given these preconditions, consider a reasonable pregenerated ranking which conflates every group of size x of documents in a ranking of n distinct documents. That means that, for example, for $x = 2$, a baseline ranking of $(1, 2, 3, 4, 5, 6, 7, 8)$ would become $((1, 2), (3, 4), (5, 6), (7, 8))$ in the pregenerated ranking, with the documents in each conflated group having the same pregen value as each other. The number of pairs of documents within a group of size x, g_n is given by:

$$g_n = \frac{1}{2}x(x-1)$$

Within each conflated group in the pregen, we consider that the documents in that conflation will be, on average, randomly ordered in the search engine's output. That means that within those groups, the number of concordant pairs and discordant pairs will be equal, and so the number of discordant pairs in the group, g_d, is half of the group size:

$$g_d = \frac{1}{2}(\frac{1}{2}x(x-1))$$

Since there are $\frac{n}{x}$ of these groups in the total list of documents, the total number of discordant pairs in the ranking n_d is given by:

1907	Space
223, 1, 1, 1, 1, 29, 25, 155, 157, 60, 5, 48, 78, 136, 29	Punctuation
16, 50, 16, 17, 5, 4, 1, 5, 7, 8	Digits 0 - 9
41, 30, 1, 3, 1, 190, 1, 29, 1, 28, 11, 4, 1	Punctuation
1048, 134, 549, 502, 1204, 205, 306, 384, 669, 21, 88, 515, 333	Letters a - m
737, 854, 481, 12, 858, 643, 850, 199, 100, 315, 38, 225, 8,	Letters n - z
28, 13, 29, 93	Punctuation
15	Unicode

Figure 1: Relative symbol frequencies for the printable ASCII character encoding on Chicken Smoothie titles

$$n_d = \frac{n}{x} g_d$$

Because this pregen behaves like a truncation, we know that these are the only discordant pairs in the ranking, so n_c is computed by subtracting the number of discordant pairs from the total number of pairs in the pregen of n documents:

$$n_c = \frac{1}{2} n(n-1) - n_d$$

Substituting these values into Kendall's Tau gives the expression:

$$\tau = \frac{(\frac{1}{2}n(n-1) - (\frac{n}{x}\frac{1}{2}\frac{1}{2}x(x-1))) - (\frac{n}{x}\frac{1}{2}\frac{1}{2}x(x-1))}{\frac{1}{2}n(n-1)}$$

Combining the two occurences of the expression for n_d gives:

$$\tau = \frac{\frac{1}{2}n(n-1) - \frac{n}{x}(\frac{1}{2}x(x-1))}{\frac{1}{2}n(n-1)}$$

Performing the division simplifies the expression to:

$$\tau = 1 - \frac{\frac{n}{x}(\frac{1}{2}x(x-1))}{\frac{1}{2}n(n-1)}$$

Cancelling terms in the numerator and denominator gives the value of Kendall's Tau for a pregenerated ranking of n distinct values which conflates each group of x elements:

$$\tau = 1 - \frac{x-1}{n-1}$$

We can reverse this expression to derive the value of x for a given τ for this theoretical ranker:

$$-\tau + 1 = \frac{x-1}{n-1}$$

$$(-\tau + 1)(n-1) = x - 1$$

$$(1-\tau)(n-1) + 1 = x$$

x is a useful value to examine, because we can consider its relationship to the number of search results which the search engine presents on a single page. If those numbers are of similar magnitude, our search engine could resort just the documents that appear on a given search result page using their complete titles, and achieve a high overall ranking accuracy.

9 Results

The performance of each encoding scheme is graphed against the standard and restricted baselines of the Chicken Smoothie and Ancestry.com post title collections in figure 2, and the performance on the Chicken Smoothie post titles is displayed in table 1. The performance is measured using the conflation group size metric we defined in the previous section. In the standard baseline, we have included the "strtrunc" encoding, which is a simple truncation of the lowercased input. The graphs also show the "optimal" encoding, which is the conflation score expected if the input document titles are evenly distributed, and the pregen value is computed by sorting all of the documents in the collection and then numbering them in sequential order.

Where arithmetic string encoding has been used, it always outperforms the radix string encoding on the same character encoding, on our test data. This is true even for the Ancestry.com document collection, where the character probability model has been left unchanged from that computed for the Chicken Smoothie collection, suggesting that this unigram character model is generally applicable to English text.

The Ancestry.com post title dataset shows an unusual spike in performance around the 56-bit mark for the ASCII-printables pregen on the standard baseline. This is due to a large number of distinct post titles in this collection which happen to share a long prefix. This prefix is the string "Looking for " (as in "Looking for John Smith"), which occurs in over 200,000 documents in the Ancestry.com collection. The sharp improvement in performance corresponds to the position at which the pregen ranking begins to encode the characters that appear after this long prefix. The next-best performing pregen, base-40, can only encode 12 characters in 64-bits, which happens to coincide with the length of this prefix, so it conflates the ordering of these 200,000+ documents. Further difficulties are encountered with documents having the prefix "Looking for info", which is two characters longer, outstripping even the compression capability of the arithmetic-encoded ASCII printables encoding.

The 64-bit pregens on the Chicken Smoothie post title collection achieve a conflation group size of 64 documents on the standard baseline, and 52 documents

Scheme	Bits	Average conflations Standard baseline
Strtrunc	8	10771
	16	4109
	24	2837
	32	2649
	64	586
Base40	8	5640
	16	3147
	24	1505
	32	1023
	64	877
Asciiprintables	8	7779
	16	2686
	24	2224
	32	352
	64	111
Asciiprintablesarith	8	4680
	16	1697
	24	474
	32	232
	64	64

Scheme	Bits	Average conflations Restricted baseline
Base37	8	5411
	16	2589
	24	793
	32	259
	64	121
Base37arith	8	4400
	16	1486
	24	310
	32	176
	64	52
Base36	8	6059
	16	3278
	24	1513
	32	1453
	64	1325
Base32	8	4754
	16	2508
	24	518
	32	240
	64	67
Base32arith	8	4365
	16	1456
	24	308
	32	179
	64	55

Table 1: Pregen performance on Chicken Smoothie post titles for various encodings and pregen bit sizes

on the restricted baseline, both using an arithmetic encoding of the baseline's character set. On the Ancestry.com post title collection, the accuracy is 134 documents on the standard baseline, and 104 documents on the restricted baseline.

Not shown on the graphs is the performance of a pregen ranking for sorting Chicken Smoothie posts by author name. As author names are much shorter than document titles on average, an accuracy of 3 documents is achieved on the standard baseline, and approximately 1 document on the restricted baseline.

10 Conclusion

We have demonstrated that pregenerated rankings of text attributes such as post titles can achieve a result that is, on average, sorted to within 134 documents on average, across two very different English language collections. Similar performance was achieved on both the restricted and standard baselines, demonstrating that we can preserve the ordering of punctuation characters in the final output without any major additional ranking error. We have developed a metric which allows the ranking performance of pregens to be understood. We have identified a situation in which this pregenerated ranking method performs poorly, which is the presence of long common prefixes between distinct document titles, but we expect that the application of a bigram probability model for English text will solve this problem by improving our text compression performance. We hope to quantify that improvement in the future.

References

[1] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, Volume 21, Number 4, pages pp. 514–532, 1971.

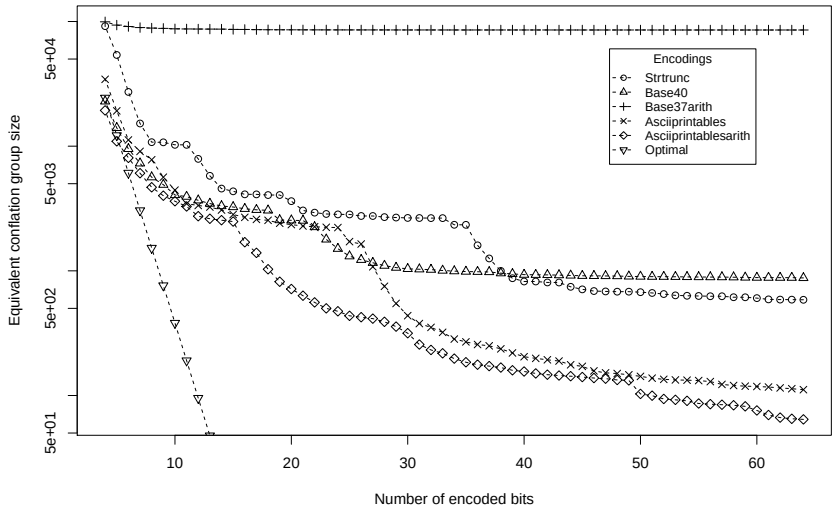
[2] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Volume 40, Number 9, pages 1098 –1101, sept. 1952.

[3] A. Moffat and A. Turpin. *Compression and coding algorithms*. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 2002.

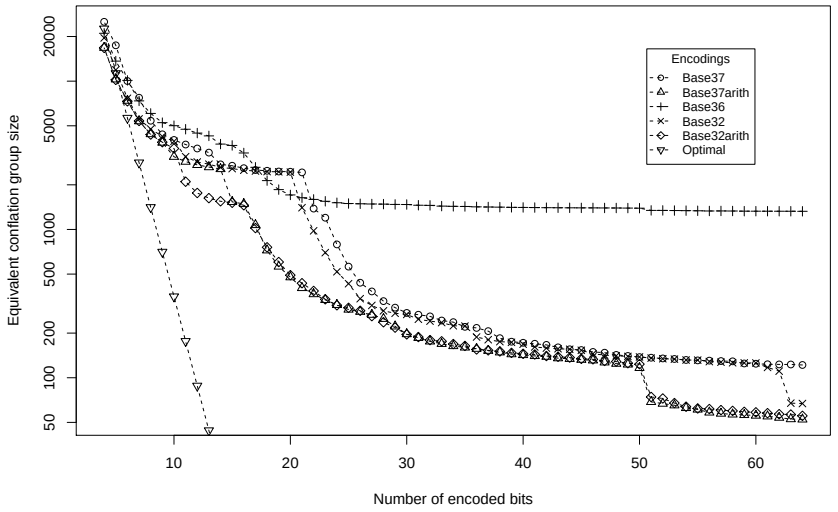
[4] Lawrence Page, Sergey Brin, Rajeev Motwani and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[5] Andrew Trotman, Xiang-Fei Jia and Shlomo Geva. Fast and effective focused retrieval. In Shlomo Geva, Jaap Kamps and Andrew Trotman (editors), *Focused Retrieval and Evaluation*, Volume 6203 of *Lecture Notes in Computer Science*, pages 229–241. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14556-8

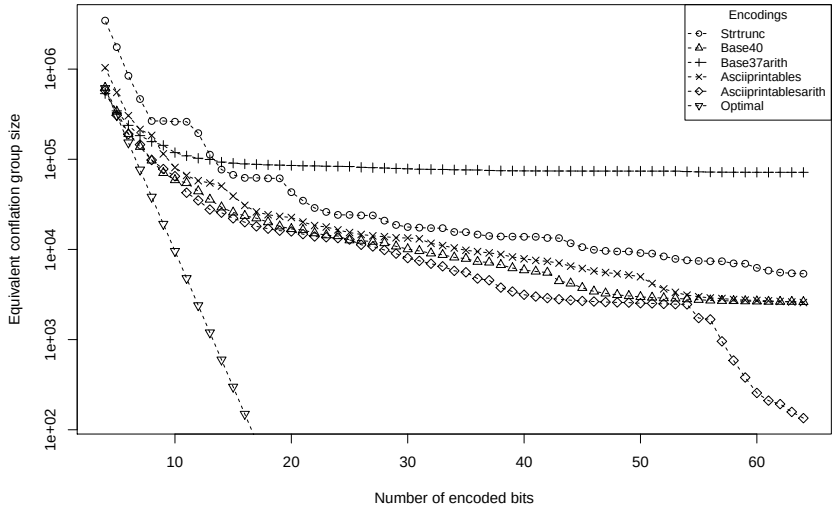
[6] Ian H. Witten, Radford M. Neal and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, Volume 30, pages 520–540, June 1987.



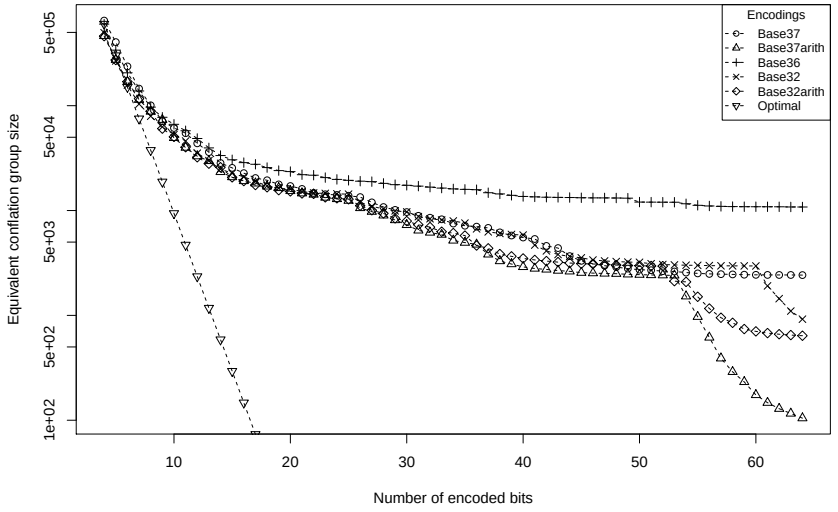
(a) Pregen ranking performance against standard baseline on Chicken Smoothie post titles



(b) Pregen ranking performance against restricted baseline on Chicken Smoothie post titles



(c) Pregen ranking performance against standard baseline on Ancestry.com post titles



(d) Pregen ranking performance against restricted baseline on Ancestry.com post titles

Figure 2: Pregen performance on sorting document titles