Faster, Minuter*

Simon Gog[1], Juha Kärkkäinen[2], Dominik Kempa[2],
Matthias Petri[3] and Simon J. Puglisi[2]

[1] Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Germany
[2] Helsinki Institute for Information Technology,
Department of Computer Science, University of Helsinki, Finland
[3] Department of Computing and Information Systems, University of Melbourne, Australia

Abstract

The FM index (Ferragina & Manzini, J. ACM, 2005) is a widely-used compressed data structure that stores a string T in a compressed form that also supports fast pattern matching queries. *Fixed-block boosting* is a relatively straightforward technique that achieves optimal index size in theory, but to date it is unclear how best to translate the method into practice. In this paper we describe several new techniques for implementing fixed-block boosting efficiently. The new indexes are consistently fast and small relative to the state-of-the-art, and thus make a good "off-the-shelf" choice for most applications.

1 Introduction

The Fast Minute (FM) index, by Ferragina and Manzini [3] is perhaps the most widely used compressed data structure, fundamental to all serious pieces of software for DNA sequence alignment (see, e.g., [6, 10]) and also extensively used for genome assembly [1]. The index stores a string T in a compressed form that also supports fast pattern matching queries. Both compression and search are achieved by exploiting structure present in the Burrows-Wheeler transform (BWT) of T.

In the 15 years since its discovery, techniques to reduce index size and to provide faster pattern searches (preferably both at the same time) have been the subject of many research articles (see, e.g., [5, 7] and references therein). One particularly straightforward scheme is the so-called *fixed-block boosting* method described by Kärkkäinen and Puglisi [8], which leads to an FM-index of size $nH_k(\mathrm{T}) + o(n)\log(\sigma)$ bits[1]. This space usage is optimal in theory, but how to best exploit *fixed block boosting* in practice has to date been unclear — a simple proof-of-concept implementation in [8] gives encouraging, if somewhat inconclusive, results on a narrow range of data.

*Supported by the Academy of Finland via grant 1294143.

[1]$nH_k(T)$ is a lower bound on the compression achievable by any statistical compressor that models symbol probabilities as a function of the k letters preceding it in the text, see [14].

In an FM-index, pattern matching is reduced to a series of *rank* queries over the BWT. The query $\mathrm{rank}(c, i)$ returns the number of occurrences of the symbol c up to position i in a string. To accelerate the rank queries, the BWT is preprocessed to build a rank index. The main idea of fixed-block boosting is to divide the BWT of T into blocks of fixed size and store a compressed rank index for each block. Due to properties of the BWT, the smaller the blocks are, the better compression is achieved. However, an overhead of $O(\sigma \log n)$ bits per block prevents too small block sizes. Kärkkäinen and Puglisi [8] show that asymptotically optimal index size can be achieved by setting the block size to $O(\sigma \log^2 n)$.

Our Contribution. In this paper we show how to implement fixed-block boosting efficiently in practice, via a number of non-trivial optimizations to the basic scheme described in [8]. The goal is to reduce the space overhead per block to as low as possible but without increasing query times. The main components of the overhead are the storage of the rank of each symbol at the block boundary and the representation of a Huffman-shaped wavelet tree, which is used as the rank index. We show that storing ranks for the symbols occurring in a block rather than for all symbols in the alphabet is enough, and describe a fast new Huffman-shaped wavelet tree variant.

The resulting indexes represent a new Pareto frontier for query time and index size in practice on a wide range of data. These new indexes give consistently strong performance in both time and space dimensions, and thus make a good "off-the-shelf" choice for most applications.

2 Overview of FM-Index

Let $\mathrm{T} = \mathrm{T}[0..n-1] = \mathrm{T}[0]\mathrm{T}[1]\ldots\mathrm{T}[n-1]$ be a string of n symbols or characters drawn from an alphabet $\Sigma = \{0, 1, .., \sigma - 1\}$. We assume that $\mathrm{T}[n-1] = 0$ and 0 does not appear anywhere else in T. In the examples, we use '\$' to denote 0.

Let n_w be the number of occurrences of a string w in T, and let $\mathrm{T}|w$ be the subsequence of T consisting of those characters that are immediately followed by w. The zeroth order and the k^{th} order empirical entropy of T are defined as

$$H_0(\mathrm{T}) = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c} \qquad H_k(\mathrm{T}) = \sum_{w\in\Sigma^k} \frac{n_w}{n} H_0(\mathrm{T}|w) \tag{1}$$

The value $nH_k(\mathrm{T})$ represents a lower bound on the number of bits needed to encode T by any compressor that considers a context of size at most k when encoding a symbol. Note that $H_{k+1}(\mathrm{T}) \le H_k(\mathrm{T}) \le \log \sigma$ for all k.

BWT. For any $i \in 0..n-1$, the string $\mathrm{T}[i..n-1]\mathrm{T}[0..i-1]$ is a *rotation* of T. Let $\mathcal{M}$ be the $n \times n$ matrix whose rows are all the rotations of T in lexicographic order. Let F be the first and L the last column of $\mathcal{M}$. The string L is the ***Burrows–Wheeler transform (BWT)*** of T. An example is given in Figure 1.

Note that L is a permutation of T and thus $H_0(\mathrm{L}) = H_0(\mathrm{T})$. Furthermore, for any $k \ge 0$, there exists a partitioning of $\mathrm{L}_1\mathrm{L}_2 \cdots \mathrm{L}_\ell = \mathrm{L}$ of the BWT L into $\ell \le \sigma^k$ blocks so that $\sum_{i=1}^{\ell} |\mathrm{L}_i| H_0(\mathrm{L}_i) = nH_k(\mathrm{T})$. In other words, by compressing each BWT block

F						L
$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

Figure 1: BWT matrix $\mathcal{M}$ for text T = BANANA$.

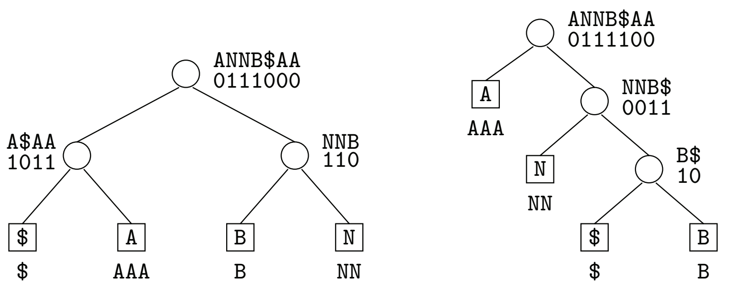

Figure 2: Balanced (left) and Huffman-shaped (right) wavelet trees.

```
Algorithm FM-Count(P[0..m − 1])
1: $b \leftarrow 0$; $e \leftarrow n$
2: for $i \leftarrow m - 1$ downto 0 do
3:     $c \leftarrow \mathrm{P}[i]$
4:     $b \leftarrow \mathrm{C}[c] + \mathrm{rank_L}(c, b)$
5:     $e \leftarrow \mathrm{C}[c] + \mathrm{rank_L}(c, e)$
6:     if $b = e$ then break
7: return $e - b$
```

Figure 3: Counting pattern occurrences using backward search.

```
Algorithm WT-Rank(c, r)
1: $v \leftarrow$ root; $q \leftarrow r$
2: while $v$ is not a leaf do
3:     if $c$ is in the left subtree of $v$ then
4:         $q \leftarrow q - \mathrm{rank_{B(v)}}(1, q)$
5:         $v \leftarrow \mathrm{leftchild}(v)$
6:     else
7:         $q \leftarrow \mathrm{rank_{B(v)}}(1, q)$
8:         $v \leftarrow \mathrm{rightchild}(v)$
9: return $q$
```

Figure 4: Rank using a wavelet tree.

to zero-order entropy, we obtain k^{th} order entropy compression for the whole text. This is called *compression boosting* [4].

The compressibility of highly repetitive texts, such as collections of similar genomes or multiple versions of the same document, is not captured by the k^{th} order entropy for small values of k. For example, $H_k(\mathrm{T}^h) \approx H_k(\mathrm{T})$ for any $h > 1$ and any $k < |\mathrm{T}|$ while it is clear that T^h is much more compressible than T. The long repeats in a highly repetitive texts manifest as short runs of the same character in the BWT [13].

Backward Search. The FM-family of compressed text self-indexes is based on a procedure called ***backward search***, which finds the range of rows in $\mathcal{M}$ that begin with a given pattern P. This range represents the occurrences of P in T. Figure 3 shows how backward search is used for counting the number of occurrences (the count query). In the algorithm, C[c] is the position of the first occurrence of the symbol c in F, and the function $\mathrm{rank_L}$ is defined as $\mathrm{rank_L}(c, j) \equiv \big|\{i \mid i < j \text{ and } \mathrm{L}[i] = c\}\big|$. The main difference between the members of the FM-family is how they implement the $\mathrm{rank_L}$-function. The best ones use wavelet trees.

Wavelet Tree. A wavelet tree of a string X over an alphabet Σ is a binary tree with leaves labelled by the symbols of Σ. Each node v is associated with the subsequence of X consisting of those symbols that appear in the subtree rooted at v. The associated strings are not stored; instead each internal node v stores a bitvector B(v) that tells for each character in the associated string whether it is in the left or right subtree of v. Figure 2 shows examples of the two commonly used variants of wavelet trees, the balanced and the Huffman-shaped.

In a balanced wavelet tree the total length of the bitvectors is $|\mathrm{X}|\lceil\log|\Sigma|\rceil$, which is exactly the length of X in bits using the standard representation. On the other hand, a Huffman-shaped wavelet tree (HWT) corresponds to a Huffman coding of X and has the smallest total bitvector length over all wavelet trees. The length of bitvectors is less than $|\mathrm{X}|(H_0(\mathrm{X})+1)$, i.e., a Huffman-shaped wavelet tree gets close to zeroth order compression.

A rank query $\mathrm{rank}_{\mathrm{X}}(c,r)$ over a wavelet tree is evaluated by a traversal from the root to the leaf labelled by c, as shown in Figure 4. The procedure involves rank queries over the bitvectors stored on the root-to-leaf path.

Bitvector Rank. There are many data structures for representing bitvectors so that rank queries can be answered in (near) constant time. They can be divided into two main categories. Uncompressing techniques leave the bitvector intact but use a small data structure on top of it. Compressing techniques reduce the space of the bitvector as well as prepare it for rank queries. The best-known compressing technique, RRR, stores a bitvector B in $|\mathrm{B}|H_0(\mathrm{B})+o(|\mathrm{B}|)$ bits and supports constant time rank queries. In practice, however, uncompressed bitvectors are substantially faster. The hybrid bitvector uses a combination of representations and offers good compression, though without as strong guarantees as RRR, and fast queries, though not quite as fast as uncompressed bitvectors. Recent experimental studies of these bitvectors can be found in [7, 9].

Higher Order and Repetitive Text Compression. Even the uncompressing variants of the FM-index require only a little space in addition to the size of the uncompressed text. The space can be reduced close to the zeroth order compressed size either by using Huffman-shaped wavelet trees or by using compressed bitvectors.

Higher order compression requires compression boosting, i.e., partitioning the BWT into blocks and compressing those blocks separately. In fixed-block boosting, the blocks are all the same size. This is not quite optimal for compression but still can be implemented in $nH_k(\mathrm{T})+o(n)\log\sigma$ bits of space, while offering fast queries and simple implementation. The division into blocks can also be done at the bitvector level, and is in fact done by the RRR and hybrid bitvectors, the use of which thus achieves higher order compression without explicit partitioning. This is known as implicit compression boosting [12].

The short runs in the BWT of highly repetitive texts are best compressed using run-length encoding. This can be done either at the BWT level as described in [13] or at the bitvector level as is done when using the hybrid bitvector. The RRR bitvectors use small block sizes that are often small enough to compress those short runs.

The RRR (with block size 63) and hybrid bitvectors have an overhead of about 10% of the uncompressed bitvector size beyond which they cannot be compressed. Thus they are often best used in combination with fixed-block boosting and Huffman-shaped wavelet trees that reduce the uncompressed size of the bitvectors.

3 Implementation

We have implemented a new version of the FM-index based on the SDSL library[2]. The top level of the index including backward searching uses the standard SDSL implementation. The bitvector rank implementation is a parameter and any of the implementations in SDSL can be used.

SDSL contains no implementation of fixed block boosting. Our implementation of fixed block boosting contains some novel features designed to reduce the space requirement and/or to speed up queries. We have also implemented a new wavelet tree variant designed specifically for fixed block boosting.

3.1 Fixed Block Boosting

Having a separate wavelet tree for each block reduces the total size of the bitvectors. It can also speed up queries because the height of the wavelet trees is smaller. The smaller the block size, the bigger these advantages are. On the other hand, each block needs some space in addition to the bitvectors and this overhead increases as the blocks get smaller. We want to minimize this overhead without sacrificing speed.

Alphabet Mapping. The wavelet tree of a block L_j contains a leaf for each symbol that appers in the block. Let $\Sigma_j = [0 \dots \sigma_j)$ be the *block alphabet* representing these symbols in the order of the leaves in the wavelet tree. To implement the rank query, we need a mapping $\gamma_j : \Sigma \to \Sigma_j \cup \{\bot\}$ from the global alphabet to the block alphabet, where $\bot$ is a special value indicating that the symbol does not appear in the block. To implement an access query, we also need the inverse mapping $\gamma_j^{-1} : \Sigma_j \to \Sigma$.

The inverse mappings are implemented as separate arrays for each block. The forward mappings are implemented as a single two-dimensional array in symbol-wise order, i.e., for a given symbol c, the values $\gamma_{[1..\ell]}(c)$ are stored consecutively.

The implementation assumes a byte alphabet no larger than 256, and uses the value 255 for $\bot$. If some block L_j happens to contain all 256 possible symbols, we map two symbols to 254. Whenever $\gamma_j(c) = 254$, we check if $\gamma_j^{-1}(254) = c$ and if not, change the value to 255.

Block Boundary Ranks. Let L_j be the block that contains a given position i and let s_j be the starting position of L_j in L. Since all blocks have the same size, computing j and s_j is easy. A rank query $\mathrm{rank}_{\mathrm{L}}(c, i)$ is implemented differently depending on whether L_j contains at least one occurrence of c or not. If it does, we compute the rank using

$$\mathrm{rank}_{\mathrm{L}}(c, i) = \mathrm{rank}_{\mathrm{L}}(c, s_j) + \mathrm{rank}_{\mathrm{L}_j}(c, i - s_j)\,.$$

The first term is obtained from an array $R_j[0..\sigma_j)$, where $R_j[\gamma_j(c)] = \mathrm{rank}_{\mathrm{L}}(c, s_j)$. The second term is computed using the wavelet tree of L_j.

If c does not appear in L_j, we find the nearest block L_k, $k > j$, that contains c by scanning $\gamma_{[j+1..k]}(c)$ for the first non-$\bot$ value. Then

$$\mathrm{rank}_{\mathrm{L}}(c, i) = \mathrm{rank}_{\mathrm{L}}(c, s_k) = R_k[\gamma_k(c)]\,.$$

[2]`https://github.com/simongog/sdsl-lite`

This approach achieves a potentially significant space saving by storing the value $\mathrm{rank}_{\mathrm{L}}(c, s_j)$ only if c occurs in L_j, since often σ_j is much smaller than σ.

Superblocks. The BWT is partitioned into superblocks each consisting of a number of blocks. Each superblock stores a mapping from the global alphabet to a superblock alphabet and the ranks at the superblock boundaries for all symbols. The implementations of these are simpler and less optimized than the ones for the blocks.

All the block data structures within a superblock are implemented completely separately for each superblock. In particular, each superblock can use a different block size. Our implementation tries multiple different block sizes for each superblock and chooses the most space efficient one.

The separation of the superblock data structures means that they can be constructed separately offering possibilities for parallel, distributed or (semi)external construction. For example, if the compressed index fits in RAM but the uncompressed BWT does not, we need only one superblockful of the BWT in RAM at a time during the construction.

3.2 Wavelet Tree

A good survey of wavelet tree implementation variants can be found in [2]. Our implementation is essentially a pointerless wavelet tree based on canonical Huffman code. However, we store some additional information to achieve the speed of pointer-based wavelet trees.

Tree Structure. We use a Huffman shaped wavelet tree based on the canonical Huffman code. In such a tree, all the nodes are packed to the right (or to the left in some versions) without gaps and all the internal nodes are to the right of the leaves on the same level. Since the number of nodes on level k is twice the number of internal nodes on level $k-1$, storing the number of leaves on each level is a complete representation of the structure.

In a rank query we need to find the path to the ith leaf. Using the above representation, it is easy to find the level k that contains the ith leaf. Furthermore, we can compute an integer j such that the ith leaf would be the node number j on the level k if the tree was a complete binary tree. Then the bits in the k-bit binary representation of j indicate the turns on the path from the root to the ith leaf.

Bitvectors. All the bitvectors on a level are concatenated together into a single level bitvector. We also concatenate all the level bitvectors together into a single wavelet tree bitvector. We store the sizes of the level bitvectors in order to locate them. Furthermore, we also concatenate all the wavelet tree bitvectors within a superblock. Each block stores the position of its wavelet tree bitvector in the superblock bitvector. The superblock bitvector can be implemented with any of the SDSL bitvectors supporting rank and access queries.

Computing a rank query $\mathrm{rank}_{\mathrm{B}}(1, i)$ requires locating the starting position s of B in the concatenated bitvector $\widehat{\mathrm{B}}$. Then

$$\mathrm{rank}_{\mathrm{B}}(1, i) = \mathrm{rank}_{\widehat{\mathrm{B}}}(1, s+i) - \mathrm{rank}_{\widehat{\mathrm{B}}}(1, s)\,.$$

In a pointerless wavelet tree, s is computed without storing any extra information. The bit vector of a left child starts at the same position on the level bitvector as its parent's bitvector on the previous level, where the position is measured as a distance from the right end of the level bitvector. Similarly, the bitvector of a right child ends at the same position as its parent's. The size of each bitvector can be computed by counting the number of zeros (left child) or ones (right child) in the parent bitvector. The counting can be done with two rank queries at the bitvector boundaries.

The above procedure requires three bitvector rank queries for each level of the wavelet tree while a pointer-based wavelet tree needs just one rank query per level. However, two of the three queries are at the bitvector boundaries. Thus only one rank query per level is needed if we store the bitvector boundary ranks, which needs less space than a full pointer-based wavelet tree. We are not aware of a previous wavelet tree implementation with this type of optimization. To reduce the number of bits needed, we do not store the absolute rank values but the difference to the rank value at the parent boundary (already computed during the descend).

3.3 The Final Data Structure

Let us now summarize the components of the implementation and their sizes. We assume that the maximum alphabet size is 256, the maximum block size is 2^{16} and the maximum superblock size is 2^{24}.

For each superblock we store:

1. The mapping from the global alphabet to the superblock alphabet (σ bytes).
2. The ranks at the superblock boundaries using 4σ bytes. If the BWT is longer than 2^{32}, it is further divided into hyperblocks of size 2^{32}.
3. The alphabet mappings from the superblock alphabet to the block alphabets using σ_s bytes per block, where σ_s is the size of the superblock alphabet.
4. The concatenated wavelet tree bitvectors.
5. An array with a variable-sized entry for each block (see below).
6. An array with a 14 byte entry for each block containing the block alphabet size, the number of wavelet tree levels, the bitvector rank at the start of the wavelet tree bitvector and pointers to the two arrays above.

The variable-sized block entry for a block with a block alphabet size σ_b consists of:

5.1 The inverse alphabet mapping using σ_b bytes.
5.2 The block boundary ranks using $3\sigma_b$ bytes.
5.3 The wavelet tree structure and the level bitvector sizes using three bytes per wavelet tree level.
5.4 The bitvector boundary ranks using $2(\sigma_b - 1)$ bytes.

The relative sizes of the different components are shown in Figure 5 for several files. The missing component is the bitvectors implemented with hybrid bitvectors in this experiment. Here as well as in all of our experiments, the superblock size was fixed to 1 MiB. The block size was chosen separately for each superblock to minimize the total space. A more detailed study of the effect of the block and superblock sizes will be provided in an extended version of this paper.

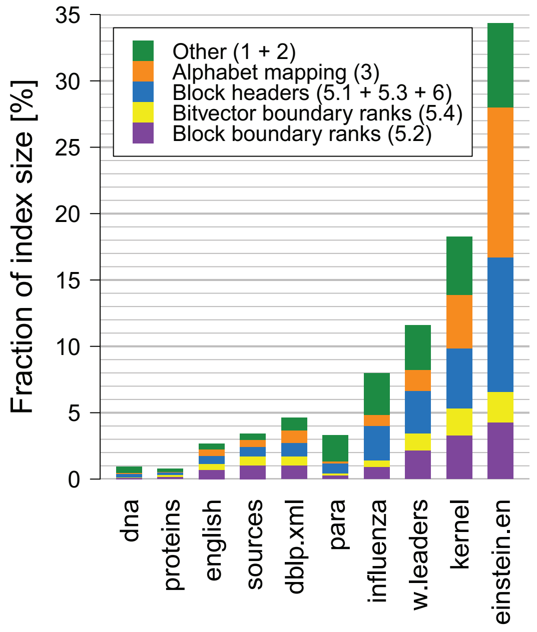

Figure 5: The percentage of the total index size for individual components of the new wavelet tree (with hybrid bitvector as a bitvector representation) for different testfiles. The numbers refer to description in Section 3.3.

Name	σ	$n/2^{20}$	n/r	n/z
dna	16	200	1.63	15.0
proteins	25	200	1.93	8.5
english	225	200	2.91	15.0
sources	230	200	4.40	18.3
dblp.xml	96	200	7.09	29.9
para	5	410	27	184
influenza	15	148	51	199
w.leaders	89	44	82	267
kernel	227	2048	304	1127
einstein.en	199	1198	707	2420

Table 1: Files used in the experiments. The files are from the Pizza & Chili standard and repetitive corpus (available from `http://pizzachili.dcc.uchile.cl/`), except we use larger versions of kernel and einstein.en testfiles. The values of n/r (average length of the run in BWT) and n/z (average length of phrase in the LZ77 factorization) are included as measures of repetitiveness.

Perhaps the most important novelty with respect to prior implementations is that we store only σ_b instead of σ_s boundary ranks per block. Figure 5 shows the effectiveness of this optimization. In the figure, the alphabet mappings need σ_s bytes per block and the block boundary ranks would need three times that without the boundary rank optimization. For all files, the actual space is much smaller.

Another novel optimization is that we achieve the speed of pointer-based wavelet trees (one bitvector rank query per level) using just $2\sigma_b$ bytes of additional space, which was never more than about 2% of the total index size in our experiments.

4 Experiments

We performed experiments on a 3.4 GHz Intel Core i7-4770 CPU equipped with 8 MiB L3 cache and 16 GiB of DDR3 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 14.04, 64bit) running kernel 3.19.0. All programs were compiled using `g++` version 4.9.2 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

We implemented the new version of FM-index as described in Section 3 and compared its performance for count queries to a number of other indexes, including other variants of FM-index. The results, including more detailed description of competing indexes, are presented in Figure 6.

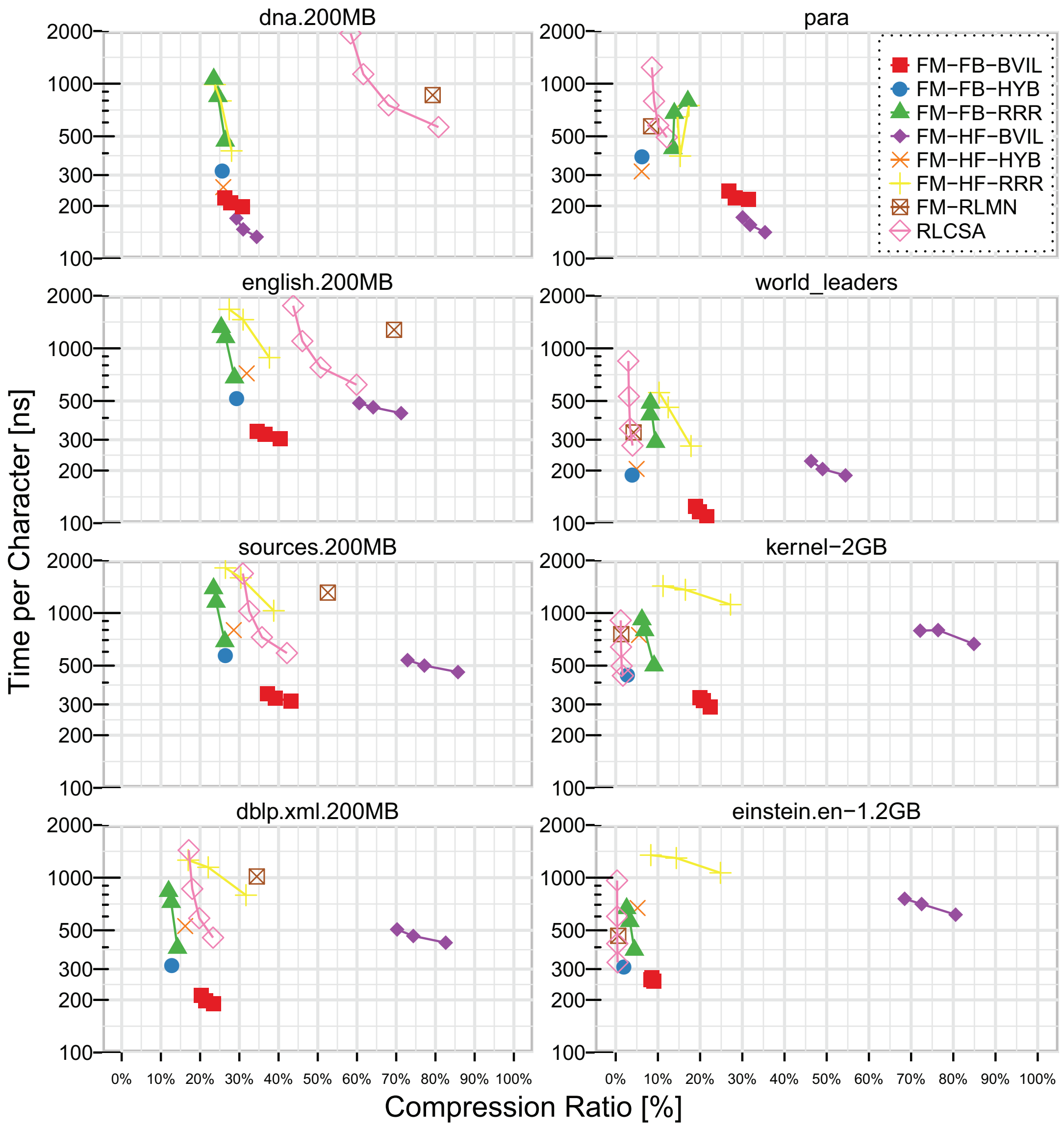

Figure 6: Time/space tradeoffs on standard (left) and repetitive (right) collections for count queries using the methodology of Ferragina et al. [5] which measures mean time in nanoseconds to process one symbol during a single count query over 5×10^4 queries on 20-length patterns randomly extracted from the indexed text. Space is given with respect to the original size of the input text. The text indexes included in the experiments are primarily a Huffman-shaped wavelet tree based FM-index (FM-HF-*) and a fixed blocked partitioned wavelet tree (FM-FB-*). For each FM-index type we use (1) RRR compressed bitvectors (FM-*-RRR) using block sizes 15, 31 and 63 [7, 15, 16]; (2) optimized uncompressed bitvectors with interleaved rank samples (FM-*-BVIL) using block sizes 256, 512, and 1024, providing a rank space overhead of 25%, 12.5% and 6.25% respectively; and (3) the hybrid encoding method [9] using a superblock size of 16 (FM-*-HYB). Additionally, we use an FM-index based on a run-length encoded BWT (FM-RLMN) [11] and the RLCSA index of Mäkinen et al. [13]. All code except RLCSA (`http://iki.fi/jouni.siren/rlcsa`) and the fixed block wavelet tree are part of the SDSL library. All optional structures not needed for count queries are excluded.

The new FM-index achieves superior performance compared to a single wavelet tree independently of the underlying bitvectors representation and simultaneously achieves very significant compression for nearly all types of data. For example: (i) plugging the high-speed uncompressed bitvectors from SDSL library to our FM-index sets a new speed record for count queries (within the space achievable by an FM-index), (ii) plugging the high-compression RRR representation to the new FM-index allows reducing the space beyond what is achievable with a single wavelet-tree RRR.

The lone exception, where the new FM-index is slightly slower (and achieves only marginally better compression) is DNA data which, due to even distribution of (small number of) symbols, does not benefit from high-order entropy compression.

Plugging the hybrid bitvector into the new FM-index results in a particularly effective combination. The resulting index achieves all-around excellent compression and is never much slower than other indexes at the corresponding compression level.

References

[1] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Proc. WABI*, volume 7534 of *LNCS*, pages 225–235. Springer, 2012.

[2] F. Claude, G. Navarro, and A. O. Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015.

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

[4] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.

[5] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Experiment. Algor.*, 13:1.12–1.31, 2009.

[6] P. Flicek and E. Birney. Sense from sequence reads: Methods for alignment and assembly. *Nat. Methods*, 6(11 Suppl):S6–S12, Nov. 2009.

[7] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Softw., Pract. Exper.*, 44(11):1287–1314, 2014.

[8] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. SPIRE*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011.

[9] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. DCC*, pages 302–311. IEEE, 2014.

[10] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[11] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM*, volume 3537 of *LNCS*, pages 45–56. Springer, 2005.

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

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

[14] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[15] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, volume 7276 of *LNCS*, pages 295–306. Springer, 2012.

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