Overview and comparison of three techniques for inverted index compression

Golomb coding, Binary Interpolative Coding, and Simple-9

Silvia Imeneo

Information Retrieval A.Y. 2023-2024
Data Science and Scientific Computing

Abstract. Search engines heavily rely on the inverted index data structure for information retrieval. In most of the scenarios where these indexes are used nowadays, the volume of data that is stored and that needs to be scanned is so huge that inverted index compression techniques are indispensable. This is why multiple studies have been carried out to find new methods that could efficiently compress an inverted index while maintaining an acceptable decoding speed.
This document provides an overview of three lossless compression techniques used for compressing the postings lists of an inverted index: the Golomb coding, the Binary Interpolative Coding, and the Simple-9 Coding. The description of their procedure is presented together with examples to ease their understanding. A final comparison is also provided to highlight where each method stands in the trade-off between saved space and needed query processing time[1].

1 Introduction

1.1 The inverted index

When dealing with a collection of textual documents, in order to perform a search on them, we need a data structure that allows us to retrieve the information that we look for. If we see our textual documents as a set of terms, for each distinct term t, we can build a sequence S_t which sorts all the identifiers of the documents where that term appears. This sequence S_t is called *inverted list*, or *postings list*, of the term t, and the set of inverted lists for all the distinct terms is the data structure that we need: the *inverted index*[1].
An inverted index consists of two parts: the dictionary and the postings lists. For each unique term in the collection, the dictionary maps that term with the

[1]We remind that other techniques exist for dictionary compression, but they will not be covered in this work.

location on the disk where its postings list is. The postings list for any given term is a vector and is often represented as $\{\langle d_1, f_1\rangle, \langle d_2, f_2\rangle, ..., \langle d_n, f_n\rangle\}$ where a pair $\langle d, f\rangle$ means that the given term is present f times in document d. The inverted list is usually sorted sequentially by the document IDs[2].

Query resolution consists of searching the vocabulary for the query terms, then fetching and **decoding** the corresponding inverted lists. At this stage, if we use *Boolean queries*, we perform intersection (AND), union (OR) and/or complement (NOT) operations on the sets of document numbers contained in the inverted lists and extract the resulting documents. If we use *ranked queries* instead, we compute a similarity score between the query and each document in the collection, and then present some amount of top-scoring documents as answer to the query[3][4]

1.2 Why compressing

The inverted index is at the core of large-scale search engines, and, in a typical use case, it indexes millions of documents, resulting in several billions of integers[1]. Given this huge volume of data, techniques for compressing the inverted index becomes necessary for several reasons:

- Less disk space is needed to store the inverted index.
- Higher use of caching is possible: we can fit a lot more information into the main memory, saving disk seeks.
- Faster query processing is achieved: on current hardware, with most compression schemes data can be decoded faster than it can be delivered from the disk, resulting in a net decrease in access time if it is stored compressed[4].

2 Examples of compression techniques

There are plenty of techniques that have been developed to compress inverted indexes. They are generally divided between *lossless*, where all information is preserved, and *lossy*, where some information is discarded.

Compression techniques can also be classified based on the object of the compression, being it: a single integer; a whole inverted list; or the whole inverted index[1].
In this work, we are going to focus on three lossless techniques: Golomb coding, Binary Interpolative Coding, and Simple-9.

2.1 Golomb coding

Introduced in 1966 by Solomon Golomb, this technique belongs to the class of algorithms that focus on compressing a single integer. The integer x that we want to encode is represented as two parts:

- a quotient, calculated as $q = \lfloor (x-1)/k \rfloor$
- a remainder, calculated as $r = x - (q * k) - 1$

k is the base to which the Golomb code is calculated and its choice is crucial for the final result. We will see later how to choose it.

To obtain the Golomb encoded form of our integer x, we proceed as follows[5][2]:

- We calculate q and r;
- We calculate $b = \lfloor log_2(k) \rfloor$ and $p = 2^{b+1} - k$
- Then:
  - If $r < p$: the Golomb code is constructed as the unary code of q concatenated with the binary representation of r.
  - If $r \geq p$: the Golomb code is constructed as the unary code of q concatenated with the binary representation of $r + p$.

If, for example, we want to encode the integer 5, using as base $k = 2$, so if we look for $G_k(x) = G_2(5)$, what we do is:

- We calculate:
  - $q = \lfloor \frac{(x-1)}{k} \rfloor = \lfloor \frac{(5-1)}{2} \rfloor = 2$
  - $r = x - (q * k) - 1 = 5 - (2 * 2) - 1 = 0$
- We calculate:
  - $b = \lfloor log_2(k) \rfloor = \lfloor log_2(2) \rfloor = 1$
  - $p = 2^{b+1} - k = 2$
- Since $r < p$, the Golomb code is 110.0, given by the concatenation of:
  - the unary code of $q = 2$, which is 110
  - the binary representation of $r = 0$, which is 0

Regarding the choice of k, if we make a bad choice, the result of the Golomb encoding can be very large, making decoding quite slow [2]. Some researchers suggest assuming that the integers of an inverted list follow a Bernoulli model, which has a geometric distribution starting at 0. The optimal value of k depends then on $p = P(x = 0)$, the probability of success in a given Bernoulli trial. Based on this, the optimal value for k has been estimated to be $k \approx 0.69/p$ [2].

Golomb coding is highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than the occurrance of large values[5].

2.2 Binary Interpolative Coding

Presented by Moffat and Stuiver in 1996, the Binary Interpolative Coding (BIC) is a technique used to compress a postings list by exploiting locality, meaning the tendency for terms to appear in clusters in the collection rather than at random.[3]. The non-randomness is a consequence of the fact that, in most databases, records are appended in chronological sequence, and topics move into and out of favour[3]. The BIC algorithm as written by Moffat and Stuiver is represented in Figure 1.

```
Binary_Interpolative_Code($L, f, lo, hi$)
/* Array $L[1 \ldots f]$ is a sorted list of $f$ document numbers, all in the range
  $lo \ldots hi$. */
1. If $f = 0$ then return.
2. If $f = 1$ then call Binary_Code($L[1], lo, hi$) and then return.
3. Otherwise, calculate
       $h \leftarrow (f + 1)$ div 2,
       $f_1 \leftarrow h - 1$,
       $f_2 \leftarrow f - h$,
       $L_1 \leftarrow L[1 \ldots (h - 1)]$, and
       $L_2 \leftarrow L[(h + 1) \ldots f]$.
4. Call Binary_Code($L[h], lo + f_1, hi - f_2$).
5. Call Binary_Interpolative_Code($L_1, f_1, lo, L[h] - 1$).
6. Call Binary_Interpolative_Code($L_2, f_2, L[h] + 1, hi$).
7. Return.
```

Figure 1: Binary Interpolative Coding.

The key idea of the Binary Interpolative algorithm is to exploit the order of the already-encoded elements to compute the number of bits needed to represent the elements that will be encoded next [1]. It is designed in a recursive way and it exploits the fact that postings lists are sorted sequences where every next value is always higher and no duplicates are present.

With this technique, we start from a sequence S and, at each iteration, we want to encode the sub-sequence $S'_{l,r}$, for which we always know the following five parameters[6]:

- the **indexes** of the delimiting left (l) and right (r) elements of the sub-sequence: $S'_{l,r} = s'_l, ..., s'_r$;
- the number n of elements in the sub-sequence: $|S'_{l,r}|$;
- *low*: a lower-bound to the **lowest value** in the sub-sequence ($low \leq s'_l$);
- *hi*: an upper-bound to the **highest value** in the the sub-sequence ($hi \geq s'_r$).

When starting, at the first iteration the values of the five parameters are: $l = 1$; $r = n$; $n = |S|$; $low = 0$; $hi = s_n$.

Before showing how to proceed, it is important to keep in mind that, in the Binary Interpolative Coding, the encoding of an integer s'_i is not fixed but **depends on the distribution of the other integers in its sub-sequence** S'. More specifically, it depends on those five parameters described above. This means that the same integer may be encoded differently in its occurrences over different postings lists of the inverted index[6].

We can now present the procedure. At each step of this algorithm:

- We first encode the middle element, s'_m, where $m = \lfloor \frac{l+r}{2} \rfloor$. We said that we do it by using the parameters and this is because the actual number that we encode is not the one corresponding to the middle position, but it is $s'_m - low - m + 1$.
- We then recursively encode the two sub-sequences to the left and to the right of s'_m (so $s'_l, ..., s'_{m-1}$ and $s'_{m+1}, ..., s'_r$) by using the properly recomputed five parameters $\langle n, l, r, low, hi \rangle$[6].

With the exception of the values of the first iteration (which both the encoder and the decoder must know), all values for the subsequent iterations can be easily recomputed from the previous ones. In particular:

- for the left sub-sequence $s'_l, ..., s'_{m-1}$, the parameter low is the same of the previous step, since s'_l has not changed; and we can set $hi = s'_m - 1$, since $s'_{m-1} < s'_m$ given that we assumed the integers to be distinct and increasing;
- for the right sub-sequence $s'_{m+1}, ..., s'_r$, the parameter hi is the same as before, since s'_r has not changed; and we can set $low = s'_m + 1$, since $s'_{m+1} > s'_m$;
- the parameters l, r and n are recomputed accordingly.

In order to know the space needed for this encoding, we can obser that:

- in the first half of S', so to the left of s'_m, we have $m - l$ distinct values and the smallest one is larger than the lower bound low;
- in the second half of S', so to the right of s'_m, we have $r - m$ distinct values and the largest one is smaller than the upper bound hi.

This means that the value that we want to encode, s'_m, lies in the range $[low + m - l, hi - r + m]$, thus we can encode it by using $\lceil log_2(intv) \rceil$ bits, where $intv$ is the size of that interval, so $intv = hi - low - n + 1$[1].

For a clearer understanding, we can consider an example[1]. We start from the sequence $S = [3, 4, 7, 11, 13, 15, 21, 25, 36, 38, 54]$. The initial values for the

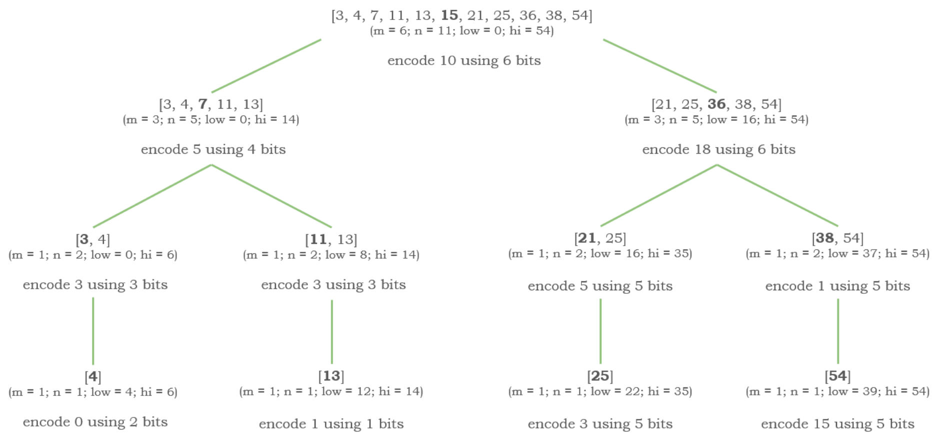

Figure 2: The recursive calls performed by the Binary Interpolative Coding algorithm.

five parameters here are: $l = 1$, $r = 11$, $n = 11$, $low = 0$, $hi = 54$. Figure 2 represents, as a binary tree, the recursive calls performed by the BIC algorithm.

We start by extracting the middle element, the one that we want to encode (in bold in Figure 2). Since $m = \lfloor \frac{l+r}{2} \rfloor = 6$, at the first run we have that $s'_m = 15$. Based on what said so far, we know that the element that we are actually encoding is not 15 but is $s'_m - low - m + 1$, which is $15 - 0 - 6 + 1 = 10$. This encoding is performed using $\lceil log_2(hi - low - n + 1) \rceil = \lceil log_2(54 - 0 - 11 + 1) \rceil = 6$ bits.
We proceed moving to the left sub-sequence, $S'_{l,m-1} = [3, 4, 7, 11, 13]$, for which the values of the five parameters are now: $l = 1$, $r = 5$, $n = 5$, $low = 0$, $hi = 14$. For this sub-section, we know that we can encode the middle value, $s'_m = 7$, by encoding $s'_m - low - m + 1 = 7 - 0 - 3 + 1 = 5$, and by using $\lceil log_2(hi - low - n + 1) \rceil = \lceil log_2(14 - 0 - 5 + 1) \rceil = 4$ bits. We keep proceeding in this way by iterating the encoding on each new left and right sub-sequences.

By *pre-order* visiting the tree, we can eventually obtain the sequence of values that we have actually encoded, so $[10, 5, 3, 0, 3, 1, 18, 5, 3, 1, 15]$, and the associated bits needed for the encoding $[6, 4, 3, 2, 3, 1, 6, 5, 5, 5, 5]$ [1][2].

2.3 Simple-9

Presented by Moffat and Anh in 2005, this technique aims at encoding a list of *d*-gaps, meaning the list of the differences between the values of consecutive documentIDs.
For terms that occur in many of the documents in the collection, the list of *d*-gaps is long, but on average the values in the list are small, since the sum of

[2]Different ways for encoding the integer of interest can be applied in this technique (like minimal binary encoding) but we are not going to cover them.

4-bit selector	integers	bits per integer	wasted bits
`0000`	28	1	0
`0001`	14	2	0
`0010`	9	3	1
`0011`	7	4	0
`0100`	5	5	3
`0101`	4	7	0
`0110`	3	9	1
`0111`	2	14	0
`1000`	1	28	0

Figure 3: The 9 different ways of packing integers in a 28-bit segment.

the d-gaps cannot exceed the total number of documents in the collection. On the other hand, terms with short inverted lists can contain large d-gaps, but cannot contain very many of them. If the inverted list for a term t contains f_t entries, then the average d-gap in that list cannot exceed N/f_t[4].

In the Simple-9 algorithm, we split the sequence of d-gaps into fixed-memory units and then see how many integers can be packed in one unit (trying, of course, to pack in it as many integers as possible)[1]. We work with 32-bit memory words[3]. Out of them:

- 4 bits are dedicated to store the *selector*;
- the remaining 28 bits are dedicated to store the data.

This *selector* gives the information about how many integers are stored in the 28 bits **assuming that each integer takes the same number of bits**. For example, if each integer takes 1 bit to be stored, than we can store $28/1 = 28$ integers. If each integer takes 3 bit, then we can store $28/3 = 9$ integers and we will have 1 wasted bit. In total, we can have 9 different ways of packing integers in a 28-bit segment when using the Simple-9 technique, and they are summarized in Figure 3.

Different memory words can store different amounts of integers, but within a memory word each integer is represented using exactly the same number of bits[4].

To see how this algorithm works, we can consider an example[4]. We start from the postings list $S = [4, 10, 11, 12, 15, 20, 21, 28, 29, 42, 62, 63, 75, 95]$, and compute the list of d-gaps, $D = [4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12, 20]$, which has 14 integers in total. By checking Figure 3, we see that, according to the second selector, we could store the whole list in our 28 bits if each integer could be coded using 2 bits only. In order for that to be possible, the value of each of the 14 integers should be at most 3 [4], but that is not our case. So we check

[3]A memory word is an ordered set of bytes or bits and it is the normal unit in which information may be stored, transmitted, or operated on within a given computer.

[4]Because, starting from the integer 4, we need 3 or more bits to encode the number.

the third selector, and we can see that we could store the first 9 integers of our d-gaps list if each one of them took 3 bits of space, so if the value of each integer were at most 7. This matches our case and so we can proceed with the first 32-bit word. Here:

- The first 4 bits are used to code the third selector, 0010
- The 9 integers are coded as 011, 101, 000, 000, 010, 100, 000, 110, 000[5]
- The last bit is not used.

Now we have left to encode the remaining five d-gaps, which are $13, 20, 1, 12, 20$. Based on Figure 3, those 5 integers could be encoded if each one of them took 5 bits, so if the maximum value of each integer was below $2^5 = 32$. Since our highest value is 20, we can proceed with this encoding and our second 32-bit word will be:

- 0100 to encode the fifth selector,
- 01100, 10011, 00000, 01011, 10011 to encode the 5 integers,
- three unused bits.

In this way, we are able to encode our 14-integer d-gaps list using two 32-bit memory words.
One consideration to be made on this technique is that numbers greater than 2^{28} cannot be represented in this system, because a single integer can use at most 28 bits to be encoded. This means that the number of documents in the collection is limited to approximately 256 million. Nevertheless, Moffat and Anh do not see this as a serious limitation, since very large collections are typically partitioned into manageable sub-collections[4].

3 Comparison

Now that we have described each technique, we want to compare their performance presenting the results published by Moffat and Anh[4].
Their test system makes use of an impact-sorted index, where the $f_{d,t}$ values are factored out, and each inverted list consists of a sequence of blocks. Within a block all pointers have the same impact value, which is stored only once per block. The blocks in each inverted list are sorted in decreasing order of impact, and are not necessarily all examined while any particular query is being resolved[4].

In their work, Moffat and Anh used four test collections, all drawn from the document sets distributed as part of the long-running TREC information retrieval project [6]. These collections are:

[5]Allowing for the fact that a d-gap of 1 maps to a code of 000.

[6]The Text REtrieval Conference (TREC) is an ongoing series of workshops focusing on a list of different information retrieval research areas Wikipedia

- The WSJ collection, which is the concatenation of the Wall Street Journal subcollections on disks one and two of the TREC corpus;
- The collection labelled TREC12, which is the concatenation of all nine subcollections on the first two TREC disks, including the WSJ data;
- The collection wt10g, which is 10 GB of web data collected as part of the Very Large Collection TREC track in 2000;
- The collection .GOV, which is the 18 GB collection that was built by crawling the .gov domain in 2002.

Details of each collection can be seen in Figure 4

	Collection			
Attribute	*WSJ*	*TREC12*	*wt10g*	*.GOV*
Size (MB)	508.5	2071.7	10511.1	18537.6
Documents (10^3)	173.4	741.9	383.4	1247.8
Terms (10^3)	295.6	1133.9	2320.0	5486.9
Pointers (10^6)	38.6	137.1	92.8	360.1
Blocks (10^3)	543.6	1909.6	3304.2	7832.9

Figure 4: Test collections used, showing their size in megabytes; the number of documents they contain; the number of distinct terms (after case-folding and stemming); the number of document pointers in the inverted index; and the number of blocks in the impact-sorted inverted index.

What Moffat and Anh compute is the **compression effectiveness** and the **query throughput rate**, which is affected by the decoding speed.
Regarding the first one, the total cost of the index is the cost of storing all the equal-impact blocks for all of the inverted lists: the document numbers stored in each block are a sorted subset of the integers from 1 to N (the number of documents in each collection), and each inverted list contains as many as 10 blocks (meaning 10 different impact values). The compression effectiveness is measured in bits per pointer averaged across the whole index.
Regarding the query throughput rate, the value that the authors present for each technique and collection is the average of the elapsed time (in milliseconds) between when a query enters the system and when a ranked list of the top 1,000 answers is finalized. The average is taken over 10,000 random queries containing on average three terms each.

The results are showed in Figure 5 and in Figure 6

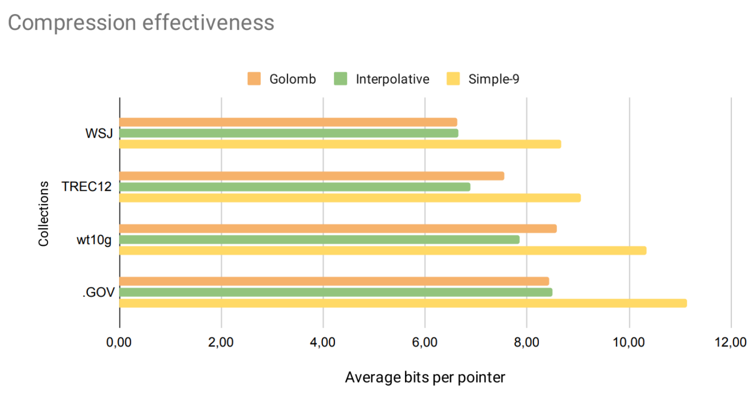

Figure 5: Compression effectiveness, measured in bits per pointer averaged across the whole index.

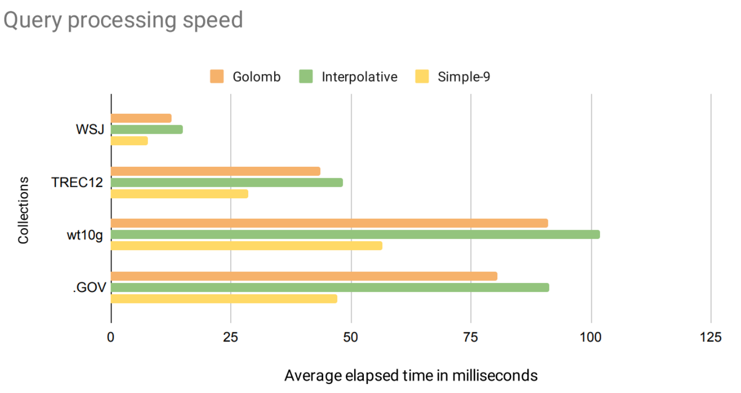

Figure 6: Query throughput rate, as the average elapsed time (in milliseconds) between when a query enters the system and when a ranked list of the top 1,000 answers is finalized.

We can see that, in terms of compression effectiveness, the Binary Interpolative coding is the most efficient one, closely followed by the Golomb technique. On the other hand, when we consider the time needed for the decoding phase and for the whole query processing in general, the BIC is the slowest method and the Simple-9 is by far the fastest one.

4 Conclusion

The results shown in Figure 5 and 6 are a clear representation of the trade-off that we have to consider when choosing an inverted index compression technique: is saving space our main focus? Or do we care more about saving time instead?

Each method has its perks. The Golomb code is optimal if the documents containing the term are randomly scattered. It is also relatively straightforward to implement, and has been found to provide good compression effectiveness on typical document collections[4]. The Binary Interpolative Code is sensitive to localized clustering, and it can reduce a whole run of unit *d*-gaps to just a few bits of output. The Simple-9 method works very well on long inverted lists, where the majority of the *d*-gaps are uniformly small [4].

Finding the solution that is the best for each scenario is not possible. Pros and cons must always be considered and the chosen solution should always be determined by considering the actual data that we have.

References

[1] Giulio Ermanno Pibiri and Rossano Venturini. "Techniques for Inverted Index Compression". In: *ACM Computing Surveys 53:1-36* (2020).

[2] Andrew Trotman. "Compressing inverted files". In: *Information Retrieval, 6:5-19* (2003).

[3] Alistair Moffat and Lang Stuiver. "Binary Interpolative Coding for Effective Index Compression". In: *Information Retrieval, 3:25-47* (2000).

[4] Alistair Moffat and Vo Ngoc Anh. "Inverted Index Compression Using Word-Aligned Binary Codes". In: *Information Retrieval, 8:151-166* (2005).

[5] Wikipedia, Golomb coding.

[6] Paolo Ferragina and Fabrizio Luccio. *Computational Thinking.* Springer, 2018. Chap. 9 - Integer Encoding.