On Elias-Fano for Rank Queries in FM-Indexes

Ma, Danyang

2021

Ma, D, Puglisi, S J, Raman, R & Zhukova, B 2021, On Elias-Fano for Rank Queries in FM-Indexes. in A Bilgin, MW Marcellin, J SerraSagrista & JA Storer (eds), 2021 DATA COMPRESSION CONFERENCE (DCC 2021). IEEE Data Compression Conference, IEEE Computer Society, pp. 223-232, Data Compression Conference (DCC), Snowbird, United States, 23/03/2021. https://doi.org/10.1109/DCC50243.2021.00030

http://hdl.handle.net/10138/336880
10.1109/DCC50243.2021.00030

unspecified
acceptedVersion

On Elias-Fano for Rank Queries in FM-Indexes*

Danyang Ma[1], Simon J. Puglisi[2], Rajeev Raman[1], and Bella Zhukova[2]

[1] Department of Informatics, University of Leicester, United Kingdom
[2] Helsinki Institute of Information Technology (HIIT),
Department of Computer Science, University of Helsinki, Finland

Abstract

We describe methods to support fast rank queries on the Burrows-Wheeler transform (BWT) string S of an input string T on alphabet Σ, in order to support pattern counting queries. Our starting point is an approach previously adopted by several authors, which is to represent S as $|\Sigma|$ bitvectors, where the bitvector for symbol c has a 1 at position i if and only if $S[i] = c$, with the bitvectors stored in Elias-Fano (EF) encodings, to enable binary rank queries. We first show that the clustering of symbols induced by the BWT makes standard implementations of EF unattractive. We then engineer several improvements to EF that go some way to alleviating this problem, and go on to describe two new EF-inspired bitvectors that have superior practical performance.

1 Introduction

For a pattern P of length $|P| = m$, an FM-index for text T answers a $\mathsf{count}(P)$ query, returning the number of occurrences of P in T, by executing m rank queries on S, the Burrows-Wheeler Transform (BWT) of T. A rank query $\mathsf{rank}_X(i, c)$ on string X returns the number of occurrences of symbol c in prefix $X[0, i-1]$ (we drop the subscript X when it is clear from context). The main distinguishing feature of different FM-index implementations is the way they represent S and support rank queries on it.

One popular approach, explored by several authors (e.g., [9, 11, 8, 19, 5]), is to store S as σ bitvectors, where the bitvector for symbol c has a 1 at position i if and only if $S[i] = c$. The query $\mathsf{rank}_S(i, c)$ is then answered simply as $\mathsf{rank}(i, 1)$ on the bitvector corresponding to symbol c, with the particular choice of bitvector representation leading to different index size/query time tradeoffs. More complex data structures to support rank on S have also been extensively studied (see [13]).

In this paper, we investigate the use of the Elias-Fano (EF) bitvector [7] to support rank on S. This, in itself, is also not new, but we show experimentally that the nature of the bitvectors induced by the BWT confound the use of standard Elias-Fano representations as implemented in popular software libraries. With this in mind, we describe several optimizations to — and variations on — the Elias-Fano scheme, with the aim of improved performance. In particular, our main contributions are:

*Funded in part by Academy of Finland grant 319454.

- We describe optimizations to EF that quicken rank query time by up to 30% in practice, a significant amount considering the wide use of EF in compressed data structures. Using our improved EF implementation to represent S as σ bitvectors, we derive an FM-index we call $\mathsf{EFFM}_{\mathsf{sdmod}}$.
- Preliminary experimental analysis of $\mathsf{EFFM}_{\mathsf{sdmod}}$ led us to design and implement two variants aimed at improved performance. Both are EF-like, but aim at removing E-F's dependency on select in different ways. These bitvectors enable indexes that outperform baseline methods in our experiments.

We emphasise that these results are not especially tied to FM indexes, and may be of wider interest, in particular to support predecessor and rank queries on bit strings where 1s are sufficiently *clustered*, e.g., in inverted files. Nonetheless, we use the FM-index as a vehicle for experimentation throughout.

This paper is structured as follows. Section 2 provides an overview of Elias-Fano bitvectors along with our practical optimisations to it. Our two new bitvector representations suitable for use in FM indexes are then described in Section 3. Our main performance comparison is in Section 4. Section 5 summarises related work. Reflections and avenues for future work are then laid out in Section 6.

2 Elias-Fano

Elias-Fano. The "Elias-Fano (EF)" data structure [7] represents a bit-string X of length m with n 1s. We describe it by considering X as the characteristic vector of a set $\{x_0, \ldots, x_{n-1}\} \subseteq \{0, \ldots, m-1\}$. We choose an integer *bucket size* $\ell \geq 1$ and divide x_i into a *quotient* value $q_i = \lfloor x_i/2^\ell \rfloor$ and a *remainder* value $r_i = x_i \bmod 2^\ell$. The EF representation stores the sequence of remainders $r_0, \ldots, r_{n-1}$ in an array $L[0..n-1]$ where each entry has width ℓ bits (the *lower* part of EF). In addition, all x_i with the same quotient value j are said to belong to the j-th bucket. The size $s \geq 0$ of each bucket is written in unary as 0^s1, and the unary representations of the bucket sizes are concatenated to form a bit-string U (the *upper* part of EF).

We now discuss the space used by EF. L takes $n\ell$ bits. Quotients need $q = w - \ell$ bits each, where $w = \lceil \lg m \rceil$. Descriptions of EF [14, 7] vary regarding the precise choice of q, including $\lceil \lg(1.44n) \rceil$ [14], $\lfloor \lg n \rfloor$ [7] and $\lfloor \lg n \rfloor + 1$ (`sdsl`). In each case, U is represented using $n + 2^q$ bits, as there can be up to 2^q buckets. We observe that all 2^q buckets need not necessarily exist — the largest quotient is $\lfloor (m-1)/2^\ell \rfloor$, so we can represent U using just $n + \lfloor (m-1)/2^\ell \rfloor + 1$ bits, for an overall space usage of $(\ell+1)n + \lfloor (m-1)/2^\ell \rfloor + 1$ bits. Minimizing the function $f(\ell) = m2^{-\ell} + n\ell$ wrt ℓ gives the optimal ℓ as $\lg((m \ln 2)/n) \sim \lg(0.69m/n)$, we suggest rounding this value to choose ℓ. This is very similar to the (fractional) $q = \lg(1.44n)$ obtained by [14], but our formula handles the case that m is not a power of 2 better. As shown in Table 1, on our data sets, all of the representations above have an additive overhead, compared to the ideal $\mathcal{B}(m,n) = \lg \binom{m}{n}$ bits, of $0.48n$ bits to $1.54n$ bits.

In all the above cases, $\ell = \lg(m/n) + O(1)$ and U takes $O(n)$ bits. The space usage of EF is therefore always $n \lg(m/n) + O(n)$ bits. We now discuss how to perform rank

m	n	[14]	`sdsl`	New 1	New 2
3355443200	209715200	1.16	0.88	0.60	0.60
47185920000	209715200	1.31	1.03	0.63	0.50
48234496000	209715200	1.27	0.99	0.61	0.51
20132659200	209715200	1.54	1.26	0.73	0.48

Table 1: The additive overhead of four combinations of choice of ℓ and representation of bitvector U (representing the EF buckets) relative to $\mathcal{B}(m,n)$, given in bits/n. New 1 and 2 are the modified representation of U, with ℓ chosen according to `sdsl` and the formula given here, respectively.

and select operations on X[1] — the details are standard and can be found, e.g., in [7, 14, 13]. To perform $\mathsf{select}(i,1)$ on X, we directly index L to find the remainder of x_i. We then perform a $\mathsf{select}(i,0)$ on U to find the bucket that x_i is in, thus finding the quotient of x_i. The overall time is $O(1)$. To perform $\mathsf{rank}(i,1)$ on X, we first find the id of the bucket i lies in, and then perform two select operations on U, with consecutive arguments. The first one gives the count of 1s up to the start of the bucket that x_i lies in. The second one tells us the size and endpoints of the subarray of L comprising this bucket. Finally, we perform a rank operation within the bucket. Since the maximum bucket size is $2^\ell = O(m/n)$, we can perform rank within a bucket via binary search to obtain the following lemma.

Lemma 1 ([14]) *A bit-vector X of length m with n 1s can be represented using $n\log(m/n) + O(n)$ bits, supporting select in $O(1)$ and rank in $O(\log(m/n))$ time.*

We remark that rank can be sped up further in theory [2]. In addition, we note that the number of buckets is always $\Theta(n)$, so the average bucket size is $O(1)$. If we believe the keys to be evenly distributed into buckets, we can even perform linear search in buckets during rank, and indeed the code of both [14] and `sdsl` does this. We also remark that the space usage of an EF data structure can be reduced to $\mathcal{B}(m,n) + O(n/(\log n)^{O(1)})$ bits, while still supporting rank and select in the same time, by combining the results of Patrascu [15] with ideas from [17, Theorem 4.6].

EFFM-index. Given a text T, the FM-index [3] represents the string $S = \mathsf{BWT}(T)$, $|S| = n$ in a manner that supports the operation $\mathsf{rank}_S(i,c)$ for any $0 \le i < n$ and $c \in \Sigma$ in time t_{rank}. Using small additional data structures of size $O(\sigma \lg n)$ bits, the FM-index can support $\mathsf{count}(P)$ queries on T for any pattern P in time $O(|P|t_{\mathsf{rank}})$. As noted in the introduction, we represent T using σ bit-vectors each of length n, $b_0, \ldots, b_{\sigma-1}$, where $b_i[j] = 1$ iff $S[j] = i$. The concatenation of these bit-vectors will be called B. As has been observed already in [9], $\mathsf{count}(P)$ can be answered using $O(|P|)$ rank_1 queries on either B or the individual bit-vectors b_i. Our approach, called the EFFM-index, is to either store B in an EF data structure, or the individual b_is in σ EF data structures. Using Lemma 1 to represent B or the b_is, we get:

[1] When operating on bit-vectors, rank and select refer to $\mathsf{rank}(\cdot,1)$ and $\mathsf{select}(\cdot,1)$, unless explicitly stated otherwise.

Lemma 2 (See also [13]) *A text T of length n with alphabet size σ can be represented:*

i. using $n \lg \sigma + O(n) + O(\sigma \lg n)$ bits, supporting count(P) *queries in $O(|P| \lg \sigma)$ time, and*

ii. using $nH_0(T)+O(n)+O(\sigma \lg n)$ bits and supporting count(P) *queries in $O(-\lg \Pr(P))$ time, where $P = p_1 \dots p_{|P|}$, and $\Pr(P) = \prod_{i=1}^{|P|} \Pr(p_i)$, and $\Pr(p_i) = n_i/n$, where n_i is the number of times p_i appears in T.*

Implementing the EFFM-Index. Many standard implementations of the FM-index store S in a *wavelet tree (WT)* [6]. A "balanced" WT stores S as a collection of $\lg \sigma$ bit-vectors, and achieves bounds similar to Lemma 2(i), without the $O(n)$ term in the space bound, but adding a potentially large lower-order term of $\lg \sigma \cdot o(n)$ bits, while a "Huffman-shaped" WT does the same wrt Lemma 2(ii). In principle, the EFFM-index should be fast compared to WT-based FM-indices because the WT makes $O(\lg \sigma)$ rank queries per symbol in P, and these rank queries should make non-local memory accesses. By contrast, EF rank queries perform binary search on a bucket of size $O(\sigma)$, which, unless σ is very large, should show good locality (indeed, the average bucket size is only $O(1)$, so one could hope for even faster rank queries).

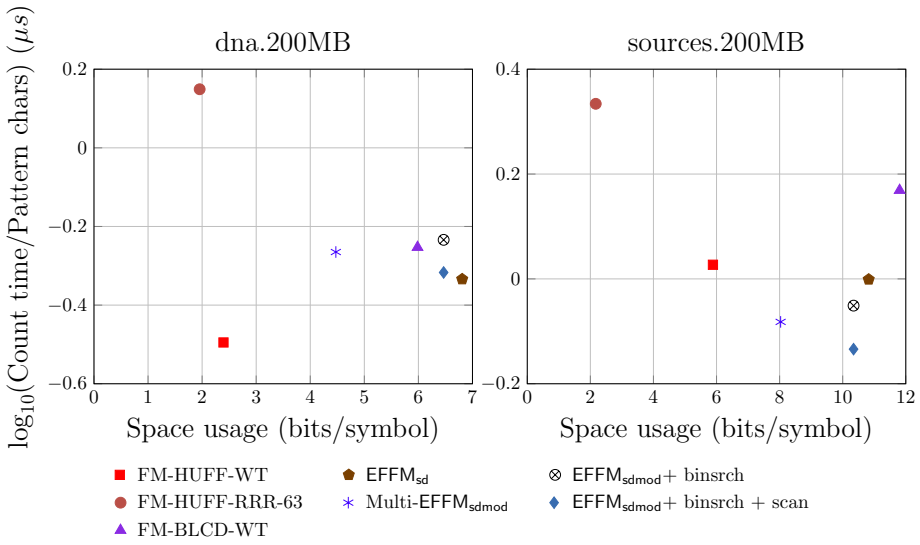

Figure 1: Time-space tradeoff for various `sdsl` FM-index implementations and EFFM variants (y-axis is on log scale).

We performed some experiments on EFFM. The experimental setup and implementation details are described in Section 4: we summarize the main points here. The base implementation EFFM$_{\text{sd}}$ represents B using the `sdsl` implementation of EF (the `sd_vector` class) as is – recall that this uses linear search for rank within a bucket. In EFFM$_{\text{sdmod}}$ we made the small change to U to reduce space that we described previously, and tried a variety of methods for calculating rank within a bucket, explained in Section 4. We report on just two now: binary search (using two select

operations as described above, "EFFM$_{\text{sdmod}}$ + binsrch") and optimized binary search (using only one explicit select operation during rank, and replacing the other one by a sequential scan of U "EFFM$_{\text{sdmod}}$ + binsrch + scan"); EFFM$_{\text{sdmod}}$ below refers to the latter unless explicitly stated otherwise. We also tried Multi-EFFM, where each b_i is stored in a separate EF instance; since bucket sizes can vary widely across different b_i, the rank function here dynamically chooses among different implementations of rank within a bucket. These were compared against three `sdsl` FM-index implementations, using a balanced WT (`FM-BLCD-WT`), a Huffman-shaped WT (`FM-HUFF-WT`) and a Huffman-shaped WT using the RRR compressed bit-vector (`FM-HUFF-RR`), which has been shown to achieve H_k compression [10]. Representative results are shown in Figure 1: the datasets are described in Table 2 (XML and English were similar to Sources). We note:

- EFFM$_{\text{sdmod}}$ is slightly more space-efficient than EFFM$_{\text{sd}}$, as expected.
- EFFM$_{\text{sd}}$ slows down significantly on Sources ($\sigma = 230$) relative to DNA ($\sigma = 16$), while EFFM$_{\text{sdmod}}$ maintains its performance. This is because Sources is H_k-compressible (shown by the space usage of `FM-HUFF-RRR`), and for H_k-compressible data, the occurrences of a given character c in S will be clustered together, rather than scattered across S, implying that 1s in B (and therefore in b_i's) are clustered together. This means that buckets in L will tend to be either quite full or completely empty, and linear search is a poor option.
- Replacing one of the two selects on U with a linear scan of U has a significant impact, as the scanned distance is typically small, and select is relatively slow.
- When σ is large, EFFM$_{\text{sdmod}}$ uses less space than `FM-BLCD-WT`, implying that the extra $O(n)$ term in the former is smaller than the $\lg \sigma \cdot o(n)$ term in the latter.

`FM-HUFF-WT` dominates EFFM$_{\text{sdmod}}$ on DNA. While `FM-HUFF-WT` is a litle slower than EFFM$_{\text{sdmod}}$ on Sources, its space usage is far lower (the gap in space usage with respect to Multi-EFFM is smaller, but still noticeable). Given that `FM-HUFF-WT` is "off-the-shelf" software (albeit well-engineered and optimized), this cannot be said to be a good outcome. `FM-HUFF-WT` is fast partly because the symbols in the pattern have a similar distribution as the symbols in S / T, and so the number of levels of the WT that a rank operation must traverse is usually much less than $\lg \sigma$. In addition, the searches will have a degree of temporal locality.

3 New Approaches

In this section we discuss two new approaches to the EFFM-index.

Zero-Suppressed EFFM. The starting point of this approach is the observation from the previous section that buckets in L tend to be either empty or relatively full when T is H_k compressible. From a practical perspective, if $\ell = 8$ (i.e. maximum bucket size = 256), if a bucket has ≥ 32 remainders in it, then it is actually more space-efficient to store it as a characteristic bit-vector of length 256, which says which remainders are actually present. Storing a bucket as a characteristic bit-vector has

two advantages: 1) rank within a bucket will be extremely fast, and 2) each bucket will have a fixed-width representation, eliminating the need for select in U to find bucket boundaries. We now describe an EFFM variant based on this idea, called $\mathsf{EFFM_{zs}}$.

For simplicity, the description assumes that σ and n are powers of 2, and we describe the variant where B is stored in a single instance of E-F. We choose a parameter $0 \leq \ell \leq (\lg n + \lg \sigma)$, and divide B into buckets of size 2^ℓ. L is now a bit-vector that consists of the concatenation of all buckets that are not all zero, and is of length $2^\ell \cdot nz_\ell$, where nz_i is the number of non-zero buckets when $\ell = i$ (when $\ell = 0$ we take L to be the empty string). U is now a bit-vector of length $(n\sigma)/2^\ell$, and $U[i] = 1$ iff the i-th bucket is not all zeros. To execute $\mathsf{rank}_B(i, 1)$, we first perform $j = \mathsf{rank}_U(\lfloor i/2^\ell \rfloor, 1)$. This tells us that there are j non-empty buckets before the bucket containing i. Now if $U[\lfloor i/2^\ell \rfloor] = 0$, then $\mathsf{rank}_B(i, 1) = \mathsf{rank}_L(j \cdot 2^\ell, 1)$, otherwise $\mathsf{rank}_B(i, 1) = \mathsf{rank}_L(j \cdot 2^\ell + i \bmod 2^\ell, 1)$. Thus, rank_B is reduced to two rank operations on U and L respectively, and should be fast.

We now discuss space usage. Noting that $nz_i \leq n$ for all i, L uses $\leq n2^\ell$ bits in the worst case, and U uses $(n\sigma)/2^\ell$ bits. Choosing ℓ so that $2^\ell = O(\sqrt{\sigma})$, we get a space usage of $O(n\sqrt{\sigma})$ bits, which can be acceptable in practice if σ is small. In the best case, however, nz_ℓ can be as small as $n/2^\ell$, and L will then occupy just n bits. We can then choose any $\ell \geq \lg \sigma$ and obtain an $O(n)$-bit space usage. Although the space usage of $\mathsf{EFFM_{zs}}$ is data-dependent, we observe that:

- As $nz_{i+1} \geq nz_i/2$, for $i = 1, \ldots, \lg(n\sigma) - 1$, $|L|$ is monotonically non-decreasing with increasing ℓ, going from 0 to $n\sigma$. $|U|$ decreases monotonically as ℓ increases. Thus the space usage of $\mathsf{EFFM_{zs}}$ is an upwardly concave function of ℓ, allowing us to choose an optimal ℓ if there is a closed-form estimate for nz_i.
- If B has r runs, then $nz_\ell \leq 2r + n/2^\ell$, and we can choose ℓ so that the overall space usage is $O(\sqrt{nr\sigma} + n)$ bits.

Partitioned-Upper EFFM. As the space usage of $\mathsf{EFFM_{zs}}$ is data-dependent, we now consider a different approach that also aims to gain speed by removing the need for select on U. We partition U into two bit-vectors U_{nz} and U_{sz}, and assume that ℓ is chosen as in standard E-F, namely $\ell = \lg \sigma + O(1)$. $U_{nz}[i] = 1$ iff the i-th bucket is non-empty. U_{sz} comprises the concatenation of the unary encodings of all the sizes of the *non-empty* buckets, so a bucket with size $s \geq 1$ is encoded as $0^{s-1}1$ (note that $|U_{sz}| + |U_{nz}| = n + \lceil m/2^\ell \rceil = |U|$). It is easy to see that $\mathsf{select}_U(i, 1)$ can be simulated by a rank on U_{nz} and a select on U_{sz}. The key observation is that $\mathsf{select}_{U_{sz}}$ can be supported rapidly while not using too much space.

We represent U_{sz} as an *array* with each entry of width ℓ bits; $U_{sz}[i]$ now is the size of the i-th non-empty bucket (minus one). The length of U_{sz} is $n' = nz_\ell$. $\mathsf{select}_{U_{sz}}(i)$ simply returns the partial sum of the first i bucket sizes. Using standard techniques, such as explicitly storing every $k = O(\lg n / \lg \sigma)$-th prefix sum explicitly using $O(\lg n)$ bits and performing table lookup on segments of U_{sz} of size $O(k)$, we can represent U_{sz} in $O(n'\ell) = O(n' \log \sigma)$ bits and support this operation in $O(1)$ time. Since $n' \leq n$, the worst-case space usage of this representation is $O(n \log \sigma)$ bits. However, for compressible texts, we would expect $n' \ll n$. We call this representation $\mathsf{EFFM_{pu}}$.

File	n	σ	H_0	ℓ (ZS)	%NZB (ZS)	ℓ (PU)	%NZB (PU)
DNA	209,715,200	16	1.97	10	25.07%	4	22.21%
English	209,715,200	225	4.52	7	3.51%	8	4.27%
Sources	209,715,200	230	5.47	6	2.40%	8	4.41%
XML	209,715,200	96	5.26	7	4.14%	7	4.14%

Table 2: On the left, the main characteristics of our datasets. On the right, parameters chosen for $\mathsf{EFFM}_{\mathsf{zs}}$ and $\mathsf{EFFM}_{\mathsf{pu}}$, and percentages of non-zero buckets for each dataset shown.

4 Implementation and Empirical Evaluation

Test Machine and Environment. We used a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.10.0-38-generic. Programs were in C++11, compiled using `g++` version 5.4.0. All given runtimes for the `count` queries were obtained using the Linux `getrusage(RUSAGE_SELF)` facility and the space with `sdsl`'s `size_in_bytes()` method.

As testing datasets we used 200MB files from the Pizza&Chili corpus. We refer to these files as DNA, English, Sources, and XML, where e.g. DNA refers to the file `DNA.200MB` on the site. Table 2 shows characteristics of the files. Patterns were generated randomly by choosing 50000 patterns of length 20 from each file.

Coding details. Our new implementations are as follows.
$\mathsf{EFFM}_{\mathsf{sdmod}}$: represents B using a modification of `sdsl`'s `sd_vector`. First, U's length was reduced[2] as in Section 2. Next, rank in a bucket was implemented as below:

- Using binary search, either using two selects as described in Section 2 ("$\mathsf{EFFM}_{\mathsf{sdmod}}$ + binsrch") or by using only one explicit select during rank, and replacing the other by a sequential scan of U ("$\mathsf{EFFM}_{\mathsf{sdmod}}$ + binsrch + scan").
- We used MMX/SSE/BMI2 x86 intrinsics to speed up linear search in the buckets ("$\mathsf{EFFM}_{\mathsf{sdmod}}$ + x86"). This was only used when $\ell \leq 8$, and was used to compare 8 remainders against the query value using built-in instructions on 64-bit MMX registers. Since the remainders were stored in $\ell \leq 8$-bit fields in integers, we read 8 remainders at a time using SDSL's `get_int` method, and spread them into 8-bit fields using a `PDEP` instruction.
- Two-level search: checking every k-th remainder (we used $k = 16$) in the bucket using linear search, and comparing against the k intervening remainders using x86 intrinsics (as above). This is called "$\mathsf{EFFM}_{\mathsf{sdmod}}$ + multilevel + x68".

$\mathsf{EFFM}_{\mathsf{zs}}$: Our implementation takes the bit-vector to be represented and empirically calculates the value of ℓ that gives the smallest overall size. This can be done in time linear in the number of words of the input bit-vector, as follows. The space usage of $\mathsf{EFFM}_{\mathsf{zs}}$ for a particular ℓ is fully determined by the number of 1 bits in U. Given U for a particular ℓ, we can calculate U for $\ell + 1$ by simply ORing disjoint pairs of

[2]Due to time constraints, our final experiments only used the value of ℓ chosen by `sdsl`.

consecutive bits in U. We then pack the resulting bits, resulting in a new bit-vector of half the size.
EFFM$_{\text{pu}}$: ℓ is chosen as in `sdsl`. select on the array U_{sz} is also accelerated using x86 intrinsics if $\ell \leq 8$. We choose the approach for performing rank in the bucket depending on ℓ: when $\ell \leq 3$ linear search is used, when $4 \leq \ell \leq 8$ we use multi-level + x86, and binary search otherwise.
Multi-EFFM$_*$: These simply have σ instances of the corresponding EF variant.

Results. Figure 2 shows the size and query times for the indexes. We note:

- For all data sets, EFFM$_{\text{zs}}$ is the fastest index tested, usually narrowly shading Multi-EFFM$_{\text{zs}}$. EFFM$_{\text{zs}}$ is around four times faster than FM-HUFF-WT on all data sets except DNA, where it is nonetheless still the fastest index. The space usage of Multi-EFFM$_{\text{zs}}$ is surprisingly good: we would like to understand this better. The choice of ℓ can be unexpected, e.g. $\ell = 10$ for DNA (cf. Table 2).
- In all cases except DNA, EFFM$_{\text{pu}}$ outperforms EFFM$_{\text{sdmod}}$, both in terms of space and time. This shows that the re-engineered U in EFFM$_{\text{pu}}$ does indeed support select faster, and uses less space to boot (this is explained by the low proportion of non-empty buckets in these datasets, see Table 2). However, Multi-EFFM$_{\text{pu}}$'s improvement over Multi-EFFM$_{\text{sdmod}}$ is smaller. Overall, Multi-EFFM$_{\text{pu}}$ gives relevant performance tradeoffs on English and Sources.
- The overhead of the space usage of `FM-HUFF-WT` wrt H_0 is about 0.4 bits/symbol, while that of Multi-EFFM$_{\text{sdmod}}$ is about 2.5-2.6 bits/symbol. This is expected, because EFFM targets $\mathcal{B}(n\sigma, n)$, which is $nH_0 + \Omega(n)$. EFFM$_{\text{pu}}$ comes closer on all data sets except DNA (which could be because DNA has a high fraction of non-empty buckets). We should be able to reduce the gaps for both EFFM variants by about 0.5 bits/symbol (cf. footnote 2).

5 Related Work

As noted at the start of this article, representing the BWT as σ bitvectors (Elias-Fano encoded, or not) is an idea with some history. Grabowski et al. [5] credit the idea to Mäkinen and Navarro [9]. However, the highly-related idea of storing the Ψ function as σ bitvectors appears in even earlier papers on compressed text indexing, including Sadakane's compressed suffix array [18] and in a paper by Grossi, Gupta, and Vitter [6], a variation on which is implemented in [8].

Sirén et al. applied the σ-bitvectors approach to several practical compressed indexes, starting from the RLCSA [11], which uses run-length- and gap-encoded bitvectors instead of Elias-Fano. In GCSA2 [19], an index intended for genomic data, plain bitvectors are used for frequent characters and Elias-Fano for infrequent ones.

Gog, Petri, and Moffat [4] describe a CSA for large alphabets using a hybrid encoding: each bitvector is broken into blocks of $k = 128$ ones and then each block is encoded as either Elias-Fano, run-length encoding, or as a plain uncompressed bitvector. More recently, Arroyuelo and Sepúlveda [1] have described an alphabet-partitioning structure that uses an Elias-Fano bitvector over the BWT for each subalphabet.

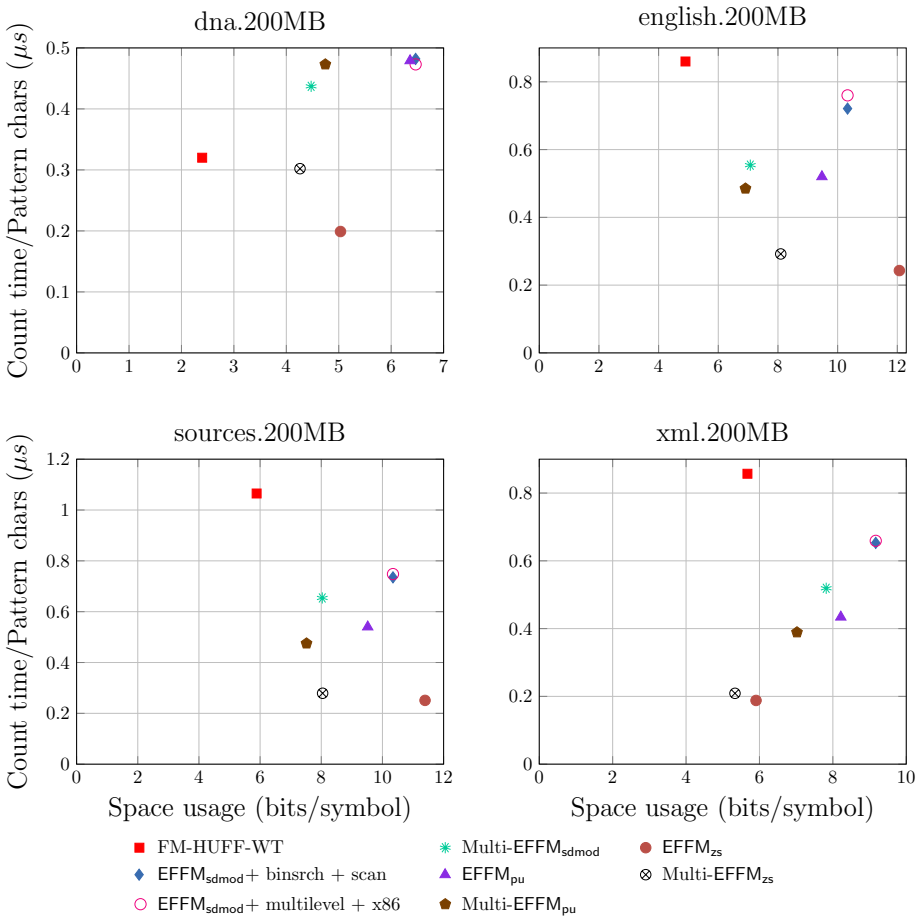

Figure 2: Time-space tradeoffs for various `sdsl` FM-indexes and EFFM variants.

The approaches used to represent bit-vectors in Section 3 are also not entirely new. The approach used in EFFM$_{zs}$ is similar to the approach used by Na et al. [12], and arguably even has precursors in the van Emde Boas data structure [20]. The novelty is that due to the structure of our input, we can use this approach to get $\mathcal{B}(m, n)$ space in practice. The partitioning of U into U_{nz} and U_{sz} in EFFM$_{pu}$ is also a known technique [16], but getting a select-less EF using $O(n \lg \sigma)$ bits may be new.

6 Conclusions and Future Work

The representation of the BWT string, S, as a series of σ bitvectors is an old idea. In this paper we have shown that, when implemented carefully, this approach can outperform wavelet-tree-based FM-indexes, particularly for query time. There are many avenues for future work. In particular, we plan to explore the use of our indexes for representing and querying labelled trees and automata, where alternative Burrows-Wheeler-based implementations have already gained some traction [19]. Finally, we believe our exploitation of AVX instructions is far from complete, and note that this avenue is largely unexplored in succinct data structres in general.

References

[1] D. Arroyuelo and E. Sepúlveda. A practical alphabet-partitioning rank/select data structure. In *Proc. SPIRE*, LNCS 11811, pages 452–466. Springer, 2019.

[2] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4):23:1–23:19, 2014.

[3] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[4] S. Gog, A. Moffat, and M. Petri. CSA++: fast pattern search for large alphabets. In *Proc. ALENEX*, pages 73–82. SIAM, 2017.

[5] S. Grabowski, M. Raniszewski, and S. Deorowicz. Fm-index for dummies. In *Proc. BDAS*, pages 189–201, 2017.

[6] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.

[7] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications. *SIAM J. Comp.*, 35(2):378–407, 2005.

[8] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *Proc. DCC*, pages 292–301. IEEE, 2014.

[9] V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays, C-2004-20. Technical report, University of Helsinki, 2004.

[10] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE*, LNCS 4726, pages 229–241. Springer, 2007.

[11] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

[12] J. C. Na, J. E. Kim, K. Park, and D. K. Kim. Fast computation of rank and select functions for succinct representation. *IEICE Trans. IS*, 92-D(10):2025–2033, 2009.

[13] G. Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[14] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*. SIAM, 2007.

[15] M. Patrascu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE, 2008.

[16] N. Rahman and R. Raman. Rank and select operations on binary strings. In *Encyclopedia of Algorithms - 2008 Edition*. Springer, 2008.

[17] R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

[18] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA*, pages 225–232. ACM/SIAM, 2002.

[19] J. Sirén. Indexing variation graphs. In *Proc. ALENEX*, pages 13–27, 2017.

[20] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.