Compressing integer lists with Contextual Arithmetic Trits

Yann Barsamian, André Chailloux

Compressing integer lists with Contextual Arithmetic Trits

Yann Barsamian✉
Inria (Paris, France)
EPI COSMIQ
yann.barsamian@teacher.eursc.eu

André Chailloux
Inria (Paris, France)
EPI COSMIQ
andre.chailloux@inria.fr

September 5, 2022

Abstract

Inverted indexes allow to query large databases without needing to search in the database at each query. An important line of research is to construct the most efficient inverted indexes, both in terms of compression ratio and time efficiency. In this article, we show how to use trit encoding, combined with contextual methods for computing inverted indexes. We perform an extensive study of different variants of these methods and show that our method consistently outperforms the Binary Interpolative Method — which is one of the golden standards in this topic — with respect to compression size. We apply our methods to a variety of datasets and make available the source code that produced the results, together with all our datasets.

Contents

1 Introduction

When faced with a large database, storing the information contained in its documents is one of the concerns, but the most prominent one is searching the database. One of the standard tools to help queries is an *inverted index*. Each document in the database has an associated identifier, and the index stores, for each word, the (sorted) list of identifiers whose associated documents contain this word. A sorted list of this kind is known as a *posting list*.

Techniques for efficiently representing those posting lists have to reach a difficult balance between two main parameters: the size of the inverted index, and the time needed to query it. For instance, the *cmix* compression algorithm [56] is extremely space efficient but constructing and querying the inverted index takes a prohibitive amount of time — 22 hours for a few MB. Regarding more efficient algorithms, the technique that yielded the best compression ratio was the Binary Interpolative Method [18] from 1996 — where a few MB are handles in a few seconds. Its compression ratio has been recently outperformed, on some datasets, by the Packed+ANS2 method [17].

In this article, we focus on the size of the resulting index, *i.e.*, we focus on the *compression ratio* but for algorithms with a running time comparable to the Binary Interpolative Method. We present a representation for the posting lists (transformation into a *trit list*) that consistently outperforms those methods (up to 13%) on databases ranging from a few MB (*e.g.*, personal e-mails) to many GB (*e.g.*, large Information Retrieval databases employed for the World Wide Web).

We make available the source code that produced the results, together with all our datasets.

1.1 Formal statement of the problem

We want to answer the following question: what is the best compression algorithm that can be crafted, that targets inverted indexes? The posting lists we want to compress can each be viewed as a binary matrix, where the bit 0 means that the word does not appear in the document and the bit 1 means that the word appears in the document[1]. An example is given in Figure 1.

		Document															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Words	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
	2	0	1	0	0	0	0	1	1	0	1	1	0	1	0	0	0
	3	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	5	0	0	0	1	1	1	0	0	1	0	0	0	0	1	0	1

Figure 1: Posting lists represented as a binary matrix M such that $M[i, j] = 1 \Leftrightarrow$ the word i appears in the document j.

Storing this full table requires nbWords $*$ nbDocuments bits, which is 80 bits in this example[2]. This would be fine if we had to encode any kind of binary table but those arising from inverse indexes have much more structure. Exploiting this structure has been the key for deriving more

[1] Posting lists can be generalized to the case where we store the number of times a word appears in a document. We do not consider this generalization in our work.

[2] This assumes we know the values nbWords and nbDocuments.

and more successful compression algorithms. We present below two kinds of structure that are identified and exploited in most real life posting lists.

Sparcity and Clustering

Sparcity. The most frequent structure that we see is that inverse indexes are very sparse. Typically, in the databases we consider, posting lists have an average density below 0.3%. This means it is more efficient to store the list of documents IDs (*i.e.*the position of 1s) in which each word appears. Since these form an increasing sequence for each word, it is often more efficient store the gaps between each document ID and to work on the corresponding gap lists. The matrix given in Figure 1 thus becomes the arrays in Figure 2, where the transformation between position and gaps is pictured.

Document lists

$$\text{Words} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} \begin{bmatrix} (12,16) \\ (2,7,8,10,11,13) \\ (2,3,4) \\ (11) \\ (4,5,6,9,14,16) \end{bmatrix}$$

Gap lists

$$\text{Words} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} \begin{bmatrix} (12,4) \\ (2,5,1,2,1,2) \\ (2,1,1) \\ (11) \\ (4,1,1,3,5,2) \end{bmatrix}$$

Figure 2: From posting lists to document lists and gap lists. We took the posting lists from Figure 1 and derived the document lists and the gap lists.

Most efficient methods work on the gap lists, even though one of the best-performing ones, the Binary Interpolative Method [18], works directly on the document lists.

Clustering. Inverted indexes have a property which is called clustering: the 1s appearing in the gap lists are much closer to each other than what they should be if the document IDs were uniformly distributed. This depends of course on the ordering of the documents and can be explained as follows: suppose for example that the documents are newspaper articles sorted by date. If a tornado happens at a current date then the word 'tornado' will appear in many articles close together *i.e.*close to this date, but will appear rarely at other times. This clustering implies not only that small gaps appear more often but also that they are close to each other on average. Using this structure is the key for giving the most efficient compression algorithms.

Different performance measures for posting list compression

There are two main aspects that we can care about when compressing. First, of course is the size of the compression: the smaller the size, the better the compression algorithm is. We can also be interested in efficiency but here, many questions arise: do we care about the time to build the index? to decompress it? to query it? if so on which kind of queries? This makes comparing different compression algorithms difficult because each of them has different upsides and downsides.

In this work, we are mostly interested in the first aspect, *i.e.*the space efficiency of the compression algorithm. We take everything into account when measuring the size of the compressed database and the results we provide correspond to actual file sizes from which we can recover the original posting lists.

Concentrating mainly on the compression efficiency is really advantageous regarding benchmarking. Indeed, the same compress list compressed using a specific algorithm should yield the same result independently on the code, or the computer it runs on. This makes it therefore very easy to compare the compression performance of different algorithms. This is contrast with timing benchmarks which vary depending on implementation and hardware.

1.2 State of the art

Even though some compression techniques encode the gaps one by one, most compression techniques take advantage of local properties of the posting lists. One way to do it is to encode the posting lists by blocks (methods that resort to this technique sometimes dynamically adapt the block sizes to enhance the compression ratio), another way to do it is to search similarities between the posting lists in order to reorder the document IDs. A large portion of the state-of-the-art also covers ways to speed up the encoding, the decoding, and/or the queries in the index; usually, this hinders compression performance, sometimes by a factor that can be greater than 2.

Before this article, the state-of-the-art techniques that yielded the best compression ratios were the Binary Interpolative Coding (Interp) [18] and the Packed+ANS2 method [17]. For 20 years, Interp was undefeated, and it is only recently that, on some datasets, it has been outperformed. There has then been an extensive amount of work to improve compression algorithms but only Packed+ANS2 outperformed Interp regarding compression ratio; as we will show, if one is only interested in the compression ratios, the gains are lesser than what was shown in the article presenting the Packed+ANS2 method, and are not present on all kinds of datasets.

> “[W]e have shown that the two-decade-old benchmark set by Interp can be consistently beaten, a milestone outcome for index compression.”
>
> A. Moffat & M. Petri [17, Section 5]

Another recent paper [23] claimed to match and even outperform Interp on some datasets. While reproducing these results, however, we found out that this was not the case. In short, we view their idea mostly as a reordering method which is outperformed by another one, namely the graph bisection algorithm [9].

We reproduced the results of the two papers we just mentioned [17, 23] by using the code they provide. By doing so, we found some minor issues that explains why we thought that a more careful analysis of their results was necessary:

- First, they use the same base code from [21] and since they operate on gap blocks (of size 128 or 256), they rightfully compare their methods to a variant of Interp: Block-Interp. However, the Block-Interp implementation they use stores useless information that hinders its compression ratio. In short, the authors compare to a non-optimal version of Block-Interp[3]. Comparing to the best Block-Interp method changes the claims made.
- Second, both these articles use only 2 posting lists for comparison: Gov2 and ClueWeb09. Moreover, those posting lists are truncated in both articles (posting lists of size less than

[3]This fact was already known by the programmers of the original base code, as they write in a comment: `// XXX wasting one word!` regarding this issue. The original programmers certainly didn't need some optimal bounds for the Block-Interp method and this inefficiency was probably overlooked by the subsequent papers using this method. We discuss this issue more in detail in Appendix A.

128 are excluded in [17], and posting lists of small size are removed in [23] until 10% of the posting are removed[4]).

- Finally, these results do not use the best method for document reordering, namely the graph bisection algorithm [9].

In conclusion, we consider that the Interp and the Packed+ANS2 methods are the two best compression algorithm for posting lists in terms of pure compression performance, and most of our results will therefore only mention those two methods — we give in Appendix A comparisons against more methods to prove this claim.

1.3 Contributions

Our paper may be viewed as the happy wedding between trits — as exploited by Elias in 1955 [39, p. 31] — and a coding satisfying Shannon's theorem [46]. We chose arithmetic coding, as also described by Elias [36, pp. 61–62], to match as closely as possible the entropy. We note that Asymmetric Numeral Systems (ANS) [38] could also have been chosen, if one wishes to have quicker running times and is willing to lower the compression ratios. First, we transform each gap array to a trit list, then we encode this trit list with an arithmetic contextual method: the knowledge of the last trits gives a good insight into the possible values of the next ones.

We note that previous works on inverted index compression also used contextual encodings: contextual arithmetic encoding on bit vectors [4], contextual arithmetic encoding on words [16], contextual ANS encoding on the result of binary packing blocks of integers [17]. Most of the previous works make two passes on the dataset. One to set the model, and one to encode the dataset with the model (static arithmetic coding). In this article, we used static arithmetic coding but also adaptive arithmetic coding, that requires only one pass over the dataset, and turned out to give better compression rates on most cases.

- From a theoretic point of view, we present a new compression algorithm: Arithmetic Coding with Hybrid Contexts on Trits. This method outperforms Interp consistently on all the datasets we tested but one and the gain with respect to the Interp method goes up to 13%— this method also outperforms Packed+ANS2 on all datasets.
- In order to perform our benchmarking, we re-implemented many other compression techniques (around 30). Since we are only interested in the space compression, we managed to make for this metric a fair comparison with other methods. By doing this, we also saw that the Block-Interp method used in [17, 23] stores too much information and can be easily optimized. We use for our tests around 15 posting lists which we make publicly available in order to facilitate future comparisons with our work.

[4] Comparisons on the full dataset are made in [22], where Interp is shown to remain undefeated.

1.4 Overview of our methods and results

Trit encoding

The trit encoding of a number x is thus its binary representation, minus the leading "1", plus a closing "2", *e.g.*, $19_{10} \to 10011_2 \to 00112$.
The "2" at the end of each encoded integer serves as a delimiter between two successive integer encodings.

The trit encoding by itself doesn't have very good compression rates. On top of that, it doesn't take advantage of the clustering the databases we consider. We therefore add a contextual method on this trit encoding to get our efficient compression methods.

Arithmetic coding [36, pp. 61–62] for trits. Suppose you have a sequence of n trits with an estimate of the proportion of "0", "1" and "2" symbols denoted respectively $P(0), P(1), P(2)$ with $P(0)+P(1)+P(2) = 1$. If the trit sequence has n_i symbols i for $i \in \{0, 1, 2\}$ with $n_0+n_1+n_2 = n$, then a perfect arithmetic encoder will encode your sequence using

$$\lceil n_0 \log_2(P(0)) + n_1 \log_2(P(1)) + n_2 \log_2(P(2)) \rceil$$

bits. This is optimal for random trit sequences with respective symbol frequencies of $0, 1, 2$ equal to $P(0), P(1), P(2)$. However, our trit encoding of the gaps of the posting list are not independent because of the clustering. We therefore add some context in our arithmetic coding.

Context arithmetic coding for trits. In context coding, we don't use absolute probabilities $P(0), P(1), P(2)$ but use instead some probabilities that depend on the previously decoded trits — the context. In order not to store too many probabilities, we only use as context the last γ encoded trits and use the probabilities $P(i|C)$ as our probabilities in our arithmetic coding[5].

For example, if $\gamma = 8$ and our encoding so far is 201021**02022021**, we will use the probabilities $P(0|02022021), P(1|02022021), P(2|02022021)$ in our arithmetic encoder to encode the next trit. This method requires to store $2 \cdot 3^\gamma$ probabilities $P(i|C)$[6] which implies that we cannot have a very large depth γ.

Hybrid contexts. The above presents contexts of size γ where we store each trit of the context. Instead we write $\gamma = k + w$ for some non-negative integers k, w and consider the following context encoding:

- On the last k trits of the context, store for each position whether you have a "2" symbol encoded by T or another symbol (*i.e.*"0" or "1") encoded by N meaning not "2". There are hence 2^k possibilities.
- On the w previous trits, store only the total number of "2" symbols. This is an integer in $\{0, \dots, w\}$.

[5] That doesn't say how we encode our first γ trits since there is no or not enough context. We use an arithmetic encoder with less context (or no context at all for the firts trit) to handle these first trits.

[6] For each C, we only store $P(0|C)$ and $P(1|C)$ since we know that $P(2|C) = 1 - P(0|C) - P(1|C)$. How we encode these probabilities is also very important and will be discussed when we perform a full description of our method.

Our contexts C are therefore elements of $\{N,T\}^k \times \{0,\ldots,w\}$. If we take our example, with $\gamma = 8$ and $k = 3, w = 5$, and our encoding so far is 201021$\underbrace{02022}_{w}\underbrace{021}_{k}$, we will use the probabilities $P(0|NTN,3), P(1|NTN,3), P(2|NTN,3)$ in our arithmetic encoder. This means we have to store $2 \cdot ((w+1)2^k)$ probabilities which can be significantly smaller than $2 \cdot 3^\gamma$. Our arithmetic encoding will be less efficient for the same γ but we will be able to take much larger values of γ which will make our encoder better overall.

Dynamic vs. static estimation of the probabilities $P(i|C)$. Finally, an important aspect of our method is to determine how we compute our estimates for $P(i|C)$. There are two main ways to compute these estimates:

1. **Static:** when compressing the posting list, we first read all the gap lists of the posting list to determine these probabilities exactly. In order to decompress, we need these probabilities so we have to store them in the index.
2. **Dynamic:** we start from $P(i|C) = \frac{1}{3}$ for each i and C — it is highly probable that better initial estimates are possible — and start our encoder with these probabilities. Each time we encode a new trit i with context C, we update our probabilities[7] and continue our context arithmetic encoding with these new probabilities. This method has the drawback of having less precise estimates for $P(i|C)$, it has 2 advantages though:
    - The probabilities $P(i|C)$ can be reconstructed from scratch at the decoding phase, by initializing the decoder again with $P(i|C) = \frac{1}{3}$ for each i and C and updating the probabilities as you decode. This means we don't need to store these probabilities which reduces the storage of the compressed posting lists.
    - We don't need to make a first pass on the posting list in order to compute these $P(i|C)$ which improves the running time of the method.

We compared these 2 methods which are very close in terms of storage, the dynamic estimation is even usually slightly better.

To summarize, the method we use is a context arithmetic encoder, where we use a hybrid context and dynamic estimation of the probabilities. We perform also other optimizations: regarding the ordering of the documents, we use the bisection method, which we also use for concurrent methods for a fair comparison. We also analyze the use of batching in Section 5.3, *i.e.*splitting the words set into batches depending on their density in the posting list and apply a compression on each batch separately. We also analyze whether sorting the words depending on the density helps or not.

We made an extensive analysis of our methods and of other methods existing in the literature for a variety of methods. We focused on posting lists that come from mailboxes as it was one of our first motivations but also consider larger posting lists that come from the web.

We managed to break the Interp barrier up to 13%, even if we sometimes achieve only a few percent of gains. Keep in mind that we were only interested in the raw compression size while other methods could have other advantages in terms of multiple queries or running times.

The remainder of this paper is organized as follows: in Section 2 we introduce the existing techniques. In Section 3 we present our compression algorithm. In Section 4 we show the datasets

[7] Again, there are several ways of doing this update and we will describe our choices when describing the details of our methods.

and our results. In Section 5 we present variants of our compression algorithm. Section 6 summarizes the work and presents some future directions.

2 Existing compression methods

To compress the posting lists, the literature gives us a lot of different approaches. Section 2.1 shows what has been done in a first approach, to find a suitable encoding of the integer lists. Section 2.2 gives a rapid insight into another approach which enhances the clustering property of the posting lists, by reordering the document IDs. Section 2.3 gives a reference to a really different path that may be taken if one is willing to have more efficient ways to make complex queries: the suffix trees.

Last but not least, there are of course a lot of other approaches to compression, that do not target integer lists. The arithmetic coding, that we use in this work, is one of the multiple possible general-purpose algorithms to compress data. We note that the state-of-the-art for general-purpose compression is the cmix [56] algorithm, a successor of the PAQ8 [44] algorithm. The interested reader may find in this latter reference an introduction to arithmetic coding, that is used together with machine learning in those algorithms, to predict the input characters based on a static dictionary and on the previous characters read from the file to compress.

2.1 Compressing posting lists

We will suppose that, given a database of N documents, the document IDs are in $\{1, 2, \ldots, N\}$. If $N < 2^{32}$, this means that each posting list can be represented by a sequence of 32-bit unsigned integers. "Raw" databases often use this simple strategy, because it is easy to use [58, 51]. However, as already discussed, in a given database, the gaps tend to be small. We can thus, instead, use more specialized techniques.

Universal codes. Some techniques encode each gap independently. In this category there are some codes that operate on a bit-by-bit basis, *e.g.*, Unary, Gamma, Delta [40, 37], Zeta [3], Golomb, Rice [43, 42] codes. Those codes are presented in detail in, *e.g.*, [31]. However, codes that operate on a byte-by-byte basis are often preferred, because they allow faster decoding: Variable Byte [27] and its variants, *e.g.*, Varint GB [7], Varint-G8IU [26]. It is also possible to operate on different bases, *e.g.*, Variable Nibble [24, Section 5], which outputs nibbles.

Block codes. Instead of treating the sequence gap by gap, a relatively recent idea is to treat them by block. In this category, the Simple family of codes encode a variable-size number of gaps in a fixed-size output (*e.g.*, 32 bits for Simple9 [2], 64 bits for Simple8b [1]). The Quantities, Multipliers, and eXtractor (QMX) [28] code almost falls in this category, as it outputs most of the time 128 bits, and sometimes 256.

Other types of codes treat a fixed-size number of gaps (*e.g.*, 128 gaps) and output a variable-size number of bytes. The most notable encoding of this type is the Binary Packing, closely related to the Frame-of-Reference (FOR) [12] encoding. It first outputs b, the number of bits required to store the largest of the 128 gaps in binary notation, then outputs the gaps as 128 b-bits numbers. Sometimes, one gap in a block has a big binary magnitude, whereas all the others have a small one. This is called an *exception*. Different authors have come with different solutions to mitigate the cost of such exceptions, and the resulting encodings are called Patched Frame-of-reference (Pfor) [35]. Codes in this category can exhibit high throughput thanks to the use of vectorization, *e.g.*, SIMD-BP128[14, 57].

Another possibility to mitigate the cost of big gaps is to dynamically adapt the block size: Vector of Split Encoding (VSE) [25], Adaptive Frame-of-reference (AFOR) [8, 53].

Last but not least, it is possible to combine one of these methods with another compression method, *e.g.*, Packed+ANS2 [17].

Working directly on the document sequence. Another possibility is to work on the sequence of document IDs, instead of working on the gap sequence. The Quasi-Succinct Indices [29] is a relatively recent re-engineering of the Elias–Fano [10, 11] encoding. This encoding depends on the maximal possible value encoded, so it is quite natural to apply the same block techniques that we have seen before: encoding document sequences by fixed-size chunks is more efficient (*e.g.*, with chunks of 128 IDs), and dynamically adapting the chunk size offers yet another improvement in the compression ratio: it is the Partitioned Elias–Fano encoding [21].

Last, but not least, comes the Binary Interpolative Encoding (Interp) [18]. This technique has set the standard for compression ratio since 1996, and has been recently outperformed, on some datasets, by the Packed+ANS2 method [17]. We note that a variant of Interp exist, that yields better running time but not more compression [5].

Working on the bit vector. Another possibility is to work on the *bit vector*, sometimes also called *bitmap*, associated with the document sequence. There is a wide literature covering this subject, but few papers make the link between this possibility and the previous ones. Indeed, working on bitmaps is mostly done in the database community, whereas working on posting lists is mostly done in the information retrieval area.

We mention it here for completeness, but also because our method is the pendant on trit vectors to what has been applied on bit vectors in the past. Indeed, Bookstein *et al.* [4] worked on contextual methods on bit vectors, and we present, in this article, contextual methods on trit vectors. This method, as other methods that rely on models, can be enhanced with batching [19], see Section 5.3.

A recent study [30] compared 21 compression methods for integer lists, and the interested reader will find in this study references to methods that work on bit vectors.

Remark: as a rule of thumb, most methods that work on bit vectors take more space than methods that work on document sequences, unless the sequences are very dense.

2.2 Reordering the document IDs

As argued before, if the gaps are small — or equivalently, if the document IDs in the posting lists are close to one another — the compression methods will usually benefit from it. Another path has hence been taken to reduce the size of an inverted index: to reorder the document IDs.

The interested reader may find more information about this strategy in [13, Table 1]. In the present paper, we used the state-of-the-art algorithm of [9], more precisely the implementation from [15].

A recent strategy called Clustered Elias–Fano (CPEF) [23, 61] divides the full dataset into clusters, and computes a *reference list* for each cluster. Each document sequence S is then coded as two sequences: the intersection I between S and the reference list of its cluster, and of course $S \setminus I$. For clustering, the authors used the same algorithms as the ones usually used in other reordering strategies. To encode the resulting sequences, they use Partitioned Elias–Fano [21]. What is novel is that the sequences to encode are now different. We slightly modified their algorithm to improve the compression ratio, see Appendix A.

Last but not least, some authors take advantage of the highly repetitive nature of some document collections, *e.g.*, versioned documents: universal indexes for Highly Repetitive Document Collections (uiHRDC) [6] and Pattern Identification Sequentially (PIS) [34]. It would be great to see if those algorithms can be used on our datasets to improve compression. Indeed, a previous e-mail is usually included when we write an answer or when we transfer it, so e-mail datasets are no doubt suitable for such algorithms. Unfortunately, we were only able to test the uiHRDC algorithm, and it gives less good results, because their algorithm was optimized for in-memory size and not for output-file size.

2.3 Suffix trees

Other data structures (*e.g.*, suffix arrays) allow more efficient searches when performing advanced queries, at the cost of additional memory overhead. In this article, we focus on the memory requirement of the index, and will thus not go into the details of these data structures. The interested reader may find more information about this data structure in, *e.g.*, [20].

3 Detailed presentation of our method and implementation details

The main pseudo-code of our method is detailed in Figure 3. In this section, we will detail the different parts.

3.1 Pre-processing

As explained in Section 2.2, there is a rich literature that explains that most compression methods can be used together with a method that reorders the document IDs. We thus first apply graph bisection on each dataset [9, 15]. There has to be somewhere, stored, a bijection between the actual documents and $\{1, \ldots, \text{nbDocuments}\}$. We believe that this reordering do not change the size of this bijection (this bijection is not accounted for in our results).

For our method that uses adaptive arithmetic coding, there is another reordering that yields superior compression rates: a reordering on the word IDs. We store the words by increasing density.

Remark: this time, we believe that the cost of storing the bijection between the actual words and $\{1, \ldots, \text{nbWords}\}$ might be slightly more — because standard techniques that apply, *e.g.*, when the words are sorted alphabetically, are no longer available. We note that if we use a stable sort, the original ordering can be retrieved without needing to store additional data — even though we did not implement it.

3.2 Choosing the contexts

When using a static arithmetic encoding, we make two passes over the data. In a first pass, we collect, for each context, the number of trits that have this context. In the second pass, we will encode those trits according to those occurrences (transformed into probabilities).

We distinguish two kinds of contexts. We would like to have $\gamma = k + w$ trits of context available, but of course, this is impossible for the first trits to encode. We thus resort, for those trits, to smaller contexts. The first trit is encoded with an empty context, the second trit is encoded with a context of only one trit, the third trit is encoded with a context of only two trits, and so on, until we reach $kInit$ trits of context. Then, the context of a trit is the last $kInit$ trits. This way, we encode the

```
0   // Initialization
1   Pre-process the posting lists                                   Section 3.1
2   Set the model parameters according to global properties         Section 3.4
3   Foreach possible context c ∈ C,                                 C is described in Section 3.2
4       Foreach trit t ∈ {0, 1, 2},
5           Initialize a counter for (c, t) at 0
6   // First pass on the data: collect statistics
7   Foreach gap array g in the posting lists,
8       Transform g into a trit list l
9       Initialize a context c to ∅
10      Foreach trit t in l
11          Increment the counter of (c, t)
12          Update c according to t
13  // Write the posting lists properties in the file
14  Foreach gap array g in the posting lists,
15      Write |g| in the file                                       Section 3.5
16  Write the model parameters in the index
17  Foreach possible context c,
18      Convert the counters for (c, 0), (c, 1) and (c, 2) to probabilities p(c)
19      Write p(c) in the file                                      Section 3.3
20  // Second pass on the data: encode the trit lists
21  Foreach gap array g in the posting lists,
22      Transform g into a trit list l
23      Initialize a context c to ∅
24      Foreach trit t in l
25          Arithmetic encoding of t according to p(c)
26          Update c according to t
```

Figure 3: Pseudo-code for our static arithmetic encoding method.

first trits in each trit list until we have encoded $k+w$ trits. Recall that we split γ in two parts $k+w$ because having 3^γ different contexts was not efficient. This explains why we have yet another parameter $kInit$: fixing $kInit$ to $\gamma-1$ is possible but is not efficient, thus we use another, smaller, value.

As we saw in Section 1.4, in the contexts, we "merge" 0s and 1s. This means that we view the contexts [120] and [020], for example, to be the same one, denoted as [NTN] ("T" for two and "N" for not-two). This allows to use bigger values for k and w at the cost of having less precise contexts, and overall this is more efficient.

First trits — merge 0s and 1s

The first $k+w$ trits of each trit list l are encoded with an initial arithmetic contextual method — where the context of the i-th trit (we start counting at 1) is the $min(i-1, kInit)$ previous trits, where the trits 0 and 1 are undistinguished.
There is a number of occurrences to track for the initial contexts equal to:

$$\sum_{i=0}^{kInit} 3 \times 2^i = 3 \times \frac{1-2^{kInit+1}}{1-2}$$

Rest of the trit list — merge 0s and 1s

The rest of the trits are encoded with another contextual arithmetic method — where the context of a trit is the k previous trits (0 and 1 are undistinguished) and the number of 2 in the w trits before those k trits.
There is a number of occurrences to track for the general contexts equal to:

$$3 \times (w+1) \times 2^k.$$

3.3 Storing the probabilities

For each context c, we now have the number of times, in the trit lists to encode, where we have a 0 with this context ($= occ(c,0)$), where we have a 1 with this context ($= occ(c,1)$), and where we have a 2 with this context ($= occ(c,2)$).

We must now find a good encoding of all those numbers $occ(c,t)$. We could keep things exact and store, for each context, the three values. We found that it was more effective to let the model be a little bit inexact, but only store two values. To do so, we normalize, for each context, those three values (*e.g.*, we take integer values $occ'(c,0), occ'(c,1)$ and $occ'(c,2)$ such that $\sum_t occ'(c,t) = 255$ and such that for each trit u, $occ'(c,u) \approx occ(c,u) \times \frac{255}{\sum_t occ(c,t)}$). We can then encode those three values on only 16 bits by encoding $occ'(c,0)$ and $occ'(c,1)$ on 8 bits each (they are values no greater than 255), and deducing $occ'(c,2) = 255 - occ'(c,0) - occ'(c,1)$.

The more bits we use to encode those normalized occurrences, the closest this approximate model will be to the actual data, but the more space it takes to store it in the index for decoding.

3.4 Choosing the parameters

A crucial point in every method that needs parameters is the choice of those parameters.

Static Arithmetic Coding

Our parameters, for the static arithmetic encoding, are:

- $k, w, kInit$ for the contexts
- the number of bits on which to normalize the occurrences

We made experiments on our datasets that showed that (a) the best value for k was growing in accordance to nbPointers, (b) a value for w close to k was the best choice, (c) all $kInit$ choices neither too small nor too big with respect to k were all good choices, and (d) normalizing the occurrences on 8 bits was a very good choice.

We thus choose our parameters to be:

$$k, w = k + 1, kInit = \lceil k/3 \rceil \text{ and number of bits } = 8,$$

where k is chosen as big as possible without making the storage space of the model, in bits, going further than 2% of nbPointers.

As a small optimization, we remarked that, for any database with nbDocuments documents in total, if the last $\lceil \text{nbDocuments} \rceil - 1$ trits of context are all different from 2, then the next trit will always be a 2 (because there is no gap bigger than nbDocuments). We thus made sure that this "free 2" was available to the eyes of our encoder, by making sure that $k + w$ would be equal to this value, in the case where a bigger value for w would have been selected automatically.

Adaptive Arithmetic Coding

Unlike in the static case, the symbol occurrences we are handling are always estimates of the true ones, based on what has been read so far (the encoding is done in one pass, thus we have never access to the occurrences of the symbols in the full message). First, we provide initial estimates of the probabilities, and then, we periodically "forget" some of the occurrences that have been read so far, in order to adapt to the latest symbols read [41]. Last, we must also find good values of the parameters, exactly as in the static case. When using an adaptive arithmetic encoding, we do not need to store the model in the index, so the choice of those parameters is not according to a balance between the precision of the model and its size. Instead, it is a balance between the precision of the model and the cost associated with the time needed to initialize it.

Our parameters, for the adaptive arithmetic encoding, are:

- $k, w, kInit$ for the contexts
- the initial estimates of the probabilities $P(i|C)$, for each trit i and each context C
- the number of steps N before dividing by two the occurrences

After some experiments, we chose our parameters to be:

$$\begin{cases} w = k = \left\lfloor \dfrac{\log \text{nbPointers}}{a} + b + 0.5 \right\rfloor, \text{ where } a = 1.67264 \text{ and } b = -2.24758, \\ kInit = min(2 \times k - 1, 16), \\ P(i|C) = \dfrac{1}{3} \text{ thus initial occurrences of 1,} \\ \text{and number of bits } = k \end{cases},$$

the number of bits set to k meaning that we halve all the occurrences every $N = 2^k$ steps.
Remark: the values for a and b have been set thanks to a linear fitting.

3.5 Storing the posting lists lengths

We store explicitly, for each posting list, its total length — the number of gaps in it — before encoding it. Indeed, to be able to decode the output of our arithmetic encoder, it suffices to know in advance when to stop decoding. Here, the number of gaps in a posting list is equal to the number of "2" in its associated trit list. As soon as we have decoded that number of "2", we know we reached the end of the encoding.

Storing those lengths is not mandatory, though, and we picture a variant in Section 5.

We tried different strategies to encode those lengths, and the best performing one is to encode those lengths with the Delta method.

Remark: in our datasets, the encoding of those lengths remains negligible in front of the encoding of the posting lists. It is possible, though, that on other datasets, another method would be better to encode those lengths.

4 Methodology and results

4.1 Methods studied and implemented

Our goal in this paper was to search for the best compression method on inverted indexes, without any further requirement. There are already a lot of implementations that can be directly used to compress inverted indexes, but their purpose is usually different: they also want to minimize the time elapsed during queries in the index. By doing so, they usually add an overhead. Needless to say, each team working on the subject has a different way of dealing with those overheads, and it is most of the time not possible to compare the compression ratios given in different papers. Furthermore, it is sometimes useful to combine different techniques, and it cannot be done without a common framework. We thus re-implemented methods from the state-of-the-art, in `Java`. The result is the repository `https://gitlab.inria.fr/ybarsami/compress-lists`, that combines the following compression methods:

- the methods presented in this paper
- the Interp method [18], together with its variants. We rewrote the method with all the optimizations, as described in the paper — note that some optimizations are not applied in other papers that use it.
- the methods Binary Packing [12], FastPFOR [14], NewPFD, OptPFD [32], Simple9 [2], Simple16 [33], directly taken from the JavaFastPFOR library [57].

- the methods QMX [28], Simple8b [1], whose implementation straightforwardly derive from the one of Simple9 from the JavaFastPFOR library in the previous point.
- the method AFOR3 [8], directly taken from the SIRENe library [53].
- the Quasi-Succinct indices [29], also known as the Elias–Fano method, and its refinements with partitioning (PEF [21]) and with clustering (CPEF [23]). The partition algorithm is a simple `Java` translation of the code from the original repository [60]. The clustering algorithm has been entirely rewritten from the original paper, in order to be able to use not only the Elias–Fano method, but other ones as well. We also modified the code to add two optimizations, see Appendix A.
- a variant of the Packed+ANS2 method [17]. We re-implemented the algorithm described in the article, and, in our variant, we use arithmetic coding instead of ANS. This allows us to gain space in the index with respect to the original implementation (thus, providing more than fair comparisons), see Appendix A.
- the Four-State Markov models that work on bit vectors from [4]. They are contextual arithmetic methods, so we used the arithmetic coder from the Reference arithmetic coding library [59], and re-implemented the ideas of the paper from their description.
- the Bernoulli, Golomb and Rice [43, 42] methods, which also use the arithmetic coding.
- and finally the Unary, Binary, Delta, Gamma [40, 37], Zeta [3], Variable Byte [27] and Variable Nibble [24] methods, which are straightforward to implement given their expressions. It is worth noting that for these methods, we explicitly use the fact that we never have to code the integer 0. The encoding is thus the same as described in "Managing Gigabytes" [31], even though in other libraries, the authors chose to be able to encode 0.

That being said, there still remains a lot to be said about how to encode an inverted index with a given method.

4.2 Datasets

To validate our approach, we used different datasets. Some datasets are commonly used in the state-of-the-art, and we added e-mails datasets, because one of our concerns was the use of inverted index to search through e-mails. Indeed, one of the applications we have in mind, for this work, is an encrypted mail service. In such a service, the compressed lists would be stored in the cloud, and would have to be sent to the users that want to use it without using permanent storage on their device. In that case, unlike most works on inverted index compression, we need efficient compression even for small datasets.

- our personal mails: we computed the inverted indexes of 3 personal e-mail boxes, 2 of which are from the authors.
- the Enron corpus of e-mails [49, 50]: this corpus contains professional e-mail boxes of 150 persons. We cleaned the original folders by removing duplicate e-mails, as suggested [49, Section 2]. We present results on the e-mails of 4 users who have the biggest e-mail boxes in the corpus: the e-mails of Dasovich, Jones, Kaminski, and Shackleton.
- manuals / books: we took four commonly used databases for inverted indexes: the Authorized Version (King James') Bible [48], a bibliographic dataset "GNUbib", a collection of law documents "Comact" (the Commonwealth Acts of Austria), and a collection of documents from the Text Retrieval Conference (TREC disks 4 and 5 [52]).

Document collection	Mail1	Mail2	Mail3
Number of documents	32 380	7 691	26 195
Number of words	90 611	48 455	116 765
Number of gaps	6 169 953	1 394 581	4 979 238
Total size (MB)	82.6	20.4	n/a

Table 1: Main properties of our personal e-mails datasets.

Document collection	Dasovich	Jones	Kaminski	Shackleton	ENRON
Number of documents	15 748	10 646	11 348	10 576	284 982
Number of words	77 871	27 691	48 174	30 250	518 412
Number of gaps	3 606 327	1 270 340	1 701 536	1 404 902	42 253 227
Total size (MB)	64.9	16.7	21.5	17.3	526

Table 2: Main properties of datasets from the Enron corpus.

- web crawling: we took two commonly used databases, which are substantially bigger than the other datasets: Gov2 and ClueWeb09 [51].

We do not index directly all the words in the datasets, but we first normalize the text: case-folding ("C" is treated as "c"), accent removal ("é" is treated as "e"), splitting of unicode characters representing two letters ("œ" is treated as "oe"). The resulting text is then split into words (which are any sub-string contained into non alphanumerical characters), and those words are finally stemmed (no stop-words were removed — frequent words have posting lists which are the most compressed in indexes). We used the Porter2 algorithm, improved from the Porter algorithm [47], as implemented in the Snowball library [62]. To index e-mails in MIME format, we made the following choices:

- we first parse the e-mails with Apache Mime4j [54]
- we index the headers (date, subject, from, to, cc, bcc)
- we do not index attached files
- when we encounter a `text/html` part, we first extract the text from the html code with `jsoup` [55] (to avoid indexing words in tags)
- when we find a url in the text ("http://...", "https://..." or "mailto:..."), we remove what is after question marks (parameters of mailto or of php webpages), and remove what is between two consecutive "/" if the number of characters exceeds 30 (to avoid indexing what is usually hashed text).

While this hand-crafted normalization process is not perfect, it follows the "Keep It Simple Stupid" idea.

Tables 1, 2, 3 and 4 give the main properties of those different datasets.

4.3 Results

We compare here 5 different methods:

Document collection	Bible	GNUbib	ComAct	TREC
Number of documents	31 102	64 267	261 829	528 155
Number of words	9 423	51 910	296 351	1 098 349
Number of gaps	705 989	2 228 876	12 919 692	119 802 501
Total size (MB)	4.44	14.1	135	1 643

Table 3: Main properties of manuals / books datasets.

Document collection	Gov2	ClueWeb09
Number of documents	25 205 179	50 220 423
Number of words	1 107 205	1 000 000
Number of gaps	5 979 715 441	15 641 166 521
Total size (MB)	n/a	n/a

Table 4: Main properties of web crawling datasets.

1. The (Optimized) Interp method [18].
2. A variant of the Interp method that works by blocks of 128 integers (to allow better comparison against the next method, that also uses blocking).
3. A variant of the Packed+ANS2 method [17], that we call Packed+Arithmetic, that uses arithmetic coding instead of ANS.
4. Our TC Method: TritContext method, with the generic parameters presented in Section 3.4.
5. Our TCA Method: TritContextAdaptative method, with the generic parameters presented in Section 3.4.

Tables 5, 6, 7 and 8 present our results. Before applying the different methods, we first apply to the datasets the bisection (b) algorithm [9, 15] and sort (s) the resulting integer lists by increasing length, hence the (bs) in the dataset names. The gain corresponds to the gain in percentage of our best method (in bold) compared to the Interp method. All results are written in bits/pointer and include the size required to compute the number of occurrences for each word, encoded using the Delta [40, 37] method. All values are rounde at ± 0.001 bits/pointer. The gain is rounded $\pm 0.01\%$.

	Mail1 (bs)	Mail2 (bs)	Mail3 (bs)
Interp	3.012	4.295	4.334
Block-Interp	3.093	4.414	4.441
Packed+Arithmetic	2.868	4.276	4.272
TC	2.692	4.097	4.114
TCA	**2.618**	**3.935**	**4.026**
Gain	13.10%	8.39%	7.12%

Table 5: Results on our personal e-mails datasets. On each dataset, we applied the bisection (b) algorithm [9, 15] and the resulting integer lists were sorted (s) by increasing length.

	Kaminski(bs)	Jones(bs)	Shackleton(bs)	Dasovich(bs)	ENRON(bs)
Interp	4.520	3.843	4.073	4.038	4.043
Block-Interp	4.607	3.907	4.144	4.084	4.107
Packed+Arithmetic	4.608	3.916	4.170	4.105	4.051
TC	4.440	3.695	3.979	3.945	3.868
TCA	**4.323**	**3.572**	**3.890**	**3.882**	**3.761**
Gain	4.37%	7.06%	4.49%	3.86%	6.96%

Table 6: Results on datasets from the Enron corpus. On each dataset, we applied the bisection (b) algorithm [9, 15] and the resulting integer lists were sorted (s) by increasing length.

	Bible(bs)	GNUbib(bs)	ComAct(bs)	TREC(bs)
Interp	**5.326**	4.108	3.894	4.529
Block-Interp	5.429	4.296	4.048	4.603
Packed+Arithmetic	5.624	4.277	4.023	4.628
TC	5.358	4.037	3.824	4.552
TCA	5.354	**4.015**	**3.798**	**4.489**
Gain	-0.52%	2.25%	2.46%	0.90%

Table 7: Results on manuals / books datasets. On each dataset, we applied the bisection (b) algorithm [9, 15] and the resulting integer lists were sorted (s) by increasing length.

	Gov2(bs)	Gov2(us)	ClueWeb09(bs)	ClueWeb09(us)
Interp	2.690	3.485	4.309	4.999
Block-Interp	2.722	3.522	4.338	5.028
Packed+Arithmetic	2.708	3.462	4.274	4.982
TC	2.579	3.344	4.123	4.803
TCA	**2.575**	**3.311**	**4.119**	**4.763**
Gain	4.30%	5.00%	4.41%	4.72%

Table 8: Results on web crawling datasets. On each dataset, we applied the bisection (b) algorithm [9, 15] or we let the documents Ids sorted with a lexicographical sorting on the corresponding URIs (u) [25, Section 5], and the resulting integer lists were sorted (s) by increasing length.

5 Detailed analysis, interpretation and variants of our method

5.1 Choosing the parameters

In Section 3.4, we identified automatic parameters to be set with probabilities normalized on 8 bits, and such that the total size of the model would not exceed, in bits, 2% of nbPointers.

We also set our parameters $w, kInit$ such that $w = k + 1$ and $kInit = \lceil k/3 \rceil$.

It is, in the general case, possible to find better parameters. However, of course, finding the best set of parameters would be very time consuming. For instance, on Gov2, we found that normalizing probabilities on 16 bits and having the total size of the model not exceeding 0.5% of nbPointers was slightly better.

5.2 More subtle contexts

At the beginning of our work is the empirical establishment that contexts are very useful for entropy coding. We then searched a balance between the precision of the contexts used and the size of the probabilities needed to store those contexts.

The most efficient compromise we found was to divide a context of size γ into the k last trits, and the w trits before. On the k last trits, what is looked at is whether each trit is a 2 or not, and on the w trits before, what is looked at is the number of 2 in total.

We also tested to have a larger number of sub-boxes inside the γ last trits, and it did not turn out to be useful. For instance, we would divide the γ last trits in the k last trits, where we would, as before, take the full context (just merging 0s and 1s), then look at the w_0 previous trits, only counting the number of 2 in it, then the w_1 trits before, also only counting the number of 2 in it, etc.

In a way, our experiments are a specialized version of the more general GRASP algorithm pictured in [45], which also tries to merge contexts in order to get information from a greater context without having to pay for the full price. It would be interesting to see if this more general idea could lead to better compression rates, and to analyze the kind of contexts that would be merged together.

5.3 Batching

We can refine our contextual methods with batching. We experimented and found that sometimes, a logarithmic batching (with respect to the gap list lengths) gives slightly better results (on half of the datasets), but the gains were marginal. We used 7 batches and use, for each batch, the best method between Interp, TC and TCA. Our batches have the following delimiters:

$$\{0, 0.0002, 0.001, 0.01, 0.03, 0.1, 0.2, 1\}$$

in terms of pointer density (gap list lengths divided by the nbDocuments).

5.4 Quatritlist

In all previous papers, and in previous sections, the index was always created with an explicit storage of the gap array lengths. For most methods, this knowledge is mandatory. But in our case, we can avoid storing the gap array lengths. When using arithmetic coding, in order to decode we need a way of knowing when to stop decoding. Instead of using the gap array lengths, we can thus encode a special "end" character, at the end of each gap array. If we note this special character "3", and we are now dealing with quatrits (integers in $\{0, 1, 2, 3\}$).

We tested all our methods with quatrits, and the results are equivalent to the results with the explicit storage of the gap array lengths. However, if one is dealing with a dataset for which this storage of lengths is higher, this method surely helps.

5.5 Context on the bit vector

In Section 2, we noted that previous works on inverted index compression also used contextual encodings, *e.g.*, contextual arithmetic encoding on bit vectors [4]. We thus also tested contextual encodings on the bit vector. Our experiments showed that these methods are outperformed by our

trits — even though contextual encodings on the bit vector are also outperforming both Interp and Packed+Arithmetic methods. It would be interesting to see if a more careful setting of parameters would lead to better results.

5.6 SplitB2B01 methods

Instead of directly working on trit lists, we also tried to split the trit list in different ways. In the SplitB2B01 methods, we first transform each trit list in two parts: the part that contains only the 2, and the part that contains only the 0 and 1.

SplitB2B01 methods

$$g = [4, 1, 1, 3, 5, 2] \Rightarrow l = 002221201202 \Rightarrow \begin{cases} B2 = 001110100101 \\ B01 = 001010 \end{cases}$$

$B2$ is obtained from l by replacing each 0 and 1 by a 0, and each 2 by a 1. $B01$ is obtained from T by removing all the 2.

On each of those bit vectors $B2$ and $B01$, we tried to apply different methods. Our idea was that, in $B2$, there is more clustering than in the original bit vector associated to the posting list, and it could potentially be worth trying to go in that direction. Nevertheless, we did not achieve better compression rates when doing this.

5.7 SplitTLB01 methods

The SplitTLB01 methods do not work directly on the gap array g nor on the trit list T as the context methods. Instead, they first transform the gap array g in the following way. They first build gl, the gap lengths array — the array that contains the *lengths* of the gaps (number of bits needed to write each gap in base 2) and $B01$, the sub-list of T containing only the 0 and 1. Indeed, it is easy to reconstruct T from gl and $B01$. Let us see that on the same example $g = [4, 1, 1, 3, 5, 2]$. In this array, $4_{10} = 100_2$ needs 3 bits to be written in base 2. $1_{10} = 1_2$ needs 1 bit to be written in base 2, etc. We thus have the gap length array $gl = [3, 1, 1, 2, 3, 2]$. In this example, $T = 002221201202$ and the sub-list containing only the 0 and 1 is thus equal to $B01 = 001010$.

If we want to reconstruct g, it is easy: $gl[0] = 3$ means that $g[0]$ takes 3 bits. In T, we removed the leading 1, hence in $B01$, the first 2 bits come from $g[0]$. T thus starts with 002. Then $gl[1] = 1$ means that $g[1]$ takes 1 bit. In T, we removed the leading 1, hence in $B01$, the next 0 bits come from $g[1]$. T thus starts with 0022. etc.

As in the context methods, we do not work directly on gl but we work on its associated trit list. In our example, the binary representation of the numbers in gl is $[11, 1, 1, 10, 11, 10]$. The associated trit list is thus $TL = 1222021202$ (we remove the leading 1 in each of the gap lengths as all the gap lengths gl_i verify $gl_i \geq 1$).

SplitTLB01 methods

$$g = [4, 1, 1, 3, 5, 2] \Rightarrow \begin{cases} gl = [3, 1, 1, 2, 3, 2] \Rightarrow TL = 1222021202 \\ T = 002221201202 \Rightarrow B01 = 001010 \end{cases}$$

On those two vectors TL and $B01$, we tried to apply different methods. Our idea was the same: in TL, there would be more clustering than in the original trit list T, and it could potentially be worth trying to go in that direction. Nevertheless, we did not achieve better compression rates when doing this neither.

6 Conclusion

In this article, we developed a novel idea to compress inverted indexes, that beats the Interp and Packed+ANS2 methods, two methods that are the state-of-the-art regarding compression ratios. We applied our method to a wide range of datasets, including e-mail datasets.

We understand that, in most applications, the most useful feature is not always compression. We hope that, in the future, the trits we used in this paper can be used together with other algorithms, to provide better time complexities, while maintaining the compression rate showed in this paper.

We mentioned contributions taking advantage of the highly repetitive nature of some document collections, *e.g.*, versioned documents. We hope that, in the future, it will be possible to combine those ideas with ours.

Last but not least, we think that our methods can still be enhanced. Throughout this paper, we highlighted that the contextual probabilities of the trits are the key to the success of our methods. We tried, in different ways, to reach a balance between the precision of the probabilities (bigger contexts) and the cost of the model (in case it needs to be stored) or the time needed to read sufficiently enough trits to have a good estimate of the true probabilities (in case the model is built on-the-fly). What we did not do is to look at what those probabilities look like. We think that it could probably enhance our methods if we could look at the probabilities and extract a pattern from them, or use machine learning to enhance prediction, as is done in cmix [56] and PAQ8 [44]. This is left for future work.

Acknowledgments. We would like to thank A. Moffat for unearthing the GNUbib and ComAct datasets. We would like to thank K. Merckx for porting to our attention the ENRON dataset. We would like to thank the authors of the various softwares we used in this experiment. No comparison could have been done without their previous efforts. We directly included in our code, by alphabetical order, "JavaFastPFOR" [57], "jsoup" [55], "Mime4j" [54], "Reference arithmetic coding" [59], "SIRENe" [53], and "Snowball" [62]. We also used the following softwares, by alphabetical order, "CPEF" [61] (to split the dataset by clusters), and "PISA" [58] (to reorder the datasets using graph bisection). Following their path, we also provide the source code of our experiments, see https://gitlab.inria.fr/ybarsami/compress-lists. Special thanks to the Thalys train which made this collaboration easy.

A Detailed comparison to the state-of-the-art

The Interp method was first described in 1996 [18].

In 2009, the method was claimed to be improved by remarking that blocking could improve the overall compression ratio [32, Table 1]. This was not true. Already in the original article, the authors remarked that blocks of size $2^n - 1$ were beneficial [18, Section 6], and the "balanced recursion" refinement already made recursions have these sizes whenever it is possible.

In 2017, a new method, based on clusters, claimed to beat the Interp method on one of the two datasets tested [23, Table VIII]. In 2018, a new method, Packed+ANS2, also claimed to beat the Interp method on two datasets similar from the previous paper [17, Table 5]. However, those two papers do not compare their results to the original Interp method, but to a modified version of the Interp method, where a posting list is not coded as a whole, but rather divided by blocks (of size 128), with additional overhead to enable fast queries inside those blocks. To enable fast queries, in addition to the encoding of each block of 128 integers is added, for each block, the maximum value of an integer in that block and the size, in bytes, of the encoded block. This way, one can "jump" over an encoded block without having to decode it if needed. As an example, we show the encoding of a document array of size 81, with blocks of size 32.

The first block covers the document IDs from 84 to 6601. The maximum of that block is 6601. The second block covers the document IDs from 6654 to 12644. The maximum of that block is 12644. Because the previous maximum was 6601 (and it is a known value when decoding), we remove 6601 to all document IDs before encoding that block. We hence encode document IDs between 53 and 6043. The second block covers the document IDs from 12736 to 15428. The maximum of that block is 15428. Because the previous maximum was 12644 (and it is a known value when decoding), we remove 12644 to all document IDs before encoding that block. We hence encode document IDs between 92 and 2784.

If we want to know whether 12345 is in that posting list, we do not need to decode the first block: all values in the first block are ≤ 6601, so 12345 may not be in that block. We thus use the pointer to the second block (because $6601 < 12345 \leq 12644$), and directly decode the second block to answer this question (and find out that the answer is "no").

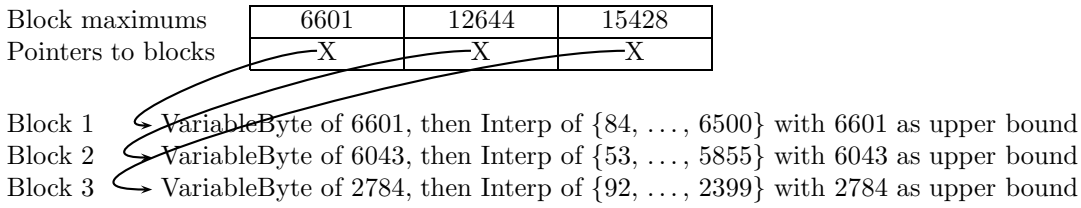
However, when encoding a block of integers (here, the blocks are of size 2^k, but this also holds in the general case) with the Interp method, it is a good idea to first encode the largest value of the block. Indeed, the Interp method requires to know an upper bound on the largest value. One could always use the number of documents as an upper bound, but this would result in a very inefficient way of encoding small blocks of the full posting list. Writing the maximum value in the block, followed by an encoding of the remaining $2^k - 1$ values is better (we use a better upper bound, and we are now left with a block of size $2^k - 1$, optimal for the recursion of the Interp method). Thus, this information is written twice in the index. And indeed, if we go through the code, we find, at line 281 in `block_codecs.hpp` from the repository of the Packed+ANS2 method [17], the following comment:

```
// XXX wasting one word!
```

The same code appears in `block_codecs.hpp` from the repository of the CPEF method [23] — although, without this comment. Wasting a word per block of 128 integers makes a noticeable difference on the resulting index size, hence, we think that the comparisons made in those two articles are misleading, and a comparison to an optimized implementation of the block-Interp

is needed, with and without enforcing fast-skipping of blocks. This comparison is done in this appendix.

We picture in Figure 4 the encoding as it is implemented in those repositories. It is obvious from this figure that the VariableByte encodings of 6601, 6043 and 2784 can be avoided, as they can be recomputed from the block maximums array. This optimization leads to a better space usage without using more time (neither to construct, nor to query) — it would even marginally use less time because a subtraction of two integers already in the memory is probably quicker than a decoding of the VariableByte method.



$\{84, \ldots, 6500\}$ are the first 31 document IDs. $\{53, \ldots, 5855\}$ are the document IDs number 33 to 63 from which 6601 was subtracted. $\{92, \ldots, 2399\}$ are the document IDs number 65 to 80 from which 12644 was subtracted.

In the actual implementation, there is a slight optimization for the values encoded with VariableByte. Because we know the block size (always 32 except for the last block whose size can be deduced because we know the full size of the gap array — here $81 \mod 32 = 17$), we can always subtract the block size to the value encoded. Indeed, the document IDs are strictly increasing, thus it is safe to subtract the block size to the maximum value, it will still be a positive number, and it is straightforward to reconstruct the value at decoding time, by adding back the block size that was subtracted.

Figure 4: Example encoding of a document array of length 81 with blocks of size 32 with the Block-Interp method as implemented in [61].

PS: For the blocked-Elias–Fano method, there is a specialized code in those repositories. We did not check whether their implementations avoid this waste.

Another difference is that in the two articles, the methods are only tested on a subset of the full dataset, where the small posting lists are removed. In [22], the same methods are tested on the same datasets, but without removing from the datasets the small posting lists, and the Interp method is shown to beat the two other methods.

The Gov2 and ClueWeb09 datasets we have at our disposal are already versions of the datasets where the small posting lists were removed (in Gov2 (resp. ClueWeb09), the posting lists have lengths no less than 100 (resp. 250) postings). These datasets are thus really close to the datasets filtered in [17]. To reproduce as closely as possible the work in [23], we also present results, in this appendix, on datasets where even more small posting lists are removed (in Gov2(*f) (resp. ClueWeb09(*f)), the posting lists have lengths no less than 14182 (resp. 16618), to match [23, Section 4.1, Table II]).

In our repository[8], we first re-implemented the different versions of Interp. Then, we re-implemented the cluster method, with two changes from the original CPEF method: (a) we only considered the frequency-based construction for the reference list, to be able to also use

[8] https://gitlab.inria.fr/ybarsami/compress-lists

the cluster method in concordance with other methods than the PEF method and (b) we found and implemented an optimization with respect to index size called "shrinking". Recall that each document sequence S is coded as two sequences: the intersection I between S and the reference list of its cluster, and of course $S \setminus I$. For our optimization, instead of coding directly the integers in $S \setminus I$, we first apply a mapping m from $S \setminus I$ to $\mathbb{N}$ where $m(x) = x - \#i \in \text{referencelist} | i < x$. Last, we implemented a method as close as possible to the Packed+ANS2 method, but using arithmetic coding instead of ANS coding. We called this method Packed+Arithmetic. The additional differences between the original method and our implementation are that (a) we did not optimize the RAM usage — there is still a huge array allocated for the probabilities (hence, we do not use "ANSmsb" from [17, Section 3.3]) and (b) we keep all the $16 \times 17/2 = 136$ contexts instead of merging them in 64 different ones. The use of 64 contexts in the original paper is done in order to keep the number of bits used for the context as low as 6 (instead of 8), thus gaining 2 bits per block. Here, we code the selector for the context with an arithmetic coder and as a result, it takes less than 6 bits, even with the 136 contexts (between 4.5 and 4.9 bits per selector on mail datasets, 4.6 bits per selector on Gov2), and permits to distinguish more contexts, thus gaining space also on the compression of the gaps. We show the results of these different methods in Table 9. As can be seen in the table, the Interp method is almost always the best, even though the Packed+Arithmetic method is close — sometimes better. Blocking makes a noticeable difference (2.7% on Mail1, 1.6% on TREC, 1.2% on Gov2, more if we add padding), and we recall that in the current article, we provide comparisons against the original Interp method, which provides the best compression ratio.

B Implementation details

Our implementation is available at `https://gitlab.inria.fr/ybarsami/compress-lists`. We use the arithmetic encoder from the Reference arithmetic coding library [59]. We slightly modified the original codebase in order to more efficiently use adaptive contexts.

All our methods work gap list by gap list. They first read a full gap list from an index, store it in main memory in an `IntsRef` (it is more or less the `Java` equivalent of a pointer on an array of `int`s), and then work on this `IntsRef`. This design choice was made in order to facilitate the use of the "SIRENe" [53] (implementation of the AFOR [8] method) library that uses this data structure, and it also facilitates the use of the "JavaFastPFOR" [57] library, that uses `int[]` to store the gap lists.

If one is interested in the running time of the methods implemented in our repository, it would be more efficient to avoid the use of this intermediate data structure, that incurs a $2x$ overhead in memory transfers (first, read from file and store in the array, then read from this array and do the computations).

The implementation of the bit lists, trit lists and quatrit lists all use the `java.util.BitSet` data structure internally, in order to be memory efficient. This is not required as those lists may be constructed on the fly instead of being stored in main memory — there is another $2x$ overhead in memory transfers behind this choice.

All the methods are implemented inside the `gaparrayencoding` folder.

	Mail1 (b)	TREC(b)	Gov2(b)
Interp	2.323	67.83	2011
Block-Interp (padding)	2.422	69.48	2048
Block-Interp (no padding)	2.385	68.93	2035
Packed+Arithmetic (no padding)	2.212	69.30	2024
Contextual trits (no padding)	2.019	67.22	1925
Interp, clusters (no padding)	2.384	69.31	2039
Interp, shrinked clusters (no padding)	2.349	67.97	2001
EF	4.252	91.21	5883
Block-EF (no padding)	3.631	79.51	2812
PEF-approx (no padding)	2.497	78.78	2234
PEF-opt (no padding)	2.456	73.52	n/a
PEF-approx, clusters (no padding)	2.558	80.07	2268
PEF-approx, shrinked clusters (no padding)	2.515	78.60	2225
PEF-opt, clusters (no padding)	2.501	n/a	n/a
PEF-opt, shrinked clusters (no padding)	2.461	73.61	n/a

The Block-* methods use blocks of fixed-size 128. The PEF* methods use dynamic block sizes, where the block sizes are computed to minimize the total cost (opt means that the optimum is found thanks to dynamic programming, approx means it is an $(1 + \epsilon)$ approximation of that optimum, thanks to the algorithm from the original article [21]). For all block methods, "padding" means that each block encoding is padded to the end of one byte, whereas "no padding" means that there is no such padding. The "opt" algorithm takes quadratic time, and thus did not finish in a reasonable amount of time on bigger datasets. This is why some results are missing.
On each dataset, we applied the bisection (b) algorithm [9, 15].

Table 9: Total index size (in MB) for different methods on various datasets.

	Gov2(bf)	Gov2(uf)	ClueWeb09(bf)	ClueWeb09(uf)
Interp	1397	1755	6942	7994
Block-Interp	1422	1779	6991	8043
PEF	1593	1984	7870	8965
Packed+Arithmetic	1412	1765	6926	8055
Contextual trits	1354	1686	6684	7679
Clusters-Packed+Arithmetic	1421	1612	6914	7586
Clusters-Interp	1385	1569	6895	7501
Clusters-Contextual trits	1375	1534	6695	7302
Clusters-PEF	1582	1789	7829	8464
CPEF ("space-based", estimation)	1526	1725	7693	8319
(Skipping) PEF [61]	1761	2216	8717	10016
(Skipping) CPEF [61]	1795	2066	8944	9540
(Skipping) Packed+ANS2 [17]	1600	1956	7494	8635
(Skipping) Block-Interp [61]	1711	2044	7773	8812
(Skipping) Block-Interp (optimized)	1649	1982	7562	8601

The Gov2 (resp. ClueWeb09) dataset from [51] has been filtered (f) by removing all posting lists whose size was strictly less than 14 182 (resp. 16 618), see [23, Table II].
In our implementation of the CPEF method, we use the frequency-based method to compute the reference lists. For a reference list size of 800K for Gov2 (resp. 1600K for ClueWeb09), the authors report [23, Figure 7] a 3% (resp. 1.5%) gain in the index space. In their paper, this percentage takes into account an overhead that is not present in our paper (to allow block skipping). We thus took this percentage w.r.t. the value given by their implementation to output the "CPEF, estimation" line in the first part of this table.
In the implementation of [61], we only show the results for the "docs", not the "freqs" (we had to artificially add freqs to the dataset [51] to be able to use [61], to chose to put all equal to 1). We note that we modified the Packed+ANS2 [17] implementation to avoid modeling the "freqs". Indeed, in the original implementation, a unique model is used to encode both the "docs" and the "freqs" part — the space compression ratio we show is thus better than the one that you would obtain by using their unmodified implementation.
When removing the "redundant max", we instrumented the code to compute the size of the unnecessary VariableByte encodings, see Figure 4.
On each dataset, we applied the bisection (b) algorithm [9, 15] or we let the documents Ids sorted with a lexicographical sorting on the corresponding URIs (u) [25, Section 5], and the resulting integer lists were sorted (s) by increasing length.

Table 10: Total index size (in MB) for different methods on various filtered datasets.

	Gov2(b)	Gov2(u)	ClueWeb09(b)	ClueWeb09(u)
Interp	2011	2605	8425	9773
Block-Interp	2048	2632	8481	9830
Packed+Arithmetic	2024	2588	8357	9740
Contextual trits	1925	2475	8054	9313
(Skipping) Block-Interp [61]	2481	3068	9551	10887
(Skipping) Packed+ANS2 [17]	2221	2803	8958	10359
(Skipping) Block-Interp (optimized)	2401	2988	9305	10641

Table 11: Results on web crawling datasets. On each dataset, we applied the bisection (b) algorithm [9, 15] or we let the documents Ids sorted with a lexicographical sorting on the corresponding URIs (u) [25, Section 5].

Books and articles — Inverted index compression techniques

[1] V. N. Anh and A. Moffat. "Index Compression Using 64-Bit Words". In: *Software — Practice & Experience* 40.2 (2010), pp. 131–147. DOI: 10.1002/spe.948.

[2] V. N. Anh and A. Moffat. "Inverted Index Compression Using Word-Aligned Binary Codes". In: *Information Retrieval* 8.1 (2005), pp. 151–166. DOI: 10.1023/B:INRT.0000048490.99518.5c.

[3] Paolo Boldi and Sebastiano Vigna. "Codes for the World Wide Web". In: *Internet Mathematics* 2.4 (2005), pp. 407–429. URL: https://projecteuclid.org:443/euclid.im/1150477666.

[4] A. Bookstein, S. T. Klein, and T. Raita. "Modeling Word Occurrences for the Compression of Concordances". In: *ACM Transactions on Information Systems* 15.3 (1997), pp. 254–290. DOI: 10.1145/256163.256166.

[5] C.-S. Cheng, J. J.-J. Shann, and C.-P. Chung. "Unique-order interpolative coding for fast querying and space-efficient indexing in information retrieval systems". In: *Information Processing & Management* 42.2 (2006), pp. 407–428. DOI: 10.1016/j.ipm.2005.02.002.

[6] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. "Universal indexes for highly repetitive document collections". In: *Information Systems* 61 (2016), pp. 1–23. DOI: 10.1016/j.is.2016.04.002.

[7] J. Dean. "Challenges in building large-scale information retrieval systems: invited talk". In: *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining*. WSDM '09. Association for Computing Machinery, 2009. DOI: 10.1145/1498759.1498761. URL: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/WSDM09-keynote.pdf.

[8] R. Delbru, S. Campinas, and G. Tummarello. "Searching Web Data: An Entity Retrieval and High-Performance Indexing Model". In: *Journal of Web Semantics* 10 (2012), pp. 33–58. DOI: 10.1016/j.websem.2011.04.004.

[9] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. "Compressing Graphs and Indexes with Recursive Graph Bisection". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. Association for Computing Machinery, 2016, pp. 1535–1544. DOI: 10.1145/2939672.2939862.

[10] P. Elias. "On binary representations of monotone sequences". In: *Proceedings of the 6th annual Princeton Conference on Information Sciences and Systems*. Department of Electrical Engineering, Princeton University, 1972, pp. 54–57. URL: https://books.google.fr/books?id=BIcpAQAAMAAJ.

[11] R. M. Fano. *On the number of bits required to implement an associative memory*. Tech. rep. Computation Structures Group Memo 61. (This reference is not available online.) Massachusetts Institute of Technology, 1971.

[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. "Compressing relations and indexes". In: *Proceedings of the 14th International Conference on Data Engineering*. ICDE '98. IEEE Computer Society, 1998, pp. 370–379. DOI: 10.1109/ICDE.1998.655800.

[13] A. Kane and F. Wm. Tompa. "Skewed Partial Bitvectors for List Intersection". In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '14. Association for Computing Machinery, 2014, pp. 263–272. DOI: 10.1145/2600428.2609609.

[14] D. Lemire and L. Boytsov. "Decoding Billions of Integers Per Second Through Vectorization". In: *Software — Practice & Experience* 45.1 (2015), pp. 1–29. DOI: 10.1002/spe.2203.

[15] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. "Compressing Inverted Indexes with Recursive Graph Bisection: A Reproducibility Study". In: *Advances in Information Retrieval*. Springer International Publishing, 2019, pp. 339–352. DOI: 10.1007/978-3-030-15712-8_22.

[16] A. Moffat. "Word-based text compression". In: *Software — Practice & Experience* 19.2 (1989), pp. 185–198. DOI: 10.1002/spe.4380190207.

[17] A. Moffat and M. Petri. "Index Compression Using Byte-Aligned ANS Coding and Two-Dimensional Contexts". In: *Proceedings of the 11th ACM International Conference on Web Search and Data Mining*. WSDM '18. Association for Computing Machinery, 2018, pp. 405–413. DOI: 10.1145/3159652.3159663.

[18] A. Moffat and L. Stuiver. "Exploiting clustering in inverted file compression". In: *Proceedings of the 6th Data Compression Conference*. DCC '96. IEEE Computer Society, 1996, pp. 82–91. DOI: 10.1109/DCC.1996.488313.

[19] A. Moffat and J. Zobel. "Parameterised Compression for Sparse Bitmaps". In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '92. Association for Computing Machinery, 1992, pp. 274–285. DOI: 10.1145/133160.133210.

[20] G. Navarro and V. Mäkinen. "Compressed Full-text Indexes". In: *ACM Computing Surveys* 39.1 (2007). DOI: 10.1145/1216370.1216372.

[21] G. Ottaviano and R. Venturini. "Partitioned Elias-Fano Indexes". In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '14. Association for Computing Machinery, 2014, pp. 273–282. DOI: 10.1145/2600428.2609615.

[22] G. E. Pibiri. "Space and Time-Efficient Data Structures for Massive Datasets". PhD Thesis. University of Pisa, 2018. URL: http://pages.di.unipi.it/pibiri/papers/phd_thesis.pdf.

[23] G. E. Pibiri and R. Venturini. "Clustered Elias-Fano Indexes". In: *ACM Transactions on Information Systems* 36.1 (2017). DOI: 10.1145/3052773.

[24] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. "Compression of inverted indexes for fast query evaluation". In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '02. Association for Computing Machinery, 2002, pp. 222–229. DOI: 10.1145/564376.564416.

[25] F. Silvestri and R. Venturini. "VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming". In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. CIKM '10. Association for Computing Machinery, 2010, pp. 1219–1228. DOI: 10.1145/1871437.1871592.

[26] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. "SIMD-Based Decoding of Posting Lists". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM '11. Association for Computing Machinery, 2011, pp. 317–326. DOI: 10.1145/2063576.2063627.

[27] L. H. Thiel and H. S. Heaps. "Program design for retrospective searches on large data bases". In: *Information Storage and Retrieval* 8.1 (1972), pp. 1–20. DOI: 10.1016/0020-0271(72)90024-1.

[28] Andrew Trotman. "Compression, SIMD, and Postings Lists". In: *Proceedings of the 19th Australasian Document Computing Symposium*. ADCS '14. Association for Computing Machinery, 2014, pp. 50–57. DOI: 10.1145/2682862.2682870.

[29] S. Vigna. "Quasi-Succinct Indices". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. WSDM '13. Association for Computing Machinery, 2013, pp. 83–92. DOI: 10.1145/2433396.2433409.

[30] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. "An Experimental Study of Bitmap Compression vs. Inverted List Compression". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Association for Computing Machinery, 2017, pp. 993–1008. DOI: 10.1145/3035918.3064007.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishing, San Francisco, 1999. URL: https://people.eng.unimelb.edu.au/ammoffat/mg/.

[32] H. Yan, S. Ding, and T. Suel. "Inverted Index Compression and Query Processing with Optimized Document Ordering". In: *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. Association for Computing Machinery, 2009, pp. 401–410. DOI: 10.1145/1526709.1526764.

[33] J. Zhang, X. Long, and T. Suel. "Performance of Compressed Inverted List Caching in Search Engines". In: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. Association for Computing Machinery, 2008, pp. 387–396. DOI: 10.1145/1367497.1367550.

[34] Z. Zhang, J. Tong, H. Huang, J. Liang, T. Li, R. J. Stones, G. Wang, and X. Liu. "Leveraging Context-Free Grammar for Efficient Inverted Index Compression". In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '16. Association for Computing Machinery, 2016, pp. 275–284. DOI: 10.1145/2911451.2911518.

[35] M. Zukowski, S. Heman, N. Nes, and P. Boncz. "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of the 22nd International Conference on Data Engineering*. ICDE '06. IEEE Computer Society, 2006, pp. 59–70. DOI: 10.1109/ICDE.2006.150.

Books and articles — Encoding

[36] N. Abramson. *Information theory and coding*. McGraw-Hill, 1963.

[37] J. L. Bentley and A. C.-C. Yao. "An almost optimal algorithm for unbounded searching". In: *Information Processing Letters* 5.3 (1976), pp. 82–87. DOI: 10.1016/0020-0190(76)90071-5.

[38] J. Duda. "Asymmetric Numeral Systems". PhD Thesis. Jagiellonian University, 2009. URL: https://ruj.uj.edu.pl/xmlui/bitstream/handle/item/38751/duda_asymmetric_numeral_systems.pdf.

[39] P. Elias. "Predictive coding — Part II". In: *IRE Transactions on Information Theory* 1.1 (1955), pp. 24–33. DOI: 10.1109/TIT.1955.1055116.

[40] P. Elias. "Universal Codeword Sets and Representations of the Integers". In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203. DOI: 10.1109/TIT.1975.1055349.

[41] R. G. Gallager. "Variations on a theme by Huffman". In: *IEEE Transactions on Information Theory* 24.6 (1978), pp. 668–674. DOI: 10.1109/TIT.1978.1055959.

[42] R. G. Gallager and D. C. van Voorhis. "Optimal source codes for geometrically distributed integer alphabets (Corresponde)". In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 228–230. DOI: 10.1109/TIT.1975.1055357.

[43] S. Golomb. "Run-length Encodings (Correspondence)". In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401. DOI: 10.1109/TIT.1966.1053907.

[44] B. Knoll and N. de Freitas. "A Machine Learning Perspective on Predictive Coding with PAQ8". In: *Proceedings of the 22nd Data Compression Conference*. DCC '12. IEEE Computer Society, 2012, pp. 377–386. DOI: 10.1109/DCC.2012.44.

[45] M. Mrak, D. Marpe, and T. Wiegand. "A context modeling algorithm and its application in video compression". In: *Proceedings of the 2003 International Conference on Image Processing*. Vol. 3. ICIP '03. IEEE Computer Society, 2003, pp. III–845–III–848. DOI: 10.1109/ICIP.2003.1247377.

[46] C. E. Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

Books and articles — Other

[47] M. F. Porter. "An algorithm for suffix stripping". In: *Program* 14.3 (1980), pp. 130–137. DOI: 10.1108/eb046814.

Datasets

[48] L. Andrewes, J. Overall, H. à Saravia, R. Clarke, J. Layfield, R. Tighe, F. Burleigh, G. King, R. Thomson, W. Bedwell, E. Lively, J. Richardson, L. Chaderton, F. Dillingham, R. Andrewes, T. Harrison, R. Spaulding, A. Bing, J. Harding, J. Rainolds, T. Holland, R. Kilby, M. Smith, R. Brett, D. Fairclough, W. Thorne, T. Ravis, G. Abbot, R. Eedes, G. Tomson, Sir H. Savile, J. Peryn, R. Ravens, J. Harmar, J. Aglionby, L. Hutten, W. Barlow, J. Spenser, R. Fenton, R. Hutchinson, W. Dakins, M. Rabbet, T. Sanderson, J. Duport, W. Branthwaite, J. Radcliffe, S. Ward, A. Downes, J. Bois, R. Ward, T. Bilson, and R. Bancroft. *Authorized Version (King James') Bible*. John Norton & Robert Barker, 1611. URL: http://www.gutenberg.org/cache/epub/30/pg30.txt.

[49] B. Klimt and Y. Yang. "The Enron Corpus: A New Dataset for Email Classification Research". In: *Proceedings of the 15th European Conference on Machine Learning*. ECML '04. Springer Berlin Heidelberg, 2004, pp. 217–226. DOI: 10.1007/978-3-540-30115-8_22.

[50] The Cognitive Assistant that Learns and Organizes (CALO) project. *Enron Email Dataset*. 2015. URL: http://www.cs.cmu.edu/~enron/.

[51] D. Lemire and L. Boytsov. *Integer Compression 2014 Dataset*. 2014. URL: https://lemire.me/data/integercompression2014.html.

[52] United States National Institute of Standards and Technology (NIST). *TREC disks 4 and 5*. 1997. URL: https://trec.nist.gov/data/cd45/index.html.

Softwares

[53] R. Delbru and J. Kotowski. *SIREn: Efficient semi-structured Information Retrieval for Lucene/Solr/Elasticsearch*. 2014. URL: https://github.com/sirensolutions/siren.

[54] The Apache Software Foundation. *Apache JAMES Mime4j*. 2004. URL: https://james.apache.org/mime4j/.

[55] J. Hedley. *jsoup: Java HTML Parser*. 2004. URL: https://jsoup.org/.

[56] B. Knoll. *cmix*. 2014. URL: https://github.com/byronknoll/cmix.

[57] D. Lemire and M. Taro. *JavaFastPFOR: A simple integer compression library in Java*. 2014. URL: https://github.com/lemire/JavaFastPFOR.

[58] A. Mallia, M. Siedlaczek, J. Mackenzie, T. Suel, and G. Ottaviano. *PISA: Performant Indexes and Search for Academia*. 2014. URL: https://github.com/pisa-engine/pisa.

[59] Project Nayuki. *Reference arithmetic coding*. 2011. URL: https://github.com/nayuki/Reference-arithmetic-coding.

[60] G. Ottaviano, R. Venturini, and N. Tonellotto. *Data Structures for Inverted Indexes*. 2014. URL: https://github.com/ot/ds2i.

[61] G. E. Pibiri. *Clustered Elias-Fano Indexes*. 2017. URL: https://github.com/jermp/clustered_elias_fano_indexes.

[62] M. Porter and R. Boulton. *Snowball*. 2002. URL: http://snowballstem.org/.