# A Data-Aware FM-index *

Hongwei Huo† Longgang Chen† Heng Zhao† Jeffrey Scott Vitter‡ Yakov Nekrich§
Qiang Yu†

**Abstract**

In this paper we present some practical modifications of the higher-order entropy-compressed text indexing method of Foschini et al. [6] based upon the Burrows-Wheeler transform and the FM-index. Our method, called FM-Adaptive, applies a wavelet tree to the entire BWT. It partitions each bit vector of nodes in the wavelet tree into blocks and applies the hybrid encoding along with run-length Gamma code rather than the fixed-length code of [14] to each block while explores data-aware compression. FM-Adaptive retains the theoretical performance of previous work and introduces some improvements in practice. At the same time, broad experiments indicate that our index achieves superior performance, especially in terms of compression, in comparison to the state-of-the-art indexing techniques. The source code is available online.

## 1 Introduction.

Massive data sets are being produced at unprecedented rates from sources like the World-Wide Web, genome sequencing, XML, e-mail, satellite data, and business records. A larger part of the data consists of text in the form of a sequence of symbols representing not only natural language, but also music, program code, multimedia streams, biological sequences, and myriad forms of media. A full-text index is a data structure that stores a text (a.k.a. string) in preprocessed form so that it can support fast string matching queries. The best-known full-text indexes are the suffix tree [18, 25] and suffix array [17], which support pattern matching queries in optimal or almost-optimal time. Both structures use $O(n)$ words of storage, which is $O(n \log n)$ bits, which is larger than the raw text size $n|\Sigma|$ bits, where $\Sigma$ represents the text alphabet. It is substantially larger than the size of the text in fully compressed form, which we approximate by $nH_k(T)$, for some $k$, where $nH_k(T)$ represents the $k$th-order empirical entropy of the text $T$ of length $n$.

The field of compressed or succinct data structures attempts to build data structures whose space is provably close to the size of the data in compressed format and that still provides fast query functionality [12]. Theoretical breakthroughs about 15 years ago led to the development of a new generation of space-efficient search indexes. The compressed suffix array (CSA) [8–10, 22–24] and the FM-index [3–6] have been developed to achieve this desired goal of compressed text indexing, and their query time is proportional to the query pattern size plus the product of the output size and a small polylog function of $n$. The former maintains a succinct representation of the full suffix array, and the latter is based upon the Burrows-Wheeler Transform (BWT) [1] data compressor. Both are described in the appendix. Moreover, these indexes are self-indexes. That is, they provide random access to any part of the original text, and thus the text becomes redundant and can be discarded.

Simpler implementations for the CSA and FM-index achieve higher-order compression without explicit partitioning into separate contexts. In fact, the original BWT compressor was typically implemented by encoding the BWT-transformed text, $T_{bwt}$, with the move-to-front heuristics [3, 4]. Foschini et al. [6] proposed using a single wavelet tree (see §2.4) to encode the entire $T_{bwt}$ and CSA lists rather than using a separate wavelet tree for each partition or context. They showed that the wavelet tree analysis of the CSA and FM-index by Grossi et al. [8], which established the desired space bound of $nH_k(T)+o(n)$ bits, also applies within a factor of 2 when a single wavelet tree is applied to the entire sequence rather than having separate wavelet trees applied to individual contexts. The bit vector comprising each wavelet tree node is encoded using run-length encoding; every run is then represented using the Gamma code. Mäkinen and Navarro [15] made further theoretical improvements and eliminated the factor of 2 and achieved $nH_k(T)+o(n)$ bits of space. Huo et al. [13] proposed a new representation of the neighbor function $\Phi$ of the CSA and used a two-level scheme plus a lookup table to support fast queries. The survey by Navarro and

*Supported by NSFC grants 61173025 and 61373044, and RFDPC grant 20100203110010.

†Xidian University, Xi'an 710071, China. Hongwei Huo is the corresponding author. E-mail: hwhuo@mail.xidian.edu.cn.

‡The University of Kansas, Lawrence, Kansas 66047.

§University of Waterloo, Waterloo ON N2L 3G1, Canada.

Mäkinen [19] discusses index construction and other developments, and Ferragina et al. [2] report experimental comparisons.

The *rank* and *select* operations are basic building blocks of the FM-index, the CSA, and other succinct data structures. FM-indexes do not require the select operation and can focus on rank queries. There are many data structures for representing bit vectors so that rank queries can be answered efficiently, and wavelet trees generalize the notion of bit vectors so that each entry in the vector is from a multisymbol alphabet $\Sigma$. They can be classified into two main categories: the plain representation (uncompressed vectors) and the compressed representation. The former keeps the vector intact but uses a sublinear data structure on top of it, which usually leads to an amount of space similar to that of the uncompressed text. The latter exploits the compressibility of the vectors while supporting a rank operation. An example of the latter approach for bit vectors is the RRR method by Raman et al. [21], which requires a total of $nH_0(T) + o(n)$ bits for a bit vector $T$. The recent improvements on the *rank* and *select* operations on uncompressed bit vectors have been made by [7, 14, 20, 27].

In this paper we present some practical modifications of the higher-order entropy-compressed text indexing method of Foschini et al. [6], which encodes the Burrows-Wheeler transform using a single wavelet tree. Our method, called FM-Adaptive, partitions each bit vector of nodes in the wavelet tree into blocks and applies a hybrid encoding to reduce space usage while explores data-aware compression. FM-Adaptive retains the theoretical performance of previous work and introduces some improvements in practice. At the same time, broad experiments indicate that our index achieves superior performance, especially in terms of compression, in comparison to the state-of-the-art indexing techniques. The source code is available at `https://github.com/chenlonggang/Adaptive-FM-index`.

## 2 Preliminaries.

### 2.1 Empirical entropy and compressibility of a text.

Let $T$ be a string of $n$ characters from an alphabet $\Sigma$ of size $\sigma$. The empirical entropy of a text T provides a lower bound to the number of bits needed to compress $T$ using any compressor. The compression methods based upon Huffman coding assume that characters are independent and identically distributed (i.i.d.), thus their compression is close to the 0th-order empirical entropy of the text. The Huffman codeword length for each symbol is typical around 1.1–1.2$H_0$, where $H_0$ is the 0th-order empirical entropy of $T$, defined as follows:

$$H_0(T) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log \frac{n_i}{n}$$

where $n_i$ is the number of occurrences of character $i$ in $T$.

For most of data, the similar compressors are effective. However, the assumption that the characters in a text are randomly drawn (i.i.d.) is sometimes unrealistic in some situations, as a character may be closely related to its preceding $k$ characters. For example, for the text "*Th* White House*", the '*' is likely to represent the character '*e*'. That is, the '*' depends upon the characters around. For $T = abcdabcdabcd$, if we use a 0th-order compressor, then the compression ratio is close to $H_0$, which is 2 bits for each character. However, $T[i]$ depends heavily upon the $T[i+1]$ in this example, i.e., if $T[i] = a$, then we are certain $T[i+1] = b$. Thus we can achieve a greater compression if the codeword for each character uses the information of its preceding $k$ characters.

For any length-$k$ string $w \in \Sigma^k$, let $w_s$ is the string obtained by concatenating the single characters immediately following each occurrences of $w$ inside $T$. The $k$th-order empirical entropy of $T$ is defined as

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_s| H_0(w_s)$$

where $H_k(T)$ represents a lower bound to the compression we can achieve using codewords which depend upon the $k$ most recently seen characters.

Let's continue the above example, for $T = abcdabcdabcd$, we have

$$\begin{aligned} H_0(T) = & -(1/4)\log(1/4) - (1/4)\log(1/4) - \\ & (1/4)\log(1/4) - (1/4)\log(1/4) = 2 \\ H_1(T) = & (1/12)(3H_0(w_a) + 3H_0(w_b) + \\ & 3H_0(w_c) + 2H_0(w_d)) \\ = & (1/4)H_0(bbb) + (1/4)H_0(ccc) + \\ & (1/4)H_0(ddd) + (1/6)H_0(aa) = 0 \end{aligned}$$

and all higher-order empirical entropies of $T$ are 0. This means if we choose a character uniformly at random from $T$ to guess, then the uncertainty is 2. If we know the preceding character before we guess, then we are certain of the answer.

If we are computing any higher-order empirical entropy of a string $T$, it would be 0, since all the corresponding $H_0$ terms would be 0. We will see in the following sections that regardless whether the neighbor function $\Phi$ of the CSA or the BWT and $LF$ mapping of the FM-index, both link the current position to the

previous or following position of its neighbor, which provides not only the possibility of restoring the text, but also leads to a compression to the higher-order entropy of the text.

**2.2 The Burrows-Wheeler transform.** Let $T$ be a string of $n$ characters from an alphabet $\Sigma$ of size $\sigma$ and $P$ a query pattern of length $p$. The string $T$ has $n$ suffix, which starts at each of the $n$ locations in the text. The $i$th suffix, which start at position $i$, is denoted by $T[i..n]$. The suffix array $SA[1..n]$ of $T$ is an array of $n$ integers that gives the sorted order of the suffixes of $T$. That is, $SA[i] = j$ if $T[j..n]$ is the $i$th smallest suffix of $T$ in lexicographical order. Similarly, the inverse suffix array is defined by $SA^{-1}[j] = i$. All the suffixes prefixed by $P$ occupy a contiguous range in the sorted array $SA$.

In the CSA [9, 10], the suffix array values are indirectly encoded by instead storing the $\Phi$ function, where $\Phi(i) = SA^{-1}[SA[i] + 1]$. The $\Phi$ function can be compressed into optimal space (in entropy sense), and then each $SA$ value can be computed by referring to a small portion of the $\Phi$ function in $O(\text{polylog}n)$ time. More recently, Huo et al. [13] gave a practical implementation of the CSA by encoding the differences $\Phi(i) - \Phi(i-1)$ using Elias's Gamma coding, in which they used a remarkable property of $\Phi$, that is, an increasing sequence of positions within each $\Sigma$ list.

| $i$ | $SA$ | $\Phi$ | $LF$ | $F$ | | $L$ |
|---|---|---|---|---|---|---|
| 0 | 7 | 3 | 5 | # | *abaaba* | *b* |
| 1 | 2 | 4 | 6 | *a* | *abab#a* | *b* |
| 2 | 5 | 5 | 7 | *a* | *b#abaa* | *b* |
| 3 | 0 | 6 | 0 | *a* | *baabab* | # |
| 4 | 3 | 7 | 1 | *a* | *bab#ab* | *a* |
| 5 | 6 | 0 | 2 | *b* | *#abaab* | *a* |
| 6 | 1 | 1 | 3 | *b* | *aabab#* | *a* |
| 7 | 4 | 2 | 4 | *b* | *ab#aba* | *a* |

Figure 1: Example of the BWT of $T = abaabab\#$.

The Burrows-Wheeler transform (BWT) of $T$ is an invertible permutation of $T$, denoted by $L$, such that $L[i]$ is the character in the text just preceding the $i$th lexicographically smallest suffix of $T$. That is, $L[i] = T[SA[i] - 1 \mod n]$. Intuitively, the sequence $L[i]$ is easier to compress because adjacent characters often share higher-order contexts, and thus space can be reduced even further to about $nH_k$ bits. The $LF$ function [3] stands for last-to-first column mapping since the character $L[i]$ in the last column of Figure 1 is located in the column $F$ at position $LF(i)$, i.e., $L[i] = F[LF(i)]$. In the example of Figure 1, $L[6]$ and $F[LF(6)] = F[3]$ both correspond to the third $a$ in the string $abaabab\#$. Thus we can walk backwards through the text $T$ using the function $LF$. That is, if $T[k] = L[i]$, then $T[k-1] = L[LF(i)]$.

The FM-index and the CSA are closely related: the $LF$ function and the CSA neighbor function $\Phi$ are inverses of one another. That is, $SA[LF(i)] = SA[i] - 1$; equivalently $LF(i) = SA^{-1}[SA[i] - 1] = \Phi^{-1}(i)$.

Since the BWT does not change the distribution of characters, the 0th-order empirical entropy of $T$ remains the same. However, it tends to move characters with similar contexts close together and thus the resulting string $L$ has a good locality (being consecutive piecewise in the lexicographic order as shown in Figure 1) which makes the move-to-front or wavelet tree transformation more effective.

**2.3 The FM-index.** Ferragina and Manzini introduced the elegant FM-index [3, 4], based upon the Burrows-Wheeler transform. The FM-index was the first self-index shown to have both fast performance and space usage within a constant factor of the desired entropy bound for constant-sized alphabets. The core problem of the FM-index is to provide a compressed representation of $L$ together with some auxiliary structures which makes it possible to compute the $LF$ mapping efficiently. $LF(i) = C[L(i)] + Occ(i, L(i)) - 1$, where $Occ(i, c)$ is the number of occurrences of character $c$ in the prefix $L[0, i]$, and $C[]$ is the array of length $\sigma + 1$ such that $C[c]$ is the total number of text characters which are alphabetically smaller than $c$. $C = \{0, 1, 5, 8\}$ for the example in Figure 1.

FM-index is based upon a procedure called *backward search* [4, 10], which finds the range of rows in the BWT matrix $M$ (the last three columns in Figure 1) that begin with a given pattern $P$. This range represents the occurrences of $P$ in $T$, which answers the *count* query (returns the number of occurrences of $P$ in $T$). Thus by backward search we turn the count query on $T$ into a sequence of *rank* queries on $L$. With a slight extension, we can implement the *locate* and *extract* queries, where *locate* reports the occurring positions of $P$ in $T$ and *extract* displays the text substring $T[start, start + len - 1]$, given *start* and *len*.

The *count* algorithm describes the backward search-based counting operation in which the **while** loop performs $p$ iterations from $p-1$ to 0,where $p$ is the length of pattern $P$. The algorithm maintains the following invariant: after the $k$ iterations, the variable $l$ points to the first row of the $M$ prefixed by $P[p-k, p-1]$, and the variable $r$ points to the last row of the $M$ prefixed by $P[p-k, p-1]$. After the $p$ iterations, $occ = r - l + 1$, the number of occurrences of $P$ in $T$.

*count*($P$, $l$, $r$)
Input: $P$
Output: $l$ and $r$
1 $i \leftarrow p-1$, $c \leftarrow P[p-1]$
2 $l \leftarrow C[c]$
3 $r \leftarrow C[c+1]-1$
4 **while** ($l \le r$ *and* $i \ge 1$) **do**
5 $c \leftarrow P[i-1]$
6 $l \leftarrow C[c] + Occ(c, l-1)$
7 $r \leftarrow C[c] + Occ(c, r) - 1$
8 $i \leftarrow i-1$
9 **if** $l > r$ **then**
10 **return** "pattern does not exist"
11 **else**
12 **return** $l$ and $r$

It is obvious that the running time of the *count* algorithm depends upon how the value of *Occ* is computed. By means of the wavelet tree [6, 8, 11] representation of $L$, $Occ(c, j)$ $(= rank_L(c, j))$ can be answered by performing a sequence of *rank* queries over bit vectors of the wavelet tree, that is, a traversal from the root to the leaf labeled by $c$. Thus we can retrieve character $L(i)$ and compute the number of occurrences $Occ(i, c)$ in $O(\log \sigma)$ time, provided constant-time responses to the *rank* query on the bit vectors and a (nearly) balanced wavelet tree. Therefore, the *count* algorithm runs in $O(p \log \sigma)$ time.

For $T = abaabab\#$ in Figure 1, we use pattern $P = ab$ as an example to show how the *count* algorithm works. After initialization in line 1, $i = 1$, $c = b$, $l = C[b] = 5$, $r = C[b+1]-1 = 7$. For the first iteration of the **while** loop, $c = P[0] = a$, $l = C[a] + Occ(a, 4) = 1 + 1 = 2$, $r = C[a] + Occ(a, 7) - 1 = 4$, $i = 0$. The *count* returns $l = 2$ and $r = 4$.

The *count* algorithm reports an interval in which the corresponding suffixes in $T$ are prefixed by the pattern $P$. Thus, for each value $i$ in this interval, we use the procedure *getpos*2($i$) to find the position in $T$ of the suffix, i.e. the value $(SA_l[i] + step) \mod n$ returned by *getpos*2($i$). The *locate* algorithm invokes the *getpos*2($i$) for each value in $[l, r]$, given as follows.

*locate*($P$, *ans*)
Input: $l$ and $r$
Output: *ans*
1 $ans[0..r-l] \leftarrow 0$
2 **for** $i \leftarrow l$ **to** $r$ **do**
3 $ans[i-l] \leftarrow getpos2(i)$
4 **return** *ans*

*getpos2*($i$)
1 $step \leftarrow 0$
2 **while** $(i \mod d) \neq 0$ **do**
3 $i \leftarrow LF(i)$
4 $step \leftarrow step + 1$
5 $i \leftarrow i/d$
6 **return** $(SA_l[i] + step) \mod n$

For $i = l \cdots r$, we can walk the text $T$ using $LF$ along indices $i'(LF(i)), i''(LF(i')), \cdots$, such that $SA[i] - 1 = SA[i'], SA[i'] - 1 = SA[i'']$, and so on, until a sampled position, $SA_l$, is met. If we sample the suffix array at a regular text position interval $d = \Theta((\log n)^2 / \log\log n)$, then the **while** loop is executed $O((\log n)^2 / \log\log n)$ times. Since $LF(i)$ can be answered via $O(\log \sigma)$ computations of $L()$ and $Occ()$, we have that *getpos*2 takes $O(\log \sigma((\log n)^2 / \log\log n))$ time. Thus, the *locate* algorithm runs in $O(occ \log \sigma((\log n)^2 / \log\log n))$ time. Sampling suffix array at every $(\log n)^2 / \log\log n$ points takes $\Theta(n \log\log n / \log n)$ bits of space.

We continue to have pattern $P = ab$ as an example to show how the *locate* algorithm works. As we have seen, the input for *locate* is $l = 2$ and $r = 4$. The range $[l, r]$ contains all the occurrences of $ab$ in $T$ in the example of Figure 1. *locate* invokes *getpos*2 for $i = 2$ to 4. Let $d = 4$. We use $i = 2$ to illustrate. After performing line 1, $step = 0$, then we judge the **while** condition $(2 \mod 4) \neq 0$, updating $i = LF(2) = 7$ and $step = 1$. Continuing the loop, $(7 \mod 4) \neq 0$, updating $i = LF(7) = 4$ and $step = 2$, At the time, the **while** loop exits, since the loop condition is not satisfied. Then after doing line 5, we get $i = 4/d = 4/4 = 1$. And the *getpos*2 returns $(SA_l[i] + step) \mod n = SA_l[1] + 2 = 5$, one of the positions where the pattern $ab$ occurs in $T$ shown in Figure 1.

*extract*(*start*, *len*, *seq*)
Input: *start* and *len*
Output: *seq*
1 $end \leftarrow start + len - 1$
2 $anchor \leftarrow (n - 2 - end)/e$
3 $step \leftarrow (n - 2 - end) \mod e$
4 $i \leftarrow SA_l^{-1}[anchor]$
5 **for** $j \leftarrow 0$ **to** $step - 1$ **do**
6 $i \leftarrow LF(i)$
7 **for** $j \leftarrow 0$ **to** $len - 1$ **do**
8 $seq[len - j - 1] \leftarrow L(i)$
9 $i \leftarrow LF(i)$
10 **return** *seq*

The *extract* algorithm displays substring $T[start..start + len - 1]$ of length *len* in $T$. If we sample the inverse suffix array at every $e = \Theta((\log n)^2 / \log\log n)$, the **for** loop in line 5

is executed $O((\log n)^2/\log\log n)$ times. The **for** loop in line 7 is executed *len* steps to collect the text characters. Thus, the *extract* algorithm takes $O((len + (\log n)^2/\log\log n)\log\sigma)$ time.

For $T = abaabab\#$ in Figure 1, we give an example to show how the *extract* algorithm works. Assume that the inverse suffix array sampling $SA_l^{-1}[] = [0, 7, 6]$ and the sampling size $e = 3$. We want to display $T[2..4]$ for $start = 2$ and $len = 3$. After performing the first three lines, we have $end = 4$, $anchor = 0$, and $step = 2$. After performing the **for** loop in line 5, we have $i = 2$. Then we go into **for** loop in line 7. After the first iteration, we $seq[2] = L(2) = b$ and $i = LF(2) = 7$; next $seq[1] = L(7) = a$ and $i = LF(7) = 4$; and finally $seq[0] = L(4) = a$ and $i = LF(4) = 1$. Thus $T[2..4] = aab$.

THEOREM 2.1. *Let $T$ be a string of $n$ characters from an alphabet $\Sigma$ of size $\sigma$ and $P$ a query pattern of length $p$.*

*(i) We can implement count query in $O(p\log\sigma)$ time.*

*(ii) We can implement locate query in $O(occ((\log n)^2/\log\log n)\log\sigma)$ time, where occ is the number of the occurrences of $P$ in $T$.*

*(iii) We can implement extract query in $O((len + (\log n)^2/\log\log n)\log\sigma)$ time, where len is number of characters to display.*

**2.4 The Wavelet tree.** The wavelet tree, an elegant data structure introduced by Grossi et al. [6, 8, 11], has become a key tool in modern full-text indexing and data compression [12]. It supports *access*, *rank*, and *select* queries on arrays of characters from $\Sigma$. A member query $access(i)$ reports the character at a given position $i$ of the array. A rank query $rank(c, i)$ counts how many times character $c$ occurs in the first $i$ positions of the array. A select query $select(c, j)$ returns the location of the $j$th occurrences of $c$.

The wavelet tree is conceptually a binary tree (often a balanced tree) in which each node consists of a vector. The root node is a bit vector of the same length as the input text and partitions the alphabet into two sets, the first half and the second half. The $i$th bit is 0 (resp., 1) if the $i$th entry is an element of the first half (resp., second half) of the alphabet, in which case it is recursively represented in the left subtree (resp., right subtree). Such a bit vector representation happens recursively at each internal node; the text at the internal node consists of the characters dispatched from the parent node, with their order in the text preserved. The collective size of the bit vectors at any given level of the tree is bounded by $n$, and they can be stored in compressed format, giving the 0th-order entropy space bound [8, 11].

## 3 Data-aware FM-index.

**3.1 The index structure for bit vectors.** Let $T$ be a string of $n$ characters from an alphabet $\Sigma$ of size $\sigma$. Applying the wavelet tree introduced by Grossi et al. [8] to the BWT of $T$, which we denote $L$, we obtain a wavelet tree representation of $L$, denoted by $WT(L)$. We conceptually concatenate the bit vectors at any given level of the tree into a single one, denoted by **B**, and then divide it into blocks of size $b$ and compress each independently by choosing one from three types of compression methods (Plain, RLG0/1, All0/1) so that the compression has the minimum space. Plain means to represent the block intact. RLG0 (resp., RLG1) encodes the block in run-length Gamma coding, in which the first run is a 0-run (resp., 1-run). (We use 0-run to mean a run of 0s.) All0 (resp., All1) means that the block consists entirely of a 0-run (resp., 1-run).

The idea of choosing among multiple encodings comes from Kärkkäinen et al. [14], but we have made several modifications. For example, we do not use fixed-length coding for each run but rather Gamma coding using blocks of size $b$. Moreover, our block size is dependent upon the statistics of the input data.

To save further space, we combine a constant number of blocks into a superblock. A bit vector **B** in $WT(L)$ is represented by six structures: $S$, $SBrank$, $SB$, $Brank$, $B$, $Header$, where $S$ is the bit vector **B** stored in the compressed form (called the encoded sequence), $SBrank$ stores the cumulative sum of 1s preceding the current superblock in **B**, $SB$ stores the offset of a superblock in the encoded sequence $S$ (i.e., the total number of bits preceding the current superblock), $Brank$ stores the cumulative sum of 1s preceding the current block inside its superblock, $B$ stores the offset relative to its superblock, and $Header$ stores the types of the encoding methods (each with at most three bits).

Figure 2 shows an example of the structures with the bit vector size $n = 84$, the block size $b = 12$, and superblock size $sb = 36$. We choose one encoding method for each block. For the block containing only 0s or only 1s, we store nothing, since we can identify them from the header labeled with All0 or All1.

To obtain $rank(i)$, we first query $SB$ and $B$ to determine the position (*offset*) of the block in $S$ to which $i$ belongs, then access the $SBrank$ and $Brank$ (base rank) to obtain the total number of occurrences of 1s preceding the block, and then decode starting from the *offset* in the $S$ to obtain the number of 1s of the first $((i+1) \mod b)$ bits (called local rank, denoted by $lrank$) within the block. It is added to the base rank to

| **B** | 001101001010 | 000000000000 | 111111111111 | 000000111111 | 111111000000 | 011111111100 | 011100100000 |
|---|---|---|---|---|---|---|---|
| *Header* | Plain | All0 | All1 | RLG0 | RLG1 | RLG0 | Plain |
| *S* | 001101001010 | null | null | 0011000110 | 0011000110 | 10001001010 | 011100100000 |
| *SBrank* | 0 | | | 17 | | | 38 |
| *Brank* | 0 | 5 | 5 | 0 | 6 | 12 | 0 |
| *SB* | 0 | | | 12 | | | 43 |
| *B* | 0 | 12 | 12 | 0 | 10 | 20 | 0 |

Figure 2: The index structure for the bit vector $\mathbf{B}$ ($n = 84$, $b = 12$, $sb = 36$).

obtain the final value of $rank(i)$, defined as follows.

$$rank(k) = SBrank[\lfloor (i+1)/sb \rfloor] + Brank[\lfloor (i+1)/b \rfloor] + lrank(S, offset, header[\lfloor i/b \rfloor], (i+1) \mod b)$$

where $sb$ and $b$ is the superblock size and block size, respectively. The $lrank$ operation returns the number of 1s of the first $((i+1) \mod b)$ bits within the block in $S$. The starting position for decoding is determined by the parameter *offset*, which is $SB[(i+1)/sb] + B[(i+1)/b]$. $(i+1) \mod b$ is the number of bits to decode.

For example, to solve $rank(66)$, we compute the starting decoding position in $S$: $offset = SB[67/36] + B[67/12] = SB[1] + B[5] = 12 + 20 = 32$, then compute the base rank: $SBrank[67/36] + Brank[67/12] = SBrank[1] + Brank[5] = 17 + 12 = 29$. We know that the first run in the encoded block is a 0-run since $header[66/12] = header[5] =$ RLG0. The number of bits needed to decode is $(i+1) \mod b = 67 \mod 12 = 7$ bits. We start to decode at the 32th position of $S$. We get #bits = 1 for the first decoding and the corresponding run is a 0-run. The number of bits to decode becomes 6 bits. We get #bits = 9 for the second decoding, and the corresponding run is a 1-run and $9 > 6$. Thus the number of 1s of the first 7 bits in the block is 6. Therefore, $rank(66) = 29 + 6 = 35$.

**3.2 Space usage.** Let $B_i$ $(i = 1, 2, \cdots, t = \sigma - 1)$ denote the bit vector of internal node $i$ of the wavelet tree, and $|B_i|$ denote the number of bits contained in $B_i$. We divide the bit vector $B_i$ into blocks, denoted by $B_i^j$, of size $b = O((\log n)^2 / \log\log n)$. $B_i^j$ can be viewed as a sequence of maximal runs of identical bits $B_i^j = b_1^{l_1} b_2^{l_2} \cdots b_m^{l_m}$ for some $m \le |B_i^j|$, where $b_i \neq b_{i+1}$ for $1 \le i < m$. The run-length Gamma coding of $B_i^j$ is the binary string $B_{i|rle,\gamma}^j = b_1 \gamma(l_1)\gamma(l_2)\cdots\gamma(l_m)$ where $b_1$ is a single extra bit necessary for decoding. The run-length Gamma encoded wavelet trees are the ones whose bit strings are run-length Gamma encoded. We compress each block $B_i^j$ by choosing the compression method that minimizes the encoding length $h(B_i^j)$ from three types of compression methods: Plain, RLG0/1, and All0/1. Obviously, $|h(B_i^j)| = \min\{|B_{i|rle,\gamma}^j|, |\text{Plain}(B_i^j)|\}$, since the blocks labeled with All0/1 store nothing. Therefore, the run-length hybrid encoded wavelet trees for the coding scheme are the ones whose bit strings are run-length hybrid encoded.

LEMMA 3.1. ( [11]) $|B_{rle,\gamma}| \le 2nH_0(B) + 2\log n + 2$

LEMMA 3.2. ( [8]) $\sum_{i=1}^{t} |B_i| H_0(B_i) = nH_0(T)$

Grossi et al. [8] introduced the wavelet tree for reducing the redundancy inherent in maintaining separate dictionaries for each symbol appearing in the text. They partition the $L$ into a table of lists $y$ (columns) and context $x$ with length $k$ (rows). All the entries across the row of a context $x$ correspond to a contiguous piece of $L$, that is, some $L_s$. A wavelet tree is built over each table row of context $x$ so that the sum of 0th-order entropies over all contexts achieves the $h$th-order entropy, $nH_h$ [8]. Note that the collection of all $L_s$ $(s = 1, 2, \cdots, \sigma^k)$ corresponding to some context $x$ is precisely $L$.

THEOREM 3.1. *We can encode $L$, the BWT of $T$, with a wavelet tree and run-length Gamma encoding using $2nH_k(T) + o(n)\log\sigma$ bits of space for any $k \le c\log_\sigma n - 1$ and any constant $c < 1$.*

*Proof.* The bits corresponding to $L_s$ regarding context $x$ form a substring (hereafter known context block $X_i^s$) of the bit vectors at each node of the wavelet tree, since the corresponding positions are mapped in the left and right child, thus the order is preserved. For blocks $B_i^j$ of length $b$ of node $i$ in the wavelet tree, we classified them into two categories: one is the blocks that are within context block $X_i^s$, and the other is the ones that cross the context block. Assume that the context block $X_i^s$ contains $t_i$ such complete blocks, so the concatenation, $B_i^1 B_i^2 \cdots B_i^{t_i}$, of these $t_i$ blocks is a substring of $X_i^s$ of length $bt_i$.

Let $S_{i,r}$ denotes the set of blocks contained in $X_i^s$ of node $i$ and coded in run-length Gamma when

$|B^j_{i|rle,\gamma}| < b$; $S_{i,p}$ denotes the set of blocks contained in $X^s_i$ of node $i$ and coded in Plain when $|B^j_{i|rle,\gamma}| \geq b$; $S_a$ denotes the set of blocks contained in $X^s_i$ of node $i$ and consisted of all 0s or 1s. Obviously, $|S_{i,r}| + |S_{i,p}| \leq t_i$.

Using the coding scheme, the space required by the substring $B^1_i B^2_i \cdots B^{t_i}_i$ in compressed form (run-length Gamma encoding) is

$$\begin{aligned}
\sum_{j=1}^{t_i} h(B^j_i) &= \sum_{j=1}^{t_i} \min\{|B^j_{i|rle,\gamma}|, |\text{Plain}(B^j_i)|\} \\
&= \sum_{k\in S_{i,r}} |B^k_{i|rle,\gamma}| + \sum_{j\in S_{i,p}} |\text{Plain}(B^j_i)| \\
&\leq \sum_{k\in S_{i,r}} |B^k_{i|rle,\gamma}| + \sum_{j\in S_{i,p}} |B^j_{i|rle,\gamma}| \\
&= \sum_{j\in S_{i,r}\cup S_{i,p}} |B^j_{i|rle,\gamma}| \\
&\leq \sum_{j\in S_{i,r}\cup S_{i,p}} (2bH_0(B^j_i) + 2\log b + 2) \\
&\leq 2|X^s_i|H_0(X^s_i) + 2t_i \log b + 2t_i
\end{aligned}$$

The first inequality is due to the fact that $|\text{Plain}(B^j_i)| \leq |B^j_{i|rle,\gamma}|$ when $j \in S_{i,p}$. The second inequality is due to Lemma 3.1. The third inequality is due to Lemma 3.2 and $|S_{i,r}| + |S_{i,p}| \leq t_i$. Here we do not count the space of the first bit $b_1$ (a single bit necessary for decoding) of the run-length Gamma coding of $B^j_i$, since we do not store the bit in run-length Gamma coding of $B^j_i$ but store it in the *Header* structure.

Notice that these context blocks $X^s_i$ are precisely those that would result if we built the wavelet tree for $L_s$ regarding context $x$. By summing over all the $(\sigma-1)$ internal nodes in the wavelet tree and over all contexts of length $k$, we proceed in evaluating our main inequality as

$$\begin{aligned}
&\sum_{s=1}^{\sigma^k}\sum_{i=1}^{\sigma-1}(2|X^s_i|H_0(X^s_i) + 2t_i\log b + 2t_i) \\
&\leq \sum_{s=1}^{\sigma^k}(2|L_s|H_0(L_s) + 2|L_s|\log\sigma\log b/b + 2|L_s|\log\sigma/b) \\
&= 2nH_h(T) + O(n\log\sigma(\log\log n)^2/(\log n)^2 + \\
&\quad n\log\sigma(\log\log n)/(\log n)^2 \\
&= 2nH_h(T) + o(n)\log\sigma
\end{aligned}$$

The first term in the inequality is due to Lemma 3.2 and the last two terms are due to the fact that $bt_i \leq |X^s_i|$ and $|X^s_i|$ lengths add up $|L_s|$ at each level of the wavelet tree. The first term in the first equality is due to [8] as described in the paragraph just before Theorem 3.1 in this Section and the last two terms are due to the fact that $\sum_{s=1}^{\sigma^k} L_s = n$ and $b = O((\log n)^2/\log\log n)$.

Moreover, we must count the space required by the blocks that across the context block $X^s_i$, which is $O(\sigma^{k+1}\log n)$ bits, since there can be at most two such blocks for a context block of a node in wavelet tree and each takes $\log n$ bits in the worst case, and there are total of $\sigma^k$ contexts, and at most $(\sigma-1)$ internal nodes in the wavelet tree, which is $o(n)$ for any $k \leq c\log_\sigma n - 1$ and any constant $c < 1$.

The space required by the $SB$, $SBrank$, $B$, and $Brank$ is $\frac{2n}{sb}\log n + \frac{2n}{b}\log(sb)$ bits, which is $o(n)$ bits for $b = O((\log n)^2/\log\log n)$ and $sb = O((\log n)^2)$. It is obvious that the *Header* takes $3n/b$ bits, which is $o(n)$ bits of space, since there are five encoding methods, each with at most three bits. Thus, the total space required by these five structures is $o(n)$.

By summing over all the space needed by all pieces, we get the space required by the whole structure, which is $2nH_k(T) + o(n)\log\sigma$. This completes the proof of Theorem 3.1.

**3.3 Speedup for rank queries.** Inspired by the storage techniques used in [26], we store $SBrank$ and $SB$ in an interleaved manner. The same technique is applied to store $Brank$ and $B$. Thus the memory access times is reduced from 4 to 2 when answering a rank query. Among the three types of encoding methods, it is simple to decode the case All0/1. For Plain, we use the solution described in [26] to obtain the number of 1s in the Plain. For RLG0/1, we use a lookup table similar to [13] to optimize the computation of the $lrank$ operation defined in §3.1, since RLG0/1 accounts for a high percentage of the three types of encoding methods.

We build two kinds of lookup tables, Z and R: Z is used to speed up the counting of the number of 0s at the beginning of a bit string, and R is used to quickly compute the value of the $lrank$ operation. As we know, to decode Elias's Gamma encoded sequence for an integer, we need first to scan and count the number of 0s, say $x$, at the beginning of the bit vector; if $x$ is zero, then the decoded number is 1; otherwise, the decoder reads the following $x+1$ bits and decodes the corresponding bits to get the integer. We can create a table, Z, of size $W = O(\log n/2)$ to accelerate the decoding. It stores the length of the 0-run at the beginning of the bit string of length $W$. We can obtain the value $x$ directly by accessing to the table once, if the table size is less than $\sqrt{n}$. We can store Z in $\sqrt{n}\log\log\sqrt{n}$ bits, since it contains $2^W$ entries and each needs at most $\log W$ bits.

Now we show how a run-length Gamma encoded bit string is constructed. Assume that we are given a raw bit string $t = 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1$. We use run-length

encoding to encode $t$ and have 1 2 1 2 3 1 1, and then we use Gamma encoding to encode the integer sequence and have $s = 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1$. If we want to obtain the raw bit string $t$, we have to do the decoding on $s$.

On the other hand, the table R (as shown in Table 1) consists of four parts, denoted $R_1$, $R_2$, $R_3$ and $R_4$, to provide fast decoding and computation of $lrank$. The $s$ in first column of Table 1 is a run-length Gamma encoded bit string.

$R_1$ stores the number of the complete Gamma encodings contained in a bit string $s$ of length $W$ (i.e., the number of runs in the raw bit string). By similar analysis as above, we know that $R_1$ uses $\sqrt{n}\log\log\sqrt{n}$ bits of space.

$R_2$ stores the cumulative sum of complete Gamma decoded numbers in a bit string $s$. It is the number of bits of the raw bit string. The space required by $R_2$ is $\frac{1}{4}\sqrt{n}\log n$ bits, since it contains $2^W$ entries and each needs at most $W/2$ bits.

$R_3$ stores the total number of complete Gamma decoded bits in a bit string $s$. Obviously, $R_3$ requires $\sqrt{n}\log\log\sqrt{n}$ bits of space.

$R_4$ stores the rank value of the raw bit string (i.e., the number of 1s in the raw bit string) assuming that the first run in the string is 1-run. For example, if the raw bit string is $1^{l_1}0^{l_2}1^{l_3}0^{l_4}1^{l_5}$, which can be completely Gamma encoded in the form $\gamma(l_1)\gamma(l_2)\gamma(l_3)\gamma(l_4)\gamma(l_5)$, the rank value we store in $R_4$ is $l_1+l_3+l_5$. $R_4$ requires $\frac{1}{4}\sqrt{n}\log n$ bits of space, since each entry needs at most $W/2$ bits.

Thus the total space required by R is $2\sqrt{n}\log\log\sqrt{n} + \frac{1}{2}\sqrt{n}\log n$ bits, which is $o(n)$ bits. Therefore, the space required by Z and R is $o(n)$ bits.

Table 1: The lookup table R ($W = 16$)

| $s$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|
| 0000000000000000 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... |
| 0000000111111110 | 1 | 255 | 15 | 255 |
| 0000000111111111 | 2 | 256 | 16 | 255 |
| ... | ... | ... | ... | ... |
| 1000000011111111 | 2 | 256 | 16 | 1 |
| ... | ... | ... | ... | ... |
| 1010101001111001 | 7 | 11 | 13 | 6 |
| ... | ... | ... | ... | ... |
| 1111111111111111 | 16 | 16 | 16 | 8 |

The value of $R_2$ achieves its maximum when 0000000111111111 or 1000000011111111 occurs. In this case, we cannot represent the value, and it would overflow since we use 8 bits to represent each entry in $R_2$. In the implementation, when $R_1 > 0$ and $R_2 = 0$, the actual value of $R_2$ is set to 256.

Moreover, we use the same index to access R in order of $(R_1, R_2, R_3, R_4)$. By interleaving storage, we can reduce four memory accesses to one memory access plus three cache accesses. Notice that $R_4$ store the rank value that takes the first run as 1-run. If the first run of a decoding is a 0-run, then the rank value is not the one in $R_4$ but the difference of corresponding entries of $R_2$ and $R_4$.

Let's take an example to show how the table R works. Assume that we are given a run-length Gamma encoded bit string $s = 1\ 010\ 1\ 010\ 011\ 1\ 1\ 001$ to decode and the current start decoding position is at $p$, and the number of bits to decode is 20. We start a lookup by querying R along the eighth row of Table 1, we know $R_1 = 7$, since $s$ contains 7 complete Gamma encodings, which are 1, 010, 1, 010, 011, 1, 1; $R_2 = 11$, since the cumulative sum of complete Gamma decoded numbers is $(1 + 2 + 1 + 2 + 3 + 1 + 1) = 11$; $R_3 = 13$, since the total number of complete Gamma decoded bits is $(1 + 3 + 1 + 3 + 3 + 1 + 1) = 13$; and $R_4 = 6$, since the corresponding raw bit string (after decoding on $s$) is 1 00 1 00 111 0 1 and contains 6 1s. As for the last three bits in $s$, 001, it is not a complete Gamma encoding, since there are 2 0s in front of 1, we should decode the following 3 bits, but we have not that bits. So we start the next lookup at the position $p + R_3(= p + 13)$ and continue this process until the cumulative number of decoded bits is up to 20.

Notice that we take the first run as 1-run. In the example described above, the first complete Gamma encoding is 1 and the corresponding decoded number is 1. We take it as 1 not 0 in the raw bit string. That is, thinking that the raw bit string is 1 00 1 00 111 0 1 not 0 11 0 11 000 1 0. If the first run is truly 1-run, then the local rank value is $1 + 1 + 3 + 1 = 6$, which can be obtained directly by querying $R_4$. Otherwise, the raw bit string is 0 11 0 11 000 1 0, the local rank value should be 5, which can be obtained by computing the difference of corresponding entries of $R_2$ and $R_4$.

In terms of the type (0-run/1-run) of the first run obtained by decoding a given run-length Gamma encoded bit string and the number of runs in $R_1$, we can determine the type of the first run in a lookup. The rules are as follows. If we have decoded even number of runs before a lookup, then the type of the first run of the lookup is the same as that of the first run in the given run-length Gamma encoded bit string. If we have decoded odd number of runs, then the type of the first run of the lookup is the opposite to that of the first run in the given run-length Gamma encoded bit string. We

can determine the type of the first run in a given run-length Gamma encoded bit string by checking the field of the encoding methods, i.e., the RLG0/1 field. Thus we do not need to keep the exact number of runs but the parity of number of runs in $R_1$. This is the reason why we can compactly store $R_1$ and $R_3$ in a byte.

**3.4 Data-aware compression.** The statistical characteristics of the input data have a significant influence on the compression. For typical English-like data, statistical experiments show that the bit vector associated to one node of the corresponding wavelet tree exhibits weaker locality than that of highly-repetitive data. That is, its 0-runs and 1-runs are generally shorter. The number of occurrences of long 0-runs or 1-runs is typically less than that in the highly-repetitive data. Therefore, for the same block size, the English-like data have more runs than do the highly-repetitive data, directly leading to a worse compression and slower encoding or decoding. Thus the decoding time will obviously increase when we increase the block size for the English data. For the English data of 100MB with different block sizes 256, 512, and 1024, the corresponding compression ratio is 30.7%, 27.4%, and 25.6%, respectively. The corresponding single counting (pattern length 20) is 40.18us, 52.71us, and 79.81us, respectively.

On the other hand, the highly-repetitive data contains a high proportion of long runs and the fewer number of runs. These characteristics may result in a good compression and fast decoding. For the kernel data with different block sizes, 256, 512, and 1024, the corresponding compression ratio is 13.4%, 9.5%, and 7.4%, respectively. The corresponding single counting (pattern length 20) is 26.59us, 26.36us, and 26.72us, respectively, almost unchanged. It turns out that the highly-repetitive data is less sensitive than the English-like data to an increase in the block size for the query time. So we can choose a larger block for the highly-repetitive data; otherwise a smaller block.

The superblock size, $sb$, is taken to be 16 times the block size, $b$, i.e., $sb = 16b$, with adaptive block size in the implementation. We choose $b = 1024$ for the highly-repetitive data, 512 for the English-like data, and 256 otherwise. We motivate our choice of block size $b$ with statistical evidence of average runs, defined as the average length of runs of the BWT $L$, denoted by $aver$. The threshold, $aver$, capturers the characteristics of classes of input data with respect to their degree of locality. Loosely speaking, the larger the $aver$ (the more long runs there are) is, the higher the locality. Therefore, the key decisions are as follows.

$$b = \begin{cases} 256, & if\ aver \le l_1 \\ 512, & if\ l_1 < aver \le l_2 \\ 1024, & if\ aver > l_2 \end{cases}$$

These thresholds are empirical values, which can be weighed according to demand. The smaller threshold emphasizes on the query speed while the larger one on the compression. We provide the parameter, *speedlevel*, which determines three typical thresholds:

*Speedlevel* = 0: corresponding to a threshold in $(l_1, l_2) = (2, 10)$, tuning for compression, and block size, generally larger, and slower queries.

*Speedlevel* = 1: corresponding to a threshold in $(l_1, l_2) = (4, 20)$, a good trade-off between compression ratio and query time.

*Speedlevel* = 2: corresponding to a threshold in $(l_1, l_2) = (10, 50)$, tuning for query time, and block size, generally smaller, and faster queries.

Our design can automatically choose an appropriate block size in terms of the characteristics of input data.

## 4 Experiments and Discussion.

**4.1 Experimental setup and environment.** We performed experiments on a HP Z400 with a 2.53GHZ dual-core Intel Xeon W3503 equipped with 4MB L3 cache and 4GB of DDR2 main memory with 64bit Ubuntu12.04. Our program was compiled using g++ version 4.7.3 with -O3 option. We implemented the algorithm in C++, and used Mäkinen and González's *SAc.tgz* (`http://pizzachili.dcc.uchile.cl/indexes.html`) to build suffix array, and our code is available at `https://github.com/chenlonggang/Adaptive-FM-index`.

We use the standard data sets from the *Canterbury Corpus* (`http://corpus.canterbury.ac.nz/`), the *Pizza&Chili Corpus* (`http://pizzachili.dcc.uchile.cl/texts.html`), and the highly-repetitive data sets from the *Pizza&Chili repetitive Corpus* (`http://pizzachili.dcc.uchile.cl/repcorpus.html`).

**4.2 Methods being compared.** Our algorithm is one of five methods being compared:

1. FM-Adaptive, our method described in this paper;
2. FM-Hybrid, developed by Kärkkäinen et al. [14];
3. FM-RRR, developed by Gog and Petri [7];
4. RLCSA, developed by Mäkinen et al. [16]; and
5. FGGV, developed by Foschini et al. [6].

The comparisons are shown in Figure 4. We take our superblock size $sb = 16b$, where $b$ is the block size, that is adaptively set to 256, 512, or 1024, depending on the input data.

**4.3 Results for standard corpus.** Table 2 shows the comparison of FM-Adaptive and the FGGV method in [6] on the compression ratio. The test data used comes from the Canterbury Corpus. The experiments show that FM-Adaptive improves the compression results of [6]. We chose *Speedlevel* = 0 for the experiments shown in Table 2.

Table 2: Comparison of space required by FM-Adaptive and FGGV.

| | book1 | bible | E.coli | world192 |
|---|---|---|---|---|
| FGGV [6] | 3.274 | 2.051 | 2.601 | 1.975 |
| FM-Adaptive | 3.016 | 1.912 | 2.256 | 1.832 |

Table 3 summarizes some general characteristics of the standard data from the *Canterbury Corpus* (the first seven files) and the *Pizza&Chili Corpus* (the last five files).

The term *aver* is the average length of runs of $L$, which can be used for measuring the degree of compressibility of $L$. *Speedlevel* = 1 is a default value in the experiments.

Table 3: General statistics for the standard data sets.

| File | size | $\sigma$ | *aver* |
|---|---|---|---|
| book1 | 768.8KB | 82 | 1.99 |
| book2 | 610.9KB | 96 | 2.55 |
| paper1 | 53.2KB | 95 | 2.40 |
| world192 | 2.5MB | 94 | 4.02 |
| news | 377.1KB | 98 | 2.38 |
| E.coli | 610.9KB | 96 | 2.55 |
| bible | 4.6MB | 4 | 1.41 |
| DNA | 100MB | 16 | 1.59 |
| proteins | 100MB | 25 | 1.69 |
| English | 100MB | 215 | 2.88 |
| sources | 100MB | 227 | 4.29 |
| dblp.xml | 100MB | 96 | 6.96 |

Figure 3 shows the build time of the FM-Hybrid, FM-RRR, RLCSA, and our index (named FM-Adaptive) in seconds. We took the default value of block size for these indexes. The experiments show that FM-Adaptive is the fastest among the indexes, about 1.5–2 times faster.

The size of the indexes includes what is necessary for counting. The compression ratio is defined as the ratio of the indexes size to the original size of the input text. We searched for $10^4$ patterns of length 20, randomly extracted from the indexed text, performed $10^4$ *count* operations on these indexes and took the average time (in $\mu$sec) of a count as the counting query time.

Figure 4 shows the compression and query time for the indexes. For the data *aver* relatively small, FM-Adaptive is usually the smallest index, while its query speed is comparable with FM-Hybrid. When *aver* is very small, such as in dna and proteins, the advantage of FM-Adaptive on compression ratio is not very obvious, but still one of the best. When the *aver* value is larger, such as for sources and dblp.xml, FM-Adaptive still performs very well in compression, but the query time becomes worse. This effect is closely related to the encoding method we used.

For modest size *aver*, such as in book2, bible, and paper1, the better localities enhance the compressibility of their corresponding $L$. Here the run-length Gamma method is dominant. And compared with the fixed encoding of FM-Hybrid, FM-Adaptive has a much better compression ratio since the number of occurrences of small runs is generally greater than that of long runs. Although the Gamma decoding is slower than fixed-length decoding in the FM-Hybrid, the combination of the lookup tables and the interleaving storage to increase the cache hit rate results in competitive query time.

The FM-RRR occasionally compress slightly better but with significantly worse counting time. The RLCSA tends to perform poorly in terms of compression on the data sets but with a moderate query time.

Moreover, for the data with very small *aver*, if the block size is taken to be very larger, then the block would contain a lot runs so that the query time would increase. Our design can automatically choose an appropriate block size in terms of the characteristics of data distribution, being capable of adaptive data-aware.

**4.4 Results for highly-repetitive corpus.** Table 4 summarizes some general characteristics of the highly-repetitive data sets from the *Pizza&Chili repetitive corpus*.

Figure 5 shows the build time of the FM-Hybrid, FM-RRR, RLCSA, and FM-Adaptive in seconds. We can see from Figure 6 that FM-Adaptive is the fastest among the indexes, except for the influenza data. Figure 6 shows compression ratio and query time.

The *aver* values of the highly-repetitive data sets are generally larger. The corresponding block contains fewer runs. If we increase the block size, the query time

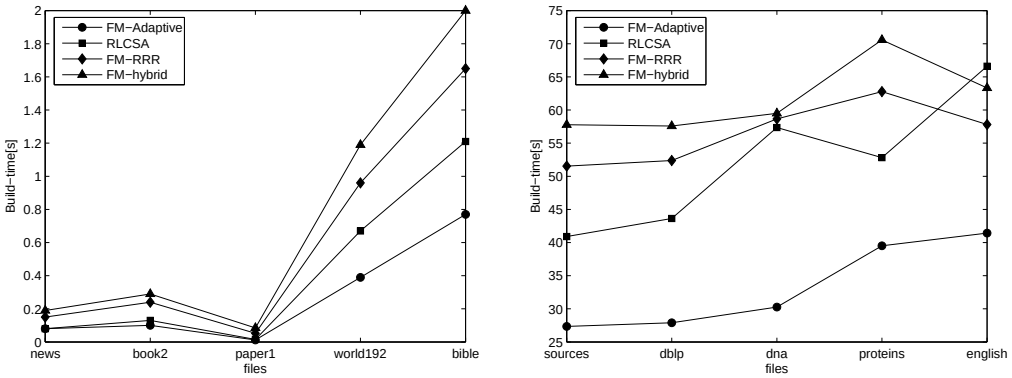



Figure 3: The build time for the standard *Canterbury corpus* (left) and *Pizza&Chili Corpus* (right) for the indexing methods described in §4.2.

Table 4: General statistics for the *Pizza&Chili* highly-repetitive data sets

| File | size | $\sigma$ | *aver* |
|---|---|---|---|
| para | 100MB | 5 | 7.97 |
| influenza | 100MB | 15 | 48.45 |
| world-leader | 40MB | 89 | 76.68 |
| kernel | 100MB | 160 | 45.05 |

would not significantly increase while the compression ratio would be significantly reduced. The block size can be automatically adjusted by our method according to the input data.

On the other hand, when runs are generally larger, the role of the lookup table is limited, so our queries would be slower than the FM-Hybrid, especially for para and influenza data. However, our compression ratio is always better than the FM-Hybrid approach, but worse than RLCSA for the world-leaders and kernel data.

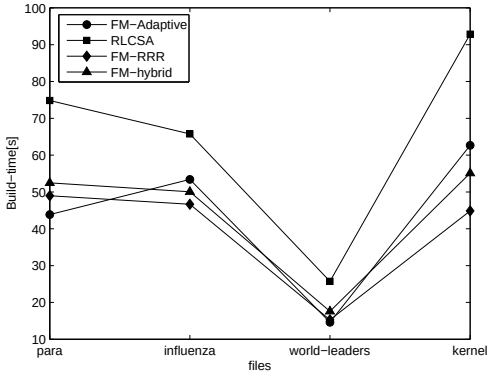



Figure 5: The build time for the *Pizza&Chili* highly repetitive corpus for the indexing methods described in §4.2.

## 5 Conclusions.

In this paper we implemented an efficient compressed indexing scheme. Our compressed index can be constructed quickly and provides new trade-offs between compression ratio and the speed of searching in a text.

**Acknowledgements.** We would like to thank Simon J. Puglisi for clarifying some issues in the paper [14] and providing their source code.

## References

[1] M. Burrows and D. J. Wheeler, *A block-sorting lossless data compression algorithm*, Tech. Report SRC-RR-124,Digital Equipment Corporation, Palo Alto, CA, 1994.

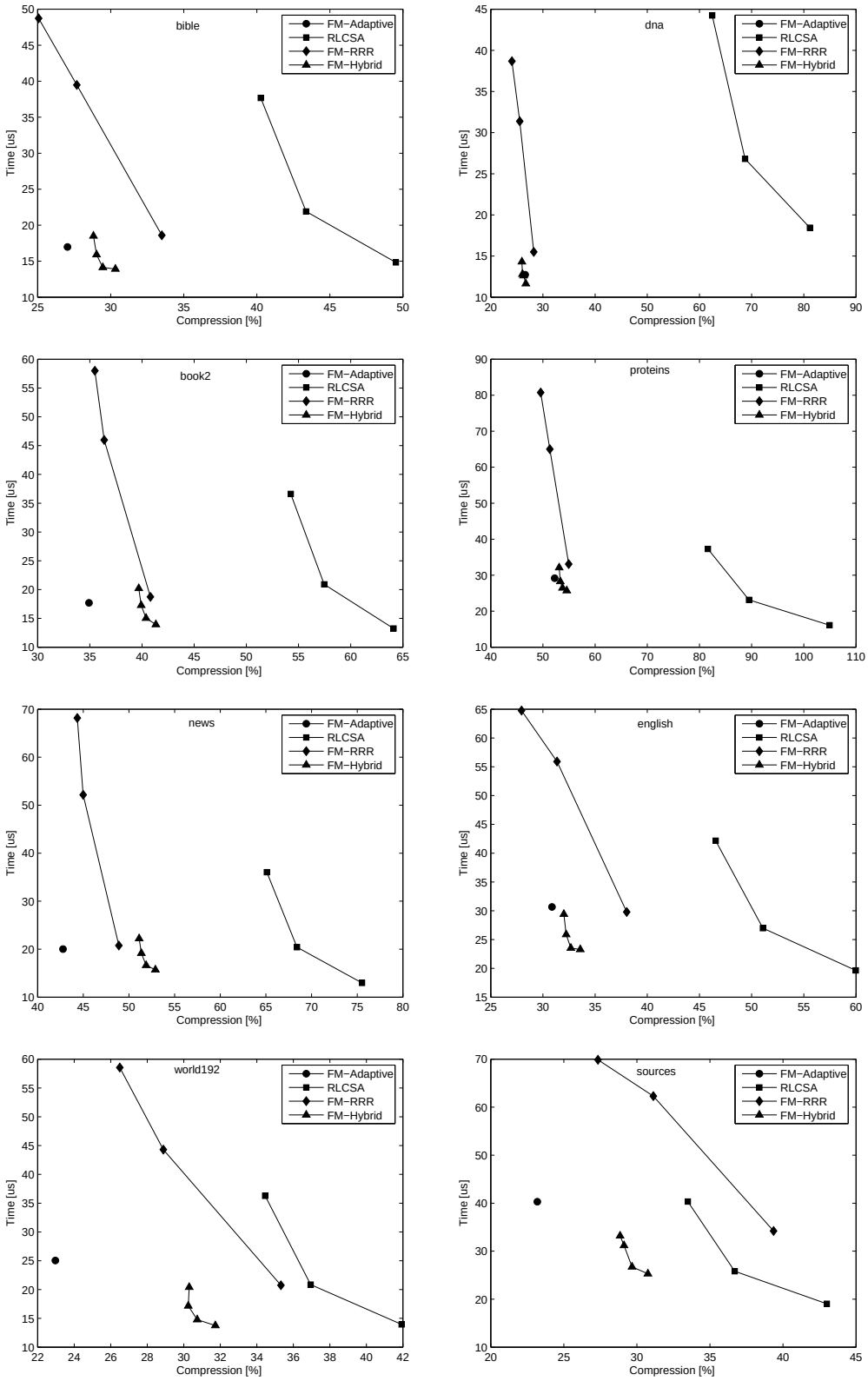
[2] P. Ferragina, R. González, G. Navarro, and R. Venturini, *Compressed text indexes: From theory to practice*, Journal of Experimental Algorithmics, 13 (2009), Article 1.12.

[3] P. Ferragina and G. Manzini, *Opportunistic data structures with applications*, In Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 390–398.

[4] P. Ferragina and G. Manzini, *Indexing compressed text*, Journal of the ACM, 52 (2005), pp. 552–581.

[5] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, *An alphabet-friendly FM-index*, In String Processing and Information Retrieval, 2004, pp. 150–160.

[6] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, *When indexing equals compression: Experiments with compressing suffix arrays and applications*, ACM



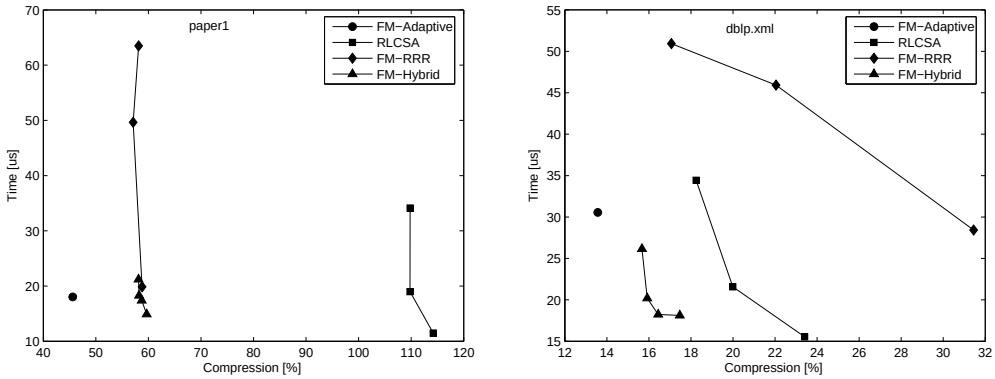



Figure 4: Compression ratio vs. counting query time on the standard *Canterbury corpus* (left) and *Pizza&Chili Corpus* (right) for the indexing methods described in §4.2.

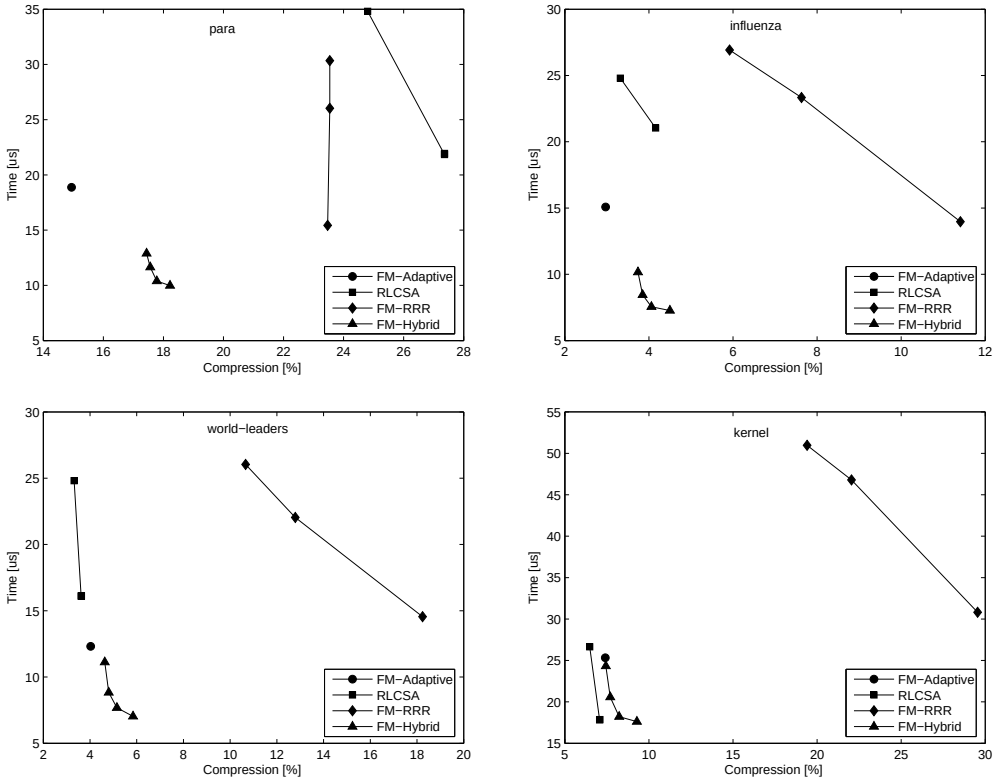



Figure 6: Compression ratio vs. counting query time on the *Pizza&Chili* highly repetitive corpus for the indexing methods described in §4.2.

Transactions on Algorithms, 2 (2006), pp. 611–639.
[7] S. Gog and M. Petri, *Optimized succinct data structures for massive data*, Software: Practice and Experience, 2013.
[8] R. Grossi, A. Gupta, and J. S. Vitter, *High-order entropy-compressed text indexes*, In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, 2003, pp. 841–850.
[9] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, In Proceedings of the thirty-second annual ACM symposium on Theory of computing, 2000, pp. 397–406.
[10] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, SIAM Journal on Computing, 35 (2005), pp. 378–407.
[11] R. Grossi, J. S. Vitter, and B. Xu, *Wavelet trees: From theory to practice*, In International Conference on Data Compression, Communications and Processing, 2011, pp. 210–221.
[12] W.-K. Hon, R. Shah, and J. S. Vitter, *Compression, indexing, and retrieval for massive string data*, In Combinatorial Pattern Matching, 2010, pp. 260–274.
[13] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich, *A Practical Implementation of Compressed Suffix Arrays with Applications to Self-Indexing*, In Data Compression Conference, 2014, pp. 292–301.
[14] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, *Hybrid Compression of Bitvectors for the FM-Index*, In Data Compression Conference, 2014, pp. 302–311.
[15] V. Mäkinen and G. Navarro, *Implicit compression boosting with applications to self-indexing*, In String Processing and Information Retrieval, 2007, pp. 229–241.
[16] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, *Storage and retrieval of highly repetitive sequence collections*, Journal of Computational Biology, 17 (2010), pp. 281–308.
[17] U. Manber and G. Myers, *Suffix arrays: a new method for on-line string searches*, SIAM Journal on Computing, 22 (1993), pp. 935–948.
[18] E. M McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM, 23 (1976), pp. 262–272.
[19] G. Navarro and V. Mäkinen, *Compressed full-text indexes*, ACM Computing Surveys, 39 (2007), Article 2.
[20] G. Navarro and E. Providel, *Fast, small, simple rank/select on bitmaps*, In Experimental Algorithms, 2012, pp. 295–306.
[21] R. Raman V. Raman and S. S. Rao, *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets*, In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, 2002, pp. 233–242.
[22] S. S. Rao, *Time-space trade-offs for compressed suffix arrays*, Information Processing Letters, 82 (2002), pp. 307–311.
[23] K. Sadakane, *Compressed text databases with efficient query algorithms based on the compressed suffix array*, In Algorithms and Computation, 2000, pp. 410–421.
[24] K. Sadakane, *New text indexing functionalities of the compressed suffix arrays*, Journal of Algorithms, 48 (2003), pp. 294–313.
[25] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica, 14 (1995), pp. 249–260.
[26] S. Vigna, *Broadword implementation of rank/select queries*, In Experimental Algorithms, 2008, pp. 154–168.
[27] D. Zhou, D. G. Andersen, and M. Kaminsky, *Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences*, In Experimental Algorithms, 2013, pp. 151–163.