Partitioned Elias-Fano Indexes

Giuseppe Ottaviano
ISTI-CNR, Pisa
giuseppe.ottaviano@isti.cnr.it

Rossano Venturini
Dept. of Computer Science, University of Pisa
rossano@di.unipi.it

ABSTRACT

The *Elias-Fano* representation of monotone sequences has been recently applied to the compression of inverted indexes, showing excellent query performance thanks to its efficient random access and search operations. While its space occupancy is competitive with some state-of-the-art methods such as γ-δ-Golomb codes and PForDelta, it fails to exploit the local clustering that inverted lists usually exhibit, namely the presence of long subsequences of close identifiers.

In this paper we describe a new representation based on partitioning the list into chunks and encoding both the chunks and their endpoints with Elias-Fano, hence forming a two-level data structure. This partitioning enables the encoding to better adapt to the local statistics of the chunk, thus exploiting clustering and improving compression. We present two partition strategies, respectively with fixed and variable-length chunks. For the latter case we introduce a linear-time optimization algorithm which identifies the minimum-space partition up to an arbitrarily small approximation factor.

We show that our partitioned Elias-Fano indexes offer significantly better compression than plain Elias-Fano, while preserving their query time efficiency. Furthermore, compared with other state-of-the-art compressed encodings, our indexes exhibit the best compression ratio/query time trade-off.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

Keywords

Compression; Dynamic Programming; Inverted Indexes

1. INTRODUCTION

The inverted index is the data structure at the core of most large-scale search systems for text, (semi-)structured data, and graphs, with web search engines, XML and RDF databases, and graph search engines in social networks as the most notable examples. The huge size of the corpora involved and the stringent query efficiency requirements of these applications have driven a large amount of research with the ultimate goal of minimizing the space occupancy of the index and maximizing its query processing speed. These are two conflicting objectives: a high level of compression is obtained by removing the redundancy in the dataset, high speed is obtained by keeping the data easily accessible and by augmenting the dataset with auxiliary information that drive the query processing algorithm. The effects of space reduction on the memory hierarchy partially mitigate this dichotomy. Indeed, memory transfers are an important bottleneck in query processing, and, thus, fitting more data into higher and faster levels of the hierarchy reduces the transfer time from the slow levels, and, hence, speeds up the algorithm [5]. In fact, in the last few years the focus has shifted from disk-based indexes to in-memory indexes, as in many scenarios it is not possible to afford a single disk seek. However, these beneficial effects can be nullified by slow decoding algorithms. Thus, research has focused its attention on designing solutions that best balance decoding time and space occupancy.

In its most basic and popular form, an inverted index is a collection of sorted sequences of integers [6, 16, 27]. Compressing such sequences is a crucial problem which has been studied since the 1950s; the literature presents several approaches, each of which introduces its own trade-off between space occupancy and decompression speed [15, 17, 19, 22, 26]. Most of these methods only allow sequential decoding, so the lists involved while processing a query need to be entirely decoded. This can be avoided by using the standard technique of splitting each sequence into blocks of fixed size (say, 128 elements) and encoding each block independently, so that it is possible to avoid the decompression of portions which are not necessary for answering the query. Still, a block must be fully decoded even if just one of its values is required.

Recently Vigna [23] overcame this drawback by introducing a new data structure called *quasi-succinct index*. This index hinges on the Elias-Fano representation of monotone sequences [10, 11], a conceptually simple and elegant data structure that supports fast random access and search operations, combining strong theoretical guarantees and excellent practical performance. Vigna showed that quasi-succinct indexes can compete in speed with mature and highly engineered implementations of state-of-the-art inverted indexes. In particular, Elias-Fano shines when the values accessed are scattered in the list; this is very common in conjunctive queries when the number of results is significantly smaller

SIGIR'14, July 6–11, 2014, Gold Coast, Queensland, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2257-7/14/07 ...$15.00.
http://dx.doi.org/10.1145/2600428.2609615.

than the sequences being intersected. A typical scenario, for example, is intersecting edge sets in large social network graphs, as Facebook's Graph Search, which has in fact adopted an implementationof Elias-Fano indexes [8].

Experiments show however that Elias-Fano indexes can be significantly larger than those obtained using state-of-the-art encoders. This inefficiency is caused by its inability to exploit any characteristics of the sequence other than two global statistics: the number of elements in the list, and the value of the largest element. More precisely, Elias-Fano represents a monotone sequence of m integers smaller than u with roughly $m\lceil\log\frac{u}{m}\rceil + 2m$ bits of space, regardless of any regularities in the sequence. As an extreme toy example consider the sequence $S = [0, 1, 2, \ldots, m-2, u-1]$ of length m. This sequence is highly compressible since the length of the first run and the value of u, which can be encoded in $O(\log u)$ bits, are sufficient to describe S. Elias-Fano requires instead $\lceil\log\frac{u}{m}\rceil + 2$ bits for *every* element in the sequence, which is not one single bit less than it would require to represent a random sequence of m sorted elements smaller than u. Even if this is just a toy example, it highlights an issue that occurs frequently in compressing posting lists, whose compressibility is caused by large *clusters* of very close values.

In this paper we tackle this issue by partitioning the sequences into contiguous *chunks* and encoding each chunk independently with Elias-Fano, so that the encoding of each chunk can better adapt to the local statistics. To perform random access and search operations, the lists of chunk endpoints and boundary elements are in turn encoded with Elias-Fano, thus forming a two-level data structure which supports fast queries on the original sequence. The chunk endpoints can be completely arbitrary, so we propose two strategies to define the partition. The first is a straightforward uniform partition, meaning that each chunk (except possibly the last) has the same fixed size. The second aims at minimizing the space occupancy by setting up the partitioning as an instance of an optimization problem, for which we present a linear-time algorithm that is guaranteed to find a solution at most $(1+\epsilon)$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$.

We perform an extensive experimental analysis on two large text corpora, namely Gov2 and ClueWeb09 (Category B), with several query operations, and compare our indexes with the plain quasi-succinct indexes as described in [23], and with three state-of-the-art list encodings, namely Binary Interpolative Coding [17], the PForDelta variant OptPFD [26], and Varint-G8IU [22], a SIMD-optimized Variable Byte code. Respectively, they are representative of best compression ratio, best compression ratio/processing speed trade-off, and highest speed in the literature.

In our experiments, we show that our indexes are significantly smaller than the original quasi-succinct indexes, with a small query time overhead. Compared to the others, they are slightly larger but significantly faster than Interpolative, both faster and smaller than OptPFD, and slightly slower but significantly smaller than Varint-G8IU.

Our contributions. We list here our main contributions.

1. We introduce a two-level representation of monotone sequences which, given a partition of a sequence into chunks, represents each chunk with Elias-Fano and stores the endpoints of the chunks and their boundary values in separate Elias-Fano sequences, in order to support fast random access and search operations.
2. We describe two partitioning strategies: an uniform strategy, which divides the sequence into chunks with a fixed size, and a strategy with variable-length chunks, whose endpoints are chosen by solving an optimization problem in order to minimize the overall space occupancy. More precisely, we introduce a linear-time dynamic programming algorithm which finds a partition whose cost is at most $(1 + \epsilon)$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$. In the experiments we show that this ϵ-optimal strategy gives significantly smaller indexes than the uniform strategy, which in turn are significantly smaller than the indexes obtained with non-partitioned Elias-Fano. Indeed, the latter are more than 23% larger on ClueWeb09 and more than 64% larger on Gov2.
3. We show with an extensive experimental analysis that the partitioned indexes are only slightly slower than non-partitioned indexes. Furthermore, in comparison with other indexes from the literature, the ϵ-optimal indexes dominate in both space and time methods which are in the same region of the space-time trade-off curve, and obtain spaces very close to the methods which give the highest compression ratio. More precisely, Binary Interpolative Coding is only 2%-8% smaller but up to 5.5 times slower; OptPFD is roughly 12% larger and almost always slower; Varint-G8IU is 10%-40% faster but more than 2.5 times larger.

2. BACKGROUND AND NOTATION

Given a collection $\mathcal{D}$ of documents, the *posting list* of a term t is the list of all the document identifiers, or *docIds*, that contain the term t. The collection of posting lists for all the terms in the collection is called the *inverted index* of $\mathcal{D}$; the set of such terms is usually called the *dictionary*. Posting lists are often augmented with additional information about each docId, such as the number of occurrences of the term in the document (the *frequency*), and the set of positions where the term occurs. Since the inverted index is a fundamental data structure in virtually all modern search systems, there is a vast amount of literature describing several variants and query processing strategies; we refer the reader to the excellent surveys and books on the subject [6, 16, 27].

In the following, we adopt the *docId-sorted* variant, meaning that each posting list is sorted by docId; this enables fast query processing and efficient compression. In our experiments we focus our attention on posting lists storing docIds and frequencies; we do not store positions or other additional data, since they have different nature and often require specialized compression techniques [25], thus they are outside of the scope of this paper. We also ignore additional per-document or per-term information, such as the mappings between docIds and URLs, or between termIds and actual terms, as their space is negligible compared to the index size.

Query Processing. Given a term query as a (multi-)set of terms, the basic query operations are the boolean conjunctive (AND) and disjunctive (OR) queries, retrieving the documents that contain respectively all the terms or at least one of them. In many scenarios the query-document pairs can be associated with a *relevance score* which is usually a function of the term frequencies in the query and in the document, and other global statistics. Instead of the full set of matches, for scored queries it is often sufficient to retrieve the k highest scored documents for a given k. A widely used relevance score is BM25 [18], which we will use in our experiments.

There exist two popular query processing strategies, dual in nature, namely *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 $\mathsf{NextGEQ}_t(d)$ operator, which returns the smallest docId in the list of t that is greater than or equal to d. A fast implementation of the function $\mathsf{NextGEQ}_t(d)$ is crucial for the efficiency of this process. The trivial implementation that scans sequentially the whole posting lists is usually too slow; a common solution resorts to *skipping* strategies. The basic idea is to divide the lists in small blocks that are compressed independently, and to store additional information about each block, in particular the maximum docId present in the block. This allows to find and decode only the block that may contain the sought docId by scanning the list of maxima, thus skipping a potentially large number of useless blocks. In the following we call *block-based* the indexes that adopt this technique.

Solving scored queries can be achieved with DAAT by computing the relevance score for the matching documents as they are found, and maintaining a priority queue with the top-k matches. This can be very inefficient for scored disjunctive queries, as the whole lists need to be scanned. Several query processing strategies have been introduced to alleviate this problem. Among them, one the most popular is WAND [3], which augments the index by storing for each term its maximum impact to the score, thus allowing to skip large segments of docIds if they only contain terms whose sum of maximum impacts is smaller than the top-k documents found up to that point. Again, WAND can be efficiently implemented in terms of $\mathsf{NextGEQ}_t(d)$.

3. RELATED WORK

Index compression ultimately reduces to the problem of representing sequences, specifically strictly monotone sequences for docIds, and positive sequences for the frequencies. The two are equivalent: a strictly monotone sequence can be turned into a positive sequence by subtracting from each element the one that precedes it (also known as *delta encoding*), the other direction can be achieved by computing prefix sums. For this reason most of the work assumes that the posting lists are delta-encoded and focuses on the representation of sequences of positive integers.

Representing such sequences of integers in compressed space is a crucial problem which has been studied since the 1950s with applications going beyond inverted indexes. A classical solution is to assign to each integer an uniquely-decodable variable length code; if the codes do not depend on the input they are called *universal codes*. The most basic example is the *unary code*, which encodes a non-negative integer x as the bit sequence 0^x1. The unary code is efficient only if the input distribution is concentrated on very small integers. More sophisticated codes, such as *Elias Gamma/Delta codes* and *Golomb/Rice codes* build on the unary code to efficiently compress a broader class of distributions. We refer to Salomon [19] for an in-depth discussion on this topic.

Bit-aligned codes can be inefficient to decode as they require several bitwise operations, so byte-aligned or word-aligned codes are usually preferred if speed is a main concern. *Variable byte* [19] or *VByte* is the most popular byte-aligned code. In VByte the binary representation of a non-negative integer x is split into groups of 7 bits which are represented as a sequence of bytes. The lower 7 bits of each byte store the data, whereas the eighth bit, called the continuation bit, is equal to 1 only for the last byte of the sequence. Stepanov *et al.* [22] present a variant of variable byte (called *Varint-G8IU*) which exploits SIMD operations of modern CPUs to further speed up decoding.

A different approach is to encode simultaneously *blocks* of integers in order to improve both compression ratio and decoding speed. This line of work has seen in the last few years a proliferation of encoding approaches which find their common roots in *frame-of-reference* (For) [14]. Their underlying idea is to partition the sequence of integers into blocks of fixed or variable length and to encode each block separately. The integers in each block are encoded by resorting to codewords of fixed length. A basic application of this technique (called also binary packing or packed binary [2,15]) partitions the sequence of integers into blocks of b consecutive integers (e.g., $b = 128$ integers); for each block, the algorithm encodes the range enclosing the values in the block, say $[l, r]$, then each value is subtracted l and represented with $h = \lceil\log(r - l + 1)\rceil$ bits. There are several variants of this approach which differentiate themselves for their encoding or partitioning strategies [9,15,21]. For example, *Simple-9* and *Simple-16* [1,2,26] are two popular variants of this approach.

A major space inefficiency of For is the fact that the presence of few large values in the block forces the algorithm to encode all its integers with a large h, thus affecting the overall compression performance. To address this issue, *PForDelta* (PFD) [28] introduces the concept of *patching*. In PFD the value of h is chosen so that h bits are sufficient to encode a large fraction of the integers in the block, say 90%. Those integers that do not fit within h bits are called *exceptions* and encoded separately with a different encoder (e.g., Simple-9 or Simple-16). Yan *et al.*'s introduce the OptPFD variant [26], which selects for each block the value of h that minimizes the space occupancy. OptPFD is more compact and only slightly slower than the original PFD [15,26].

A completely different approach is taken by *Binary Interpolative Coding* [17], which skips the delta-encoding step and directly encodes strictly monotone sequences. This method recursively splits the sequence of integers in two halves, encoding at each split the middle element and recursively the two halves. At each recursive step the range that encloses the middle element is reduced, and so is the number of bits needed to encode it. Experiments [21,24,26] have shown that Binary Interpolative Coding is the best encoding method for highly clustered sequence of integers. However, this space efficiency is paid at the cost of a very slow decoding algorithm.

Recently, Vigna [23] introduced *quasi-succinct indexes*, based on the Elias-Fano representation of monotone sequences, showing that it is competitive with delta-encoded block-based indexes. Our paper aims at making this representation more space-efficient.

Optimal partitioning algorithms. The idea of partitioning a sequence to improve compression dates back to Buchsbaum *et al.* [4], who address the problem of partitioning the input of a compressor C so that compressing the chunks individually yields a smaller space than compressing the whole input at once. Their paper discusses only the case of compressing a large table of records with gzip but their solution can be adapted to solve the more general problem stated above. Their approach is to reduce this optimization problem to a dynamic programming recurrence which is solved in $\Theta(m^3)$ time and $\Theta(m^2)$ space, where m is the input size.

Silvestri and Venturini [21] resort to a similar dynamic programming recurrence to optimize their encoder for posting lists. They obtain an $O(mh)$ construction time by limiting the length of the longest part to h. Ferragina *et al.* [12] significantly improve the result in [4] by computing in $O(m \log_{1+\epsilon} m)$ time and $O(m)$ space a partition whose compressed size is guaranteed to be at most $(1+\epsilon)$ times the optimal one, for any given $\epsilon > 0$, provided that the compression ratio of C on any portion of the input can be estimated in constant time.

In this paper we apply the same ideas in [12] to the Elias-Fano representation and we exploit some of its properties to compute *exactly*, as opposed to estimating, its encoding cost in constant time. Then, we improve the optimization algorithm reducing its time to $O(m)$ while preserving the same approximation guarantees.

4. SEARCHABLE SEQUENCES

The *Elias-Fano representation* [10, 11] is an elegant encoding for monotone sequences, which provides good compression ratio and efficient access and searching operations. Consider a monotonically increasing sequence $S[0, m-1]$ of m non-negative integers (i.e., $S[i] \leq S[i+1]$, for any $0 \leq i < m-1$) drawn from an *universe* $[u] = \{0, 1, \ldots, u-1\}$.

Given an integer ℓ, the elements of S are conceptually grouped into buckets according to their $\lceil \log u \rceil - \ell$ *higher* bits. The number of buckets is thus $\frac{u}{2^\ell}$. The cardinalities of these buckets (including the empty ones) are written into a bitvector H with negated unary codes; it follows that H has length at most $m + \frac{u}{2^\ell}$ bits. The remaining ℓ *lower* bits of each integer are concatenated into a bitvector L, which thus requires $m\ell$ bits. It is easy to see that H and L are sufficient to recover $S[i]$ for every i: its $\lceil \log u \rceil - \ell$ higher bits are equal to the number of 0s preceding the ith 1 in H, and the ℓ lower bits can be retrieved directly from L.

While ℓ can be chosen arbitrarily between 0 and $\lceil \log u \rceil$, it can be shown that the value $\ell = \lfloor \log \frac{u}{m} \rfloor$ minimizes the overall space. Summing up the lengths of H and L, it follows that the representation requires at most $m \lceil \log \frac{u}{m} \rceil + 2m$ bits.

Despite its simplicity, it is possible to support efficiently powerful operations on S. For our purposes we are interested in the following.

- $\mathsf{Access}(i)$ which, given $i \in [m]$, returns $S[i]$;
- $\mathsf{NextGEQ}(x)$ which, given $x \in [u]$, returns the smallest element in S which is greater than or equal to x.

The support for these operations requires to augment H with an auxiliary data structure to efficiently answer Select_0 and Select_1 operations, which, given an index i, return the position of respectively the ith 1 or the ith 0 in H. See [23] and references therein for more details on the implementation of these standard operations.

To access the ith element of S we have to retrieve and concatenate its higher and lower bits. The value of the higher bits is obtained by computing $\mathsf{Select}_1(i) - i$ in H, which represents the number of 0s (thus, buckets) ending before the ith occurrence of 1. The lower bits are directly accessed by reading ℓ consecutive bits in L starting from position $i\ell$.

The operation $\mathsf{NextGEQ}(x)$ is supported by observing that $p = \mathsf{Select}_0(h_x) - h_x$ is the number of elements of S whose higher bits are smaller than h_x, where h_x are the higher bits of x. Thus, p is the starting position in S of the elements whose higher bits are equal to h_x (if any) or larger. $\mathsf{NextGEQ}(x)$ is identified by scanning the elements starting from p.

Several implementations of Elias-Fano supporting efficiently these operations are available[1].

All the operations can be implemented in a stateful cursor, in order to exploit locality of access by optimizing short forward skips, which are very frequent in query processing. An additional convenient cursor operation is Next, which advances the cursor from position i to $i+1$.

Note that Elias-Fano requires just *weak* monotonicity of S, so if only the Access operation is needed, the space occupancy can be reduced to $m \lceil \log \frac{u-m+1}{m} \rceil + 2m$ bits by turning S into a weakly monotone sequence $S'[i] = S[i] - i$ and encoding S' with Elias-Fano. At query time we can recover the ith entry of S by computing $\mathsf{Access}_{S'}(i) + i$.

The *quasi-succinct indexes* of Vigna [23] are a direct application of Elias-Fano; the posting lists are immediately representable with Elias-Fano as the docIds are monotonically sorted. Since Elias-Fano is not efficient for dense sequences, a bitvector is used instead when it is convenient to do so; the Access and $\mathsf{NextGEQ}$ can be supported efficiently with small additional data structures. The frequencies can be turned into a strictly monotone sequence by computing their prefix sums, and the ith frequency can be recovered as $\mathsf{Access}(i) - \mathsf{Access}(i-1)$. Moreover, since the query processing needs only Access on the frequencies, we can use the trick mentioned above to reduce the space occupancy of the representation. In positional indexes, a similar transformation can be used to represent the term positions.

4.1 Uniform partitioning

As we argued above Elias-Fano uses roughly $\lceil \log \frac{u}{m} \rceil + 2$ bits per element regardless the sequence being compressed. Notice that $\log \frac{u}{m}$ is the logarithm of the average distance among consecutive elements in the sequence. Apart from this average distance, Elias-Fano does not exploit in any way the nature of the underlying sequence and, thus, for example it does not make any distinction between the two extreme cases of a randomly generated sequence and a sequence formed by only a long run of consecutive integers. While paying $\lceil \log \frac{u}{m} \rceil$ bits per element is the best we can hope for in the former case, we would expect to achieve a better space occupancy in the latter. Indeed, a sequence of integers is intuitively more compressible than a random one when it contains regions of integers which are very close to each other. The presence of these regions is typical in posting lists. Consider for example a term which is present only in few domains. If the docIds are assigned by sorting the documents by their URLs, then the posting list of this term is, apart from few outliers, formed by clusters of integers in correspondence of those domains. Observe that, since the elements within each region are very close to each other, the average distance intra-region is much smaller than the global average distance.

[1]Such as the open-source `http://github.com/facebook/folly`, `http://github.com/simongog/sdsl`, `http://github.com/ot/succinct`, and `http://sux.di.unimi.it`.

The above observation is the main motivation for introducing a two-level Elias-Fano. The basic idea is to partition the sequence S into m/b chunks of b consecutive integers each, except possibly the last one. The first level is an Elias-Fano representation of the sequence L obtained by juxtaposing the last element of each chunk (i.e., $L = S[b-1], S[2b-1], \ldots, S[m-1]$). The second level is the collection of the chunks of S, each represented with Elias-Fano. The main advantage of having this first level is that the elements of the jth chunk can be rewritten in a smaller universe of size $u_j = L[j] - L[j-1] - 1$ by subtracting $L[j-1]+1$ from each element. Thus, the Elias-Fano representation of the chunk requires $\lceil \log \frac{u_j}{b} \rceil + 2$ bits per element. Since the quantity $\frac{u_j}{b}$ is the average distance of the elements within the chunk, we expect that this space occupancy is much smaller that the one obtained by representing the sequence in its entirety, especially for highly compressible sequences. Observe that part of this gain vanishes due to the cost of the first level which is $\lceil \log \frac{u}{m/b} \rceil + 2$ bits every b original elements. Indeed, the first level stores m/b integers drawn from a universe of size u.

This two-level representation introduces a level of indirection in solving the operations Access and NextGEQ. The operation Access(i) is solved as follows. Let j be the index of the chunk containing the ith element of S (i.e., $j = \lfloor i/b \rfloor$) and k be its offset within this chunk (i.e., $k = i \bmod b$). We access $L[j-1]$ and $L[j]$ on the first level to compute the size of the universe u_j of the chunk as $L[j] - L[j-1]$ (or $L[j]$ if j is the first chunk). Knowing u_j and b suffices for accessing the kth element of the jth chunk. If e is the value at this position, then we conclude that the value $S[i]$ is equal to $L[j]+1+e$. The operation NextGEQ(x) is solved as follows. We first compute the successor of x, say $L[j]$, on the first level. This implies that the successor of x in S is within the jth chunk and, thus, it can be identified by solving NextGEQ($x - L[j] - 1$) on this chunk.

An important distinction with block-based indexes is that the choice of b does not affect significantly the efficiency of the operations: while block-based indexes may need to scan the full block to retrieve one element, the chunks in our representation are searchable sequences themselves, so the performance does not degrade as b gets larger; it actually gets better, as fewer block boundaries have to be crossed during a query.

In our implementation we use different encodings to overcome the space inefficiencies of Elias-Fano in representing dense chunks. The jth chunk is dense if the chunk covers a large fraction of the elements in the universe $[u_j]$ (or, in other words, b is close to u_j). Indeed, the space bound $b\lceil \log \frac{u_j}{b} \rceil + 2b$ bits becomes close to $2u_j$ bits whenever b approaches u_j. However, we can always represent the chunk within u_j bits by writing the characteristic vector of the set of its elements as a bitvector. Note that Vigna [23] also uses this technique but only for whole lists, which are very unlikely to be so dense except for very few terms such as stop-words. In our case, instead, we expect dense chunks to occur frequently in representing posting lists, because they can be contained inside dense clusters of docIds. Hence, besides Elias-Fano, we adopt two other encodings chosen depending on the relation between u_j and b. The first encoding addresses the extreme case in which the chunk covers the whole universe (i.e., whenever u_j is equal to b). The first level gives us the values of u_j and b which are enough by themselves to derive all the elements in the chunk without the need of encoding further information. Operations Access and NextGEQ become trivial: both Access(i) and NextGEQ(i) are equal to i. The second encoding is used whenever the size of the Elias-Fano representation of the chunk is larger than u_j bits (i.e., whenever $b > \frac{u_j}{4}$). In this case we encode the set of elements in the chunk by writing its characteristic vector in u_j bits. The Access and NextGEQ operations can be reduced to the standard Rank and Select operations on bitvectors; their implementation is described in detail in [23].

4.2 Optimal partitioning

Splitting S into chunks of fixed size is likely to be suboptimal, since we cannot expect the dense clusters in S to appear aligned with the uniform partition. Intuitively it would be better to allow S to be partitioned freely, with chunks of variable size. It is not obvious however how to find the partition that minimizes the overall space occupancy: on one hand, the chunks should be as large as possible to minimize the number of entries in the first level of the representation and, thus, its space occupancy; on the other hand, the chunks should be as small as possible to minimize the average distances between their elements, and, thus, the space occupancy of the second level of the representation.

An optimal partition can be computed in $\Theta(n^2)$ time and space by solving a variant of dynamic programming recurrence introduced in [4]. However, these prohibitive complexities make this solution unfeasible for inputs larger than few thousands of integers. This is the main motivation for designing an approximation algorithm which reduces the time and space complexities to linear at the cost of finding slightly suboptimal solutions. More precisely, in this subsection we present an algorithm that identifies in $O(m \log_{1+\epsilon} \frac{1}{\epsilon})$ time and linear space a partition whose cost is only $1+\epsilon$ times larger than the optimal one, for any given $\epsilon \in (0,1)$. Observe that the time complexity is linear as soon as ϵ is constant.

Before entering into the technical details of our solution, it is convenient to fix precisely the space costs involved in our representation. The space occupancy of a given partition P of k chunks $S[i_0, i_1 - 1]\ S[i_1, i_2 - 1] \ldots S[i_{k-1}, i_k]$, with $i_0 = 0$ and $i_k = m-1$, is $C(P) = \sum_{h=0}^{k-1} C(S[i_h, i_{h+1} - 1])$ bits, where $C(S[i,j])$ is the cost of representing the chunk $S[i,j]$. Each of these costs $C(S[i,j])$ is the sum of two terms: a fixed cost F to store information regarding the chunk in the first level and the cost of representing its elements in the second level. Concerning the fixed cost F, for each chunk we store three integers in the first level: the largest integer within the chunk, the size of the chunk, and the pointer to its second-level Elias-Fano representation. Thus, we can safely upper bound this cost F with the quantity $2\log u + \log m$ bits. Instead, the cost of representing the elements in $S[i,j]$ is computed by taking the minimum between the costs of the three possible encodings introduced in the previous subsection. Depending on the size of the universe $u' = S[j] - S[i-1]$ (or, $u' = S[j]$, if $i = 0$) and the number of elements $m' = j - i + 1$, these three costs are i) $m'\ell + m' + \frac{u'}{2^\ell}$ bits with $\ell = \lfloor \log \frac{u'}{m'} \rfloor$, if $S[i,j]$ is encoded with Elias-Fano; ii) m' bits, if $S[i,j]$ is encoded with its characteristic vector; iii) 0 bits, if $m' = u'$ and, thus, $S[i,j]$ covers the whole universe. A crucial property to devise our approximation algorithm is the monotonicity of the cost function C, namely, for any i, j and k with $0 \le i < j < k \le m$, we have $C(S[i,j]) \le C(S[i,k])$.

In the following we first use the algorithm in [12] to obtain a solution which finds a $(1+\epsilon)$-approximation in time $O(m \log_{1+\epsilon} \frac{U}{F})$, where U is the cost in bits of representing S as a single partition. Then, we improve the algorithm to obtain a linear time solution with the same approximation guarantees. Following [12], it is convenient to recast our optimization problem to the problem of finding a shortest path in a particular directed acyclic graph (DAG) $\mathcal{G}$. Given the sequence S of m integers, the graph $\mathcal{G}$ has a vertex $v_0, v_1, \ldots, v_{m-1}$ for each position of S plus a dummy vertex v_m marking the end of the sequence. The DAG $\mathcal{G}$ is complete in the sense that, for any i and j with $i < j \leq m$, there exists the edge from v_i to v_j, denoted as (v_i, v_j). Notice that there is a one-to-one correspondence between paths from v_0 to v_m in $\mathcal{G}$ and partitions of S. Indeed, a path $\pi = (v_0, v_{i_1})(v_{i_1}, v_{i_2}) \ldots (v_{i_{k-1}}, v_{i_m})$ crossing k edges corresponds to the partition $S[0, i_1-1]\ S[i_1, i_2-1] \ldots S[i_{k-1}, m-1]$ of k chunks. Hence, by assigning the weight $w(v_i, v_j) = C(S[i, j-1])$ to each edge (v_i, v_j), the weight of a path is equal to the cost in bits of the corresponding partition. Thus, a shortest path on $\mathcal{G}$ corresponds to an optimal partition of S. Computing a shortest path on a DAG has a time complexity proportional to the number of edges in the DAG. This is done with a classical elegant algorithm which processes the vertices from left to right [7]. The goal is to compute the value $M[v]$ for each vertex v which is equal to the cost of a shortest path which starts at v_0 and ends at v. Initially, $M[v_0]$ is set to 0, while $M[v]$ is set to $+\infty$ for any other vertex v. When the algorithm reaches the vertex v in its left-to-right scan, it assumes that $M[v]$ has been already correctly computed and extends this shortest path with any edge outgoing from v. This is done by visiting each edge (v, v') outgoing from v and by computing $M[v] + w(v, v')$. If this quantity is smaller than $M[v']$, the path that follows the shortest path from v_0 to v and then the edge (v, v') is currently the best way to reach v'. Thus, $M[v']$ is updated to $M[v] + w(v, v')$. The correctness of this algorithm can be proved by induction and, since each edge is relaxed exactly once, its time complexity is proportional to the number of edges in the DAG.

Unfortunately our DAG $\mathcal{G}$ is complete and, thus, it has $\Theta(n^2)$ edges, so this algorithm by itself does not suffice to obtain an efficient solution for our problem. However, it can be used as the last step of a solution which performs a non-trivial pruning of $\mathcal{G}$. This pruning produces another DAG $\mathcal{G}_\epsilon$ with two crucial properties: i) its number of edges is substantially reduced from $\Theta(n^2)$ to $O(m \log_{1+\epsilon} \frac{U}{F})$, for any given $\epsilon \in (0, 1)$; ii) its shortest path distance is (almost) preserved since it increases by no more than a factor $1 + \epsilon$.

The pruned graph $\mathcal{G}_\epsilon$ is constructed as the subgraph of $\mathcal{G}$ consisting of all the edges (v_i, v_j) such that at least one of the following two conditions holds: i) there exists an integer $h \geq 0$ such that $w(v_i, v_j) \leq F(1+\epsilon)^h < w(v_i, v_{j+1})$; ii) (v_i, v_j) is the last outgoing edge from v_i (i.e., $j = m$).

Since w is monotone, these conditions correspond of keeping, for each integer h, the edge of $\mathcal{G}$ that better approximates the value $F(1+\epsilon)^h$ from below. The edges of $\mathcal{G}_\epsilon$ are called $(1+\epsilon)$-*maximal* edges. We point out that, since there exist at most $\log_{1+\epsilon} \frac{U}{F}$ possible values for h, each vertex of $\mathcal{G}_\epsilon$ has at most $\log_{1+\epsilon} \frac{U}{F}$ outgoing $(1+\epsilon)$-maximal edges. Thus, the total size of $\mathcal{G}_\epsilon$ is $O(m \log_{1+\epsilon} \frac{U}{F})$. Theorem 3 in [12] proves that the shortest path distance on $\mathcal{G}_\epsilon$ is at most $1 + \epsilon$ times larger than the one in $\mathcal{G}$. Thus, given $\mathcal{G}_\epsilon$, a $(1+\epsilon)$-approximated partition can be computed in $O(m \log_{1+\epsilon} \frac{U}{F})$ time with the above algorithm.

We show now how to further reduce this time complexity to $O(m \log_{1+\epsilon} \frac{1}{\epsilon})$ time without altering the approximation guarantees. Let $\epsilon_1 \in (0, 1]$ and $\epsilon_2 \in (0, 1]$ be two parameters to be fixed later. We first obtain the graph $\bar{\mathcal{G}}$ from $\mathcal{G}$ by keeping only edges whose weight is no more than $L = F + \frac{2F}{\epsilon_1}$ plus the first edge outgoing from every vertex whose cost is larger than L. Then, we apply the pruning above to $\bar{\mathcal{G}}$ by fixing the approximation parameter to ϵ_2. The only difference here is that we force the pruning to retain the m edges in $\bar{\mathcal{G}}$ of cost larger than L. In this way we obtain a graph $\bar{\mathcal{G}}_{\epsilon_2}$ having $O(m \log_{1+\epsilon_2} \frac{L}{F}) = O(m \log_{1+\epsilon_2} \frac{1}{\epsilon_1})$ edges. We can prove that the shortest path distance in $\bar{\mathcal{G}}_{\epsilon_2}$ is at most $(1+\epsilon_1)(1+\epsilon_2)$ times larger than the one in $\mathcal{G}$. This implies that the partition computed with $\bar{\mathcal{G}}$ is a $(1+\epsilon)$-approximation of the optimal one, by setting $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{3}$ so that $(1+\epsilon_1)(1+\epsilon_2) \leq 1 + \epsilon$. This result is stated in the following lemma.

Lemma 1. *For any $\epsilon_1 > 0$ and $\epsilon_2 > 0$, the shortest path distance in $\bar{\mathcal{G}}_{\epsilon_2}$ is at most $(1+\epsilon_1)(1+\epsilon_2)$ times larger than the one in $\mathcal{G}$.*

Proof. Let $\pi_{\mathcal{G}}$, $\pi_{\bar{\mathcal{G}}}$ and $\pi_{\bar{\mathcal{G}}_{\epsilon_2}}$ denote the shortest paths in $\mathcal{G}$, $\bar{\mathcal{G}}$ and $\bar{\mathcal{G}}_{\epsilon_2}$, respectively. We know that the weight $w(\pi_{\bar{\mathcal{G}}_{\epsilon_2}})$ of $\pi_{\bar{\mathcal{G}}_{\epsilon_2}}$ satisfies $w(\pi_{\bar{\mathcal{G}}_{\epsilon_2}}) \leq (1+\epsilon_2) w(\pi_{\bar{\mathcal{G}}})$. It remains to prove that $w(\pi_{\bar{\mathcal{G}}}) \leq (1+\epsilon_1) w(\pi_{\mathcal{G}})$ which allows us to conclude that $w(\pi_{\bar{\mathcal{G}}_{\epsilon_2}}) \leq (1+\epsilon_1)(1+\epsilon_2) w(\pi_{\mathcal{G}})$.

We do so by showing that there exists a path π' in $\bar{\mathcal{G}}$ such that its weight is no more than $(1+\epsilon_1)$ times the weight of the shortest path $\pi_{\mathcal{G}}$ of $\mathcal{G}$. The thesis follows immediately because $\pi_{\bar{\mathcal{G}}}$ is a shortest path, so by definition $w(\pi_{\bar{\mathcal{G}}}) \leq w(\pi')$.

The path π' is obtained by transforming $\pi_{\mathcal{G}}$ so that its edges are all contained in $\bar{\mathcal{G}}$. Note that this transformation is not actually performed by the algorithm but it only serves for proving the lemma.

This transformation is done as follows. For each edge (u_i, v_j) in $\pi_{\bar{\mathcal{G}}}$, either $w(v_i, v_j) \leq L$, so there is nothing to do, or $w(v_i, v_j) > L$, in which case we need to substitute it with a subpath from v_i to v_j whose edges are all in $\bar{\mathcal{G}}$. This subpath can be found greedily, starting from v_i and traversing always the longest edge until we reach v_j. It can be proved that the weighting function w satisfies $w(v_i, v_k) + w(v_k, v_j) \leq w(v_i, v_j) + F + 1$ for any $0 \leq i < k < j \leq m$; intuitively, this means that by breaking an edge into two shorter edges we lose at most $F + 1$ bits. By combining these properties with the fact that all the edges in the subpath except possibly the last have cost at least L, it follows that the number of edges in this subpath cannot be larger than $\frac{w(v_i, v_j) - F}{L - F} + 1$ and its cost is at most $w(v_i, v_j) + \left(\frac{w(v_i, v_j) - F}{L - F} + 1\right)(F + 1) \leq w(v_i, v_j) + \epsilon_1 w(v_i, v_j)$, thus proving the thesis. $\square$

It remains to describe how to generate the pruned graph $\bar{\mathcal{G}}_{\epsilon_2}$ in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time directly without explicitly constructing $\mathcal{G}$ which, otherwise, would require quadratic time. This is done by keeping $k = O(\log_{1+\epsilon} \frac{1}{\epsilon})$ windows $W_0, \ldots, W_k$ sliding over the sequence S, one for each possible exponent h such that $F \leq F(1+\epsilon)^h \leq L$. These sliding windows cover potentially different portions of S that start at the same position i but have different ending positions. Armed with these sliding windows, we generate the $(1+\epsilon)$-maximal edges outgoing from any vertex v_i on-the-fly as soon as the shortest path algorithm visits this vertex. Initially, each sliding window W_j starts and ends at position 0. During the execution, every time the shortest path algorithm visits the next vertex v_i, we advance the starting position of each W_j by one position and its ending position until the cost of representing the currently covered portion of S is larger than $F(1+\epsilon)^j$. It is easy to prove that if at the end of these moves the window W_j covers $S[i, r]$, then (v_i, v_r) is the $(1+\epsilon)$-maximal edge outgoing from the vertex v_i for the weight bound $F(1+\epsilon)^j$. Notice that with this approach we generate all the maximal edges of $\bar{\mathcal{G}}_{\epsilon_2}$ by performing a scan of S for each sliding window. Every time we move the starting or the ending position of a window we need to evaluate the cost of representing its covered portion of S. This can be done in constant time with simple arithmetic operations. Thus, it follows that generating the pruned graph requires constant time per edge, hence $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time overall.

We conclude the section by describing how to modify the first-level data structure to support arbitrary partitions: together with the first-level sequence L with the last element of each block, we write a second sequence E which contains the positions of the endpoints of the partition. This sequence can be again represented with Elias-Fano. The NextGEQ operation can be supported as before, while for Access(i) we can find the chunk of i with NextGEQ on E. Both L and E have as many elements as the number of the chunks in the partition.

5. EXPERIMENTAL ANALYSIS

We performed our experiments on the following 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.

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

Table 1: Basic statistics for the test collections

For each document in the collection the body text was extracted using Apache Tika[2], and the words lowercased and stemmed using the Porter2 stemmer; no stopwords were removed. The docIds were assigned according to the lexicographic order of their URLs [20]. Table 1 reports the basic statistics for the two collections.

Indexes tested. We compare three versions of Elias-Fano indexes, namely EF single, EF uniform, and EF ϵ-optimal, that are respectively the original single-partition representation, a partitioned representation with uniform partitions, and a variable-length partitioned representation optimized with the algorithm of Section 4.2. The implementation of the base Elias-Fano sequences is mostly faithful to the original description and source code [23].

To validate Elias-Fano indexes against the more widespread block-based indexes, we tested three more indexes, namely Interpolative, OptPFD, and Varint-G8IU, which are representative of best compression ratio, best compression ratio/processing speed trade-off, and highest speed in the literature [15, 21]. For the last two use the code made available by the authors of [15]. Actually, recent experiments [15] show that there exist methods faster than Varint-G8IU by roughly 50% in sequential decoding. However, they require very large blocks (from 2^{11} to 2^{16}) making block decoding prohibitively expensive if the lists are sparsely accessed.

For these three indexes we encoded the lists in blocks of 128 postings; we keep in a separate array the maximum docId of each block to perform fast skipping through linear search (we experimented with falling back to binary search after a small lookahead, but got worse results). Blocks of postings and blocks of frequencies are interleaved, and the frequencies blocks are decoded lazily as needed. The last block of each list is encoded with variable bytes if it is smaller than 128 postings to avoid any block padding overhead.

All the implementations of posting lists expose the same interface, namely the Access, NextGEQ, and Next operations described in Section 4. The query algorithms are C++ templates specialized for each posting list implementation to avoid the virtual method call overhead, which can be significant for operations that take just a handful of CPU cycles. For the same reason, we made sure that frequent code paths are inlined in the query processing code.

Testing details. All the algorithms were implemented in C++11 and compiled with GCC 4.9 with the highest optimization settings. We do not use any SIMD instructions or special processor features, except for the instructions to count the 1 bits in a word and to find the position of the least significant bit; both are available in modern x86-64 CPUs and exposed by the compiler through portable builtins. Apart from this, all the code is standard C++11. The tests were performed on a machine with 24 Intel Xeon E5-2697 Ivy Bridge cores (48 threads) clocked at 2.70Ghz, with 64GiB RAM, running Linux 3.12.7.

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.

The source code is available at `http://github.com/ot/partitioned_elias_fano/tree/sigir14` for the reader interested in replicating the experiments.

5.1 Space and construction time

Before comparing the spaces obtained with the different indexes, we have to set the parameters needed by the partitioned EF indexes, namely the chunk size for EF uniform and the approximation parameters ϵ_1, ϵ_2 for EF ϵ-optimal.

For the former, we show in Figure 1 the space obtained by EF uniform on Gov2 for different values of the chunk size. The minimum space is obtained at 128, which is also a widely used block size for block-based indexes; in all the following experiments we will use this chunk size.

A little more care has to be put in choosing the parameters ϵ_1 and ϵ_2 since they significantly affect the construction time, as shown in Figure 2. First we fix ϵ_1 at 0, thus posing no upper bound on the chunk costs, and let ϵ_2 vary. Notice that the $(1+\epsilon_2)$ approximation bound is very pessimistic: even setting ϵ_2 as high as 0.5, meaning a potential 50% overhead, the actual overhead is never higher than 1.5%. Hence, we set $\epsilon_2 = 0.3$ to control the construction time.

[2] `http://tika.apache.org/`

	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
EF single	7.66 (+64.7%)	7.53 (+83.4%)	3.14 (+32.4%)	19.63 (+23.1%)	7.46 (+27.7%)	2.44 (+11.0%)
EF uniform	5.17 (+11.2%)	4.63 (+12.9%)	2.58 (+8.4%)	17.78 (+11.5%)	6.58 (+12.6%)	2.39 (+8.8%)
EF ϵ-optimal	4.65	4.10	2.38	15.94	5.85	2.20
Interpolative	4.57 (−1.8%)	4.03 (−1.8%)	2.33 (−1.8%)	14.62 (−8.3%)	5.33 (−8.8%)	2.04 (−7.1%)
OptPFD	5.22 (+12.3%)	4.72 (+15.1%)	2.55 (+7.4%)	17.80 (+11.6%)	6.42 (+9.8%)	2.56 (+16.4%)
Varint-G8IU	14.06 (+202.2%)	10.60 (+158.2%)	8.98 (+278.3%)	39.59 (+148.3%)	10.99 (+88.1%)	8.98 (+308.8%)

Table 2: Overall space in gigabytes, and average bits per docId and frequency

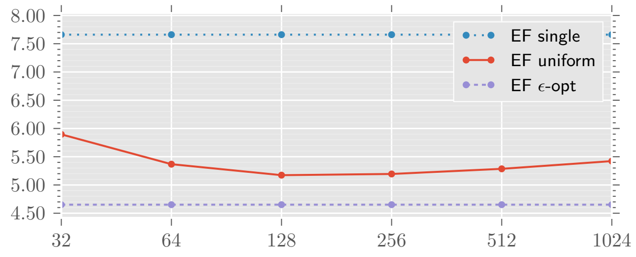
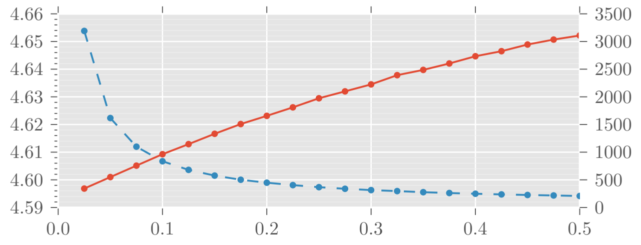
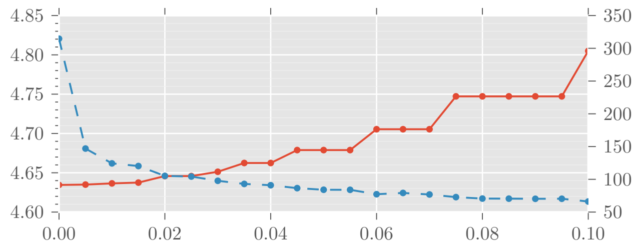


Figure 1: Index size in gigabytes for Gov2 with EF uniform at different chunk sizes

After fixing ϵ_2, we let ϵ_1 vary. Notice the sharp drop in running time without a noticeable increase in space as soon as ϵ_1 is non-zero: it is a direct consequence of the algorithm going from $O(m \log m)$-time to $O(m)$-time. Again, the spread between the worse and best solutions found is smaller than 1%. In the following, we set $\epsilon_1 = 0.03$. We found that with these parameters, the average chunk length on Gov2 for docId sequences is 231 and for frequencies 466. On ClueWeb09 they are respectively 142 and 512. For brevity we omit the plots for ClueWeb09, which are very similar to the ones for Gov2.



(a) $\epsilon_1 = 0$, varying ϵ_2 from 0.025 to 0.5



(b) $\epsilon_2 = 0.3$, varying ϵ_1 from 0 to 0.1

Figure 2: Influence of the parameters ϵ_1 and ϵ_2 on the EF ϵ-optimal indexes for Gov2. Solid line is the overall size in gigabytes (left scale), dashed line is the construction time in minutes (right scale).

Table 2 shows the index space, both as overall size in gigabytes and broken down in bits per integers for docIds and frequencies. Next to each value is shown the relative loss/gain (in percentage) compared to EF ϵ-optimal. The results confirm that partitioning the indexes indeed pays off: compared to EF ϵ-optimal, EF single is 64.7% larger on Gov2 and 23.1% larger on ClueWeb09. The optimization strategy also produces significant savings: EF uniform is about 11% larger on both Gov2 and ClueWeb09, but still significantly smaller than EF single.

Compared to the other indexes, Varint-G8IU is by far the largest, 2.5 to 3 times larger than EF ϵ-optimal; it is particularly inefficient on the frequencies, as it needs at least 8 bits to encode an integer. On the other end of the spectrum, Interpolative confirms its high compression ratio and produces the smallest indexes, but the edge with EF ϵ-optimal is surprisingly small: only 1.8% on Gov2 and 8.3% on ClueWeb09. To conclude the comparison, we observe that OptPFD loses more than 10% w.r.t. EF ϵ-optimal: 12.3% on Gov2 and 11.6% on ClueWeb09. Since, as we will show in the following, EF ϵ-optimal is also faster than OptPFD, this implies that our solution dominates OptPFD on the trade-off curve.

Regarding construction time, for all the indexes except EF ϵ-optimal, Gov2 can be processed on a single thread in 8 to 10 minutes, while ClueWeb09 in 24 to 30 minutes; in both cases the construction is essentially I/O-bound. For EF ϵ-optimal, instead, the algorithm that computes the optimal partition, despite linear-time, has a noticeable CPU cost, raising the time for Gov2 to 95 minutes and for ClueWeb09 to 284 minutes. However, since the encoding of each list can be performed independently, using all the 24 cores of the test machine makes EF ϵ-optimal construction I/O-bound as well. Parallelization does not improve the construction time of the other encoders.

5.2 Query processing

To evaluate the speed of query processing we randomly sampled two sets of 1000 queries respectively from TREC 2005 and 2006 Efficiency Track topics, drawing only queries whose terms are all in the collection dictionary. The two dataset have quite different statistics, as will be apparent in the results.

In his experimental analysis, Vigna [23] provides some examples showing that Elias-Fano indexes are particularly efficient in conjunctive queries that have sparse results. To make the analysis more systematic, we define as *selective* any query such that the fraction of the documents that contain all its terms over those that contain at least one of them is small (we set this threshold to 0.5%). For both datasets, selective queries among TREC 2005 queries are at least 58%, and among TREC 2006 queries at least 78%, hence making up the most part of the samples.

The query times were measured by running each query set 3 times, and averaging the results. All the times are reported in milliseconds. In the timings tables, next to each timing is reported in parentheses the relative percentage against EF ϵ-optimal. Not very surprisingly, Interpolative is always 50% to 500% slower than the others, and Varint-G8IU is 10% to 40% faster, so for the sake of brevity we will focus the following analysis on the Elias-Fano indexes and OptPFD.

Boolean queries. We first analyze the basic disjunctive (OR) and conjunctive (AND) queries. Note that these queries do not need the term frequencies, so only the docId lists are accessed. For these types of operations, we measure the time needed to *count* the number of results matching the query.

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	80.7 (+8%)	175.0 (+10%)	261.0 (+0%)	444.0 (−2%)
EF uniform	72.1 (−3%)	154.0 (−3%)	254.0 (−3%)	435.0 (−4%)
EF ϵ-optimal	74.5	159.0	261.0	451.0
Interpolative	121.0 (+62%)	257.0 (+62%)	399.0 (+53%)	680.0 (+51%)
OptPFD	69.5 (−7%)	148.0 (−7%)	235.0 (−10%)	398.0 (−12%)
Varint-G8IU	67.4 (−10%)	143.0 (−10%)	222.0 (−15%)	375.0 (−17%)

Table 3: Times for OR queries

Times for OR queries are reported in Table 3. Unsurprisingly, as OR needs to scan the whole lists, block-based indexes perform better than Elias-Fano indexes, since they are optimized for raw decoding speed. However, the edge is not as high as one could expect, ranging from 7% to 17%. In sequential decoding Varint-G8IU can be even double as fast as OptPFD [15], however in this task it is not even 10% faster. The reason can be most likely traced back to branch misprediction penalties: the cost of decoding an integer is in the order of 2-5 CPU cycles; at each decoded docId, the CPU has to decide whether it is equal to the current candidate docId, or if it must be considered as a candidate for the next docId. The resulting jump is basically unpredictable, and the branch misprediction penalty on modern CPUs can be as high as 10-20 cycles (specifically at least 15 for the CPU we used [13]), thus becoming the main bottleneck of query processing.

Among Elias-Fano indexes, EF ϵ-optimal and EF uniform are either negligibly slower or slightly faster than EF single; we believe that the additional complexity is balanced by the higher memory throughput caused by the smaller sizes.

Table 4 reports the times for AND queries. Again, EF ϵ-optimal is competitive with EF single, except for selective queries where the overhead is slightly higher, touching 14%. EF uniform is slightly slower, which is likely caused by the smaller chunk sizes compared to EF ϵ-optimal. This will also be the case in the other queries. Compared to OptPFD, EF ϵ-optimal is 14% to 26% faster in all cases on general queries. The gap becomes even higher for selective queries, ranging from 34% to 40%, confirming the observations made in [23].

Ranked queries. In order to analyze the impact of accessing the frequencies in query time, we measured the time required to find the top-10 results for AND and WAND [3] queries

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	2.1 (+10%)	4.7 (+1%)	13.6 (−5%)	15.8 (−9%)
EF uniform	2.1 (+9%)	5.1 (+10%)	15.5 (+8%)	18.9 (+9%)
EF ϵ-optimal	1.9	4.6	14.3	17.4
Interpolative	7.5 (+291%)	20.4 (+343%)	55.7 (+289%)	76.5 (+341%)
OptPFD	2.2 (+14%)	5.7 (+24%)	16.6 (+16%)	21.9 (+26%)
Varint-G8IU	1.5 (−20%)	4.0 (−13%)	11.1 (−23%)	14.8 (−15%)

(a) All queries

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	1.1 (−11%)	2.5 (−9%)	9.2 (−14%)	11.1 (−13%)
EF uniform	1.3 (+8%)	3.0 (+11%)	11.3 (+6%)	13.0 (+2%)
EF ϵ-optimal	1.2	2.7	10.7	12.7
Interpolative	6.0 (+399%)	14.3 (+430%)	49.9 (+368%)	61.0 (+379%)
OptPFD	1.6 (+34%)	3.8 (+40%)	13.0 (+22%)	17.0 (+33%)
Varint-G8IU	1.1 (−11%)	2.5 (−6%)	8.8 (−18%)	11.3 (−12%)

(b) Selective queries

Table 4: Times for AND queries

using BM25 [18] scoring.

Results for scored AND, reported in Table 5, and for WAND, reported in Table 6, are actually very similar. As before, we note that the overhead of EF ϵ-optimal against EF single is small, ranging between 2% and 13% for all queries, and between 8% and 18% for selective queries. Also as before, EF uniform is about 10% slower than EF ϵ-optimal; this is likely caused by the optimal partitioning algorithm placing chunk endpoints around dense clusters of docId, hence making the query algorithms crossing fewer chunk boundaries. Compared to OptPFD, EF ϵ-optimal is slightly slower on Gov2 and slightly faster on the larger ClueWeb09 on all queries. On selective queries, however, it is never slower, and up to 23% faster, thus providing further evidence that Elias-Fano indexes benefit significantly from selectiveness, even for non-conjunctive queries.

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	4.0 (−2%)	8.4 (−7%)	22.8 (−9%)	24.6 (−13%)
EF uniform	4.4 (+7%)	9.8 (+8%)	27.3 (+9%)	31.0 (+9%)
EF ϵ-optimal	4.1	9.0	25.1	28.4
Interpolative	14.1 (+242%)	38.6 (+327%)	99.1 (+295%)	132.0 (+365%)
OptPFD	3.9 (−7%)	9.2 (+1%)	25.8 (+3%)	31.6 (+11%)
Varint-G8IU	2.6 (−38%)	5.5 (−39%)	15.8 (−37%)	18.0 (−37%)

(a) All queries

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	1.7 (−16%)	4.1 (−14%)	12.5 (−18%)	16.2 (−17%)
EF uniform	2.1 (+7%)	5.2 (+9%)	16.3 (+7%)	21.1 (+8%)
EF ϵ-optimal	2.0	4.8	15.2	19.4
Interpolative	10.2 (+412%)	25.9 (+439%)	80.7 (+430%)	99.7 (+412%)
OptPFD	2.3 (+18%)	5.7 (+18%)	18.8 (+23%)	23.1 (+19%)
Varint-G8IU	1.4 (−32%)	3.3 (−32%)	10.6 (−30%)	13.6 (−30%)

(b) Selective queries

Table 5: Times for AND top-10 BM25 queries

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	8.8 (−4%)	15.8 (−7%)	31.5 (−7%)	41.2 (−13%)
EF uniform	9.7 (+6%)	18.2 (+7%)	36.9 (+9%)	51.2 (+8%)
EF ϵ-optimal	9.2	17.1	34.0	47.4
Interpolative	28.0 (+203%)	62.6 (+267%)	123.0 (+262%)	200.0 (+322%)
OptPFD	8.7 (−6%)	16.7 (−2%)	35.6 (+5%)	52.1 (+10%)
Varint-G8IU	6.1 (−34%)	11.1 (−35%)	24.0 (−30%)	34.3 (−28%)

(a) All queries

	Gov2		ClueWeb09	
	TREC 05	TREC 06	TREC 05	TREC 06
EF single	8.5 (−8%)	12.2 (−9%)	21.0 (−19%)	34.2 (−15%)
EF uniform	9.8 (+6%)	14.4 (+7%)	27.3 (+6%)	43.0 (+7%)
EF ϵ-optimal	9.2	13.5	25.8	40.1
Interpolative	33.8 (+265%)	54.7 (+307%)	115.0 (+345%)	179.0 (+346%)
OptPFD	9.3 (+0%)	14.1 (+4%)	29.9 (+16%)	45.7 (+14%)
Varint-G8IU	6.3 (−32%)	9.2 (−32%)	19.2 (−26%)	30.0 (−25%)

(b) Selective queries

Table 6: Times for WAND top-10 BM25 queries

6. CONCLUSION AND FUTURE WORK

We introduced two new index representations, EF uniform and EF ϵ-optimal, which significantly improve the compression ratio of Elias-Fano indexes at a small query performance cost. EF ϵ-optimal is always convenient except when construction time is an issue, in which case EF uniform offers a $\approx 12\%$ worse compression ratio without this additional construction overhead. Furthermore, EF ϵ-optimal offers a better space-time trade-off than the state-of-the-art OptPFD.

Future work will focus on making partitioned Elias-Fano indexes even faster; in particular it may be worth exploring the different space-time trade-offs that can be obtained by varying the average chunk size. It would also be interesting to devise faster algorithms to compute optimal partitions preserving the same approximation guarantees.

Acknowledgements

This work was supported by Midas EU Project (318786), by MIUR of Italy project PRIN ARS Technomedia 2012 and by eCloud EU Project (325091).

We would like to thank Sebastiano Vigna for sharing with us his C++ implementation of Elias-Fano indexes, which we used to validate our implementation.

7. REFERENCES

[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1), 2005.

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2):131–147, 2010.

[3] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[4] A. Buchsbaum, G. Fowler, and R. Giancarlo. Improving table compression with combinatorial optimization. *Journal of the ACM*, 50(6):825–851, 2003.

[5] S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In *CIKM*, 2007.

[6] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information retrieval: implementing and evaluating search engines.* MIT Press, Cambridge, Mass., 2010.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2009.

[8] M. Curtiss and et al. Unicorn: A system for searching the social graph. *VLDB*, 6(11):1150–1161, Aug. 2013.

[9] R. Delbru, S. Campinas, and G. Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *J. Web Sem.*, 10:33–58, 2012.

[10] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[11] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA*, 1971.

[12] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, 61(1):51–74, 2011.

[13] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/microarchitecture.pdf`.

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

[15] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice & Experience*, 2013.

[16] C. D. Manning, P. Raghavan, and H. Schülze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[17] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1), 2000.

[18] S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146, 1976.

[19] D. Salomon. *Variable-length Codes for Data Compression.* Springer, 2007.

[20] F. Silvestri. Sorting out the document identifier assignment problem. In *ECIR*, pages 101–112, 2007.

[21] F. Silvestri and R. Venturini. VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, pages 1219–1228, 2010.

[22] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[23] S. Vigna. Quasi-succinct indices. In *WSDM*, 2013.

[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., 1999.

[25] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *SIGIR*, pages 147–154, 2009.

[26] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[27] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.