Enhancing Wikipedia Search Performance Using Elias Gamma Code

Ali Bagherzandi and Kerim Yassin Oktay

Department of Computer Science, University of California, Irvine,
{zandi,koktay}@ics.uci.edu

1. Motivation

Project 4 and 5 were aimed to build a real search engine, however the inefficiencies both in retrieving the relevant documents for the queried phrase and the time required to answer the query kept it far from being practical. We noticed that these two problems originate from similar sources: In fact in order to make the query response time reasonable, we ignored words with corpus frequency higher than 50'000 and this effected retrieving the relevant documents by ignoring many words that normally should not be ignored because they are key words to perform a finer search. For instance, in project 5, the phrase “search engine will retrieve” was equivalent to “retrieve” whereas both “search” and “engine” is basically characterizing the above phrase and should not be thrown away. Unfortunately when we tried to increase the threshold we faced with unacceptably long response time. For instance for the phrase “search engine will retrieve”, our server took (ms) to respond even if we threw away the word “will”.

We realized that almost all of the query response time is being spent on reading the posting lists for query terms from the disk and as the corpus frequency of a term increases this time also increases leading to a very bad performance. Therefore we thought maybe by using an elegant encoding we can reduce the length of the posting lists and hence their loading time. The method we come up with is called gamma coding.

2. Elias Gamma Code

Elias gamma code or for short gamma coding is a coding system developed by Peter Elias to encode positive integers. It is most commonly used to encode integers whose upper-bound cannot be determined beforehand or to compress data in which small values are much more frequent than large values[1].

Gamma coding minimizes the space needed to store numeric data on a file by minimizing the “wasted” space and adapt the length of the code on the finer grained *bit* level. When numeric data is stored in predefined format (e.g. byte, integer, etc.) each number takes a predefined amount of space (e.g. for byte 1 byte, for integer 4 bytes, etc.) Therefore a lot of space is “wasted” simply because most of the numbers being stored require much less space. Gamma encoding enables us to store an integer n using exactly $1 + 2\log_2(n)$ bits. This saves a huge amount of space when numbers being saved are small.

Note that since everything is stored as a sequence of 0’s and 1’s, therefore one cannot simply shift the numbers in the sequence to fill up the “wasted” space; Because it would lead to loosing the track of starting and ending points of the numbers.

In order to encode using gamma coding, two values are stored consecutively for each number: *length* and *offset*. *Length* is the bit-size of the number and *offset* is the difference between the number and the largest power of two which is smaller than the number. More precisely, consider integer k, the *length* value in gamma encoding of k is $\lfloor log_2(k) \rfloor$ and the *offset* value is $k - 2^{\lfloor log_2(k) \rfloor}$. However to keep track of the starting and ending point of the numbers, length is stored in *unary* followed by a 0 and offset is stored in *binary*. Therefore for example, length value of 13 is 3 and stored in unary followed by a 0, i.e. “1110” and offset value for 13 is 5 (13-8 = 5) and it is represented in binary as "101". Table 1 shows some examples. Decoding is much simple: after *length* and *offset* are read, the number n is reconstructed by computing $2^{length} + offset$ [2].

	number	unary code	length	offset	Gamma code	
	0	0				
	1	10	0		0	
	2	110	10	0	10,0	
	3	1110	10	1	10,1	
	4	11110	110	00	110,00	
	9	1111111110	1110	001	1110,001	
	13		1110	101	1110,101	
	24		11110	1000	11110,1000	
	511		111111110	11111111	111111110,11111111	
	1025		11111111110	0000000001	11111111110,0000000001	

3. Implementation

In order to compress our posting lists, we first eliminated unnecessary characters (e.g. “[”, “]”, “,”, etc) from our posting file. Then for each posting list, we stored the first docId in the list as the base and for the rest docId’s we stored only the increments in the docId’s rather than the docId themselves. Since the docId’s in the posting lists are stored in sorted order, this way the numbers we stored are much less than the docId's. And finally to store all the numbers in posting lists including base docId and increments, we used gamma coding technique.

To compress the posting lists using gamma encoding, we compute gamma encoding of each number, and store it in a buffer (e.g. a int[] or byte[]), when the encoding for the next number is added to buffer we use bitwise shift and bitwise or to utilize the wasted space in the byte array. Finally we wrote the buffer to the file when the sum of the length of the encoded numbers reaches to multiple of 32.

With this technique, we are able to compress our 3.31 GB posting file to 399 MB which gives a compression rate equal to 8.2.

We note that we did the above process after we have created our posting lists. However it can also be done in MapReduce while the posting lists are being created.

As we mentioned, the decompression part is much easier and requires only computing $2^{length} + offset$ after length and offset has been read. Besides this we need also compute the actual docId's using base docId and the increments. But over all the time complexity of this process is O(n) and since the computations are done in main memory it is much faster.

4. Comparison

With this enhancement, we were able to reduce the response time drastically, and be able to respond to almost all queries instantly. Figure 1 shows how much this method could speed up the program. The x-axis is the average corpus frequency for certain groups of terms, and y-axis is the retrieval time in milliseconds. The red graph shows retrieval time when there is no compression is used and the blue graph shows retrieval time when gamma compression is used.

To produce this figure, we first computed the respond time for all the 6 million terms in our index file using both with and without gamma compression technique. However since we are only interested in the relation between retrieval time and corpus frequency we, we picked 13000 different corpus frequency that we had, and bucketized them into 50 different group. i.e. all we put all the terms out of 6 million terms that had the least 260 low CF into the first bucket, all the terms that had the second least 260 low CF into the second bucket, and so forth. Finally the average retrieval time (w/ and w/o gamma compression) is computed over all the terms in a given bucket. For completeness we also show what happens in the worst case, Figure 2 shows the change in the retrieval time for the terms which fall into high end of corpus frequency spectrum in our data set. Both of theses graphs are also plotted by our program.

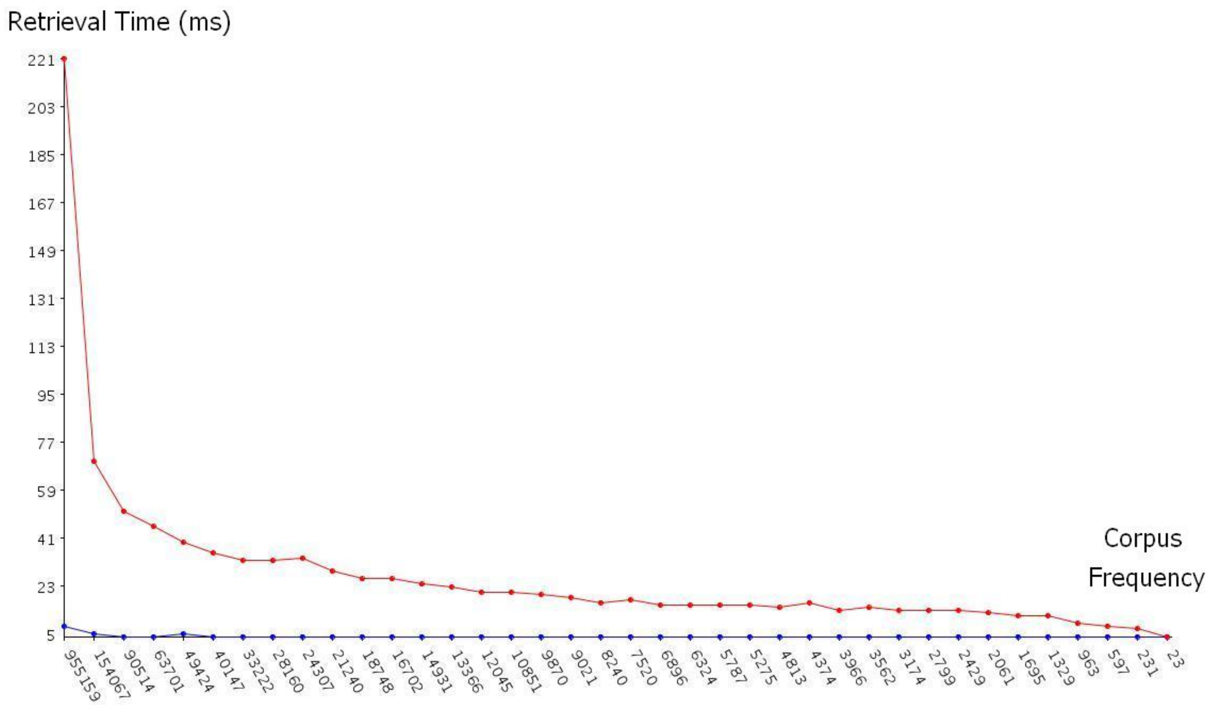

Figure 1: Retrieval Time vs. Corpus Frequency

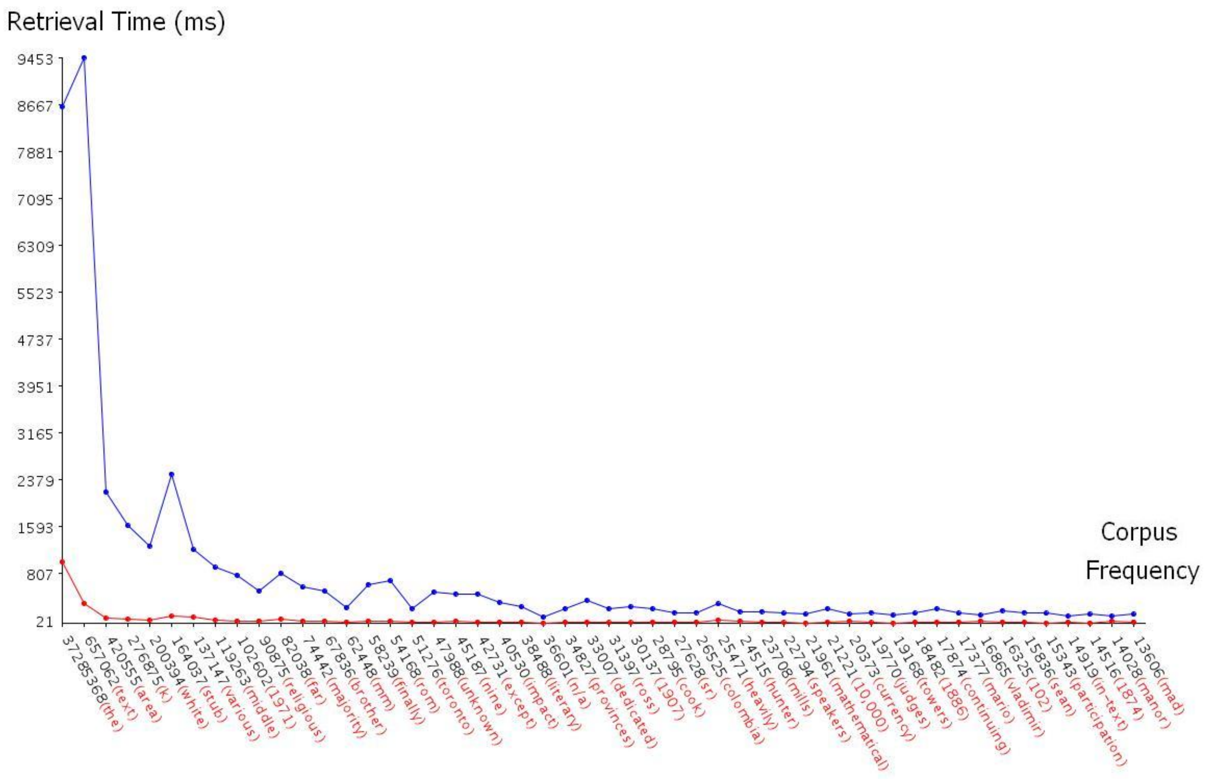

Figure 2: Retrieval time for some sample terms that have high corpus frequency

5. Conclusion

Using gamma encoding technique and storing only the increments for the docId’s in a sorted posting list, we were able decrease the size of our posting lists which lead to a posting file 8 times smaller. Also loading the compressed posting lists form disk to main memory requires lower time, and this decreased the response time of our program drastically. In fact the time complexity becomes so low that we are able to remove any threshold for ignoring the common words and take into account more words in the query phrase, which leads to retrieval of more relevant documents.

6. References

1- http://en.wikipedia.org/wiki/Elias_gamma_coding
2- http://nlp.stanford.edu/IR-book/html/htmledition/gamma-codes-1.html