Algorithmica manuscript No.
(will be inserted by the editor)

Fixed Block Compression Boosting in FM-Indexes: Theory and Practice

Simon Gog · Juha Kärkkäinen · Dominik Kempa · Matthias Petri · Simon J. Puglisi

Received: – / Accepted: –

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 and also supports fast pattern matching queries. In this paper, we describe new FM-index variants that combine nice theoretical properties, simple implementation and improved practical performance. Our main theoretical result is a new technique called *fixed block compression boosting*, which is a simpler and faster alternative to optimal compression boosting and implicit compression boosting used in previous FM-indexes. We also describe several new techniques for implementing fixed-block boosting efficiently, including a new, fast, and space-efficient implementation of wavelet trees. Our extensive experiments show the new indexes to be consistently fast and small relative to the state-of-the-art, and thus they make a good "off-the-shelf" choice for many applications.

Supported by the Academy of Finland via grant 294143.

S. Gog
Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Germany
E-mail: gog@kit.edu

J. Kärkkäinen
Department of Computer Science, University of Helsinki, Finland
E-mail: juha.karkkainen@cs.helsinki.fi

D. Kempa
Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, Finland
E-mail: dominik.kempa@cs.helsinki.fi

M. Petri
Department of Computing and Information Systems, University of Melbourne, Australia
E-mail: matthias.petri@unimelb.edu.au

S. J. Puglisi
Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, Finland
E-mail: simon.puglisi@cs.helsinki.fi

Keywords Text indexing · Wavelet tree · FM-index · Compression boosting · Suffix array · Pattern matching

1 Introduction

The FM-index, by Ferragina and Manzini [8], is perhaps the most widely used compressed data structure. It has risen to particular prominence in the field of bioinformatics, where it is now fundamental to all serious pieces of software for DNA sequence alignment (see, e.g., [21,12]). It is also used for genome assembly [3,2,1] and extensively for the discovery of repetitive structures in genomic data [22]. Elsewhere, in data compression, it is the index underlying state-of-the-art methods for Lempel-Ziv factorization [20,18]. The key virtue of the index, in these and other applications, is that it 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., [7,14] and references therein).

The main components of most FM-indexes are:

- The BWT [4]: an invertible permutation of the text T. A procedure called *backward search* [8] turns a pattern matching query on T into a sequence of *rank* queries on the BWT.
- The *wavelet tree* [15]: a representation of the BWT that turns a BWT rank query into a sequence of rank queries on bitvectors.
- A bitvector *rank index*, which supports fast rank queries on bitvectors.

The total length of standard wavelet tree bitvectors is equal to the size of the original, uncompressed text in bits. All other data structures can be made to fit in less space: asymptotically less in theory, and significantly less in practice. Basic zero-order compression is achieved either with compressed bitvector rank structures, such as RRR [30], or Huffman-shaped wavelet trees [16]. For higher order compression, we can use a technique called *compression boosting* [6,10], where the BWT is partitioned into blocks of varying sizes based on the context of symbols in T, and there is a separate, zero-order compressed wavelet tree for each block. The total size of the resulting data structure is

$$nH_k(T) + o(n)\log\sigma \tag{1}$$

bits for small enough k (see Section 3.2), where $nH_k(T)$ is a lower bound on the size of k^{th} order compressed text.

One can try to improve the compression by carefully choosing the partitioning of the BWT into blocks. Assuming each block is zero-order compressed and there is a certain type of overhead per block, the linear time algorithm of Ferragina et al. [6] finds the optimal partitioning into context blocks. The algorithm has been implemented and is used in the construction of the Alphabet-Friendly FM-index [9,10]. We call this approach *optimal context-block boosting*. Ferragina, Nitto and Venturini [11] describe an approximation algorithm for arbitrary (non-context) blocks, but to our knowledge this approach has never been implemented. Mäkinen and Navarro [24] show that k^{th} order compression is achieved without partitioning the BWT if the underlying bitvectors are compressed with RRR [24]. This phenomenon called *implicit compression boosting* results from the partitioning of the bitvectors into blocks by the RRR technique. All of these approaches achieve the size in (1) but do not provide a general asymptotic improvement.

Main Results. The main theoretical contribution of this article is a technique called *fixed-block compression boosting*. It is similar to the context-block boosting mentioned above, but divides the BWT into blocks of fixed sizes without any regard to the symbol contexts. Such a division is inoptimal in general, but we show that it cannot be much worse than the optimal one. In particular, it too achieves the size in (1) with an appropriate selection of the block size. What we gain by using fixed instead of variable size blocks is simpler and faster data structures. The RRR-structure underlying implicit compression boosting uses blocks of fixed size too but the analysis of the boosting effect based on small blocks on bitvectors does not trivially extend to larger blocks over larger alphabets.

As a second contribution, we show how to implement fixed-block boosting efficiently in practice, via a number of non-trivial optimizations to the basic scheme. 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. We also describe a fast and effective method for optimising the choice of block size at indexing time.

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.

Roadmap. In the next section we review the main components and operation of the FM-index. Section 3 then details the ways in which compression can be brought to the data structure. Section 4 describes and analyses our new technique: fixed-block compression boosting. Section 5 describes different ways of optimizing the basic scheme in order to improve practical performance, and provides experimental justification for these design choices. Finally, in Section 6, we compare the space usage and pattern matching times of implementations of our new indexes to those of several baseline schemes on a wide range of data sets.

2 Overview of the FM-Index

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

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

Fig. 1 BWT matrix $\mathcal{M}$ for text $T = \texttt{BANANA\$}$.

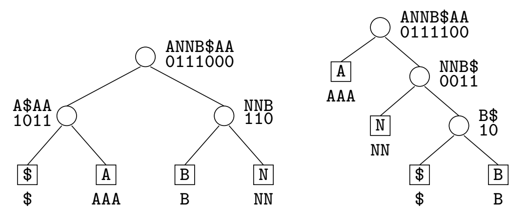

Fig. 2 Balanced (left) and Huffman-shaped (right) wavelet trees.

```
Algorithm FM-Count($P[0..m-1]$)
1: $b \leftarrow 0$; $e \leftarrow n$
2: for $i \leftarrow m-1$ downto 0 do
3:     $c \leftarrow P[i]$
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 $e - b$
```

Fig. 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 $q$
```

Fig. 4 Rank using a wavelet tree.

BWT. For any $i \in \{0 \ldots n-1\}$, the string $T[i..n)T[0..i)$ 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 $\mathcal{M}$, both taken to be strings of length n. The string L is the *Burrows–Wheeler transform (BWT)* of T. An example is given in Figure 1.

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, $\mathrm{C}[c]$ is the position of the first occurrence of the symbol c in F, and the function rank_L is defined as

$$\mathrm{rank}_L(c,j) := \left|\{i \mid i < j \text{ and } L[i] = c\}\right| .$$

The primary difference between the members of the FM-family of indexes is how they implement the rank_L-function. The standard way is to use a wavelet tree.

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 (further described in Section 3 below). In a balanced wavelet tree the total length of the bitvectors is between $|X|\lfloor\log|\Sigma|\rfloor$ and

$|X|\lceil\log|\Sigma|\rceil$. The latter is exactly the length of X in bits using a standard representation of symbols. Notice that the symbol at any position in the string X can be recovered from the wavelet tree and thus there is no need to store X in any other form. A rank query $\mathrm{rank}_X(c,r)$ 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 constant time. The simplest ones add a small data structure on top of the bitvector. The space overhead can be made asymptotically sublinear, and is a few percent of the bitvector size in practical implementations.

Summary. The simple FM-index described above needs $n\log\sigma$ bits of space for the wavelet tree bitvectors, $o(n\log\sigma)$ bits for the bitvector rank data structures and $O(\sigma\log n)$ bits for the C array and the wavelet tree structure. It can answer a count query for a pattern of length m in $O(m\log\sigma)$ time.

3 Compressing the FM-Index

We will next look at how the space needed by the FM-index can be reduced further by compression.

3.1 Zero-Order Entropy

Recall that T is a string of length n over an alphabet Σ of size σ. For each $c\in\Sigma$, let n_c denote the number of occurrences of c in T. The zero-order empirical entropy [26] of T is

$$H_0(T)=\sum_{c\in\Sigma}\frac{n_c}{n}\log\frac{n}{n_c}=\log n-\frac{1}{n}\sum_{c\in\Sigma}n_c\log n_c. \tag{2}$$

The value $nH_0(T)$ represents a lower bound on the number of bits needed to encode T by any compressor that encodes a symbol without regarding the context in which the symbol appears. Since the BWT L is a permutation of T, we have $H_0(L)=H_0(T)$.

Huffman coding is a text compression method that encodes each symbol with a bitstring whose length depends on the symbol frequencies: more frequent symbols are encoded with shorter bitstrings. The length of a Huffman coded string X is less than $|X|(H_0(X)+1)$. In a Huffman-shaped wavelet tree, the Huffman code of a symbol describes the path from the root to the leaf representing that symbol. Thus the total length of the bitvectors in a Huffman-shaped wavelet tree of a string is the same as its Huffman coded size.

An alternative way to achieve zero-order compression is to compress the bitvectors. The best-known compressed bitvector rank data structure, RRR, stores a bitvector B in $|B|H_0(B)+o(|B|)$ bits and supports constant time rank queries. The bitvectors of a balanced wavelet tree compressed using RRR have size $nH_0(T)+o(n)\log\sigma$ [9].

3.2 Higher Order Entropy

Let n_w be the number of occurrences of a string w in T, and let $T|w$ be the subsequence of T consisting of those characters that appear in the *(right) context of w*, i.e., that are immediately followed by w. Here T is taken to be a cyclic string, so that each character has a context of every length. The k^{th} order empirical entropy is

$$H_k(T) = \sum_{w \in \Sigma^k} \frac{n_w}{n} H_0(T|w) = \frac{1}{n}\left(\sum_{w\in\Sigma^k} n_w \log n_w - \sum_{w\in\Sigma^{k+1}} n_w \log n_w\right)$$

The value $nH_k(T)$ 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 $H_{k+1}(T) \le H_k(T) \le \log\sigma$ for all k.

A remarkable property of L, the BWT of T, is that all symbol of $T|w$ appear as a contiguous substring of L for any w (recall that entropy is not affected by permuting the order of symbols); we call this substring of L the *w-context block* of L. For example, if $T = \texttt{BANANA\$}$, then $T|\texttt{A} = \texttt{BNN}$ and $L[1..3] = \texttt{NNB}$ (see Figure 1). Thus we get the following result.

Lemma 1 ([26]) *For any $k \ge 0$, there exists a partitioning of $L_1L_2\cdots L_\ell = L$ of the BWT L of T into $\ell \le \sigma^k$ blocks so that*

$$\sum_{i=1}^{\ell} |L_i| H_0(L_i) = nH_k(T) \; .$$

In other words, by compressing each BWT block to zero-order entropy level, we obtain k^{th} order entropy compression for the whole text. This is called *compression boosting* [6].

The above compression technique translates directly to the compression of the FM-index: Divide the BWT into context blocks using context of length k and implement a separate wavelet tree for each block. There is an additional space overhead of $O(\sigma \log n)$ bits per block from having many blocks and wavelet trees instead of just one. The total overhead is $o(n)$ bits for $k \le ((1-\varepsilon)\log_\sigma n) - 1$ and any constant $\varepsilon > 0$.

It may not be optimal to use the same context length in all parts of L. Ferragina et al. [6] show how to find an optimal partitioning with varying context length in linear time. The resulting compression is at least as good as with *any* fixed k.

Mäkinen and Navarro [24] show that the boosting effect is achieved with the RRR bitvector rank index without any explicit context partitioning. This is called *implicit compression boosting*. First, they observe that instead of partitioning the BWT, we could partition the bitvectors and obtain the same boosting effect. Second, the RRR technique partitions the bitvectors into blocks of size $b = (\log n)/2$ and compresses each independently. The RRR partitioning is not optimal, but Mäkinen and Navarro show that the overhead due to the inoptimality is at most $2\sigma\ell b \le \sigma^{k+1}\log n = o(n)$ under the assumptions mentioned above.

Theorem 2 ([10,24]) *The FM-index either with explicit boosting and optimal partitioning [10] or with implicit boosting [24] can be implemented in $nH_k(T) + o(n)\log\sigma$ bits of space for any $k \le ((1-\varepsilon)\log_\sigma n) - 1$ and any constant $\varepsilon > 0$.*

3.3 Repetitive Texts

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, $H_k(T^h) \approx H_k(T)$ for any $h > 1$ and any $k < |T|$ while it is clear that T^h is much more compressible than T. The long repeats in a highly repetitive text manifest as short runs of the same character in the BWT [25].

These short runs can be effectively compressed using run-length encoding. This can be done either at the BWT level as described in [25] or at the bitvector level.

The two methods of encoding bitvectors that efficiently handle runs of equal symbols are RRR [30] (with large block size) and so-called hybrid bitvectors, described in [17]. The RRR bitvectors use small block sizes that are often small enough to compress those short runs, whereas hybrid bitvectors explicitly use run-length encoding.

However, RRR and hybrid bitvectors have an overhead of about 10% of the uncompressed bitvector size below which they cannot be compressed. Because of this, they are often best used in combination with compression boosting and Huffman-shaped wavelet trees that first reduce the size of the uncompressed bitvectors.

4 Fixed Block Compression Boosting

In this section, we show that the compression boosting effect can also be achieved by partitioning the BWT into blocks of fixed sizes without any regard to symbol context.

Let $H(x,y) = |B|H_0(B)$, where B is a bitvector containing x 0's and y 1's. Let $|X|_c$ denote the number of occurrences of a symbol c in a string X. The following lemma shows what can happen to the total zero-order entropy when two strings are concatenated.

Lemma 3 *For any two strings X and Y over an alphabet Σ,*

$$\begin{aligned} 0 &\le |XY|H_0(XY) - |X|H_0(X) - |Y|H_0(Y) \\ &= H(|X|,|Y|) - \sum_{c\in\Sigma} H(|X|_c, |Y|_c) \le H(|X|,|Y|) \le |XY| \; . \end{aligned}$$

Proof The last two inequalities are trivial and the first is a standard application of Gibb's inequality. We will prove the equality part. For brevity, we write $x = |X|$, $y = |Y|$, $x_c = |X|_c$ and $y_c = |Y|_c$. Using (2), we can write the left-hand side terms as follows

$$\begin{aligned} (x+y)H_0(XY) &= (x+y)\log(x+y) - \sum_{c\in\Sigma}(x_c+y_c)\log(x_c+y_c) \\ xH_0(X) &= x\log x - \sum_{c\in\Sigma} x_c \log x_c \\ yH_0(Y) &= y\log y - \sum_{c\in\Sigma} y_c \log y_c, \end{aligned}$$

and the right-hand side terms as follows

$$H(x,y) = (x+y)\log(x+y) - x\log x - y\log y$$
$$H(x_c,y_c) = (x_c+y_c)\log(x_c+y_c) - x_c\log x_c - y_c\log y_c.$$

From this it is easy to see that the terms on both sides match. □

In other words, the concatenation cannot reduce the total entropy, and the entropy can increase by at most one bit per character. Furthermore, the maximum increase happens only if the two strings have the same length and no common symbols.

Using the above lemma we can bound the increase in entropy when we switch from a context-block partitioning to a fixed block partitioning.

Lemma 4 *Let $X_1X_2\cdots X_\ell = X$ be a string partitioned arbitrarily into ℓ blocks. Let $X_1^bX_2^b\cdots X_m^b = X$ be a partition of X into blocks of size at most b. Then*

$$\sum_{i=1}^{m} |X_i^b| H_0(X_i^b) \le \sum_{i=1}^{\ell} |X_i| H_0(X_i) + (\ell-1)b\ .$$

Proof Consider a process, where we start with the first partitioning, add the block boundaries of the second partitioning, one by one, and then remove the block boundaries points of the first partitioning (that are not block boundaries in the second). By Lemma 3, adding block boundaries cannot increase the total entropy, and removing each block boundary can increase the entropy by at most b bits. □

If we assume the same number of blocks in the two partitionings, the very worst case increase in the entropy is $n-b$ bits.

This increase in entropy can be reduced by reducing the block size in the fixed block partitioning (thus increasing the number of blocks). In particular, if we set the block size to $b = \sigma(\log n)^2$, we obtain the following result.

Theorem 5 *The FM-index with explicit boosting and blocks of fixed sizes can be implemented in $nH_k(T) + o(n)\log\sigma$ bits of space for any $k \le ((1-\varepsilon)\log_\sigma n) - 1$ and any constant $\varepsilon > 0$.*

Proof Using context block boosting with fixed context length k and RRR to compress the bitvectors, the size of the FM-index is $nH_k(T) + o(n)\log\sigma$ bits. When we switch from context blocks to fixed blocks with block size $b = \sigma(\log n)^2$, we must add two types of overhead. First, by Lemma 4, the total entropy increases by at most $\sigma^k b = \sigma^{k+1}(\log n)^2 \le n^{1-\varepsilon}(\log n)^2 = o(n)$ bits. Second, the space needed for everything else but the bitvector rank indexes is $O(\sigma\log n)$ bits per block. In total, this is $O(n/\log n) = o(n)$ bits. Thus the total increase in the size of the FM index is $o(n)$ bits. □

Thus, we have the same theoretical result as with context block boosting or implicit boosting.

The advantages of fixed block boosting compared to context block boosting are:

- To compute $\text{rank}_L(c,r)$, we have to find the block containing the position r. With fixed blocks this is simpler and faster than with varying size context blocks.

– Computing the optimal partitioning is complicated and expensive in practice. With fixed blocks, construction is much simpler and faster.

Explicit boosting (with either context blocks or fixed blocks) enables faster queries than implicit boosting for the following reasons:

– Compressed bitvector rank indexes are slower than uncompressed ones by a significant constant factor. Explicit boosting can achieve higher-order compression with Huffman-shaped wavelet trees allowing the use of the faster uncompressed rank indexes.
– With implicit boosting, i.e., with a single wavelet tree for the whole BWT, the average count query time for a pattern P is $\Theta(|P|\log\sigma)$ with a balanced wavelet tree and $\Theta(|P|H_0(T))$ with a HWT. With explicit boosting and HWTs, the average query time is reduced down to $O(|P|H_k(T))$.

5 Engineering the FM-index

In this section we describe a number of techniques used to obtain a small and fast FM-index based on the fixed block boosting principle.

We will focus on the details of rank queries over BWT. At the highest level, the index uses backward search without any changes, i.e., the number of pattern occurrences in a text is computed by repeatedly asking rank queries on the wavelet tree(s) of the BWT. Similarly, we do not assume any particular encoding of bitvectors in the wavelet trees, i.e., any of the compressed (RRR, hybrid), or uncompressed encoding could be plugged in, resulting in a different time-space tradeoff.

To implement the above approach, we use the SDSL library[1]. It contains a generic implementation of backward search as well as a number of different bitvector rank implementations.

5.1 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 appears in the block. Let $\Sigma_j = [0 \ldots \sigma_j)$ 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 $\gamma_j^{-1} : \Sigma_j \to \Sigma$.

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

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.

```
Algorithm WT-Rank(L, c, i)
1: $j \leftarrow \lfloor i \,/\, blocksize \rfloor$
2: $s_j \leftarrow j \times blocksize$
3: $c' \leftarrow \gamma_j(c)$
4: if $c' = 254$ and $\gamma_j^{-1}(254) \neq c$ then
5:     $c' \leftarrow 255$
6: return $R_j[c'] + \text{WT-Rank}(L_j, c', i - s_j)$
```

Fig. 5 Rank query over BWT implemented as a wavelet tree with the fixed block boosting technique. For simplicity we assume there are no superblocks.

The implementation assumes a byte alphabet no larger than 256, and uses the value 255 for $\perp$. 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 $\text{rank}_L(c,i)$ 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

$$\text{rank}_L(c,i) = \text{rank}_L(c,s_j) + \text{rank}_{L_j}(c,i-s_j)\,.$$

The first term is obtained from an array $R_j[0..\sigma_j)$, where $R_j[\gamma_j(c)] = \text{rank}_L(c,s_j)$. 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-$\perp$ value. Then

$$\text{rank}_L(c,i) = \text{rank}_L(c,s_k) = R_k[\gamma_k(c)]\,.$$

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

The pseudo-code of the rank function is given in Figure 5.

Superblocks. To reduce the space for alphabet mapping, 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. Then, we only need to store the mapping from the superblock alphabet to block alphabet for each block. Since often the superblock alphabet is significantly smaller than the global alphabet, with appropriate superblock size the space for the alphabet mapping (including the superblock mapping) is reduced. This additional level of indirection increases the query time but only very slightly, since the superblock headers are very small and thus tend to stay entirely in cache.

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.

5.2 Wavelet Tree

A good survey of wavelet tree implementation variants has been made by Claude et al. [5]. 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 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 $\mathrm{rank}_B(1,i)$ requires locating the starting position s of B in the concatenated bitvector $\widehat{B}$. Then

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

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 descent).

5.3 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 6. 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 2^{20}.

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 6 shows the effectiveness of this optimization. In the figure, the alphabet mappings need σ_s bytes per block and the block boundary ranks would be three times larger without the boundary rank optimization. For all files, the actual space is much smaller.

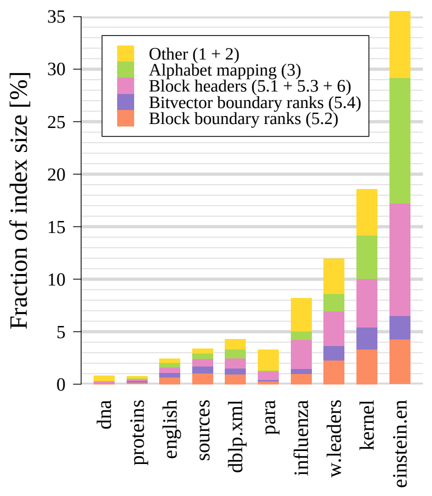

Fig. 6 The percentage of the total index size for individual components of the new wavelet tree (with hybrid bitvector as a bitvector representation) for different testfiles. The numbers refer to description in Section 5.3.

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.

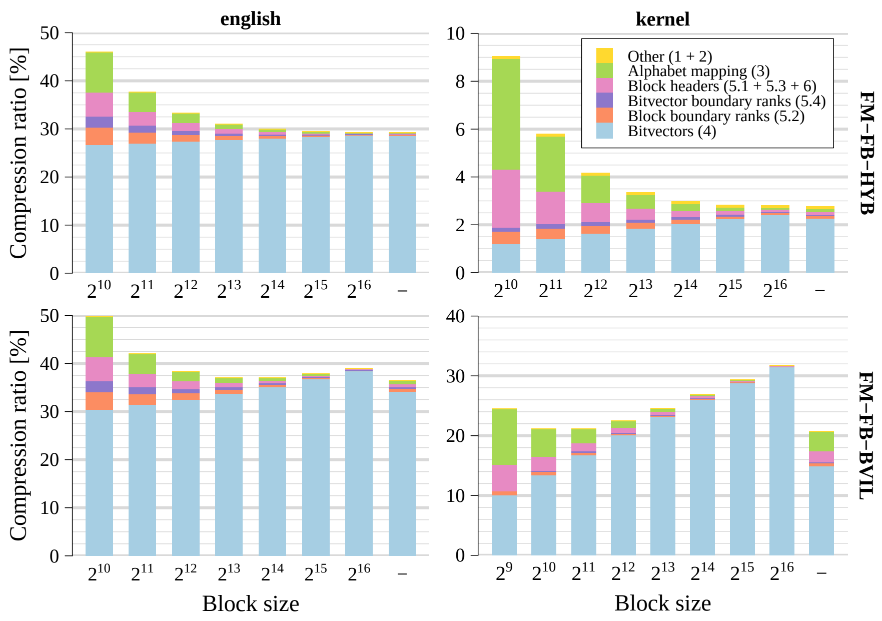

Fig. 7 The effect of the block size on the space used by the new wavelet tree using fixed block boosting technique on english text (example of non-repetitive data) and highly-repetitive kernel input data. FM-FB-HYB denotes the FM-index using hybrid vector to implement rank over bitvectors, and FM-FB-BVIL is the FM-index using fast uncompressed bitvector implementation. Also shown is the space usage of the data structure where each superblock uses optimal block size. The numbers refer to description in Section 5.3.

Construction. A major challenge when building the above data structure is computing the optimal block size. The tradeoff occurs between the compressibility of bitvectors and the overhead due to block headers. As demonstrated in Figure 7, the optimal block size depends on both the type of input data and the chosen bitvector rank implementation. Furthermore, all the block data structures within a superblock are implemented completely separately for each superblock and thus each superblock can use a different block size. As seen in Figure 7, such non-uniform encoding of superblocks can slightly improve the compression in some cases.

One possible solution when processing a superblock is to try multiple different block sizes and choose the most space efficient one. This approach requires computing and compressing all bitvectors for each tried block size and thus slows down the construction by a factor equal to the number of tested block sizes.

To speed up the construction we use the following observation. Consider two blocks of text X_1 and X_2 of equal length over the alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ and assume that X_1 consists of a roughly equal number of randomly interleaved a's and b's, and X_2 consists of similarly distributed c's and d's. Let B_1 and B_2 be the bitvectors associated with the root of the Huffman-shaped wavelet trees of X_1 and X_2. Consider now the Huffman-shaped wavelet tree of the concatenation X_1X_2. Clearly in this case it is a balanced binary tree, i.e., a root has two children, each of which has exactly two children. The possible cases are:

- Symbols `a` and `b` are in the same child of a root. Then `c` and `d` are the other child of a root and the bitvector associated with the root consists of two runs of 0's and 1's of total length $|X_1| + |X_2|$. The bitvectors associated with the children of the root are B_1 and B_2.
- Symbols `a` and `c` are in the same child of a root. Then `b` and `d` are in the other child of a root. The bitvector associated with the root is (up to a permutation of 0's and 1's) a concatenation of bitvectors B_1 and B_2, and the bitvectors associated with children of the root are uniform bitvectors of length $|X_1|$ and $|X_2|$.
- Symbols `a` and `d` are in the same child of a root. Then `b` and `c` are in the other child of a root. This situations is analogous to previous case.

In all cases, the bitvectors in the wavelet tree of X_1X_2 are: B_1, B_2, and two uniform bitvectors of length $|X_1|$ and $|X_2|$. In other words, the bitvectors in the wavelet tree of a concatenation consist of: (a) shuffled bitvectors from individual wavelet trees and (b) uniform bitvectors, the length of which depends only on the tree shape (and thus can be derived from the symbol frequencies). In particular after applying compression (e.g., RRR), the total size of (a)-bitvectors does not change when we concatenate X_1 and X_2, and the size of (b)-bitvectors is easy to estimate by knowing symbol distribution and how much a given compression method compresses uniform bitvectors.

While the above reasoning does not easily lend itself to a rigorous formalization, it prompted us to investigate the following heuristic-method for selecting the best block size for a given superblock.

We first estimate the compressibility α of a uniform block by taking a large block of zeros and computing its size after applying a given bitvector compression method. We then process all potential candidates for block sizes in increasing order. Suppose we are investigating blocks of size 2^k. For every block we maintain an array of σ integers storing the frequencies of symbols. This information together with α allows us to compute the total compressed size of (b)-bitvectors. After including the size of block headers we obtain the total space usage of the superblock encoding for a given block size *excluding* the (a)-bitvectors. From the discussion above those occupy roughly the same space independent of the block size and thus can be ignored when estimating the space.

To advance the computation to a 2^{k+1}-size block, we only need to update the symbols frequencies. This is easy to compute, since each block of size 2^{k+1} is obtained by merging two blocks of size 2^k for which we already computed the frequencies.

While heuristic, the above method works exceptionally well in practice and for all inputs and, for the compressed bitvector representations we tried, it finds the optimal block size or the difference in compression is negligibly small. The main benefit is that it only needs to compute and compress all bitvectors of the wavelet tree once, and thus achieves the construction speed of a single wavelet tree. We use it as a default construction method for the experiments in the next section.

6 Experimental Comparison

Setup. 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 15.04, 64bit) running kernel 4.4.0. All programs were compiled using g++ version 5.3.1 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. All given runtimes were recorded with the C++11 `high_resolution_clock` timer. The statistics of datasets used in the experiments are presented in Table 8.

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 8 Files used in the experiments. The files are from the Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl/`). 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.

Indexes. In our experiments we include the following indexes:

- **FM-FB-***: the new FM-index based on the fixed-block boosting technique. It uses our method for implementing multiple wavelet trees over a fixed-block partitioned BWT described in Section 5. This is the main contribution of this paper;
- **FM-HF-***: a standard FM-index [8], i.e., a Huffman-shaped wavelet-tree built for the single-piece BWT. This is the primary baseline in our experiments;
- **FM-VB-***: a version of our new FM-index based on fixed-block boosting modified to handle variable-size blocks within a superblock. We include it to allow for a proper comparison between the fixed-block and the context-block boosting.

To store bitvectors for each FM-index above we use (1) RRR compressed bitvectors (FM-*-RRR) using block sizes 15, 31, and 63 [30,27,14]; (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 [17] using a superblock size of 16 (FM-*-HYB).

In our experiments we also include the following state-of-the-art indexes:

- **AFINDEX**: the Alphabet-Friendly FM-index [9], the original implementation of the FM-index based on explicit context-block boosting technique. This is the second essential baseline in our experiments;
- **FM-RLMN**: an FM-index based on a run-length encoded BWT [23]. The index is designed to perform well for highly compressible strings, and uses a dedicated bitvector implementation optimized for sparse bitmaps [28];
- **RLCSA**: Run-Length Compressed Suffix Array [25]. An implementation of a compressed suffix array that has been optimized for highly repetitive inputs.

All structures not needed for count queries are excluded from space measurements. All code except AFINDEX (`http://pizzachili.dcc.uchile.cl/indexes`) and RLCSA (`http://iki.fi/jouni.siren/rlcsa`) is part of the SDSL library.

To allows for a proper comparison of the fixed-block and variable context-block boosting techniques, we included our own version of context-block boosted wavelet tree. Namely, we modified our new index based on fixed-block boosting to handle variable-size blocks within a superblock. We store block sizes in a list and during query time we scan that list to find the block that contains a given position. To limit these scans we have a threshold for the maximum number of blocks in a superblock, and hence also superblocks are of variable size. To locate the superblock containing a given position we store a list of superblock starting positions, enhanced with a lookup array that reduces the scanning time. In experiments we refer to this version of explicit block boosting as FM-VB. To build FM-VB index, we implemented the suffix-tree based construction algorithm from [6].

Experiments. To measure the performance of the indexes, we replicated the methodology of Ferragina et al. [7] which measures mean time to process one symbol during a single count query over 5×10^4 queries on 20-length patterns randomly extracted from the indexed text. The queries were rerun 30 times and numbers reported represent the time to process one character averaged over all runs and all queries.

The experimental results are presented in Figure 9 and Figure 10.

First we observe that the new fixed-block boosted FM-index achieves superior performance compared to a single wavelet tree, independently of the underlying bitvector rank implementation (RRR, BVIL, or HYB) and the type of input (the only exception is the DNA data which does not benefit from high-order compression). For example: (1) plugging the high-speed uncompressed BVIL bitvector to our FM-index sets a new speed record for count queries among FM-indexes, and (2) plugging the high-compression RRR or HYB bitvector allows reducing the space beyond what is achievable with a single wavelet-tree.

Next we compare the fixed-block boosted (FM-FB) and variable-block boosted FM-indexes (FM-VB, AFINDEX). First we observe that our implementation of variable-block boosting (FM-VB) achieves essentially the same speed as the original implementation (AFINDEX) but in nearly all cases uses many times less space: a factor of 2–3 on non-repetitive data and 4–10 for highly repetitive data. This improvement is due to: (1) more space-efficient bitvector implementation, and (2) a different encoding of multiple wavelet trees (Section 5).

The isolated effect of switching from variable-block into fixed-block boosting can be observed by comparing FM-VB and FM-FB. While the index size stays essentially the same, switching to fixed-size blocks significantly improves the query time of the medium-to-high speed bitvectors: for HYB by a factor 1.15–1.65, for BVIL by 1.6–2. The query time for RRR bitvector encoding improves by up to 10%. These speedups are due to the much simpler processing of the blocks in the fixed-block variant. The difference in speedup between bitvector representations confirm the intuition: rank queries over wavelet trees with RRR encoding are dominated by the bitvector rank operations, while rank queries over simpler bitvector encodings (BVIL, HYB) are much faster, and hence leave the room for upper-level (wavelet tree) optimizations.

The combined effect of our new method for encoding multiple wavelet trees together with switching to fixed-size blocks results in the combination superior for variable-block boosting both in time and space. Plugging the HYB encoding into the

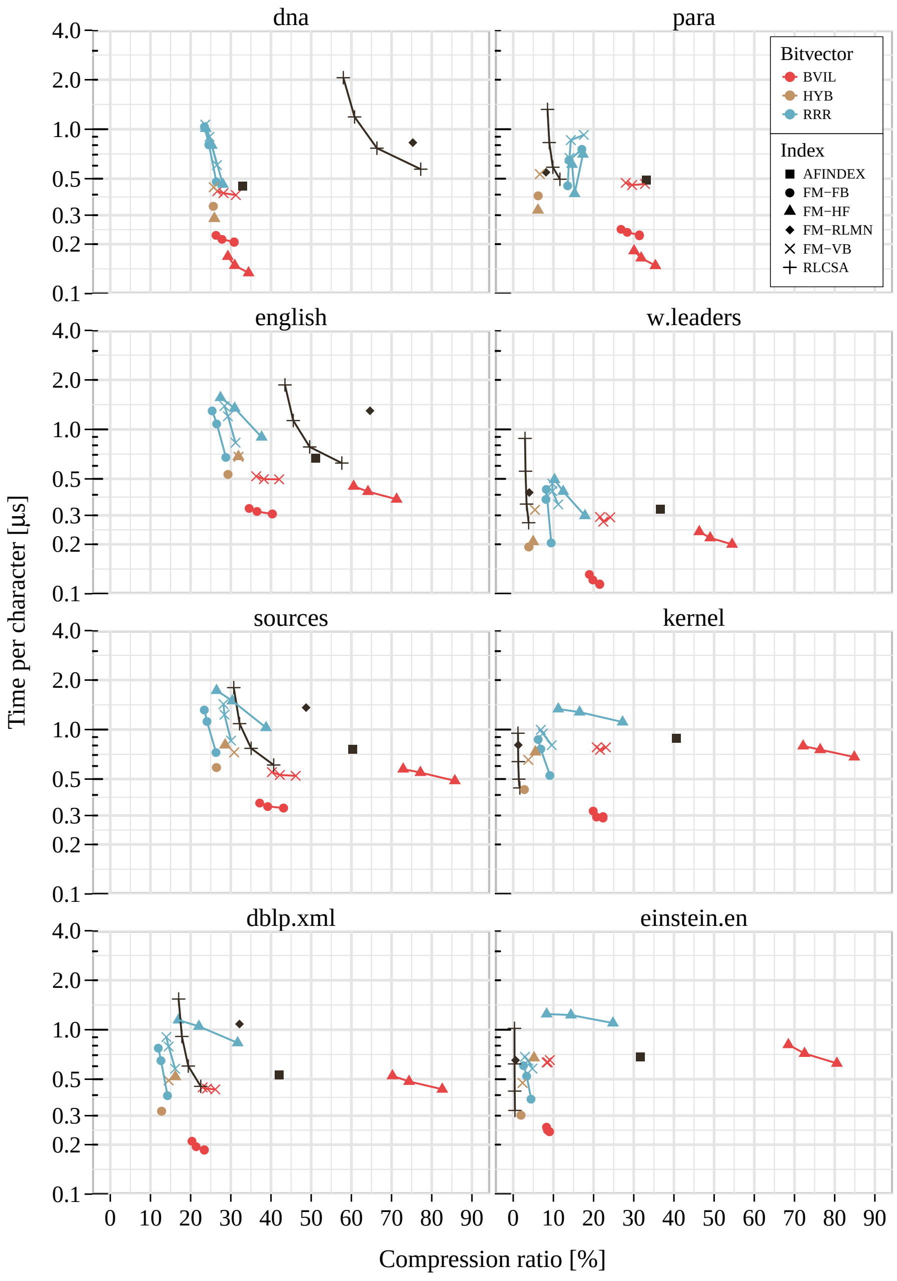

Fig. 9 Time/space tradeoffs on standard (left) and repetitive (right) collections for different index types. We measured the mean time in microseconds to process one symbol during a single count query over 5×10^4 queries on 20-length patterns extracted randomly from the indexed text. The queries were rerun 30 times and numbers reported represent the time in microseconds to process one character averaged over all runs and all queries. The space is given with respect to the original size of the input text.

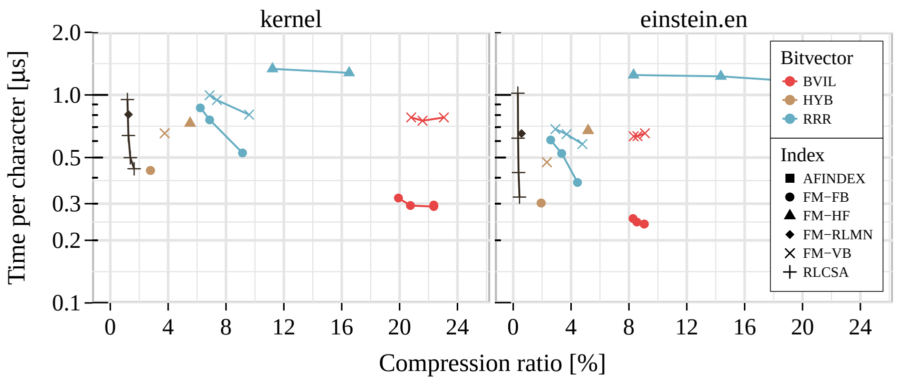

Fig. 10 Time/space tradeoffs for the two most repetitive testfiles. The graphs are as in Figure 9, except we zoom-in on the indexes achieving high compression.

new FM-index results in a particularly effective combination. The resulting index achieves all-around excellent compression and is almost always faster (and never much slower) than other indexes at the corresponding compression level. Furthermore, the construction of explicit variable-block boosted wavelet tree is slow and memory-consuming (due to the use of suffix tree), while the fixed-block wavelet tree admits a very simple and space-efficient construction.

Finally, we compare the performance of FM-FB to the two remaining specialized indexes: WT-RLMN and RLCSA. On extremely repetitive data (kernel, einstein.en), these indexes compress about three to four times better while offering similar speed (compare to FM-FB-HYB). However, on moderately repetitive inputs (para, w.leaders) they are usually already outperformed in query time and space, and on non-repetitive inputs they become much bigger and up to an order of magnitude slower.

7 Concluding Remarks

We have described new FM-index variants that are are small, have fast query times, and are easy to construct, both in theory and in practice. Experimentally we have shown the new indexes to be consistently fast and small on a wide range of data — they represent a new standard in FM-index based compressed text indexing.

There a numerous avenues for future work on FM indexes. Firstly, extending our new wavelet tree layouts, and especially their implementations, to deal with large alphabets would have immediate applications in natural language processing [29]. Perhaps more importantly, we note that a gap still exisits on highly-repetitive data between our indexes and indexes based on run-length encoding (for example, the RLCSA). Future work may aim at closing this gap, possibly via a combination of run-length encoding with the techniques described in this article.

Acknowledgements Preliminary versions of this article appeared as J. Kärkkäinen and S. J. Puglisi, “Fixed block compression boosting in FM-indexes”, Proc. SPIRE 2011 [19], and S. Gog, J. Kärkkäinen, D. Kempa, M. Petri, and S. J. Puglisi, “Faster, minuter”, Proc. DCC 2016 [13].

References

1. Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., Puglisi, S.J.: Bidirectional variable-order de Bruijn graphs. In: 12th Latin American Theoretical Informatics Symposium (LATIN 2016), Ensenada, Mexico, April 11-15, 2016, *LNCS*, vol. 9644, pp. 164–178. Springer (2016)
2. Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K.: Variable-order de Bruijn graphs. In: 2015 Data Compression Conference (DCC 2015), Snowbird, UT, USA, April 7-9, 2015, pp. 383–392. IEEE (2015)
3. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: 12th Workshop on Algorithms in Bioinformatics (WABI 2012), Ljubljana, Slovenia, September 10-12, 2012, *LNCS*, vol. 7534, pp. 225–235. Springer (2012)
4. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California (1994)
5. Claude, F., Navarro, G., Pereira, A.O.: The wavelet matrix: An efficient wavelet tree for large alphabets. Information Systems **47**, 15–32 (2015)
6. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM **52**(4), 688–713 (2005)
7. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics **13**, 1.12–1.31 (2009)
8. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4), 552–581 (2005)
9. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: 11th International Symposium on String Processing and Information Retrieval (SPIRE 2004), Padova, Italy, October 5-8, 2004, *LNCS*, vol. 3246, pp. 150–160. Springer (2004)
10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms **3**(2), Article 20 (2007)
11. Ferragina, P., Nitto, I., Venturini, R.: On optimally partitioning a text to improve its compression. Algorithmica **61**(1), 51–74 (2011)
12. Flicek, P., Birney, E.: Sense from sequence reads: Methods for alignment and assembly. Nature Methods **6**(11 Suppl), S6–S12 (2009)
13. Gog, S., Kärkkäinen, J., Kempa, D., Petri, M., Puglisi, S.J.: Faster, minuter. In: 2016 Data Compression Conference (DCC 2016), Snowbird, UT, USA, March 30 - April 1, 2016, pp. 53–62. IEEE (2016)
14. Gog, S., Petri, M.: Optimized succinct data structures for massive data. Software, Practice and Experience **44**(11), 1287–1314 (2014)
15. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: 14th Annual Symposium on Discrete Algorithms (SODA 2003), Baltimore, Maryland, USA, January 12-14, 2003, pp. 841–850. SIAM (2003)
16. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: Experiments with compressing suffix arrays and applications. In: 15th Annual Symposium on Discrete Algorithms (SODA 2004), New Orleans, Louisiana, USA, January 11-14, 2004, pp. 636–645. SIAM (2004)
17. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Hybrid compression of bitvectors for the FM-index. In: 2014 Data Compression Conference (DCC 2014), Snowbird, UT, USA, 26-28 March, 2014, pp. 302–311. IEEE (2014)
18. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-Ziv parsing in external memory. In: 2014 Data Compression Conference (DCC 2014), Snowbird, UT, USA, 26-28 March, 2014, pp. 153–162. IEEE (2014)
19. Kärkkäinen, J., Puglisi, S.J.: Fixed block compression boosting in FM-indexes. In: 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011), Pisa, Italy, October 17-21, 2011, *LNCS*, vol. 7024, pp. 174–184. Springer (2011)
20. Kempa, D., Puglisi, S.J.: Lempel-Ziv factorization: Simple, fast, practical. In: 15th Meeting on Algorithm Engineering and Experiments (ALENEX 2013), New Orleans, Louisiana, USA, January 7, 2013, pp. 103–112. SIAM (2013)
21. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: An improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)
22. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press (2015)
23. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nord. J. Comput. **12**(1), 40–66 (2005)

24. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: 14th International Symposium on String Processing and Information Retrieval (SPIRE 2007), Santiago, Chile, October 29-31, 2007, *LNCS*, vol. 4726, pp. 229–241. Springer (2007)
25. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010)
26. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM **48**(3), 407–430 (2001)
27. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: 11th International Symposium on Experimental Algorithms (SEA 2012), Bordeaux, France, June 7-9, 2012, *LNCS*, vol. 7276, pp. 295–306. Springer (2012)
28. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: 9th Meeting on Algorithm Engineering and Experiments (ALENEX 2007), New Orleans, Louisiana, USA, January 6, 2007, pp. 60–70. SIAM (2007)
29. Petri, M., Cohn, T.: Succinct data structures for NLP-at-scale. In: 26th International Conference on Computational Linguistics (COLING 2016), Osaka, Japan, December 11-16, 2016, pp. 20–21. ACL (2016)
30. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4) (2007)