Elias-Fano indexes

Jouni Petteri Moilanen 011120474

April 27, 2021

Contents

1 Introduction

This report for **Seminar on Scalability Challenges on Search Engines**, spring 2021, describes the Elias-Fano representation scheme and its modern variants. The reader is assumed to have the basic knowledge of information retrieval concepts and terms, which will not be elaborated in the text.

Elias-Fano encoding/coding/representation scheme is a way to encode a monotonically increasing sequence of integers with an upper bound with close to optimal (lower bound set by information theory) space requirements. This may not seem very important at first glance; but as we will see below, an efficient solution to this problem is crucial to modern search engines.

The name of the scheme stems from the fact it was proposed independently by Peter Elias and Robert Mario Fano in the 1970s [12]. Elias-Fano coding should not be confused with Shannon-Fano-Elias coding, a probabilistic coding scheme, or Elias delta/gamma/omega codes which are prefix codes for positive integers [11].

1.1 Inverted indexes

In a collection of text documents (a web repository for instance), each document can be considered a multiset of terms (multiset meaning a term can occur multiple times). The set of all terms in the collection is called the lexicon. The **inverted index** stores a list of document identifiers (a **postings list**) for each term. This makes it possible to resolve queries like "return all documents that contain terms x and y" efficiently. In real-life information retrieval systems, an inverted index contains a lot of other information, e.g. term/doc frequencies, positions where the term occurs in a document, etc. For this report, we concentrate on the simplest form of the inverted index containing only the document IDs.

Inverted indexes are the foundation of modern search engines. Nowadays these indexes are huge; according to a Google Search page, its search index is "well over 100,000,000 gigabytes in size" [3]. The size of indexes makes their effective and efficient compression critical for searches, allowing for both keeping a bigger portion of the indexes in memory and faster disk searches.

Pibiri and Venturini [10] list some other uses for inverted indexes:

- A postings list can represent the personal IDs of a user's friends in a social network.
- Some relational database systems store increasing sequences of row identifiers to speed up frequently executed queries.
- A list of elements with the same hash code (as in a hash collision) in a key-value storage architecture can be considered an inverted index.

1.2 The main computational problems

In most IR systems the (numeric) document identifiers, postings, are stored in ascending order. The central problem for compressing posting lists can thus be expressed as a question:

Given a sequence of monotonically increasing non-negative integers, how can we represent it as a bit vector where each integer is self-limited, in as few bits as possible? [6].

For efficient query resolution, the code must be also very quick to decompress. The query resolution - aside from decoding - at its most basic level boils down to intersecting two lists. This means that the search engine must be able to

- enumerate a list,
- pick the i-th integer, and
- skip to the first integer $\geq x$

as quickly as possible [13]. The latter two are often called, respectively, **Access(i)** and **NextGEQ(x)** in the literature.

1.3 Different solutions for different times

Pibiri and Venturiani [10] present a following timeline for the evolution of compression techniques:

Table 1. Timeline of Techniques

1949	Shannon-Fano [32, 93]
1952	Huffman [43]
1963	Arithmetic [1][1]
1966	Golomb [40]
1971	Elias-Fano [30, 33]; Rice [87]
1972	Variable-Byte and Nibble [101]
1975	Gamma and Delta [31]
1978	Exponential Golomb [99]
1985	Fibonacci-based [6, 37]
1986	Hierarchical bit-vectors [35]
1988	Based on Front Coding [16]
1996	Interpolative [65, 66]
1998	Frame-of-Reference (For) [39]; modified Rice [2]
2003	SC-dense [11]
2004	Zeta [8, 9]
2005	Simple-9, Relative-10, and Carryover-12 [3]; RBUC [60]
2006	PForDelta [114]; BASC [61]
2008	Simple-16 [112]; Tournament [100]
2009	ANS [27]; Varint-GB [23]; Opt-PFor [111]
2010	Simple8b [4]; VSE [96]; SIMD-Gamma [91]
2011	Varint-G8IU [97]; Parallel-PFor [5]
2013	DAC [12]; Quasi-Succinct [107]
2014	Partitioned Elias-Fano [73]; QMX [103]; Roaring [15, 51, 53]
2015	BP32, SIMD-BP128, and SIMD-FastPFor [50]; Masked-VByte [84]
2017	Clustered Elias-Fano [80]
2018	Stream-VByte [52]; ANS-based [63, 64]; Opt-VByte [83]; SIMD-Delta [104]; general-purpose compression libraries [77]
2019	DINT [79]; Slicing [78]

Up to the second decade of the new millennium, the main approach was to to represent the sequence as **gaps**: $x_0, x_1 - x_0, x_2 - x_1, \ldots, x_n - x_{n-1}$ in some variable length code. Among these codes were Elias gamma/delta, Variable Byte, Varint-G8IU, Simple-9/16, PForDelta (PFD), OptPFD, Binary Interpolative Coding [6].

For these kinds of codes to work well, it's good to have the gaps small and well distributed. When this was not the case naturally, it could be induced by clustering document IDs when building the index. Skip tables were also employed to enable jumps to the middle of the sequence [13]; otherwise, rank selection and predecessor search would be inefficient. Like

Sebastiano Vigna remarked the year 2013: "In retrospective, it looks a little bit contrived, doesn't it?" [13] Elias-Fano codes represented a completely different approach.

Elias-Fano encoding was proposed independently by Robert Fano in 1971 [2] and Peter Elias in 1974 [1]. It did not get much practical interest until Vigna represented his quasi-succinct indexes in 2013 [12]. After that, Elias-Fano and its modern variants have risen to one of the central techniques of index compression. These codes are the focus of this report.

The second section of the text describes the Elias-Fano representation and a few of its modern variations, concentrating on Vigna's quasi-succinct indexes and Partitioned Elias-Fano (see timeline above), as they are the foundation of the most recent Elias-Fano-based techniques. Some applications of Elias-Fano are represented in the third section and the fourth offers a brief overview of current index compression techniques.

2 Elias-Fano and its modern variants

2.1 Elias-Fano encoding

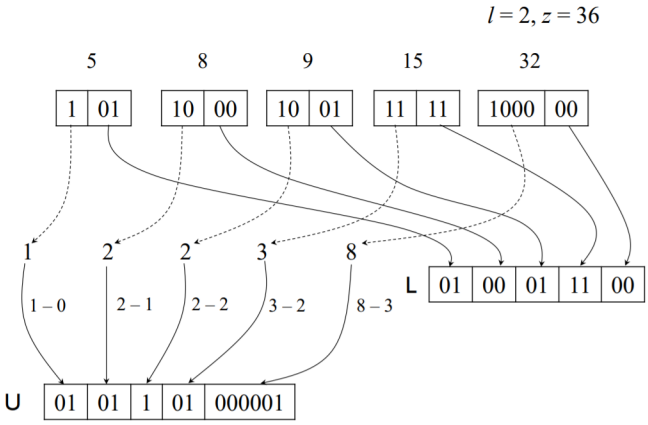
Elias-Fano representation for an array of n strictly increasing non-negative integers with an upper bound of z, i.e.

$$0 \leq x_0 < x_1 < x_2 \cdots < x_{n-2} < x_{n-1} \leq z$$

is represented in the following fashion:

- The lower (rightmost) $l = \max(0, \lfloor \log_2(z/n) \rfloor)$ bits of each integer are stored in a *lower-bit array* contiguosly and in the original order.
- The rest of the bits are stored in a *upper-bit array* as unary-coded gaps.

For instance, if we have integers $5, 8, 9, 15, 32$ with an upper bound of 36, we get the following Elias-Fano representation ($l = 2$):



(Above diagram from Information Retrieval course lecture 2 slides by Simon Puglisi)

Another way to think about the representation (resulting in the same output), discussed by Ottaviano and Venturini [5] is to consider the upper bits of the integers grouped into buckets according to their bits. In the example above, there are 2^4 buckets ranging from 0000 to 1111. The cardinalities of the buckets are written into the upper-bit array in negated unary code followed by a 0, so for empty buckets we write just 0.

So for our example we get following representation for said buckets.

- 0000: 0 - the bucket is empty
- 0001: 10 - high bits of 5, cardinality 1
- 0010: 110 - 8 and 9 fall into this bucket
- 0011: 10 - 15
- 0100-0111 are empty: 0000
- 1000: 10 - 32
- The empty buckets after the last integer are not encoded.

Put together we get 0 10 110 10 0000 10, which - to the author's relief - is identical with the example diagram's result.

It can be shown that each integer takes up at most $2 + \lceil \log_2(u/n) \rceil$ bits. This means the space taken by the whole sequence $\leq 2n + n\lceil \log_2(u/n) \rceil$ bits. In his original paper Elias proves that this is very close (less than $n/2$ bits) away from the information theoretical lower bound of $\lceil \log_2 \binom{u+n}{n} \rceil$. Thus while not strictly succinct, Elias-Fano can be called a **quasi-succinct** representation [12].

2.2 Quasi-succinct indexes (2013)

In his paper, Vigna [12] describes a new version of Elias-Fano that is specifically tailored to the needs of search engines. For the discussion that follows, we list Vigna's main definitions:

- n is the number of the documents. Each document is referenced by a **document pointer** (≥ 0). The **count** is the number of times a term appears in a document, while the **frequency** is the number of documents in which a term occurs (i.e. count > 0). The **occurrency** is the sum of the counts in the whole collection.
- In addition to the posting list, with each document we associate also the non-zero counts of terms in the document, and monotonically increasing lists of positions at which terms appear in the document.

After describing Elias-Fano (as we did in the previous section), he notes that retrieving number x_i can be done by performing i unary reads in the upper-bits array, getting to the position p. The value of upper bits is exactly $p - i$. The lower bits are at position $l \cdot i$ of the lower bit array (l being the number of lower bits stored for every number) and can be retrieved by random access.

If we assume a "helper" element $x_{-1} = 0$, a non-negative monotone sequence $x_0, \ldots, x_{n-1}$ can be expressed as sequence of gaps: $a_0 = x_1 - x_0, \ldots, a_{n-1} = x_{n-1} - x_{n-2}$. There is a bijective relationship between sequence $x_0, \ldots, x_{n-1}$ bound by u and the list of **prefix sums** of the corresponding gap sequence whose sum is bound by u. This means Elias-Fano can be used also to encode generic lists of integers.

Elias-Fano coding has a few useful properties, namely

- The distribution of the document gaps is irrelevant.
- In particular, reordering documents e.g to speed up retrieval will not affect index size.
- Scanning the index requires only a single unary read plus a few shifts.
- Search and skip can be performed on single unary (upper bits) array that consists of n ones at most $2n$ zeroes. This lends itself to some powerful ad hoc techniques.

By storing forward and skip pointers to upper bits it is possible to perform both Access and NextGEQ operations very quickly. Vigna also notes that if the represented sequence is strictly monotone, it is possible to compress further by representing x_i as $x_i - 1$ with an upper bound of $u - n$. The skip algorithm used in Vigna's proposal does not work with this format but for some components of the index, this will not matter.

For each term, document pointers are stored using the standard Elias-Fano to enable storing skip pointers, as skipping is useful in Boolean and phrasal queries, while random access is not usually necessary. The upper bound is $n - 1$, and the number of elements is the frequency. Counts of a term - being strictly positive - can be stored as their prefix sums.

Storing the positions is a more complicated matter. For the ith document pointer in the posting list of the term t with a count of c, the positions list

$$p_0, p_1, \ldots, p_{c-1}$$

is presented as

$$p_0 + 1, p_1 - p_0, \ldots, p_{c-1} - p_{c-2}$$

The representations of positions for all documents are concatenated and stored using the representation for strictly positive numbers.

In Vigna's representation, storing a bit array (bitstream) is done as follows:

- Representation metadata is stored first in a self-limiting format.
- After that, pointers, lower bits, and upper bits. The logic behind the ordering is that the length of the upper bits is the only unknown quantity. The lower bits will be at position sw after the metadata (s is the number of pointers and w their width), and the upper bits at position $sw + nl$.

Each component of the index (document pointers, counts, positions) is stored in a separate bitstream. As the representation provides constant-time access to each element, there is no need to access data not needed for an operation (e.g., counts and positions for a Boolean query). The details of the implementation (longword addressing, reading unary codes, and caching), though essential for good performance of quasi-succinct indexes, are outside the scope of this representation. For those, see section 9. of Vigna's paper.

In conclusion, Vigna states that in the experiments performed the new index provided, in general, improved performance over current methods in terms of compression, speed, or both. One drawback he mentions is that the frequencies and occurrencies of terms and the upper bound must be known before the index is built. In practice, this means caching all posting lists in some other format before building the index. As the large indices were built incrementally at the time, he does not consider this a huge problem: index segments and the relevant statistics could be compressed using gap encoding and then merged using the quasi-succinct format.

2.3 Partitioned Elias-Fano indexes (2014)

Partitioned Elias-Fano Indexes were proposed by Ottaviano and Venturini in 2014 to improve the scheme's compression rates by exploiting clustering [5] i.e the presence of long subsequences of close values. The original representation doesn't take into account any local characteristics of the sequence; the give an extreme (toy) example: a sequence of m integers $0, 1, 2, \ldots, m-2, u$ takes $2+\lceil \log_2(u/m) \rceil$ bits per element, even though the length of the run and value of u are sufficient to describe it. For the following discussion, it's also noteworthy that u/m is the average distance between consecutive integers.

The presence of regions of close values is typical to posting lists. Consider for instance a term present only in few domains. If the doc Ids are assigned by URL order, then the posting list of the term is formed mostly of clusters of integers corresponding to those domains. Since the elements within each cluster are close to each other, the average distance within clusters is much smaller than the global average distance. This is the main motivation for Partitioned Elias-Fano (PEF from now on), a two-level representation of monotone sequences.

The main idea is that the sequence S is partitioned into $j = m/b$ chunks of b consecutive integers each. The last elements of chunks are juxtaposed, resulting in a sequence $L = [S_{b-1}, \ldots, S_{m-1}]$ and coded with Elias-Fano. The chunks are also coded with EF producing the second level of the representation. The main advantage of this is that the elements of chunk j can be rewritten in a smaller universe of size $u_j = L_j - L_{j-1} - 1$ resulting in a EF coding of $2 + \lceil \log_2(u_j/b) \rceil$ bits per element. Since the average distance u_j/b within the chunk is (hopefully) much smaller than the global average, the space occupancy of the encoded chunk is also much smaller than if it was encoded as part of the entire sequence. Some of the gains are of course lost in storing m/b integers from a universe of size u at the first level.

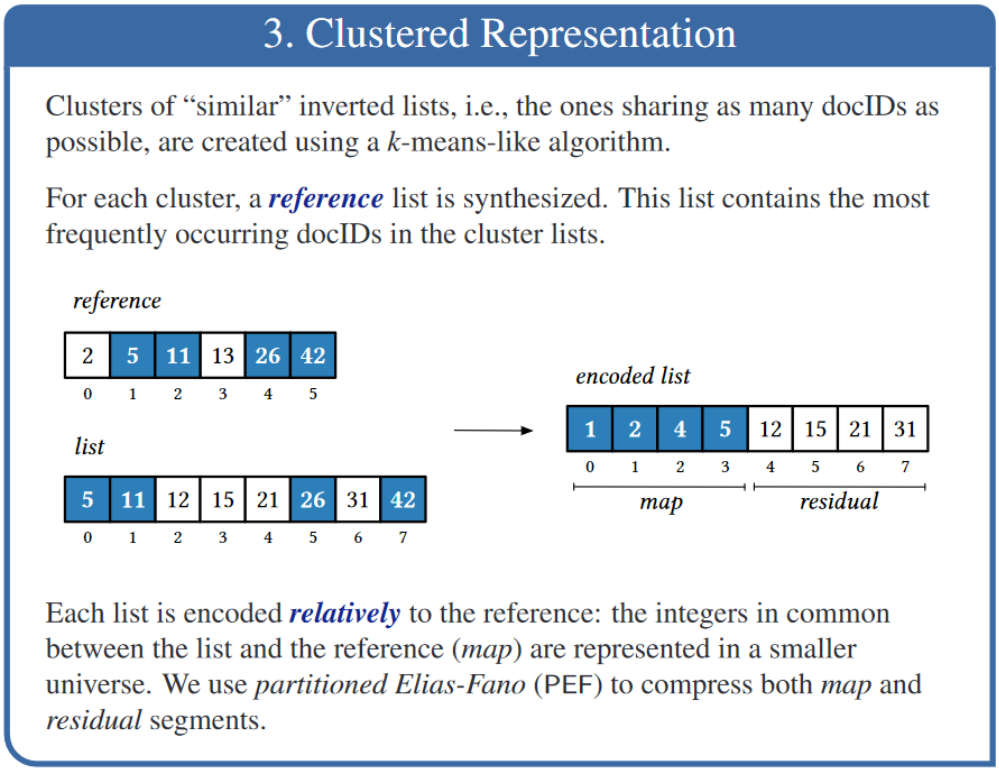
Partitioning can be done in two different ways: uniform chunking uses fixed partition size, while another approach is to choose the endpoints by optimizing the overall space required. The authors call these two representations **EF uniform** and **EF ϵ-optimal**, respectively. The name of the latter stems from a programming algorithm introduced in the article that for any $\epsilon \in (0, 1)$, finds a partition of size $\leq (1 + \epsilon)$ times the optimal size. This is desirable because while an optimal partitioning can be computed in $\Theta(n^2)$ time, for larger inputs this is infeasible in practice. The authors' solution identifies in $O(m \log_{1+\epsilon} \frac{1}{\epsilon})$ time a partition within the desired size limit. The time complexity is linear once ϵ is set.

The two representations were extensively tested against the original representation and the more widespread block-based indexes, namely Interpolative, OptPFD, and Varint-G8IU, which represented, respectively, the best compression ratio, best space/speed trade-off, and the highest speed at the time. EF uniform and EF ϵ-optimal both improved the compression ratio of original EF with not a large query cost. The latter also beat OptPDF at space-time trade-off.

2.4 Clustered Elias-Fano (2017)

Clustered Elias-Fano proposed by Pibiri and Venturini [8] builds on partitioned E-F and the fact that inverted lists are **redundant**, i.e. they share many doc IDs since - naturally - many documents share terms. Previous techniques, which represented each posting list individually, could not exploit this redundancy.

The most succinct way to represent the basic idea is to borrow a part of the authors' representation slide:



(Source: "Clustered Elias-Fano Indexes" ACM Transactions on Information Systems (TOIS), 2017)

The general scheme allows for any or even different encodings to be applied on the reference, map, and residual lists. The authors' implementation uses PEF for all three. Clustered EF poses two computational problems:

1. clustering the posting lists to maximize the number of integers shared by the lists in a cluster,
2. selecting integers to put into the reference list so that the space occupancy of the cluster is minimized.

Formally the problem can be stated as follows:

> *Let L be a set of posting lists and $CPEF(L_i, R_i)$ be the cost in bits of encoding the posting lists in its subset (cluster) L_i using reference list R_i. Determine the partition of L in c clusters $L_1, \ldots, L_c$, each non-empty, and the choice of $R_i, |R_i| > 0$ for each cluster, such that*
>
> $$\sum_{i=1}^{c} CPEF(L_i, R_i)$$
>
> *is minimum.*

The clustering part is performed by a k-means-based ad hoc algorithm. The choosing the reference list part of the problem is NP-hard, as the writers demonstrate. They propose two distinct heuristics to solve it: selecting postings by frequency within their cluster and selecting them by their contribution to the encoding space reduction. There are various performance trade-offs regarding the size of the reference list and the choice of the heuristic. For full details see the original paper [8].

In the experiments the authors describe, Clustered Elias-Fano proved highly competitive against BIC, PEF, and Varint-G8IU. Like in part 2.3, the said schemes were chosen to represent the best compression ratio, best space/speed trade-off, and the highest speed at the time. (Note how PEF has replaced OptPDF in the trade-off category.) A more detailed and reader-friendly - compared to the original paper - description of the experimental set-up and the results can be found in Pibiri's 2018 Ph.D. dissertation [7]. The source code is also available at GitHub.

3 Some applications of Elias-Fano

This section of the paper gives some examples of applications of Elias-Fano coding in loosely chronological order.

3.1 Facebook's Graph Search (2012-13)

In a seminar presentation in September 2013 [13], Vigna related how in the previous year he had visited Facebook. While talking with Mike Curtiss, who worked with their new feature Graph Search, Vigna suggested he read the preprint of his paper on quasi-succinct indexes.

In January 2013 Curtiss emailed Vigna, telling him that they had implemented an open-source C++ version of quasi-succinct representation and were using it in production "because it is faster than any other approaches". These included variants of PForDelta and Google's GroupVarint. Elias-Fano coding had made its first inroad into search engines.

Though the service was mostly discontinued in 2019, Facebook's Folly Open Source Library still contains their implementation of Elias-Fano at GitHub.

3.2 Succinct Data Structure Library (2013-present)

Succinct Data Structure Library (SDSL) is an open-source C++11 library implementing succinct data structures which can represent an object (such as a bit vector or a tree) in space close to the information-theoretic lower bound. SDSL uses Elias-Fano for compressing sparse bit vectors by "representing the positions of 1 by the Elias-Fano representation for non-decreasing sequences" (quote from the source code comments, sd_vector.hpp to be precise). The source code and the most readable (in the author's opinion) documentation available for SDSL can be found at GitHub.

3.3 Elias-Fano tries (2018)

N-gram models are a central technique in natural language processing (for reasons thereof and an explanation of N-grams, see for example [4], Chapter 3). One of the fundamental problems with large and sparse n-gram data is compressing the n-gram strings and their associated data (the index, if you will) without compromising the retrieval speed too much [7].

One of the many applications of N-grams is **query autocompletion** in search engines. It is implemented by reporting the top-k most common n-grams following the words typed by the user. This is done by traversing a data structure that stores the n-grams of previous user searches. The volume and response time requirements of search engines make it necessary that the autocompletion (which includes decoding parts of the index) is performed very, very fast [7].

A **trie** is a data structure devised for indexing and searching string dictionaries. It's one of the techniques used to store n-grams. In his 2018 Ph.D. paper [7], Pibiri presents the **Elias-Fano trie**, a compressed trie structure based on Elias-Fano. For full details, see sections 2.4 and 8.2 of his dissertation.

4 The current state of index compression

Now we'll take a brief look at the current state of techniques of index compression. In their two articles "Inverted Index Compression" (2019) [9] and "Techniques for Inverted Index Compression" (2021) [10], Pibiri and Venturini offer quite a thorough survey of different approaches and their performances.

The codes discussed can be categorized by (at least) three different criteria:

- Whether they compress the integers individually, individual lists, or the whole index together.
- Whether they are bit-aligned (processing the data bit by bit) or byte/word-aligned (hopefully self-explanatory).
- Where they land on the space/speed trade-off curve.

In the paper, 12 techniques are evaluated performance-wise in terms of compression effectiveness, sequential decoding speed, and time spent on boolean AND/OR queries. For the full list and the details of the methods, we refer the reader to the original paper. Some remarks on the results:

- Experiments were performed on Gov2, ClueWeb09, and CCNews TREC test collections. From the original collections, all lists larger than 4,096 were retained, covering over 90 % of the original data. The test queries were selected from the TREC 2005 and TREC 2006 Efficiency Track topics, retaining all queries whose terms are in the lexicon of the test collection.
- The compression effectiveness expressed as total gibibytes (GiB) and the bit-per-integer rate was measured on all three datasets. Across all of them, the most effective method was BIC with PEF a close second.
- At sequential decoding, Roaring and Slicing were the fastest methods, using 0.5 to 0.7 ns/integer. Opt-VByte, QMX, and PEF were next (0.7 to 1.3 ns on average). An additional advantage that the authors attribute to BIC, PEF, Roaring, and Slicing is that they don't require a prefix-sum computation for encoding. They state that the cost of computing the prefix-sum of the gaps is 0.5 ns per integer, which may in some cases be a more important factor than the decoding cost.
- Most of the techniques - PEF included - provided similar query efficiency/effectiveness trade-offs, except for BIC, Elias delta, and Rice, which were the slowest across the board. Roaring and Slicing outperformed all the others in terms of efficiency by a wide margin, because they exploit "the intrinsic parallelism of inexpensive bitwise instructions over 64-bit words".

Main results of the experiments in numbers and figures [10]

Table 11. Space Effectiveness in Total GiB and Bits per Integer, and Nanoseconds per Decoded Integer

Method	Gov2			ClueWeb09			CCNews		
	GiB	Bits/int	ns/int	GiB	Bits/int	ns/int	GiB	bits/int	ns/int
VByte	5.46	8.81	0.96	15.92	9.20	1.09	21.29	9.29	1.03
Opt-VByte	2.41	3.89	0.73	9.89	5.72	0.92	14.73	6.42	0.72
BIC	1.82	2.94	5.06	7.66	4.43	6.31	12.02	5.24	6.97
δ	2.32	3.74	3.56	8.95	5.17	3.72	14.58	6.36	3.85
Rice	2.53	4.08	2.92	9.18	5.31	3.25	13.34	5.82	3.32
PEF	1.93	3.12	0.76	8.63	4.99	1.10	12.50	5.45	1.31
DINT	2.19	3.53	1.13	9.26	5.35	1.56	14.76	6.44	1.65
Opt-PFor	2.25	3.63	1.38	9.45	5.46	1.79	13.92	6.07	1.53
Simple16	2.59	4.19	1.53	10.13	5.85	1.87	14.68	6.41	1.89
QMX	3.17	5.12	0.80	12.60	7.29	0.87	16.96	7.40	0.84
Roaring	4.11	6.63	0.50	16.92	9.78	0.71	21.75	9.49	0.61
Slicing	2.67	4.31	0.53	12.21	7.06	0.68	17.83	7.78	0.69

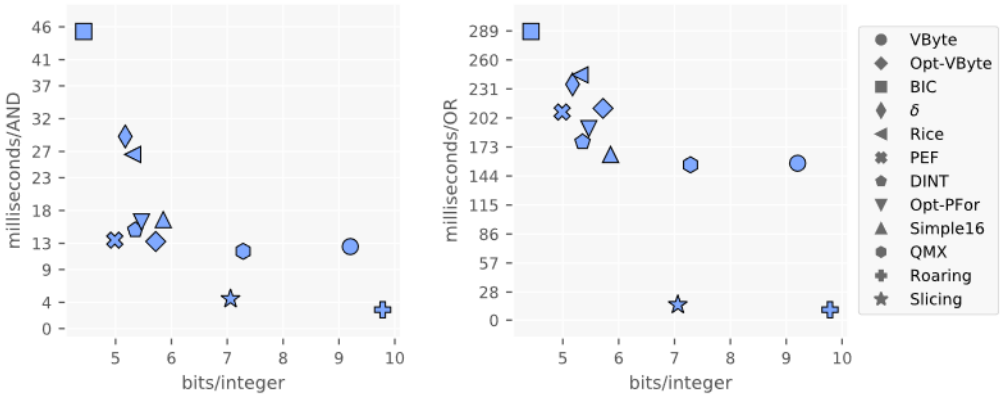


Fig. 7. Space/time trade-off curves for the ClueWeb09 dataset.

While we can see that Roaring and Slicing are in their own class in query speed, PEF holds its own in other respects, occupying a comfortable niche at the curve bend. It seems to be highly competitive "on the market" if the query efficiency is not the only priority.

5 Conclusions

As we saw in the previous section, there are quite a few competing techniques for index compression, none of which can be said to dominate. The "best" solution needs to be determined case by case, based on the application requirements and the data distribution. Elias-Fano coding, especially Partitioned Elias-Fano, seems still to be a very competitive alternative for the problem, performing well on most of the metrics described above.

Of course, it may very well be that current and future developments in computer architectures will take the research into whole new directions and render most of the schemes - Elias-Fano among them - obsolete. There were some indications of that in the query resolution part of the evaluation in the previous section. Still, at the moment, Elias-Fano variants are techniques worth knowing.

References

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

[2] R. Fano. On the number of bits required to implement an associative memory. http://csg.csail.mit.edu/pubs/memos/Memo-61/Memo-61.pdf, 1971.

[3] Google. *How Search organizes information.* https://www.google.com/search/howsearchworks/crawling-indexing/ (accessed March, 2021).

[4] D. Jurafsky. *Speech and language processing : an introduction to natural language processing, computational linguistics and speech recognition.* Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, N.J, 2nd ed. edition, 2009.

[5] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on research & development in information retrieval*, SIGIR '14, pages 273–282. ACM, 2014.

[6] G. E. Pibiri. Elias-Fano Encoding. http://pages.di.unipi.it/pibiri/slides/seminar_ef.pdf, 2016.

[7] G. E. Pibiri. *Space and Time-Efficient Data Structures for Massive Datasets.* PhD thesis, University of Pisa, 2018.

[8] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Transactions on Information Systems (TOIS)*, 36(1):1–33, 2017.

[9] G. E. Pibiri and R. Venturini. Inverted index compression. http://pages.di.unipi.it/pibiri/papers/ii_compression.pdf, 2019.

[10] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM computing surveys*, 53(6):1–36, 2021.

[11] D. D. Salomon. *Variable-length codes for data compression.* Springer, London, 2007.

[12] S. Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on web search and data mining*, WSDM '13, pages 83–92. ACM, 2013.

[13] S. Vigna. Quasi-succinct indices or the revenge of Elias and Fano. https://shonan.nii.ac.jp/archives/seminar/029/wp-content/uploads/sites/12/2013/07/Sebastiano_Shonan.pdf, 2013. NII Shonan Meeting 2013: Compact Data structures for Big Data. September 27-30, 2013.