Clustered Elias-Fano Indexes

GIULIO ERMANNO PIBIRI and ROSSANO VENTURINI, University of Pisa

State-of-the-art encoders for inverted indexes compress each posting list *individually*. Encoding *clusters* of posting lists offers the possibility of reducing the redundancy of the lists while maintaining a noticeable query processing speed.

In this paper we propose a new index representation based on clustering the collection of posting lists and, for each created cluster, building an ad-hoc *reference list* with respect to which all lists in the cluster are encoded with *Elias-Fano*. We describe a posting lists clustering algorithm tailored for our encoder and two methods for building the reference list for a cluster. Both approaches are heuristic and differ in the way postings are added to the reference list: or according to their frequency in the cluster or according to the number of bits necessary for their representation.

The extensive experimental analysis indicates that significant space reductions are indeed possible, beating the best state-of-the-art encoders.

CCS Concepts: • **Information systems** → **Data compression; Retrieval efficiency;** *Clustering;*

Additional Key Words and Phrases: Elias-Fano Encoding, Inverted Indexes, Performance

This is the accepted version of an article published in ACM Transactions on Information Systems.
Link to the version of record: https://dl.acm.org/doi/10.1145/3052773

1. INTRODUCTION

The incredibly fast growth of hardware and software technologies in the past few decades have radically changed the way data is processed. To cope with the huge amount of information processed on a daily basis, the design of efficient data structures plays a fundamental role. In all practical applications, space-efficiency and fast access to data are key driving-parameters in the design of possible solutions. As a noticeable example, it would be unfeasible to retrieve the set of documents containing a given term in a large textual collection, without the use of the so-called *inverted index*.

The inverted index can be regarded as being a collection of sorted integer sequences (posting lists), each associated to a term in the corpus dictionary [Büttcher et al. 2010; Manning et al. 2008; Zobel and Moffat 2006]. For each term, the corresponding sequence stores the identifiers of the documents containing the term. This is the data structure at the heart of modern web search engines, (semi-)structured data bases and graph search engines in social networks [Curtiss et al. 2013], just to name a few practical scenarios. Despite its simple nature, the inverted index is a popular data structure because it permits the efficient resolution of queries, such as: return all documents in the collection containing a given set of terms.

Because of the size of such textual corpora and stringent query efficiency requirements, representing the posting lists of document identifiers in compressed space while

Author's addresses: Giulio Ermanno Pibiri, Department of Computer Science, University of Pisa, email: giulio.pibiri@di.unipi.it; Rossano Venturini, Department of Computer Science, University of Pisa, email: rossano.venturini@unipi.it.

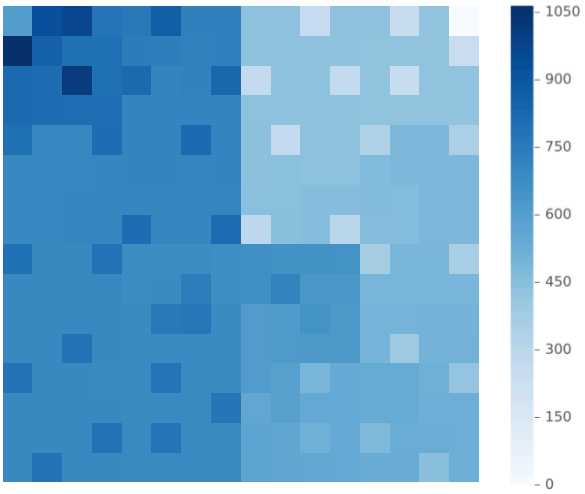


Fig. 1. Top-300 docId frequency counts for $1,066$ posting lists from the dataset Gov2, plotted along a Hilbert curve in order to help highlighting the regions of redundant postings. The color scale on the right indicates the level of frequency of postings: the darker the color the higher the frequency of a posting.

attaining to high query throughput is a fundamental research topic. Achieving both objectives is generally hard, since they are conflicting in nature: a great deal of compression usually sacrifices fast retrieval; on the contrary, high speed algorithms benefit from an augmented index representation [Broder et al. 2003; Büttcher and Clarke 2007]. A vast amount of literature describes different space/time trade-offs [Lemire and Boytsov 2013; Moffat and Stuiver 2000; Ottaviano and Venturini 2014; Salomon 2007; Stepanov et al. 2011; Yan et al. 2009].

Most encoders fall into the *block-based* family, such as OptPForDelta [Yan et al. 2009] and Varint-G8IU [Stepanov et al. 2011], in which sequential decoding is allowed for fast processing speed. The sequence is divided into blocks of consecutive identifiers, in order to avoid decompressing the entire sequence. However, it is also possible to encode directly an entire sequence without dividing it into blocks. This is the approach taken by Binary Interpolative Coding (BIC) [Moffat and Stuiver 2000] and very recent works [Ottaviano and Venturini 2014; Vigna 2013] that have demonstrated how the *Elias-Fano representation* [Elias 1974; Fano 1971] of monotonically increasing sequences combines strong theoretical guarantees with excellent practical performance, placing Elias-Fano indexes as dominant in the space/time trade-off curve.

We notice that every of the aforementioned encoders represents each sequence *individually* and, thus, none of those techniques exploits the redundancy that may exist between two or more posting lists. Indeed the space efficiency of such encoders would be better if *clusters* of posting lists were encoded together instead of separately since this offers the possibility of reducing the redundancy of the lists. Understanding how to reduce such redundancy to save index space and, at the same time, achieve very fast retrieval time is the issue addressed in this paper.

As a matter of fact, inverted indexes naturally present some amount of redundancy. The reason is that the document identifiers (docIds in the following) of "similar" documents, i.e., documents sharing a lot of terms, will be stored in the posting lists of the terms they share. More precisely, consider a document having docId d in which terms t_1 and t_2 co-occur. Then the posting lists of *both* t_1 *and* t_2 will contain d. Figure 1 gives a graphical evidence of this fact. The picture shows how many times the top-300 docIds

appear in an example cluster of $1,066$ posting lists belonging to Gov2, one of the two test collections we used for our experiments. Values are plotted along a Hilbert curve to better highlight the regions of postings having similar frequencies. The intensity of the color represents the degree of repetitiveness of a docId. Hilbert curves are *space-filling fractals* that can be used to draw a one dimensional set into two dimensions. The key characteristic of Hilbert curves is that they mostly maintain locality, i.e., clusters in two dimensions are likely to be close together if represented in one dimension.

Now generalizing to an arbitrary number of terms and documents, consider a set of terms $\{t_1, \ldots, t_k\}$ that occurs in m documents having identifiers $\{d_1, \ldots, d_m\}$. If the k terms *always co-occur* in the considered documents, then the set of integers $\{d_1, \ldots, d_m\}$ would be present in *each* of the posting lists of $\{t_1, \ldots, t_k\}$. In this special case, reducing the redundancy of the lists has an obvious solution: the redundant set $\{d_1, \ldots, d_m\}$ is encoded just once and each posting list stores a reference to it.

Unfortunately, it is very unlikely that the k terms appear in all documents. In general there would be: very few terms co-occurring in all documents and several subsets of terms co-occurring in subsets of documents. Indeed notice that only few squares in Figure 1 are much darker than others, meaning that relatively few docIds appear in all lists of the cluster. Most docIds have, instead, intermediate frequencies and constitute, therefore, the fundamental data source to be represented effectively, i.e., for which a space-efficient solution should be designed.

In this scenario, the problem of reducing the redundancy of the posting lists is much more difficult, due to two different aspects.

— We do not know which and how many terms should be clustered together in order to maximise the number of co-occuring terms.
— It is not obvious how to compactly represent the redundant docIds shared by the clustered posting lists to save index space.

Now suppose we build a *meta-list* R by selecting a subset of docIds belonging to the lists in a cluster $\mathcal{C}$. Then the integers belonging to the intersection of list $S \in \mathcal{C}$ with R can be rewritten as the positions they occupy in R (see also Figure 3). R will be the cluster *reference list*. If u denotes the last element of S, each docId in the intersection is now drawn from a universe of size $|R|$ instead of u, i.e., the number of bits necessary to represent each docId is now $\lceil \lg |R| \rceil$ instead of $\lceil \lg u \rceil$. If R is chosen such that the condition $|R| \ll u$ holds, the universe of the intersection is highly reduced and can be, therefore, encoded with much fewer bits. The representation of each list is finally compressed using a state-of-the-art encoder to save index space.

Most encoders for posting lists take advantage of universe reduction techniques as they reduce the amount of bits necessary to represent each element [Moffat and Stuiver 2000; Ottaviano and Venturini 2014].

Therefore, we are interested in the problem of determining the partition of the set of posting lists into clusters and the choice of each cluster reference list such that the overall encoding cost is minimum. This is a difficult optimisation problem. Indeed we show that the problem of selecting docIds to build the reference list such that the encoding cost of the cluster is minimised is NP-hard, already when considering a special case of the problem.

We present two heuristic approaches for building the reference list for a cluster that we validate with an extensive experimental analysis. The encoder we used to compress the representation of a posting list is Elias-Fano and, in particular, the *partitioned* variant devised by Ottaviano and Venturini [2014]. We perform several experiments on two large datasets, namely Gov2 and ClueWeb09 (Category B), and test different query algorithms such as AND, and top-k retrieval algorithms Ranked AND and WAND [Broder et al. 2003].

We compare the performance of our indexes against several encoding strategies, namely: partitioned Elias-Fano (PEF) [Ottaviano and Venturini 2014], Binary Interpolative Coding (BIC) [Moffat and Stuiver 2000] and SIMD-optimized Variable Byte (Varint-G8IU) [Stepanov et al. 2011], which are representative of best compression ratio/query processing trade-off, compression ratio and highest speed in the literature, respectively. Although partitioned Elias-Fano has proven to offer a better compression ratio/query processing trade-off than the popular PForDelta [Zukowski et al. 2006], we also compare against the optimized PForDelta implementation OptPFD [Yan et al. 2009], for completeness.

Our contributions. We list here our main contributions.

(1) We introduce a novel list representation designed to exploit the redundancy of inverted indexes by encoding a portion of a list with respect to a carefully built reference list. The representation is fully compressed using partitioned Elias-Fano, as to support very fast random access and search operations.
(2) We describe an algorithm to cluster posting lists which is tailored for the introduced list representation. Moreover, we show how the optimisation problem of selecting the proper reference meta-list for each cluster can be solved by two distinct heuristic approaches: the first selects postings by frequency within their cluster and helps controlling index building time; the second selects postings on the basis of their contributions to the overall encoding space cost reduction.
(3) We present an extensive experimental analysis, showing that our index representation is always more compact and only slightly slower than the already highly optimised partitioned Elias-Fano; very close in space or even better but much faster than Interpolative. In particular, our index is smaller by up to 11% on Gov2 and 6.25% on ClueWeb09 with respect to partitioned Elias-Fano. On Gov2 the index is also smaller than Interpolative by 5.63%, while on ClueWeb09 Interpolative is still the smallest but we *halve* the discrepancy between partitioned Elias-Fano e Interpolative, passing from 11.115% to 5.56%. We also show how to obtain interesting trade-offs by varying the size of the reference list: for smaller values our encoding is slightly more compact than partitioned Elias-Fano and Interpolative while being much faster than Interpolative (by 103% on average) and with only a small time overhead with respect to partitioned Elias-Fano (23% on average); for larger values we substantially improve over the space taken by the two competitors while being 50% faster than Interpolative but also 50% slower than partitioned Elias-Fano. As the speed of OptPFD is practically the same as the one of partitioned Elias-Fano, the above query processing considerations also apply to the comparison between our proposal and OptPFD. Our clustered representation is 55% slower than Varint-G8IU. However, our clustered indexes dominates both OptPFD and Varint-G8IU for space usage. Against the former we retain 24% of space less on Gov2 and 14.5% on ClueWeb09. Against the latter, our representation is more than 3.5 times smaller on Gov2 and more than 2.15 times on ClueWeb09.

The paper is organized as follows. Section 2 introduces background and related work. Section 3 describes our novel posting list organization and finally Section 4 presents the extensive experimental analysis validating our implementation.

2. BACKGROUND AND RELATED WORK

Given a collection $\mathcal{D}$ of documents, the *posting list* of a term t is the list of all document identifiers (docIds in short), that contain the term t. The *inverted index* of $\mathcal{D}$ is the collection $\mathcal{L}$ of all posting lists. The set of the terms $\mathcal{T}$ is usually called the collection *dictionary* or *lexicon*. This simple, yet powerful, organization allows fast processing of

queries such as: return all documents in which terms $t_1, \ldots, t_k$ appear. This kind of queries is among the most common on modern search engines and its resolution boils down to posting lists intersection. There is a huge amount of literature describing different query processing algorithms as well as augmented posting list representations. Postings lists can store additional information about each document, such as the set of positions in which the term appears in the document (in *positional* indexes) and the number of occurrences of the term in the document (term *frequency*) [Büttcher et al. 2010; Manning et al. 2008; Zobel and Moffat 2006].

In this paper we consider the docId-*sorted* version of the inverted index, in which all posting lists are monotonically increasing lists, as similarly done in recent works [Ottaviano et al. 2015; Ottaviano and Venturini 2014; Vigna 2013]. Furthermore, we ignore additional information about each document except the term frequencies, which are stored in separate lists. This makes possible fast query processing and efficient index compression.

2.1. Inverted index compression

Given a monotonically increasing integer sequence we can subtract from each element the previous one (the first integer is left as it is), making the sequence be formed by positive integers. This popular strategy is called *delta encoding* and most of the literature assumes this sequence form. Compressing such sequences of d-gaps is a fundamental problem that has been studied for decades. One of the most classical solution is to assign to each d-gap a self-delimiting (or uniquely-decodable) variable-length code. Classical examples include the unary code and more sophisticated ones such as Elias δ and γ or Golomb/Rice codes. The interested reader can find an in-depth discussion in the book by Salomon [2007].

The codes mentioned are also called *bit-aligned* as they do not represent an integer using a multiple of a fixed number of bits, e.g., a byte. This may result in an inefficiency at decoding time since they require many bitwise operations to decode an integer. This is the reason for preferring byte-aligned or word-aligned codes when decoding speed is the main concern. Variable byte (VByte) [Salomon 2007] is the most popular and simple byte-aligned code: the binary representation of a non-negative integer is split into groups of 7 bits which are represented as a sequence of bytes. In particular, the first 7 bits of each byte are reserved for the data while the last one (the 8-th), called the *continuation bit*, is equal to 1 only for the last byte of the sequence. Recent work has exploited the parallelism of SIMD instructions in modern processor to further enhance decoding speed of VByte [Stepanov et al. 2011].

Another approach to improve both compression ratio and decoding speed is to encode a *block* of contiguous integers. This line of work finds its origin in the so-called *frame-of-reference* (For) [Goldstein et al. 1998]. Once the sequence has been partitioned in blocks of fixed or variable length, then each block is encoded separately. The integers in each block are encoded using codewords of fixed length. An example of this approach is *binary packing* [Anh and Moffat 2010; Lemire and Boytsov 2013], where blocks are of fixed length, e.g., 128 integers. Given a block $[\ell, r]$, we can simply represent its integers in $u = \lceil \lg(r - \ell + 1) \rceil$ bits subtracting the lower bound ℓ from their values. Plenty of variants of this simple approach has been proposed [Delbru et al. 2012; Lemire and Boytsov 2013; Silvestri and Venturini 2010]. Among these, Simple-9 and Simple-16 [Anh and Moffat 2005, 2010; Yan et al. 2009] combine very good compression ratio and high decompression speed. The key idea is to try to pack as many integers as possible in a 32-bit word. As an example, Simple-9 uses 4 *header bits* and 28 data bits. The header bits provide information on how many elements are packed in the data segment using equal-size codewords. A header 0000 may correspond to 28 1-bit

integers; 0001 to 14 2-bits integers; 0010 to 9 3-bits integers (1 bit unused), and so on. The four bits distinguish from 9 possible configurations.

For may suffer from a key space inefficiency since the presence of few large values in a block forces the algorithm to encode all its integers with a large universe u. This has been the main motivation for the introduction of PForDelta (PFD) [Zukowski et al. 2006]. The idea is to choose a proper value of u such that u bits are sufficient to encode a large fraction, e.g., 90% of the integers in the block. This strategy is called *patching*. All integers that are unable to fit within u bits, are treated as exceptions and encoded separately with a different encoder. As the problem of choosing the best value for u is posed, Yan *et al.* introduced the OptPFD variant [Yan et al. 2009], which selects for each block the value of u that minimises its space occupancy. Extensive experimental analysis has demonstrated that OptPFD is more space-efficient and only slightly slower than the original PFD [Lemire and Boytsov 2013; Yan et al. 2009].

It is possible, however, to directly compress the monotonically increasing integer sequence without a first delta-encoding step. Binary Interpolative Coding (BIC) [Moffat and Stuiver 2000] encodes the middle element of the sequence and recursively apply this step to the two halves. At each recursive step the algorithm works with reduced ranges that permit to encode the middle element with fewer bits. Though experiments [Silvestri and Venturini 2010; Witten et al. 1999; Yan et al. 2009] have shown that BIC is the best encoding method for highly clustered sequence of integers, the price to pay is a slow decoding algorithm.

Another approach that directly encodes monotonically increasing sequences was recently proposed by Vigna [2013]. He introduced *quasi-succinct indexes* that are based on the Elias-Fano representation of monotone sequences [Elias 1974; Fano 1971], which we discuss in the next paragraph in details. The partitioned variant by Ottaviano and Venturini [2014] greatly improves on the space taken by the Elias-Fano representation of such sequences while introducing only a negligible overhead at query time.

Encoders greatly benefit from docIds-reordering strategies that are focused on finding a suitable reordering of docIds such that a posting lists result more clustered and, consequently, more compressible. An amazingly simple, but very effective strategy for web pages is to assign identifiers to the documents in $\mathcal{D}$ according to the lexicographical order of their URLs. This technique was introduced by Silvestri [2007] and is the one we have used for our test collections.

Lam, Perego, Quan, and Silvestri [2009] has explored the possibility of exploiting the redundancy of inverted indexes by encoding two lists together. Their work proposes two encoding schemes: Mixed Union (MU) and Separated Union (SU). MU stores the union of two posting lists with two additional bits per posting, indicating whether the posting belong to the first posting list (10), to the second (01) or to both (11). SU splits the representation of the union in three segments: the intersection between the two lists and the two residual parts. The terms that should be paired together are chosen by solving a *Maximum Weight Matching* problem on the graph $\mathcal{G} = (V, E)$ built as follows. Each node of V is a term; edge $(t_i, t_j) \in E$ is labeled with a value indicating how many bits would be saved with the pairing of terms t_i and t_j.

Another *multi-term indexing* strategy appeared in the work by Chaudhuri, Church, König, and Sui [2007]. Their solution builds, beside the traditional inverted index, a multi-term inverted index where each entry is composed by terms co-occurring frequently in query logs. As they observed that the distribution of terms is highly skewed in query logs, the aim of their proposal is to boost query processing speed at the price of the extra space needed to deal with another index.

Very recently, an approach based on generating a *context-free grammar* from the inverted index has been proposed [Zhang et al. 2016]. The core idea is to identify common

patterns, i.e., repeated sub-sequences of docIds, and substitute them with symbols belonging to the generated context-free grammar. Although the reorganized posting lists and grammar can be suitable to different encoding schemes, the authors preferred OptPFD [Yan et al. 2009]. The experimental analysis indicates that significant space reductions are possible (by up to 8.8%) compared to state-of-the-art encoding strategies with competitive query processing performance. By exploiting the fact that the identified common patterns can be directly placed in the final result set, decoding speed can also be improved by up to 14%.

2.2. Query processing

Inverted indexes owe their popularity to the efficient resolution of user queries. A query is formulated as a multiset of terms. For each term in the query, the corresponding posting list is accessed and combined with the ones of the other terms. The way posting lists are combined depends on *query operators*: the most common ones are boolean operators such as AND and OR. In many applications, it is preferable to associate to each document matching the user query a *score* indicating the relevance of the document to the query. This score is typically computed as a function of term frequency in the document and few other statistics. A common relevance score is BM25 [Robertson and Jones 1976], which we have used in our experiments. To limit the number of query results, only the k documents scoring better are returned (top-k retrieval). A priority queue of length k is used to collect the result set.

Two query processing strategies have gained popularity: *Term-at-a-Time* (TAAT) and *Document-at-a-Time* (DAAT). The former scans the posting list of each query term separately to build the result set, while the latter scans them concurrently, keeping them aligned by docId. We will focus on the DAAT strategy as it is the most natural for docId-sorted indexes.

The alignment of the lists during DAAT scanning can be achieved by means of the $\mathsf{NextGEQ}_t(x)$ operator, which returns the smallest docId in the list of t that is greater than or equal to x. An efficient implementation of this operation is, therefore, crucial for query processing. To avoid scanning the whole lists, a common solution resorts to *skipping*. The idea is simple: we divide each list in blocks and store some additional information for every block, e.g., the maximum docId or the maximum score. The list scanning process involves a two-level query algorithm: first the additional information is searched to identify a block, then search continues inside the block.

WAND [Broder et al. 2003] is a popular strategy that augments the index by storing for each term its maximum impact to the score. More precisely, WAND processes a query in phases by maintaining a priority queue storing the top-k documents seen so far along with a cursor for each term in the query scanning the corresponding posting list. In each phase, WAND estimates an upper bound to the scores that can be reached by the documents currently pointed by the cursors. In this way, it is able to either determine a new candidate to be inserted in the priority queue or move forward one cursor, potentially skipping several documents in its posting list. We use this algorithm in our experiments for top-k retrieval.

2.3. Clustering algorithms

The classical categorization of clustering algorithms divides them into two broad classes, dual in nature: *partitioning* and *agglomerative*. Partitioning algorithms work by partitioning the whole set of objects $\mathcal{S}$ according to an objective function f, until the desired number of clusters is reached or a stopping criterion is satisfied. On the contrary, agglomerative algorithms start by considering as many initial singleton clusters as the number of objects in $\mathcal{S}$. Then two or more clusters are merged together accord-

ing to f. This process produces a hierarchy of clusters that may be subject to further refinement.

Among the first class, k-*means* [Aggarwal and Reddy 2013] is the most popular and used clustering algorithm. In its simplest formulation [Lloyd 1982]: (1) k objects are drawn at random from $\mathcal{S}$ and considered as centroids; (2) all other objects are assigned to the closest centroid, according to a distance function D; (3) centroids are updated to be the mean of all objects in their clusters. The last two steps are repeated until the k centroids remain almost the same. Common hierarchical clustering algorithms include *Single-Link*, *Average-Link* and *Complete-Link*. We point the reader to the survey by Xu and Wunsch [2005] for an exhaustive overview on the subject. Although such algorithms are supposed to produce superior clusters in terms of cluster quality [Steinbach et al. 2000], their quadratic complexity often prevents from practical use. This has been the main reason of the success of k-means which is elegant, simple and fast [Arthur and Vassilvitskii 2007; Pelleg and Moore 2000; Steinbach et al. 2000]: its complexity is $O(kd|\mathcal{S}|i)$, where d is the dimensionality of the clustered objects and i is the number of needed iterations to converge. Since usually k and d are much less than $|\mathcal{S}|$, its complexity is very appealing in practice.

Many variants of the regular k-means algorithm have been proposed in the literature [Arthur and Vassilvitskii 2007; Pelleg and Moore 2000; Steinbach et al. 2000]. Steinbach, Karypis, and Kumar [2000] introduced a *bisecting* variant of regular k-means that computes two initial clusters and recurse on them until k clusters are formed. They proved bisecting k-means performs even better than the classical one, because it produces relatively uniformly sized clusters. We will use this approach in our own clustering algorithm. Pelleg and Moore [2000] improved on the classical formulation that needs the user to provide the number of clusters. Instead of a single value for k, their algorithm takes as input a possible range of values. In essence, the algorithm starts with k equal to the lower bound of the provided range and keep adding centroids until the upper bound is attained. Adding a new centroid implies splitting an existing one in two: the decision on which centroid to split is based on the *Bayesian Information Criterion*. During the whole process, the centroid set that achieves the highest score is stored and finally output.

As the question regarding which seeds to choose for the initialization step of k-means is posed, Arthur and Vassilvitskii [2007] proposed to select k seeds at random, one at a time, from a *non-uniform* distribution. Specifically, the first centroid c is picked uniformly at random, then the probability that x becomes the next centroid is $D(x,c)^2 / \sum_{y \in \mathcal{S} \setminus \{x\}} D(y,c)^2$. The process is repeated until k seeds are chosen. The key drawback of such approach lies in its inner sequential nature, since the k seeds must be chosen sequentially thus requiring k passes over the data. This issue was tackled by Bahmani, Moseley, Vattani, Kumar, and Vassilvitskii [2012], providing an efficient parallel implementation of the above procedure. As clear from the above formula, the greater the distance of an object to the just chosen centroid, the higher the probability of selecting that object. The intuition, confirmed by their experimental analysis, is that a "good" initial choice of centroids will place them far apart from each other. We will use this initialisation procedure in our own algorithm.

2.4. Elias-Fano sequences

The encoding we are about to describe was independently proposed by Peter Elias [1974] and Robert Mario Fano [1971], hence its name. Given a monotonically increasing sequence $S[0,n)$ of positive integers (i.e., $S[i] \leq S[i+1], \forall i \in [0,n)$) with $u = S[n-1]$, we write each $S[i]$ in binary using $\lceil \lg u \rceil$ bits. Each binary representation is then split into two parts: a *high* part consisting in the first $\lceil \lg n \rceil$ most significant

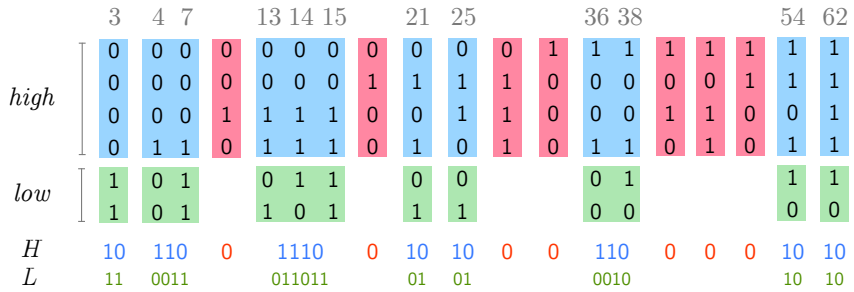


Fig. 2. Elias-Fano encoding example for the sequence $[3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$. Different colors mark the distinction between *high bits* (in blue), *low bits* (in green) and *missing high bits* (in red).

bits that we call *high bits* and a *low* part consisting in the other $\ell = \lfloor \lg \frac{u}{n} \rfloor$ bits that we similarly call *low bits*. Let us call h_i and ℓ_i the values of high and low bits of $S[i]$ respectively. The Elias-Fano representation of S is given by the encoding of the high and low parts. The array $L = [\ell_0, \ldots, \ell_{n-1}]$ is stored in fixed width and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary*[1] using a bit vector of $n + u/2^\ell \leq 2n$ bits as follows. We start from a 0-valued bit vector H and we set the bit in position $h_i + i$, $\forall i \in [0, n)$. The effect is that now the k-th unary integer m of H indicates that m integers of S have high bits equal to k. Finally the Elias-Fano representation of S is given by the concatenation of H and L and takes, overall, $EF(S[0, n)) = n\lceil \lg(u/n) \rceil + 2n$ bits. Figure 2 shows an example of encoding for the sequence $S = [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$.

While we can opt for an arbitrary split into high and low parts, ranging from 0 to $\lceil \lg u \rceil$, it can be shown that the value $\ell = \lfloor \lg(u/n) \rfloor$ minimises the overall space occupancy of the encoding.

In Figure 2 the "missing high bits" embody a graphical representation of the fact that using $\lceil \lg n \rceil$ bits to represent the high part of an integer with Elias-Fano, we have *at most* $2^{\lceil \lg n \rceil}$ distinct high parts. Not all of them could be present. In the exemplar Figure 2, we have $\lceil \lg n \rceil = 4$ and we can form 16 distinct high parts. Notice that, for example, no integer has high part equal to 0010, which are, therefore, "missing".

Despite its simplicity, it is possible to support powerful search operations on $EF(S[0, n))$. Of particular interests for our purposes are:

— $\mathsf{Access}(i)$ which returns $S[i]$ for any $i \in [0, n)$;
— $\mathsf{NextGEQ}(x)$ which returns the integer of S that is greater than or equal to x.

Both operations are supported using an auxiliary data structure that is built on bit vector H, able to efficiently answer the $\mathsf{Select}_{0/1}(i)$ query, which returns the position in H of the i-th $0/1$ bit. This auxiliary *Rank & Select* data structure is *succinct* in the sense that it is negligibly small compared to the encoding of S, requiring only $o(n)$ additional bits [Clark 1996].

Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$ in $O(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. While the low bits ℓ_i are trivial to retrieve as we need to read the range of bits $[i\ell, (i + 1)\ell)$ from L, the high bits deserve a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer of S and a zero for every distinct high part. Therefore, to retrieve the high

[1] If $U(x)$ indicates the unary representation of the integer x, its negated unary representation is the bit-wise NOT of $U(x)$. A simple example: if $x = 5$, then $U(5) = 000001$ and its negated unary is 111110.

bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $O(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$. Notice, therefore, that we only need to support Select_1 queries on bitvector H.
Linking the high and low bits is as simple as a shift followed by a bitwise OR: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) | \ell_i$, where $\ll$ is the left shift operator and $|$ the bitwise OR.

$\mathsf{NextGEQ}(x)$ is resolved as follows. Since the quantity $p = \mathsf{Select}_0(h_x) - h_x$ is the number of elements of S whose higher bits are smaller than h_x, we can start our search scanning S from position p, *skipping* a potentially large range of elements that, otherwise, would have required to be compared with x.

As the $\mathsf{Select}_{0/1}$ primitive is so fundamental for the efficient implementation of Access and NextGEQ, we briefly discuss how it can be supported on bitvector H using little space. See the paper by Vigna [2013] and references therein for an in-depth discussion. While there exists a large body of research on selection that has developed optimal, i.e., constant-time, algorithms requiring tiny space [Clark 1996], such solutions are rather complicated and not practically appealing for their high constant costs. The idea we describe here is, instead, very simple and of practical use. It actually dates back to the original work by [Elias 1974]. We have seen that the crucial step of decoding an integer, and consequently the most time-consuming one, is the reading of its high bits which are coded in negated unary. To speed up this reading, we can write the position of the next bit we would reach after reading kq, $k \geq 0$, unary codes, where q is a fixed quantum. These $\lfloor 2^{\lceil \lg n \rceil}/q \rfloor$ sampled positions are called *forward pointers* and are stored in fixed-width (as they require at most $\lg n + 1$ bits each) before the representation of the high bits. Now $\mathsf{Select}_1(i)$ will first fetch the pointer p in position $\lfloor i/q \rfloor$ and than completes exhaustively reading $i \mod q$ unary codes in H starting from position p. The very same technique can be used to store *skipping pointers* to speed up Select_0 queries too, which enable very fast NextGEQ operations. Clearly, smaller values of q permit less reads but use more space.

To enable fast query processing, we implement a *cursor* that iterates over the encoded sequence by means of the described operations. The cursor implementation is *stateful*, in order to exploit locality of access by optimizing short forward skips, which are very frequent in posting lists processing. Besides NextGEQ, another convenient cursor operation is Next, which advances the cursor by one position.

As a final note, notice that while docId-sequences are immediately compressible with Elias-Fano, frequency lists are not, as they could not be monotonically increasing. However, they can be turned into *strictly* monotone sequences by computing their prefix sums[2]. Since Elias-Fano only requires *weak monotonicity*, we can also subtract i to the value of the i-th frequency. Unfortunately, this trick cannot be applied to docId-sequences [Vigna 2013], since it makes impossible the efficient implementation of NextGEQ as previously described. At query processing, the i-th frequency can be recovered as $\mathsf{Access}(i) - \mathsf{Access}(i-1) + 1$.

2.5. Partitioned Elias-Fano

One of the most relevant characteristic of Elias-Fano is that it only depends on two parameters, i.e., the length and universe of the sequence that poorly describe the sequence itself. As docId-sequences often present groups of close identifiers, Elias-Fano fails to exploit such natural clusters. Partitioning the sequence into chunks to better exploit such regions of close docIds is the key idea of the two-level Elias-Fano representation devised by Ottaviano and Venturini [2014].

[2]Given a sequence S of n integers, the k-th prefix sum of S is defined as $S_k = \sum_{i=1}^{k} S[i]$, i.e., the sum of the first k elements of S (prefix of length k).

The core idea is as follows. They partition a sequence S of length n and universe u into n/b chunks, each of b integers. First level L is made up of the last elements of each chunk, i.e., $L = [S[b-1], S[2b-1], \ldots, S[n-1]]$. This level is encoded with Elias-Fano. The second level is represented by the chunks themselves, which can be encoded using three different strategies, that we discuss next. The main reason for introducing this two-level representation, is that now the elements of the j-th chunk are encoded with a smaller universe, i.e., $L[j] - L[j-1] - 1$. This is a uniform-partition strategy that may be suboptimal, since we cannot expect clusters of docIds be aligned to such boundaries. As the problem of choosing the best partition is posed, an algorithm based on dynamic programming is presented which, in $O(n)$ time, yields a partition whose cost (i.e., the space taken by the partitioned encoded sequence) is at most $(1+\epsilon)$ away from the optimal one, for any $\epsilon \in (0,1)$. To support variable-size partitions, another sequence E is maintained in the first level of the representation, which encodes with Elias-Fano the lengths of the chunks in the second level.

This two-level organisation introduces a level of indirection when resolving queries. However, this indirection only causes a very small time overhead compared to plain Elias-Fano on most of the queries.

As anticipated above, each chunk in the second level is encoded with one among three different strategies. One of them is, not surprisingly, Elias-Fano. The other two additional encodings come into play to overcome the space inefficiencies of Elias-Fano in representing dense chunks. We describe here the optimisation as it plays a key role in our list representation, as we are going to see next. Let us examine the j-th chunk of a list. Call b and u_j its length and universe respectively. Vigna [2013] first observed that as b approaches u_j the space bound $b\lceil \lg(u_j/b) \rceil + 2b$ bits becomes close to $2u_j$ bits. However, we can always represent the chunk with u_j bits by writing the characteristic vector of the set of its elements as a bit vector. The more b is close to u_j, the denser the chunk. The additional used encodings are chosen according on the relation between u_j and b. The first one addresses the extreme case in which the chunk covers the whole universe, i.e., whenever u_j and b are the same: in such case, the first level of the representation suffices to recover each element of the chuck which is, therefore, encoded with 0 bits. The second encoding is used whenever the size of the Elias-Fano representation of the chunk is larger than u_j bits: doing the math it is not difficult to see that this happens whenever $b > u_j/4$. In this case we can encode the set of elements in the chunk by writing its characteristic vector in u_j bits.

For completeness, we also describe the dynamic programming algorithm. The problem of determining the partition of minimum encoding cost can be seen as the problem of determining the path of minimum cost (shortest) in a complete, weighted and directed acyclic graph (DAG) $\mathcal{G}$. This DAG has n vertices, one for each integer of S, and $\Theta(n^2)$ edges where the cost $w(i,j)$ of edge (i,j) represents the number of bits needed to represent $S[i,j]$. Each edge cost $w(i,j)$ is computed in $O(1)$ by just knowing the universe and size of the chunk $S[i,j]$, as explained above.

Since the DAG is complete, a simple shortest path algorithm is inefficient already for medium sized inputs. However, it could be used on a pruned DAG $\mathcal{G}_\epsilon$, which is obtained from $\mathcal{G}$ and has the following crucial properties: (1) the number of edges is $O(n \log_{1+\epsilon} \frac{U}{F})$ for any given $\epsilon \in (0,1)$; (2) its shortest path distance is at most $(1+\epsilon)$ times the one of the original DAG $\mathcal{G}$. U represents the encoding cost of S when no partitioning is performed; F represents the fixed cost that we pay for each partition. Precisely, for each partition we have to write its universe of representation, its size and a pointer to its second-level Elias-Fano representation. F can be, therefore, safely upper bounded with $2 \lg u + \lg n$.

The pruning step retains all the edges (i,j) from $\mathcal{G}$ that satisfy the following two properties for any $i = 0, \ldots, n-1$, $j > i$: (1) there exists an integer $h \geq 0$ such that

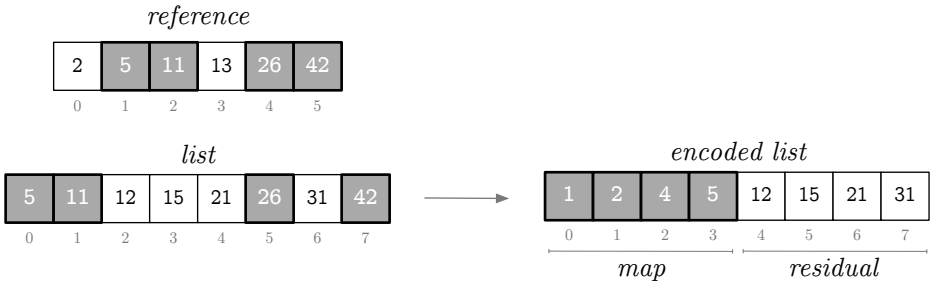


Fig. 3. List representation example: shaded boxes mark the integers falling in the intersection between list and reference; all others form the residual part.

$w(i,j) \leq (1+\epsilon)^h F < w(i,j+1)$; (2) (i,j) is the last edge outgoing from node i, i.e., $j = n$. The edges is $\mathcal{G}_\epsilon$ are the ones that better approximate the value $(1+\epsilon)^h F$ from below because the edge costs are monotone. Such edges are called $(1+\epsilon)$-maximal edges. Since for each $h \geq 0$ it must be $(1+\epsilon)^h F \leq U$, there are at most $\log_{1+\epsilon} \frac{U}{F}$ edges outgoing from each node of $\mathcal{G}_\epsilon$, thus in conclusion $\mathcal{G}_\epsilon$ has $O(n \log_{1+\epsilon} \frac{U}{F})$ edges. Now the dynamic programming recurrence can be solved in $\mathcal{G}_\epsilon$ in $O(n \log_{1+\epsilon} \frac{U}{F})$ admitting a solution whose cost is at most $(1+\epsilon)$ times larger than the optimal one [Ferragina et al. 2011].

We can further reduce this complexity to $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ conserving the same approximation guarantee as follows. Let $\epsilon_1 \in [0,1)$ and $\epsilon_2 \in [0,1)$ two approximation parameters. We first retain from $\mathcal{G}$ all the edges whose cost is no more than $L = \frac{F}{\epsilon_1}$, then we apply the pruning strategy as described above with approximation parameter ϵ_2. The obtained graph has now $O(n \log_{1+\epsilon_2} \frac{L}{F}) = O(n \log_{1+\epsilon_2} \frac{1}{\epsilon_1})$ edges, which is $O(n)$ as soon as ϵ_1 and ϵ_2 are fixed. Ottaviano and Venturini [2014] proved that the shortest path distance is no more than $(1+\epsilon_1)(1+\epsilon_2) \leq (1+\epsilon)$ times the one in $\mathcal{G}$ by setting $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{3}$. These two parameters deeply affect index construction time, as they directly control the number of edges considered by the algorithm.

To generate the pruned DAG from $\mathcal{G}$ we employ $q = O(\log_{1+\epsilon} \frac{1}{\epsilon})$ windows $W_1, \ldots, W_q$, one for each possible exponent $h \geq 0$ such that $(1+\epsilon)^h F \leq L$. Each sliding window covers a different portion of S and it slides over the sequence. We generate the q maximal edges outgoing from node i on-the-fly as soon as the shortest path algorithm visit this node. Initially all windows start and end at position 0. Every time the algorithm visits the next node i, we advance the starting position of each window by one position and the ending position j until $w(i,j)$ exceeds the value $(1+\epsilon)^j F$.

3. REPRESENTATION OF A SET OF POSTING LISTS

In this Section we introduce our novel index representation for a set of posting lists. The key idea is to exploit the implicit redundancy of the docId-lists to reduce space, via a *universe reduction* technique.

Let R be a list not necessarily belonging to $\mathcal{L}$ and $\mathcal{C} \subseteq \mathcal{L}$ a set of lists. Then the integers belonging to the intersection of list $S \in \mathcal{C}$ with R can be rewritten as the positions they occupy in R. If u denotes the universe of S, the intersection is now encoded with a universe of size $m = |R|$ instead of u. If R is chosen such that the condition $m \ll u$ holds, we are greatly reducing the universe of the intersection which can be, therefore, encoded with much fewer bits. R will be called the *reference* list for set $\mathcal{C}$. Figure 3 shows an encoding example.

This strategy introduces a partition of each list into two segments: a *map* part made up of all rewritten integers falling in the intersection with the reference, and a *residual* part consisting in all other integers. Our list representation is the juxtaposition of these two segments.

Specifically, suppose S and R share k integers. Using the space bound of plain Elias-Fano, the space taken by S in this new representation is

$$k\left\lceil \lg \frac{m}{k} \right\rceil + (n-k)\left\lceil \lg \frac{u}{n-k} \right\rceil + 2n + o(n) \text{ bits.} \quad (1)$$

Not surprisingly, the greater the number of integers shared by S and R, the better the space of our encoding. Notice that, though the average gap $\lfloor \lg(u/(n-k)) \rfloor$ will become larger, the overall residual cost will decrease too, since $(n-k)\lceil \lg(u/(n-k)) \rceil$ is monotonically decreasing for reasonably large values of u, as it is likely to be in practice. For example, substituting $m = u/8$ and $k = n/2$ in Equation 1, the resulting space is $n\lceil \lg(u/n) \rceil + 2n + o(n) - n/2$ bits, thus saving 0.5 bits per integer with respect to plain Elias-Fano.

Notice that this is a general scheme that allows for different encoding strategies to be applied on reference, map and residual lists. In our implementation we adopt partitioned Elias-Fano, hereafter indicated with PEF, to encode both map and residual segments of each list, as well as references. Furthermore, observe that nothing prevents from recursively applying the very same encoding strategy to the residual segment of each list. However, implementing such a recursive encoder is much more complicated and may prevent from practicality.

The cursor operations, NextGEQ and Next, can be efficiently implemented by means of the same operations performed by three cursors, each operating on map, residual and reference list separately. To answer a NextGEQ query on our list organisation, a naïve implementation will first perform a NextGEQ on the residual followed by another one on the map to finally return the minimum of the two. Given an integer x, notice that the map may only contain its position p_x within the reference. Therefore a NextGEQ(x) operation on the map actually involves a first $r_x =$ NextGEQ(x) on the reference followed by NextGEQ(p_x) on the map. This second NextGEQ(p_x) is necessary to verify if the searched element p_x is actually contained in the map. If it is, then r_x is the value to return, otherwise a final random-access operation must be performed on the reference to retrieve $R[\text{NextGEQ}(p_x)]$.

The outlined algorithm will always execute three NextGEQs. We argue that this complexity can be alleviated if we cache a state in our cursor implementation. In particular, we save the last accessed values in map and residual: call them y_m and y_r respectively. Depending on the relationship between the given lower bound x, y_m and y_r, it is possible to write the algorithm such that we do not always need three but just two, one or even no NextGEQ. Figure 4 offers a pictorial representation of such relations. In the following description we show in brackets the number of NextGEQs performed. If $x > y_m$ then a NextGEQ on the map is performed (2). At this point we also check if $x > y_r$. If so, a NextGEQ on the residual is performed and we return the minimum of the two values (3). If $x \leq y_m$ instead, we can directly check if $x > y_r$. If so a NextGEQ on residual is performed (1). Eventually, if both previous conditions are not satisfied, i.e., $x \leq y_m$ and $x \leq y_r$, then no NextGEQ are needed and we can just return the minimum between y_m and y_r (0). This completes the description on the NextGEQ algorithm operating on the new list representation. The Next procedure can be implemented similarly.

At this point it is clear that the reference length is a crucial parameter for our encoder. As already mentioned, the larger the number of integers shared by the reference and a list, the better the space usage but, conversely, the greater the number of integers that need to be unmapped during a search operation (two NextGEQs and one Access

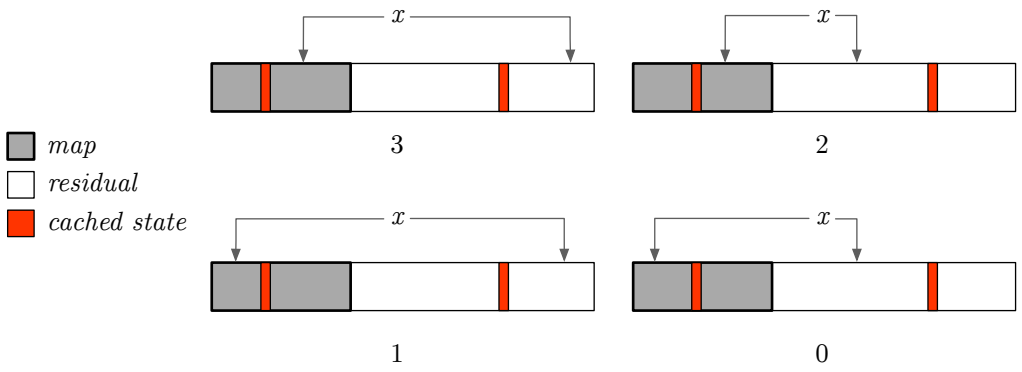


Fig. 4. NextGEQ algorithm operating on the clustered list representation. Depending on the relation between the searched value x and the cached state of our cursor, the algorithm may perform 3, 2, 1 or even 0 NextGEQs. Below each case, we report the number of performed operations.

in the worst case), affecting the retrieval efficiency. We will stress this evident trade-off in Section 4 with an extensive experimental analysis.

Frequency lists need to reflect the partition into map and residual segments too: first we store all frequencies for the map, then all frequencies for the residual. When resolving an $x' = \mathsf{NextGEQ}(x)$ query we need to know the frequency of the term in document x', we access the corresponding frequency list at position $k + j_r$ if x' falls into the residual or at position j_m if it falls into the map, where j_m and j_r denote the position of cursors operating on map and residual respectively.

We argue that two intuitive, yet fundamental, problems naturally arise from the given representation.

— Which lists to group together in set $\mathcal{C}$ under the same reference. Since the greater the intersection of a list with its reference the better our encoding, we would like to collect together lists sharing a lot of integers so that a given reference is able to "cover" a greater portion of the lists in $\mathcal{C}$. This problem reduces to the problem of clustering the index posting lists to maximise the number of integers shared by the lists in a cluster.
— Choosing which integers to put into the reference list reduces to an optimisation problem. Indeed, our goal is that of selecting the reference list for a given set $\mathcal{C}$ such that the space taken by the encoding of $\mathcal{C}$ is minimized.

These two considerations allow us to formally state the problem we are considering. Let $\mathsf{CPEF}(\mathcal{L}_i, R_i)$ be the cost, in bits, of our encoding applied to the lists in set $\mathcal{L}_i$, using reference R_i. We want to solve the following optimisation problem.

PROBLEM 1. *Determine the partition of $\mathcal{L}$ in c clusters, i.e., $\{\mathcal{L}_1, \ldots, \mathcal{L}_c\}$ subject to $\mathcal{L} = \cup_{i=1}^{c} \mathcal{L}_i$, $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset \; \forall i \neq j$ and each $\mathcal{L}_i$ non-empty, and the choice of R_i, $|R_i| > 0$, for each cluster $\mathcal{L}_i$, such that*

$$\sum_{i=1}^{c} \mathsf{CPEF}(\mathcal{L}_i, R_i) \tag{2}$$

is minimum.

This problem is difficult. In fact, suppose to have already computed the partition $\{\mathcal{L}_1, \ldots, \mathcal{L}_c\}$. Now consider a cluster $\mathcal{L}_i$ and the problem of choosing a reference of

size k such that the encoding cost of $\mathcal{L}_i$ is minimum when each list is encoded with plain Elias-Fano. This is a *k-reference selection problem* (RSP_k). Even in this simplified setting, the following theorem holds.

THEOREM 3.1. *RSP_k is NP-hard.*

PROOF. We use the hardness of the *k-Clique Problem* (CP_k) [Garey and Johnson 1979], in which the input is an undirected graph $\mathcal{G} = (V, E)$ and the output is a clique of k nodes (if one exists). In particular, the variant of CP_k for which we are asked to find, for a given $0 < \epsilon < 1$, a clique of size $\epsilon|V|$ is NP-complete for any choice of ϵ [Garey and Johnson 1979]. In our reduction we are interested in the variant where k equals $n/4$, i.e., $\epsilon = 1/4$.

Consider an instance $\mathcal{G} = (V, E)$ of CP_k. V is the set of n vertices that, without loss of generality, we can indicate with $1, \ldots, n$. E is the set of m undirected edges. The instance of RSP_k is obtained from $\mathcal{G}$ by letting vertices in V be the set of postings and by including in $\mathcal{C}$ the m posting lists, one for each edge $(u, v) \in E$, formed by the two postings u and v.

Call R the optimal solution for RSP_k on $\mathcal{C}$. The encoding of each posting list with respect to R has only the following three possible costs

(1) $2\lceil \lg(k/2) \rceil + 4$ bits when both postings are in R;
(2) $2\lceil \lg(n/2) \rceil + 4$ bits when both postings are not in R;
(3) $\lceil \lg k \rceil + \lceil \lg n \rceil + 4$ bits when one posting is in R and the other is not.

Since k is $n/4$, the costs in (2) and (3) coincide and (1) is always the smallest encoding cost. Thus, R must be such that the number of posting lists whose encoding cost is (1) is maximized. This is equivalent of saying that R is such that it maximizes the number of edges that have both vertices in R, by the one-to-one correspondence between edges of $\mathcal{G}$ and posting lists in $\mathcal{C}$. Thus, if $\mathcal{G}$ has a k-clique, R is formed by the vertices of this k-clique. □

As it is not obvious which is the strategy that selects the best clusters and references so that the encoding of the clusters is minimum, it is convenient to consider a three-step modeling, in which the first two steps are solved by a heuristic approach.

(1) Clustering $\mathcal{L}$ into $\{\mathcal{L}_1, \ldots, \mathcal{L}_c\}$, where c is unknown, to group together lists sharing as many docIds as possible.
(2) For each cluster $\mathcal{L}_i$ select the reference R_i, $|R_i| > 0$, such that $\mathsf{CPEF}(\mathcal{L}_i, R_i)$ is minimum. We will illustrate two heuristic algorithms for this problem.
(3) Encode each $\mathcal{L}_k \in \{\mathcal{L}_1, \ldots, \mathcal{L}_c\}$ with PEF.

In what follows, we deeply discuss each of these three steps. Though the three steps are strictly correlated and, therefore, cannot be completely separated from each other, this modeling yields an efficient algorithm for an approximate solution to the original problem.

3.1. Clustering

We use a modified version of k-means [Aggarwal and Reddy 2013] as our posting list clustering algorithm. We adopt a k-means-based approach because it is the only able to scale to the dimensions we are dealing with (see Table I for some basic statistics of the tested datasets). More precisely, although other clustering approaches, e.g., Single-Link, Average-Link and Complete-Link [Xu and Wunsch 2005] may produce superior clusters in terms of cluster quality [Steinbach et al. 2000], their *quadratic* complexity often prevents from practical use. As the posting lists length ranges from tens of thousands to millions, either an approximate distance metric could be used either we need

a lower-complexity algorithm. This has been the main reason of the success of k-means and, indeed, the reason for its choice as our clustering framework.

We start with a high-level overview of the algorithm. In order to avoid to supply the a-priori number of k clusters to the algorithm, we use the *bisecting* approach by Steinbach, Karypis, and Kumar [2000]: we execute an instance of 2-means and recurse on the two children clusters. More specifically, we maintain a deque Q of clusters to be split, initially containing the fictitious cluster formed by the entire dataset. At each step of recursion a cluster is picked from Q and an instance of 2-means is executed on it. Each of the two children is then inserted in Q if it needs refinement. Whenever a cluster does not need any further splitting, it is inserted in a list L of "final" clusters. When Q is empty, the algorithm stops. The number of created clusters is the length of list L. This skeleton describes a *divisive hierarchical* clustering algorithm.

Now we discuss some details. Instead of recurring on the largest of the two children [Steinbach et al. 2000], we adopt an ad-hoc criterion that meets the requirement of the encoding phase that will follow the clustering step. Since the cost of our encoding comprises the cost for the reference lists as well, we intuitively would like to create as few clusters as possible. The problem is that, in general, the bigger the cluster, the longer the reference. However, encoding lists with respect to a very long reference will produce a negligible universe reduction, thus dwarfing the quality of our encoder. Therefore, we decide that a cluster needs further splitting if its current reference is greater than a *user-defined threshold*. This threshold defines the maximum length of the reference that the algorithm builds for each cluster. The current reference size of a cluster is estimated using a fast, heuristic approach that we will describe in Section 3.2. The number of documents U in the collection clearly represents an upper bound on the possible values of this threshold. The experiments regarding how the cluster quality varies for different values of the threshold are presented and discussed in Section 4. In particular, we will determine the best choice of maximum reference size in terms of encoding cost, i.e., the number of bits per posting.

The other meaningful point to describe is the choice of the two seeds. We use the randomised approach described by Arthur and Vassilvitskii [2007]: the first seed c is drawn uniformly at random from the set $\mathcal{S}$ of approximately equal-sized posting lists, then another list x is chosen as centroid with probability $D(x,c)^2 / \sum_{y \in \mathcal{S} \setminus \{x\}} D(y,c)^2$. We follow this approach since we want the two clusters to be well far-apart from each other. Notice that a single pass over the cluster lists suffices for this task.

The last detail we would like to describe is how the distance of a list from a cluster centroid is computed. It seems natural to use a similarity measure that accounts for how many integers of a sequence S are shared with centroid C. Using set notation, a modified Jaccard coefficient $SIM = |S \cap C|/|S| \in [0,1]$ provides us this quantity. We argue that this similarity measure suffers from several problems.

First of all, not all postings should be considered for intersection but only a subset of S. This is a direct consequence of the fact that we are using PEF to encode map and residual segments in our list representation. Recall that the j-th chunk of b docIds can be encoded in three different ways according to the relation between b and its universe u_j. In particular whenever a chunk is encoded with its characteristic bit vector or with 0 bits, it is never advantageous to represent some of its elements with respect to the reference because the chunk will be broken in pieces, each encoded with a larger number of bits. This implies that we have to exclude from S all docIds except the ones belonging to chunks encoded with Elias-Fano.

The other problem is that $|S \cap C|/|S|$ completely ignores the distribution of docIds in the posting lists. In fact, $|S \cap C|$ could be large just because of docIds: (1) that are shared between S and C but *not with all other lists* in the cluster; (2) occurring very

frequently in the whole collection and, therefore, in almost each list. These issues are tackled by maintaining two counts maintained for each posting: a *local count* keeping track of how many times the posting occurs in the lists of the cluster and a *global count* that weights each posting for its own frequency in the whole collection. Notice that the combination of local and global measures is the same solution adopted by the so-called *vector-space model* [Büttcher et al. 2010; Manning et al. 2008]. If we provide analogous definitions of term-frequency (tf) and inverse-document-frequency (idf) in the inverse domain, which is made up of all posting lists of the document collection $\mathcal{D}$, then we can treat each posting list as a vector of reals. More precisely, we define as *document-frequency* $\mathsf{df}(x, S)$ the frequency of docId x in inverted list S. This measure corresponds to the tf count in the documents' domain. This count depends on how the posting lists are built. In our case each df is just 1, but it could be greater depending on the occurrences of a docId in a posting list. A real-life example of this scenario is the Twitter inverted index: a docId is appended multiple times to the posting list of a term if that term occurs multiple times in the indexed tweet [Busch et al. 2012].

The corresponding of the idf count in the inverse domain is the *inverse-term-frequency* (itf), which accounts for how many times x is appearing in whole collection: $\mathsf{itf}(x) = \lg(|\mathcal{T}|/|\{S \in \mathcal{L} : x \in S\}|)$, where $\mathcal{T}$ is the collection lexicon.

Now each posting list S is seen as a vector s of U components, the i-th one being equal to $s_i = \mathsf{itf}(i)$ if integer i belongs to S or 0 otherwise. In what follows, we implicitly refer to a list S by means of its itf vector s. As distance function we use $D = 1 - \cos(s, c)$, where $\cos(s, c) = \sum_{i=1}^{U}(s_i c_i)/\sqrt{\|s\|^2\|c\|^2}$ is the *cosine similarity* between centroid c and list s. Whenever a list is added to its closest cluster, we immediately update the centroid to be the sum of the newly added list and the centroid itself. In this way, the centroid c of each cluster takes into account the number of times posting i occurs within the cluster (local count), which is exactly $c_i/\mathsf{itf}(i)$ times.

3.2. Reference selection

Consider cluster $\mathcal{L}_i$ and the optimisation problem of synthesizing the reference R_i such that $\mathsf{CPEF}(\mathcal{L}_i, R_i)$ is minimum. We have shown in Theorem 3.1 that the simplified RSP_k is NP-hard. If m is the number of integers in $\mathcal{L}_i$, i.e., the sum of the lengths its posting lists, and n the number of its distinct integers, then an optimal solution can be computed in $\Theta(m2^n)$ time and $O(n)$ space. In fact, for each sorted subset R_i of the distinct integers of $\mathcal{L}_i$ we should keep track of $\arg\min_{R_i} \mathsf{CPEF}(\mathcal{L}_i, R_i)$. Since there are $\sum_{k=1}^{n}\binom{n}{k} = 2^n - 1$ possible ways of choosing R_i and encoding takes linear time in the number of postings, the time complexity follows. This is clearly unfeasible.

We describe two heuristic methods for reference selection: *frequency-based* and *space-based*. Before the description, we fix some common concepts.

A *weight* is assigned to each postings and its meaning is different for the two proposed reference selection methods: the frequency-based strategy considers as weights the frequencies of occurrence of the postings, the space-based strategy the number of bits by which the total encoding cost is decreased. This general framework is captured by Figure 5, in which heavier postings are represented in darker shades. Both methods select postings that will end up in the reference from a set that we call the set of *candidate postings*. Let c indicate the cardinality of this set. As already noted in Section 3.1, only postings belonging to blocks encoded with Elias-Fano should be considered as possible candidates, otherwise there is the risk of "breaking" a block encoded with much fewer bits.

Frequency-based selection. An effective method to select the postings for the reference is based on their frequencies within the cluster, i.e., how many times they occur in the posting lists belonging to the cluster. We just need to sort the set of candidate

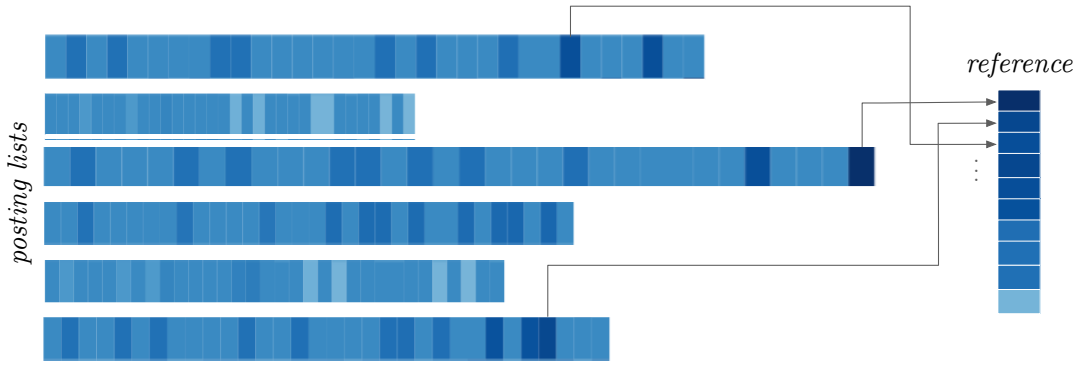


Fig. 5. Reference selection framework. The reference for a cluster of posting lists is made up of the postings having the heaviest weights, represented in the Figure with darker shades. The two proposed reference selection algorithms differ in the notion of a posting weight.

postings according to their frequencies and add to the reference the top-k most frequent postings, where k is the reference wanted dimension. The intuition behind this heuristic is that if the reference is composed by postings occurring in most lists, then it should have a good "coverage" property. On the other hand, including postings occurring only a few times, would be beneficial for few lists too while, at the same time, be detrimental for *all* other lists: the reference will grow, therefore expanding the universe of the representation of *each* map.

As the experimental Section 4 will show next, this heuristic performs well in practice especially for larger values of reference size. Its most important advantage is its speed: a single pass over the lists suffices to build frequency counts that are used as the sort-criterion. Time and space complexities are, respectively, $O(m)$ and $O(c)$.

Space-based selection. The high-level idea of the approach is to estimate the contribution Δ_x in bits of each posting x to the reduction of the overall encoding cost of $\mathcal{L}_i$ and to incrementally build R_i adding postings for decreasing values of Δ. At the beginning of the algorithm all residuals coincide with their corresponding posting lists and the reference list is empty. With a little abuse of notation, in what follows we indicate with R_i the reference *built so far* by the algorithm. Since each Δ_x depends on R_i, we should update each Δ_x whenever a posting is added to R_i. This will ruin any attempt to keep complexity manageable. Instead, we define an *epoch* as t consecutive postings additions to R_i. Within an epoch, each Δ_x is considered as static and whenever a posting is added to R_i it is then removed from the set of candidates. At the end of an epoch, we recompute $\{\Delta_x\}$ for the set of remaining candidates and update the current encoding cost. Updating the current cost corresponds to computing $\mathsf{CPEF}(\mathcal{L}_i, R_i)$. If the updated cost results lower than the previous one, then we save the current reference, thus requiring $O(c)$ space. Then we keep executing epochs until there are no more candidates to consider.

We now discuss how to compute Δ_x. Intuitively, Δ_x will result from the sum of the costs of all lists where x is present. Therefore, we first define Δ_x^S as the contribution of x to the cost of list S. Since we know S and the endpoints E representing its partition, we first binary search x on S to retrieve $rank_x$, i.e., the position of x within S, then we binary search $rank_x$ on E to retrieve the chunk endpoints where x falls in. Now that we know the chunk of x, we can estimate in $O(1)$ the cost contribution of x as the difference between current chunk cost and the old one. Let S_m and S_r be the map and residual segments of list S respectively. Evaluating Δ_x is done as follows.

(1) $\Delta_x = -\Delta_x^{R_i}$. Call m_x the position of x in R_i.
(2) For every sequence S in which x occurs:
$\Delta_x = \Delta_x - \Delta_{m_x}^{S_m} + \Delta_x^{S_r}$.

To speed up step (2), we maintain the correspondence between candidates and their block ids in the lists where they occur. The complexity of computing Δ_x during the j-th epoch is $O(\lg(j-1)t + m/n(1+\lg(j-1)t))$, since m/n represents the average frequency of each posting and during epoch j, R_i contains $(j-1)t$ postings.

Approximating $\lg n!$ using the Stirling approximation $\lg n! \sim n \lg n - O(n) + \lg n/2$, the overall complexity of the algorithm is $O(\frac{c^2}{t}(\frac{m}{n}+1)\lg c + m\frac{c}{t})$ and it uses $O(m)$ space. As it is clear, by varying t we can obtain different time/accuracy trade-offs, as we are going to illustrate in Section 4.

3.3. Encoding

Each cluster $\mathcal{L}_i$ gets encoded with respect to its reference list R_i according to the representation we have detailed at the beginning of Section 3.

In particular, only postings belonging to Elias-Fano encoded blocks are considered for intersection with the reference, all others form the residual part. Recall from Section 2.5 that a block in PEF is encoded with one among three different representations, according to the relation between its universe and size: (1) Elias-Fano; (2) the block's characteristic bit vector; (3) not encoded at all, thus taking 0 bits. Since we can calculate the Elias-Fano partitions of a sequence in linear time using the dynamic programming algorithm by Ottaviano and Venturini [2014] and determine in $O(1)$ which is the type of encoder used for each block, map and residual segments are computed in time proportional to the length of the sequence.

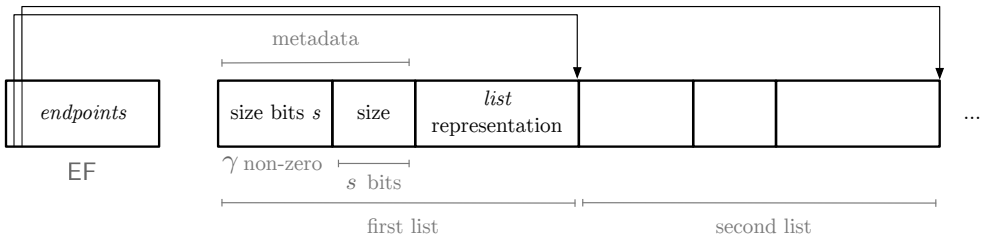
Considering all clusters, the overall complexity of the encoding step is, therefore, linear in the number of postings in the inverted index.

3.4. Index Structure

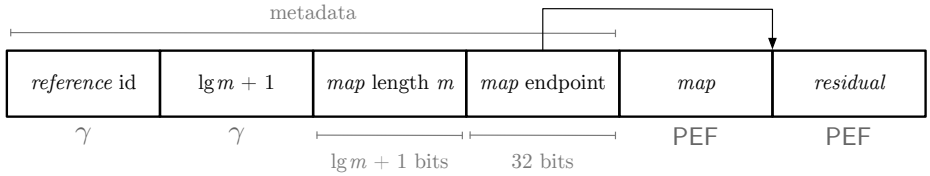
We now describe our index organisation, starting with a high-level picture: our index is made up of three large bit vectors, that we call *document*, *frequency* and *reference* bit streams. They represent the index posting, frequency and reference lists respectively.

Each bit stream results from the concatenation of the bit vectors that individually represent posting, frequency and reference lists. The three bit streams are *aligned*: the i-th frequency list is associated to the i-th document list. Document lists are stored according to cluster-identifier order, i.e., the first lists in the *documents* stream are the ones belonging to the first cluster, then the ones belonging to the second cluster follow and so on. Except for frequency lists, each list representation in the stream is enriched with a metadata information storing the size of the list. First we write in γ code the number of bits necessary to represent the size, then the size value is written uncompressed using this number of bits. Notice that since lists are non-empty we can subtract 1 to the γ encoding of the size (γ non-zero). Since we use PEF to encode both map and residual segments of our posting list organisation, we store list sizes in order to distinguish the different metadata sections of the partitioned Elias-Fano representation that we discuss later. Finally, in order to be able to access each sequence, we store the positions of the bit stream at which sequences end in an Elias-Fano encoded list of endpoints. In this way, as random access operations are supported efficiently with Elias-Fano, we can access each sequence in $O(1)$ within compressed space. Figure 6a shows how each bit stream is organised. Now we discuss how a single list is represented.

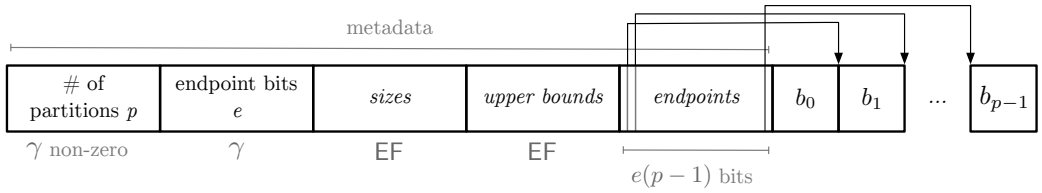
Our posting list organisation contains a metadata header section before the actual representation of its map and residual segments. This metadata section is structured



(a) Bit stream and endpoints



(b) Clustered Elias-Fano sequence



(c) Partitioned Elias-Fano sequence

Fig. 6. Bit stream; clustered and partitioned Elias-Fano sequence organisations. Below each block we report how it is encoded: γ, fixed-width, Elias-Fano (EF), partitioned Elias-Fano (PEF). In picture (b), m represents the length of the map segment; in picture (c), p indicates the number of partitions whereas e the number of bits needed to encode a pointer to a block.

as follows. First of all, we need to store the identifier of the reference list with respect to which the list is encoded. Such identifier ranges from 0 to the number of possible clusters minus 1 and is represented in γ. Then we need to record the length of the map segment, say m. We first store the quantity $\lg m + 1$ in γ, then m using $\lg m + 1$ bits. We finally need to know where the first stored segment (the map) ends to be able to distinguish between the two segments. The last metadata information is, therefore, the number of bits of the map segment written uncompressed in 32 bits. Map and residual segments are stored one after the other, both encoded as a partitioned Elias-Fano sequence that we describe next. Figure 6b shows our *clustered sequence* organisation. In this case, the size of the whole list in the stream is necessary to derive the size of the two segments, which are needed to correctly search their PEF representation.

As to be able to encode the frequency lists using Elias-Fano, each of them is transformed into a monotonically increasing integer sequence by computing its prefix sums. Since we only need to randomly access such lists, we can subtract i to the i-th frequency value and save space. Notice that we do not need to store the size of each frequency list in the bit stream, given that frequency and document streams are aligned: the size of the i-th frequency list will be the size of the i-th list of the stream.

We conclude this Section by explaining the partitioned Elias-Fano list representation. Refer to Figure 6c. As usual a metadata section precedes the representation of the blocks $\{b_0, \ldots, b_{p-1}\}$. We first write the number of partitions using γ non-zero. Then

we write the number of bits for each block endpoint: if m is the length of the bitvector resulting from the concatenation of all blocks, then we need $e = \lg m+1$ bits. Two Elias-Fano encoded blocks follow, representing blocks' sizes and upper bounds respectively. The Elias-Fano representation of a sequence is self-delimiting because we can compute exactly the number of bits of the representation if we only know the length and universe of the sequence [Vigna 2013]. In this case, both *sizes* and *upper bounds* blocks have length equal to the number of partitions; the universe of *sizes* is the length of the sequence whereas the universe of *upper bounds* is just the number of documents in the collection. The endpoints are just stored in fixed-width fashion. The concatenation of all encoded blocks terminates the list representation.

4. EXPERIMENTS

We performed our experiments on the following two datasets.

— ClueWeb09 is the ClueWeb 2009 TREC Category B test collection, consisting of 50 million English web pages crawled between January and February 2009.
— Gov2 is the TREC 2004 Terabyte Track test collection, consisting of 25 million `.gov` sites crawled in early 2004; the documents are truncated to 256 KB.

Table I. Basic statistics for the test collections.

	Documents	Terms	Postings
ClueWeb09	$50,131,015$	$92,094,694$	$15,857,983,641$
Gov2	$24,622,347$	$35,636,425$	$5,742,630,292$

Standard text preprocessing was performed on the collections. For each document, the body text was extracted using Apache Tika[3], and the words lowercased and stemmed using the Porter2 stemmer. The docIds were assigned according to the lexicographic order of their URLs [Silvestri 2007]. Table I reports the basic statistics for the two collections.

Experimental setup. Since we use partitioned Elias-Fano as building-block of our own encoder, we use the approximation parameters $\epsilon_1 = 0.03$ and $\epsilon_2 = 0.3$ as chosen in the work by Ottaviano and Venturini [2014], to tradeoff between index space and construction time as explained in Section 2.5.

To test the speed of the indexes, we use a random sampling of 1000 queries, respectively from TREC 2005 and 2006 Efficiency Track topics, selecting only queries whose terms are all in the collection dictionary. In order to smooth the effect of fluctuations during measurements, we repeat each experiment three times and consider the mean. All query algorithms were run on a single core and query times are reported in milliseconds.

All posting lists representations expose the same interface, i.e., the Access, NextGEQ, and Next operations described in Section 3. All the algorithms were implemented in standard `C++11` and compiled with `gcc` 5.3.0 with the highest optimization settings. We have preferred template specialization over inheritance to avoid virtual method call overhead, which can be disruptive for very fine-grained operations, such as the ones we consider in the following. Except for the instructions to count the number of bits set in a word and to find the position of the least significant bit, no special processor feature was used. We did not add any SIMD (Single Instruction Multiple Data) instruction to our code.

[3]http://tika.apache.org

All tests have been performed on a machine with 16 Intel Xeon E5-2630 v3 cores (32 threads) clocked at 2.4 Ghz, with 64 GB RAM, running Linux 3.13.0, 64 bits. Hardware caches have the following sizes: 32 KB (L1), 256 KB (L2) and 20 MB (L3). Levels L1 and L2 are private of each core, while L3 is shared among all the 8 cores on one socket. In addition, we have repeated all the experiments on a second machine equipped with 4 Intel i7-4790K cores (8 threads) clocked at 4 Ghz, with 32 GB RAM, running Linux 4.2.0, 64 bits to confirm the results. This second machine has the same cache sizes of the other, except for the last shared level (L3) which is 8 MB.

The data structures were saved to disk after construction, and memory-mapped to perform the queries. Before timing the queries we ensure that the index is fully loaded in memory, performing some untimed queries.

We will release the source code upon publication of the paper.

4.1. Clustering

For the clustering step, instead of considering only the posting lists containing more than a given number of postings [Dhulipala et al. 2016], we sort the lists by length in descending order and we discard the k smallest lists such that the sum of their lengths is approximately 10% of the postings of the original collections. Table II shows the modified statistics of the clustered test collections. The posting lists excluded from clustering are encoded with PEF.

Table II. Basic statistics for the clustered test collections.

	Terms	Postings	Avg ; Min ; Max posting list size
ClueWeb09	$36,985$	$14,390,169,181$	$389,081$; $16,618$; $45,857,055$
Gov2	$17,401$	$5,161,774,821$	$296,637$; $14,182$; $20,839,863$

This simple pruning strategy allows us to significantly reduce the number of processed terms from millions to tens of thousands while concentrating our effort on most of the postings, because the distribution of terms occurrences is highly skewed: relatively few lists are very long while the majority being very short.

Table III shows the number of created clusters, clustering time in minutes and the number of bits per posting by varying the reference size threshold. We recall from Subsection 3.1 that this threshold is the free parameter of our clustering algorithm and represents the maximum length of the reference that is built for each cluster. This threshold is expressed as the ratio between the universe collection U (number of documents) and a constant ranging from 1 to 32. As we can see, the smaller the reference a cluster can synthesize, the greater the number of iterations performed by the algorithm and, consequently, the number of clusters and clustering time. In particular, clustering time for ClueWeb09 is approximately four times the one of Gov2 since ClueWeb09 has roughly two times the number of posting lists and universe of Gov2. In the next Section we discuss how the number of bits per posting has been derived.

4.2. Space/time trade-offs by varying the reference size

In this Section we present a detailed analysis on *how and why* varying the reference size of the encoder can yield interesting space/time trade-offs. During the analysis we also experimentally determine the algorithm to prefer between frequency-based and space-based for building our clustered indexes and the values of reference size that give the best trade-offs.

Table III. Number of clusters, clustering time in minutes and number of bits per posting by varying the reference size threshold.

	Reference size threshold	U	$U/2$	$U/4$	$U/8$	$U/16$	$U/32$
Gov2	Number of clusters	2	5	25	70	150	264
	Minutes	5	10	23	32	34	37
	Bits per posting	2.70	2.67	2.67	2.65	2.71	2.79
ClueWeb09	Number of clusters	2	31	88	174	302	461
	Minutes	24	105	127	138	153	170
	Bits per posting	4.60	4.62	4.54	4.52	4.75	4.87

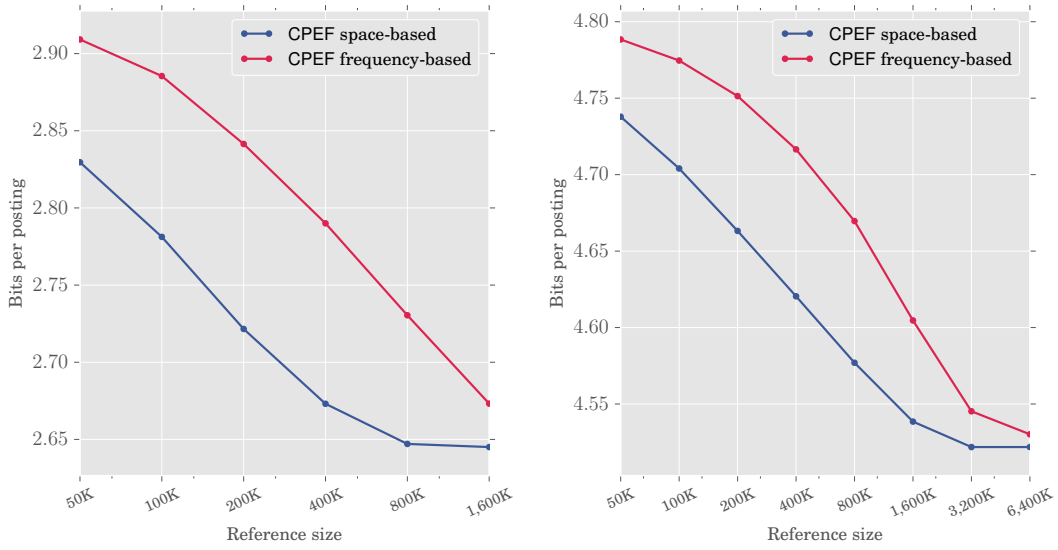
As already pointed out in Section 3, the reference selection step for each cluster deeply affects the quality of the representation in terms of both space usage and speed, as we are going to motivate next. Dealing with small references implies that the fraction of remapped postings is small too on average and this is less beneficial for index space. Conversely, as references grow in dimension, space is gradually reduced but accessing the representation of the references becomes the major bottleneck at query time. The introduced trade-off is evident: smaller references yield faster but bigger indexes while longer ones slower but smaller indexes.

To test this impact, we build several indexes for the two test collections, varying the reference size from $50{,}000$ to $1{,}600{,}000$ for Gov2 and to $6{,}400{,}000$ for ClueWeb09, doubling its size each time. These sizes represent the *maximum* reference sizes that our encoder is permitted to build.

Now, before concentrating the analysis on the mentioned space/time trade-off, we choose the clustering that yields the smallest encoding cost in terms of bits per posting. To help our decision we consider Table III. The reported number of bits per posting has been obtained by encoding each cluster with the largest reference size, i.e., $1{,}600{,}000$ for Gov2 and $6{,}400{,}000$ for ClueWeb09. For both datasets, a value of threshold equal to $U/8$ yields the most compact indexes. Therefore, in what follows, all experiments have been done using such value for the clustering algorithm.

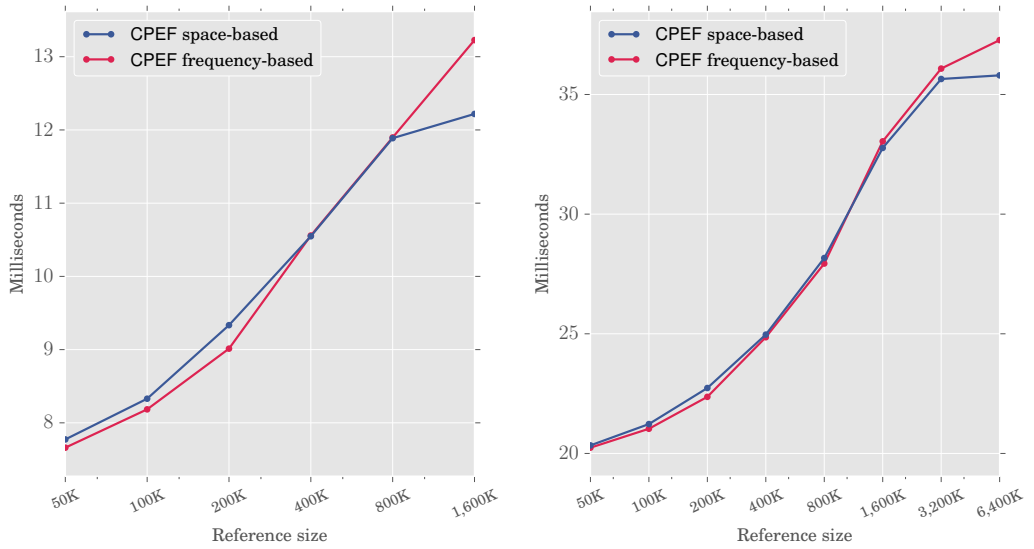
Figure 7 illustrates the number of bits per posting by varying the reference size, of both frequency-based and space-based algorithms. These plots have to be read together with the ones in Figure 8, which show mean query times for AND queries, performed on ClueWeb09 collection using the TREC 2006 sampling. Since plots for all query algorithms exhibit this kind of shape, we have chosen to show the ones for AND queries as meaningful representatives. The two pictures together confirm the trade-off introduced above: as references grow in size, the number of bits per posting decreases, but the mean query time increases as well.

By looking at Figure 7, we notice that the space-based algorithm described in Section 3.2 gives constantly better results with respect to the frequency-based one, especially for smaller reference sizes. In particular, space-based is up to 4.21% smaller for references of $200{,}000$ postings on Gov2 and up to 2.04% smaller for reference of $400{,}000$ postings on ClueWeb09. For both datasets, the advantage of space-based over frequency-based gradually reduces as references grow in size, reaching 1.05% and 0.19% for Gov2 and ClueWeb09 respectively. The reason is that the smaller the reference, the more accurately postings have to be selected. Moreover Figure 8 suggests that, not surprisingly, there is no difference in speed between clustered indexes built using the frequency-based and the space-based strategy. These two reasons together place the space-based algorithm in net advantage over the frequency-based. In order to conclude the comparison between the two, finally consider Figure 9 that reports index building



(a) Gov2 (b) ClueWeb09

Fig. 7. Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

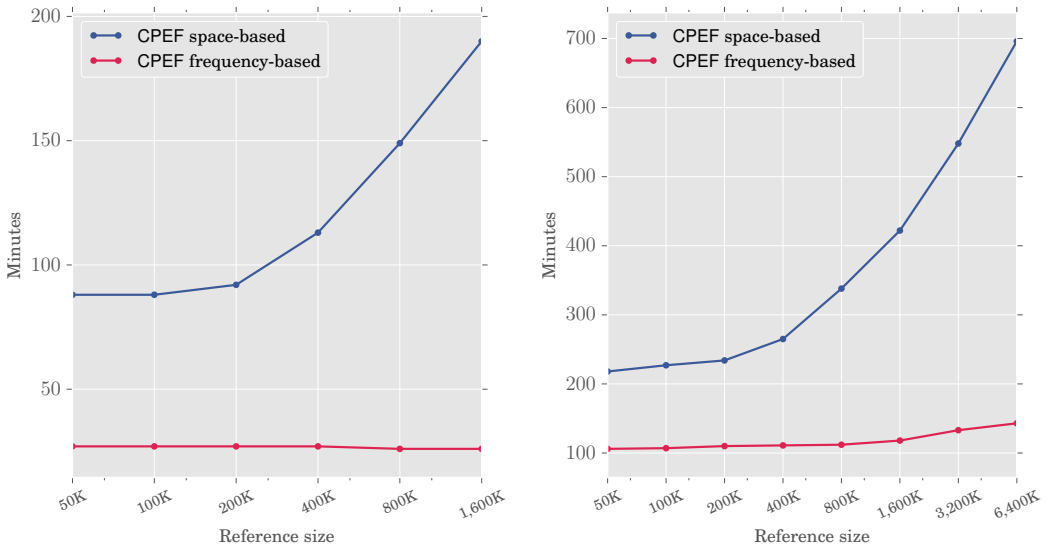


(a) Gov2 (b) ClueWeb09

Fig. 8. Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

times in minutes for Gov2 and ClueWeb09 by varying the reference size. The picture clearly shows the effectiveness of the heuristic of selecting by posting frequency for index building time. As we can see, the frequency-based algorithm is practically *insensitive* to the reference size: it is a direct consequence of the linear complexity in the number of postings, as shown in Section 3.2. On the contrary, the space-based algorithm actually starts to pay a noticeable CPU cost for mid-sized references (e.g., for 800,000). This is due to the higher number of iterations performed by the algorithm to build the reference of specified size.

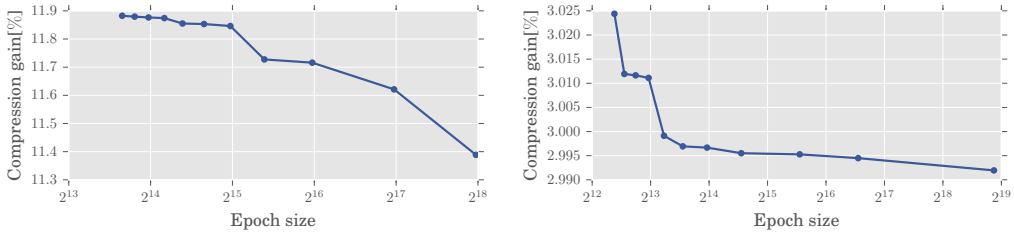
We also mentioned in Subsection 3.2 that by varying the size of an epoch, i.e., how many postings could be added to the reference under construction before evaluating the objective function, we obtain different time/accuracy trade-offs. Figures 10 and 11 illustrate the trade-off for two exemplar clusters of Gov2 and ClueWeb09 during the selection of a reference of size 200,000. The plots clearly show that shrinking the size of an epoch only results in a negligible contribution to the overall accuracy, while increasing by a significant amount the CPU cost. For this reason, it is convenient to choose a large value of epoch size, e.g., defined to be the number of candidate postings c divided by a constant. In our implementation we use $c/10$ for both datasets. As the number of candidate postings for ClueWeb09 is greater then the one for Gov2 on average, we choose different divisive constants for the plots of the two datasets as to have approximately the same epoch size.



(a) Gov2 (b) ClueWeb09

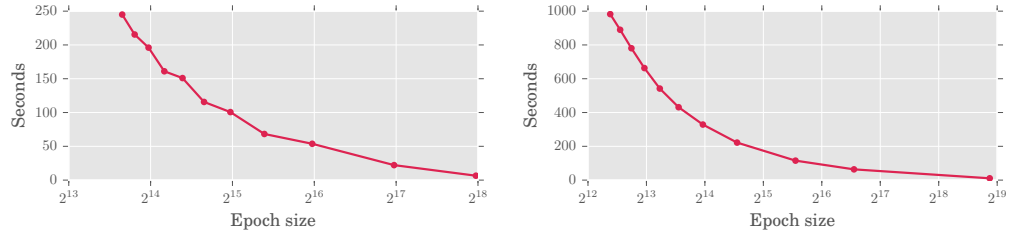
Fig. 9. Index building times in minutes by varying the reference size on Gov2 and ClueWeb09.

As a conclusion, the only advantage of the frequency-based approach over the other lies in its speed during the index construction phase. Unless index building time is the main concern, we should prefer the space-based algorithm which yields more compact indexes with no slowdown in query processing speed. Therefore from now on, we focus our attention on the space-based approach, that we indicate with CPEF in the following.



(a) Gov2 (b) ClueWeb09

Fig. 10. Compression gain in percentage against PEF for the space-based algorithm by varying epoch size, during the selection of a reference of size 200,000 from two exemplar clusters.



(a) Gov2 (b) ClueWeb09

Fig. 11. Time in seconds for the space-based algorithm by varying epoch size, during the selection of a reference of size 200,000 from two exemplar clusters.

Table IV. Space and time percentages of CPEF with respect to the value at reference size 50,000, on Gov2 and ClueWeb09.

		100K	200K	400K	800K	1,600K	3,200K	6,400K
Gov2	space	−1.71%	−3.82%	−5.53%	−6.45%	−6.52%	—	—
	time	+7.15%	+20.07%	+35.69%	+52.92%	+57.18%	—	—
ClueWeb09	space	−0.71%	−1.58%	−2.48%	−3.40%	−4.21%	−4.56%	−4.56%
	time	+4.38%	+11.79%	+22.76%	+38.50%	+61.12%	+75.28%	+76.03%

We now determine three values of reference size as representatives of the different space/time trade-offs we can obtain, in order to concentrate our analysis on these selected points. To help our decision, consider Table IV. The table reports the percentages of space and time with respect to the values in correspondence of reference size 50,000.

For both datasets, a value of reference size equal to 100,000 looses a negligible factor in query processing speed but space reductions are very poor too. Therefore, we choose this value of reference size as representative of the faster query time with respect to the other trade-off points we are going to choose. On the other hand, larger values of reference size optimise space sacrificing query processing speed. For Gov2 values of 800,000 and 1,600,000 offer practically the same space reduction but the former achieves better speed. The same holds for ClueWeb09 as well but for values 3,200,000 and 6,400,000. Values that fall in between these two extreme points tradeoff between space and time. We now select our trade-off points such that two of them optimise either space or time, the third one tries to grab the best from both. From the above discussion, we choose the following points for ClueWeb09: MIN = 100,000; MID = 400,000; MAX = 3,200,000. For Gov2 we choose instead: MIN = 100,000; MID = 200,000;

MAX $= 800{,}000$. For the rest of the experimental analysis, we concentrate on these selected trade-off points.

Table V. Cache misses at the three cache levels; average posting lists intersection with reference; references space and mean query time for AND queries on ClueWeb09 using the query set TREC 06.

	L1 32 KB	L2 256 KB	L3 20 MB	Reference intersection	Reference space	Mean query time
PEF	173.6 M	35.9 M	24.9 M	—	—	17.7
CPEF@ MIN	216.2 M	46.7 M	30.9 M	9.78%	19.3/19.7 MB	21.2
CPEF@ MID	266.5 M	58.2 M	39.3 M	18.93%	62.8/63.8 MB	25.0
CPEF@ MAX	443.6 M	88.5 M	54.4 M	44.37%	259.6/262.7 MB	35.6

The concluding part of this Section shows why the reference size sensibly affects the query processing speed of our proposal. As argued at the beginning of the Section, clustering the index inevitably brings a penalty at query time. The penalty comes precisely from the *cache misses* induced by accessing the references. The reason is that as references grow in dimension the number of integers that have been encoded with respect to them grows as well, therefore needing more reference accesses to be decoded. In particular, cache misses are spent in the first level of the partitioned Elias-Fano representation and are due to *chunk-switching* operations, i.e., whenever the distance from two consecutive searched values (jump entity) exceeds the current chunk size.

To quantify the impact of reference cache misses, we list in Table V the number of misses in million (M) for AND queries on ClueWeb09, using the query set TREC 06. Fourth column of the table reports the average intersection of posting lists with the reference while fifth column reports the sum of the space of the references accessed during the queries over the total space of the references. Cache misses have been collected with the perf Linux tool, version 3.13.11-ckt35. Levels L1 and L2 are private while L3 is shared among all the 8 cores on one socket.

As evident, we increase the number of misses at all levels going from MIN to MAX, confirming the shape of Figure 8. Most importantly, the average percentages of reference intersection reported in the fourth column are proportional to the cache misses at L3 as claimed before. From MIN to MID we have an increase of cache misses at L3 of 27.18% which corresponds to an increase of reference intersection of about 1.93 times. Instead, from MIN to MAX the increase of reference intersection is 4.54 times, therefore we should expect a corresponding increasing of cache misses of about 63.94% which is 53.9 million of cache misses, practically the same as the value reported at MAX.

We have repeated the test for the second machine, having the same data cache dimensions except for last level which is 8 MB, again shared among all cores. Results were practically the same. The reason is that all query algorithms have *no temporal locality*: even though some reference cached blocks could be reused for other queries, they will be inevitably deallocated and refetched when needed. Therefore, having a bigger cache will not bring a performance improvement at query time, unless we explicitly decide to keep in cache as many reference blocks as possible. Notice that, however, for the illustrated example, this would only be possible for the MIN point for which the cache is able to contain the whole reference working set as reported in the fifth column of Table V.

The last column of the table reports the mean query time. Again, we confirm that query processing speed sensibly depends on the number of cache misses. By the values reported in the third column, we should expect to have a slowdown, with respect to

PEF, of roughly 24%, 57.8% and 118.5% for MIN, MID and MAX respectively. Indeed mean query times report slowdown factors of 19.7%, 41.2% and 101.2%.

As discussed in Section 3, our posting list organisation may require up to three NextGEQ operations, each operating on the map, residual and reference sequences. We argue that, while this case arises in practice, it is very pessimistic and not the most frequent one. In order to avoid confusion, let us call *partial* a NextGEQ resolved on map, residual or reference and *full* the NextGEQ on the whole clustered list. Beside the worst case, a full NextGEQ may need 2, only 1 or even 0 partial NextGEQ. In Table VI we report the mean number of partial NextGEQ operations, where p_k represents the probability of performing k partial NextGEQs, $k = 1, 2, 3$ (the probability of performing no NextGEQ, i.e., $k = 0$, is minimal and does not contribute to the calculation of the mean value).

Table VI. Mean number of partial NextGEQ operations and corresponding empirical frequencies p_k for AND queries on ClueWeb09 using the query set TREC 06.

	p_1	p_2	p_3	Mean number of partial NextGEQs	Mean query time
MIN	0.85	0.04	0.11	1.26	21.2
MID	0.61	0.09	0.29	1.66	25.0
MAX	0.32	0.23	0.45	2.13	35.6

The mean number of partial NextGEQs clearly increases for growing values of reference size. More precisely, we notice an increment proportional to the values presented in Table V. In fact, the mean number of *extra* partial NextGEQs performed in the three points is 25%, 68% and 113% respectively, while the number of cache misses in excess are 24%, 57.8% and 118.5%. Finally notice that these values are also confirmed by mean query time values that report slowdown factors of 19.7%, 41.2% and 101.2%.

Moreover, it is interesting to notice how each p_i changes in relation to the reference size. In particular, p_1 is decreasing because as the reference grows, map segments grow as well thus reducing the entity of residuals and, consequently, the number of partial NextGEQs performed on them.

The crucial parameter affecting the query processing speed is, therefore, the *mean number of accessed reference lists per query*. Intuitively, if all the terms of a query belong to the same cluster, the number of accessed reference lists is just one. To give a practical evidence of this fact, we conduct the following experiment. For each query we evaluate the ratio between the number of terms and the number of distinct clusters. Doing the average of these quantities among all queries gives us the *mean number of terms per cluster* within a query, indicated with r in the following. If ρ is the mean number of terms per query in a query set, then $r \in [1, \rho]$: when $r = 1$, it means that, for each query, all terms belong to distinct clusters; on the other hand, when $r = \rho$ then all terms belong to the same cluster. We test the speed of AND queries on three sampling of 1000 queries from TREC 06, having respectively r equal to 1.07, 1.48 and 2.01. Table VII illustrates the result. As we can see, when r increases the number of reference lists accessed per query decreases and so does the mean query time. For the other query sets, i.e., TREC 06 for ClueWeb09 and TREC 05 for both Gov2 and ClueWeb09, it is not possible to obtain sufficiently diversified values for r because it is concentrated in the interval $[1.06, 1.09]$.

Table VII. Timings for AND queries by varying terms per cluster ratio on ClueWeb09 using the query set TREC 06.

r	1.07	1.48	2.01
MIN	21.2	19.5	14.1
MID	25.0	23.3	17.4
MAX	35.6	33.5	25.2

4.3. Overall comparison

In this Section we compare our proposal in the selected trade-off points against several index compression strategies: partitioned Elias-Fano (PEF) [Ottaviano and Venturini 2014], Binary Interpolative Coding (BIC) [Moffat and Stuiver 2000], optimized PForDelta (OptPFD) [Yan et al. 2009] and SIMD-based Variable Byte (Varint-G8IU) [Stepanov et al. 2011]. The competitors PEF, BIC and Varint-G8IU were chosen as they are representative of best compression ratio/processing speed trade-off, best compression ratio and highest speed in the literature, respectively. Although PEF has proven to offer a better space/time trade-off, we also report the space usage and query time of OptPFD for completeness.

We first analyse the space usage of the encoders; then we discuss their speed using different query algorithms, namely AND, top-k Ranked AND and WAND [Broder et al. 2003].

Space. Table VIII reports the number of bits per posting of CPEF in the selected trade-off points in comparison with the ones taken by the tested competitors. We show in parentheses the relative percentage against CPEF. These numbers are consistent with the ones presented in the work by Dhulipala, Kabiljo, Karrer, Ottaviano, Pupyrev, and Shalita [2016], where only posting lists having more than 4096 elements are considered.

Table VIII. Bits per posting in selected trade-off points. In parentheses we show the relative percentage against CPEF.

	MIN		MID		MAX	
PEF	2.94	(+5.60%)	2.94	(+7.91%)	2.94	(+10.95%)
CPEF	2.78		2.72		2.65	
BIC	2.80	(+0.53%)	2.80	(+2.74%)	2.80	(+5.63%)
OptPFD	3.46	(+24.54%)	3.46	(+27.27%)	3.46	(+30.85%)
Varint-G8IU	9.76	(+251.02%)	9.76	(+258.72%)	9.76	(+268.82%)

(a) Gov2

	MIN		MID		MAX	
PEF	4.80	(+2.13%)	4.80	(+3.98%)	4.80	(+6.25%)
CPEF	4.70		4.62		4.52	
BIC	4.27	(−9.22%)	4.27	(−7.58%)	4.27	(−5.56%)
OptPFD	5.29	(+12.39%)	5.29	(+14.42%)	5.29	(+16.92%)
Varint-G8IU	10.1	(+115.14%)	10.1	(+119.04%)	10.1	(+123.81%)

(b) ClueWeb09

In particular, for both datasets, our clustered representation is always better than PEF, by up to 11% on Gov2 and 6.25% on ClueWeb09. On Gov2, already with a reference of size $100{,}000$ the clustered index is slightly better than BIC and becomes up to 5.63% smaller using longer references. On the other hand, on ClueWeb09 collection, BIC is still the smallest but clustering the index is able to *halve* the discrepancy between PEF and BIC, passing from 11.12% to 5.56%.

Regarding OptPFD, our clustered representation is on average 24% smaller on Gov2 and 14.5% smaller on ClueWeb09. As Variable Byte schemes are optimized for very fast decoding speed, not surprisingly our representation is more than 3.5 times smaller than Varint-G8IU on Gov2 and more than 2.15 times on ClueWeb09.

Table IX. Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

		MIN		MID		MAX	
TREC 05	PEF	14.6	(−17.5%)	14.6	(−29.0%)	14.6	(−49.7%)
	CPEF	17.7		20.6		29.1	
	BIC	41.1	(+131.9%)	41.1	(+99.5%)	41.1	(+41.3%)
	OptPFD	14.2	(−20.2%)	14.2	(−31.4%)	14.2	(−51.4%)
	Varint-G8IU	8.9	(−49.9%)	8.9	(−56.9%)	8.9	(−69.5%)
TREC 06	PEF	17.7	(−16.6%)	17.7	(−29.1%)	17.7	(−50.3%)
	CPEF	21.2		25.0		35.6	
	BIC	55.1	(+159.7%)	55.1	(+120.8%)	55.1	(+54.7%)
	OptPFD	18.4	(−13.2%)	18.4	(−26.2%)	18.4	(−48.3%)
	Varint-G8IU	11.4	(−46.3%)	11.4	(−54.3%)	11.4	(−68.0%)

(a) ClueWeb09

		MIN		MID		MAX	
TREC 05	PEF	3.7	(−30.4%)	3.7	(−37.5%)	3.7	(−52.1%)
	CPEF	5.3		5.9		7.8	
	BIC	10.5	(+96.2%)	10.5	(+76.2%)	10.5	(+35.0%)
	OptPFD	3.5	(−34.2%)	3.5	(−40.9%)	3.5	(−54.7%)
	Varint-G8IU	2.4	(−55.4%)	2.4	(−60.0%)	2.4	(−69.3%)
TREC 06	PEF	6.1	(−27.4%)	6.1	(−35.2%)	6.1	(−49.1%)
	CPEF	8.3		9.3		11.9	
	BIC	18.5	(+122.6%)	18.5	(+98.6%)	18.5	(+56.0%)
	OptPFD	6.1	(−27.1%)	6.1	(−35.0%)	6.1	(−49.0%)
	Varint-G8IU	4.1	(−51.1%)	4.1	(−56.4%)	4.1	(−65.8%)

(b) Gov2

Table X. Timings in milliseconds for AND top-10 BM25 queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

		MIN		MID		MAX	
TREC 05	PEF	30.0	(−18.4%)	30.0	(−31.4%)	30.0	(−50.9%)
	CPEF	36.8		43.7		61.1	
	BIC	71.3	(+93.7%)	71.3	(+63.0%)	71.3	(+16.6%)
	OptPFD	23.2	(−37.1%)	23.2	(−47.0%)	23.2	(−62.1%)
	Varint-G8IU	15.0	(−59.2%)	15.0	(−65.7%)	15.0	(−75.5%)
TREC 06	PEF	29.6	(−17.1%)	29.6	(−30.0%)	29.6	(−49.5%)
	CPEF	35.8		42.4		58.7	
	BIC	88.9	(+148.5%)	88.9	(+109.9%)	88.9	(+51.6%)
	OptPFD	26.0	(−27.3%)	26.0	(−38.6%)	26.0	(−55.7%)
	Varint-G8IU	15.8	(−55.9%)	15.8	(−62.8%)	15.8	(−73.1%)

(a) ClueWeb09

		MIN		MID		MAX	
TREC 05	PEF	8.6	(−32.1%)	8.6	(−38.8%)	8.6	(−49.0%)
	CPEF	12.6		14.0		16.8	
	BIC	19.2	(+51.6%)	19.2	(+36.8%)	19.2	(+13.9%)
	OptPFD	6.0	(−52.4%)	6.0	(−57.0%)	6.0	(−64.2%)
	Varint-G8IU	4.2	(−66.9%)	4.2	(−70.2%)	4.2	(−75.1%)
TREC 06	PEF	11.9	(−29.1%)	11.9	(−35.8%)	11.9	(−45.8%)
	CPEF	16.8		18.5		21.9	
	BIC	32.7	(+94.8%)	32.7	(+76.5%)	32.7	(+48.9%)
	OptPFD	9.3	(−44.6%)	9.3	(−49.8%)	9.3	(−57.7%)
	Varint-G8IU	5.8	(−65.1%)	5.8	(−68.4%)	5.8	(−73.3%)

(b) Gov2

Table XI. Timings in milliseconds for WAND top-10 BM25 queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

		MIN		MID		MAX	
TREC 05	PEF	38.9	(−14.6%)	38.9	(−28.1%)	38.9	(−47.0%)
	CPEF	45.5		54.0		73.3	
	BIC	86.9	(+91.0%)	86.9	(+60.9%)	86.9	(+18.6%)
	OptPFD	32.8	(−27.9%)	32.8	(−39.3%)	32.8	(−55.3%)
	Varint-G8IU	22.5	(−50.6%)	22.5	(−58.4%)	22.5	(−69.3%)
TREC 06	PEF	44.0	(−15.0%)	44.0	(−26.8%)	44.0	(−46.1%)
	CPEF	51.8		60.1		81.7	
	BIC	123.3	(+138.3%)	123.3	(+105.1%)	123.3	(+51.0%)
	OptPFD	41.6	(−19.6%)	41.6	(−30.8%)	41.6	(−49.0%)
	Varint-G8IU	27.0	(−47.9%)	27.0	(−55.1%)	27.0	(−67.0%)

(a) ClueWeb09

		MIN		MID		MAX	
TREC 05	PEF	12.6	(−27.4%)	12.6	(−33.9%)	12.6	(−44.5%)
	CPEF	17.3		19.1		22.7	
	BIC	26.6	(+53.2%)	26.6	(+39.4%)	26.6	(+17.1%)
	OptPFD	10.1	(−41.9%)	10.1	(−47.1%)	10.1	(−55.6%)
	Varint-G8IU	7.3	(−58.0%)	7.3	(−61.8%)	7.3	(−67.9%)
TREC 06	PEF	17.4	(−25.5%)	17.4	(−32.5%)	17.4	(−41.4%)
	CPEF	23.3		25.7		29.6	
	BIC	43.2	(+85.3%)	43.2	(+67.8%)	43.2	(+45.8%)
	OptPFD	14.5	(−37.7%)	14.5	(−43.5%)	14.5	(−50.9%)
	Varint-G8IU	9.7	(−58.2%)	9.7	(−62.1%)	9.7	(−67.1%)

(b) Gov2

Queries. As a general overview, all query algorithms confirm the following behavior. On the ClueWeb09 test collection and for MIN points, CPEF indexes are 2.3 times faster on average (up to 2.6 times on TREC 06 for AND queries) than BIC and less than 17% on average slower than PEF indexes. On Gov2 results are slightly different: for MIN points, CPEF indexes are 84% faster on average (up to more than 2.2 times on TREC 06 for AND queries) than BIC and less than 29% on average slower than PEF indexes. Considering the MAX point, we notice the difference between the two TREC query sets, as already pointed out by Ottaviano and Venturini [2014]. While on TREC 06, we gain 50% over BIC, but loose approximately the same over PEF, on TREC 05 the advantage over BIC is reduced, ranging from 16.6% to 41.3%. This behavior is confirmed for MID points too: on TREC 06, CPEF still goes more than twice as fast as BIC, while being 29% on average worse than PEF; on TREC 05 the advantage over BIC is 77% on average. Very similar consideration holds for Gov2 too.

The comparison between CPEF and OptPFD is practically the same as the one between CPEF and PEF given that PEF is as fast as OptPFD [Ottaviano and Venturini 2014]. As already pointed out, the Varint-G8IU scheme is not competitive with all the other approaches regarding space usage, but is faster than our proposal by 55% on average. Since these considerations are valid independently of the experimented query algorithms, in what follows we treat each query algorithm separately comparing CPEF against PEF and BIC.

Table IX reports all timings in milliseconds for AND queries. In particular, on Gov2 collection, our clustered representation is twice as fast as BIC already for MIN point (retaining slightly better space too). Choosing MAX, we have a significantly reduced the space with respect to PEF, improved over BIC as the best state-of-the-art encoding for space usage, while maintaining a noticeable advantage in speed (more than 50% on TREC 06).

Accessing frequencies has an impact at query time. To measure this, we record the timings for computing the top-10 results for AND and WAND [Broder et al. 2003], using the relevance score function BM25 [Robertson and Jones 1976]. Average percentages are indeed very similar for the two query algorithms and are reported in Table X and XI respectively.

To further confirm the difference between the two query sets, consider MAX point. As an example, for top-10 AND, we notice that the clustered representation is slightly better than BIC on TREC 05 (16.6% and 13.9% respectively for ClueWeb09 and Gov2), while on TREC 06 we gain the usual 50% on average. Very similar considerations hold for the WAND algorithm as well.

5. CONCLUSIONS AND FUTURE WORK

We explored the possibility of encoding clusters of posting lists so that the redundancy of docIds is exploited for better index compression. We introduced a novel posting list organisation consisting in two segments, both encoded with partitioned Elias-Fano. One segment is rewritten with respect to the cluster reference list and, therefore, encoded with a highly reduced universe. We developed an ad-hoc clustering algorithm for posting lists and show two possible algorithms for building the reference list: one selects postings according to their frequencies within the cluster (frequency-based selection); the other selects postings according to their contribution to the overall space cost reduction (space-based selection). The space-based approach yields smaller indexes with respect to the frequency-based one especially for smaller reference sizes, at the expense of a higher index building time. Varying the reference size of the representation, we have shown different space/time trade-offs. At the two edges of the spectrum of the obtainable trade-offs, we could either prefer: (1) save 6.25% up to 11% of space over partitioned Elias-Fano and 5.63% over Binary Interpolative; (2) keep a noticeable

speed improvement (103% on average) over Binary Interpolative and a very small time overhead compared to partitioned Elias-Fano (23% on average), while retaining even less space. As the speed of the optimized PForDelta implementation (OptPFD) by [Yan et al. 2009] is practically the same as the one of partitioned Elias-Fano, the above query processing considerations also apply to the comparison between our proposal and OptPFD. Our clustered representation is 55% slower than the SIMD-optimized Variable Byte implementation (Varint-G8IU) by [Stepanov et al. 2011]. However, our clustered indexes dominates both OptPFD and Varint-G8IU for space usage. Against the former we retain 24% of space less on Gov2 and 14.5% on ClueWeb09. Against the latter, our representation is more than 3.5 times smaller on Gov2 and more than 2.15 times on ClueWeb09.

An interesting future direction could focus on designing an approximation algorithm for the reference selection problem with strong guarantees on the quality of the computed result. Moreover, selecting the clusters such that the query time is minimized is an interesting future research problem.

REFERENCES

Charu Aggarwal and Chandan Reddy. 2013. *Data Clustering: Algorithms and Applications* (1st ed.). Chapman and Hall/CRC.

Vo Ngoc Anh and Alistair Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval Journal* 8, 1 (2005), 151–166.

Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Software: Practice and Experience* 40, 2 (2010), 131–147.

David Arthur and Sergei Vassilvitskii. 2007. K-means++: the advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1027–1035.

Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable K-means++. In *Proceedings of the Very Large Database Endowment (PVLDB)*, Vol. 5. 622–633.

Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th ACM International Conference on Information and Knowledge Management (CIKM)*. 426–434.

Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. 2012. Earlybird: Real-time search at twitter. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1360–1369.

Stefan Büttcher and Charles Clarke. 2007. Index compression is good, especially for random access. In *Proceedings of the 16th ACM International Conference on Information and Knowledge Management (CIKM)*. 761–770.

Stefan Büttcher, Charles Clarke, and Gordon Cormack. 2010. *Information retrieval: implementing and evaluating search engines*. MIT Press.

Surajit Chaudhuri, Kenneth Church, Arnd Christian König, and Liying Sui. 2007. Heavy-Tailed Distributions and Multi-Keyword Queries. In *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 663–670.

David Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo.

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. 2013. Unicorn: A System for Searching the Social Graph. In *Proceedings of the Very Large Database Endowment (PVLDB)*, Vol. 6. 1150–1161.

Renaud Delbru, Stéphane Campinas, and Giovanni Tummarello. 2012. Searching web data: An entity retrieval and high-performance indexing model. *Journal of Web Semantics* 10 (2012), 33–58.

Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proceedings of the 22nd ACM Conference on Knowledge Discovery and Data Mining (KDD)*.

Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM (JACM)* 21, 2 (1974), 246–260.

Robert Mario Fano. 1971. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT* (1971).

Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2011. On optimally partitioning a text to improve its compression. *Algorithmica* 61, 1 (2011), 51–74.

Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*. 370–379.

Hoang Thanh Lam, Raffaele Perego, Nguyen Thoi Minh Quan, and Fabrizio Silvestri. 2009. Entry Pairing in Inverted File. In *Proceedings of the 10th International Conference Web Information Systems Engineering (WISE)*. 511–522.

Daniel Lemire and Leonid Boytsov. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* (2013), 1–29.

Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory (IT)* 28 (1982), 129–137.

Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval Journal* 3, 1 (2000), 25–47.

Giuseppe Ottaviano, Nicola Tonellotto, and Rossano Venturini. 2015. Optimal Space-time Tradeoffs for Inverted Indexes. In *Proceedings of the 8th Annual International ACM Conference on Web Search and Data Mining (WSDM)*. 47–56.

Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282.

Dan Pelleg and Andrew Moore. 2000. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*. 727–734.

Stephen Robertson and Sparck Jones. 1976. Relevance weighting of search terms. *Journal of the American Society for Information Science* 27, 3 (1976), 129–146.

David Salomon. 2007. *Variable-length Codes for Data Compression*. Springer.

Fabrizio Silvestri. 2007. Sorting Out the Document Identifier Assignment Problem. In *Proceedings of the 29th European Conference on IR Research (ECIR)*. 101–112.

Fabrizio Silvestri and Rossano Venturini. 2010. 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)*. 1219–1228.

Michael Steinbach, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. In *6th Annual Conference on Knowledge Discovery and Data Mining (KDD), Workshop on Text Mining*. 109–111.

Alexander Stepanov, Anil Gangolli, Daniel Rose, Ryan Ernst, and Paramjit Oberoi. 2011. SIMD-based decoding of posting lists. In *Proceedings of the 20th ACM Inter-*

national Conference on Information and Knowledge Management (CIKM). 317–326.
Sebastiano Vigna. 2013. Quasi-succinct indices. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 83–92.
Ian Witten, Alistair Moffat, and Timothy Bell. 1999. *Managing gigabytes: compressing and indexing documents and images* (2nd ed.). Morgan Kaufmann.
Rui Xu and Donald Wunsch. 2005. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks (NN)* 16, 3 (2005), 645–678.
Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*. 401–410.
Zhaohua Zhang, Jiancong Tong, Haibing Huang, Jin Liang, Tianlong Li, Rebecca J. Stones, Gang Wang, and Xiaoguang Liu. 2016. 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)*. 275–284.
Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys (CSUR)* 38, 2 (2006), 1–56.
Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. 59–70.