Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

Space-Conscious Data Indexing and Compression in a Streaming Model

Authors: P. Ferragina (ferragina@di.unipi.it)
T. Gagie (travis@mfn.unipmn.it) G. Manzini (manzini@mfn.unipmn.it)

TECHNICAL REPORT TR-INF-2008-02-01-UNIPMN
(February 2008)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.mfn.unipmn.it/`.
Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2007-05 *Scheduling Algorithms for Multiple Bag-of-Task Applications on Desktop Grids: a Knowledge-Free Approach*, Canonico, M., Anglano, C., December 2007.
2007-04 *Verifying the Conformance of Agents with Multiparty Protocols*, Giordano, L., Martelli, A., November 2007.
2007-03 *A fuzzy approach to similarity in Case-Based Reasoning suitable to SQL implementation*, Portinale, L., Montani, S., October 2007.
2007-02 *Space-conscious compression*, Gagie, T., Manzini, G., June 2007.
2007-01 *Markov Decision Petri Net and Markov Decision Well-formed Net Formalisms*, Beccuti, M., Franceschinis, G., Haddad, S., February 2007.
2006-03 *New challenges in network reliability analysis*, Bobbio, A., Ferraris, C., Terruggia, R., November 2006.
2006-03 *The Engineering of a Compression Boosting Library: Theory vs Practice in BWT compression*, Ferragina, P., Giancarlo, R., Manzini, G., June 2006.
2006-02 *A Case-Based Architecture for Temporal Abstraction Configuration and Processing*, Portinale, L., Montani, S., Bottrighi, A., Leonardi, G., Juarez, J., May 2006.
2006-01 *The Draw-Net Modeling System: a framework for the design and the solution of single-formalism and multi-formalism models*, Gribaudo, M., Codetta-Raiteri, D., Franceschinis, G., January 2006.
2005-06 *Compressing and Searching XML Data Via Two Zips*, Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S., December 2005.
2005-05 *Policy Based Anonymous Channel*, Egidi, L., Porcelli, G., November 2005.
2005-04 *An Audio-Video Summarization Scheme Based on Audio and Video Analysis*, Furini, M., Ghini, V., October 2005.
2005-03 *Achieving Self-Healing in Autonomic Software Systems: a case-based reasoning approach*, Anglano, C., Montani, S., October 2005.
2005-02 *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks*, Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D., August 2005.
2005-01 *Bayesan Networks in Reliability*, Langseth, H., Portinale, L., April 2005.
2004-08 *Modelling a Secure Agent with Team Automata*, Egidi, L., Petrocchi, M., July 2004.
2004-07 *Making CORBA fault-tolerant*, Codetta Raiteri D., April 2004.

Space-Conscious Data Indexing and Compression in a Streaming Model

Paolo Ferragina[1], Travis Gagie[2], and Giovanni Manzini[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Informatica, Università del Piemonte Orientale, Italy

Abstract. Compressed full-text indexes, data compressors, and streaming algorithms are three powerful tools for dealing with massive datasets. Surprisingly, algorithms for building compressed indexes (or files) are space-inefficient and/or incompatible with a streaming setting. Space-conscious streaming algorithms are crucial because current disk features make sequential disk accesses at least two orders of magnitude faster than random accesses. For the first time in the literature, we investigate string problems in the streaming model and prove *upper* and *lower* bounds on the complexity of building compressed indexes and compressing data in terms of the classic product: internal-memory space $\times$ number of passes over the disk data.

1 Introduction

Full-text indexes are data structures that index a text string $T[1, n]$ to support subsequent searches for arbitrarily long patterns like substrings, regexp, errors, *etc.*, and have many applications in computational biology and data mining. Recent years have seen a renewed interest in these data structures since it has been proved that full-text indexes can be compressed up to the k-th order empirical entropy of the input text T, and searched without being fully decompressed [27]. At the same time, it has been shown that modern data compressors based on full-text indexes can approach the empirical entropy of an input string without making any assumption on its generating source [13]. Such fundamental achievements have been accompanied by a large algorithmic-engineering effort (see e.g. [14, 12, 28] and references therein) that turned these theoretical breakthroughs into a *technological* advancement: compressed full-text indexes improve a suffix-tree's space by a factor of 100, and a suffix-array's space by a factor of 20, without significant slowdown in the query performance.

Clearly, data compression and indexing are mandatory when the data to be processed and/or transmitted has large size. But larger data means more memory levels involved in their storage and hence, more costly memory references. It is already known how to design an optimal external-memory (uncompressed) full-text index [11, 20], and some results on external memory compressed indexes have recently appeared in the literature [2, 5, 17]. Whichever is the index you choose (compressed or uncompressed), to use it you first need *to build it*! The sheer size of data available nowadays for mining and search applications has turned this into a hot topic because the construction/compression phase may be a bottleneck that can even prevent these indexing and compression tools from being used in large-scale applications.

Recent research [15, 18, 19, 26] has highlighted that a major issue in the construction of such data structures is the large amount of *working space* usually needed for the construction. Here working space is defined as the space required by an algorithm in addition to the space required for the input (the text to be indexed/compressed) and the output (the index or the compressed file). If the data to be indexed is too large to fit in main memory one must resort

to external memory construction algorithms. Such algorithms are known (see e.g. [9, 21, 6, 7]), but they all use $\Theta(n \log n)$ bits of working space. This amount of working space may be *two orders of magnitude* larger than the final size of the compressed index that, for typical data, is a factor 3-to-5 smaller than the original input and is anyway $O(n)$ bits in the worst case.

Given these premises, the first issue we address in this paper is the design of construction algorithms for full-text indexes which work on a disk-memory system and are *space conscious* in that their working space is as small as possible. The second issue we address concerns the way our algorithms fetch/write data onto disk: we design them to access disk data only via *sequential* scans (i.e. streaming-like). This approach is motivated from the fact that sequential I/Os are two orders of magnitude faster than random I/Os. Indeed, sequential disk access rates are currently comparable to random access rates in internal memory. This fact is routinely exploited by expert programmers to get better performance in practice, and it is currently the subject of a large body of research, named *Data Streaming*, typically dealing with sketching, geometric, or graph problem [1, 8, 10, 25]. In this paper we follow this line of research and investigate, for the first time in the literature, two string problems: building (compressed) full-text indexes and compressing data. We provide *upper* and *lower* bounds for them in terms of the product "internal-memory space $\times$ passes over the disk data".

Let us now detail our results, and compare them technically with the known literature. Our model of computation consists of an internal memory with M words (i.e. $\Theta(M \log n)$ bits), and $O(1)$ disks of unbounded capacity, supporting only *sequential* accesses to disk data.[1] The complexity of our algorithms will be measured in terms of the *number of disk passes* in the worst case. In our model sorting n items takes $\Theta(n/M)$ passes on one disk [24], and $O(\log n)$ passes on (at least) two disks [22]. Since the construction of most full-text indexes reduces to suffix-array construction, which in turn needs $(\log n)$ recursive sorting-levels [9], we can build a (compressed) full-text index via a sort-based approach in $O(\frac{n}{M} \log n)$ passes on one disk and $O(\log^2 n)$ passes on two or more disks. The working space of this approach is $\Theta(n \log n)$ bits regardless of the number of disks. Our first result is to show that in our model we can build a suffix array using a simple sort-*less* approach, that, even on a single disk, takes $O(n/M)$ passes and n bits of working space (Theorem 2). If we allow more than one disk, our algorithm is faster than the sort-based approach whenever $M = \Omega(n/\log^2 n)$. This condition holds on modern PCs for texts of size up to a few Terabytes.

Our second contribution is to show how to compute the BWT (a basic ingredient of both compressors and compressed indexes) using a sort-less approach in $O(n/M)$ passes and n bits of disk working space. This is a key contribution since the total space usage of the algorithm is $\Theta(n)$ and therefore proportional to the size of the input. Note also that at each pass over disk data we scan $\Theta(n)$ bits. This yields an additional $\Theta(\log n)$ speedup with respect to the sort based approach that at each pass scans $\Theta(n \log n)$ bits. An additional benefits of working via sequential scans is that the data can be stored on disk in compressed form. Our algorithm takes full advantage of this property since, if the input has small entropy and is therefore highly compressible, the same is true not only for the output but for the intermediate files as well. To our knowledge ours is the first construction algorithm with this feature.

Another important contribution of our paper is a space-conscious method to invert the BWT (and thus decompress the data) which uses $O(n/M)$ passes with one disk or $O(\log^2 n)$

[1] Our model is a reminiscence of the PL-LEMA model of [1], that constraints the classic External Memory model [29] to set $M = O(\text{polylog}(n))$ and to execute only a polylog number of *random* I/Os. All other I/Os must be *sequential* and thus examine the disk data in passes. In our model algorithms operate in disk passes only (i.e. no random I/Os at all) and M is an arbitrary parameter.

At the beginning of the pass, sa_{ext} and pos_{ext} are stored on disk and contain the sa and pos arrays of $T_h T_{h-1} \cdots T_1$.

1. Compute in internal memory the array $\mathsf{sa}_{int}[1, m]$ containing the lexicographic ordering of the suffixes starting in T_{h+1}. This step uses T_{h+1}, T_h and the first m entries of pos_{ext} which are on disk (these are needed since suffixes extend up to T_1). Let us call the suffixes starting in T_{h+1} *new* suffixes, and the ones starting in $T_h \cdots T_1$ *old* suffixes;
2. Compute how many old suffixes fall between two lexicographically consecutive new suffixes. This step uses sa_{int}, and a left-to-right scan of $T_h \cdots T_1$, and pos_{ext};
3. Update sa_{ext} and pos_{ext} so that they include all the suffixes of the longer string $T_{h+1} T_h \cdots T_1$.

Fig. 1. Pass $h + 1$ of the CF-algorithm, where $m = \Theta(M/\log n)$.

passes with two disks, and $\Theta(n)$ bits of disk working space (Theorem 7). This improves known *sort-based* bounds, and is based on different techniques than the ones we used to derive our construction algorithms. Further contributions of our paper are: streaming algorithms for computing the array Ψ and a sampling of the suffix array, which are both important ingredients of compressed indexes (Theorems 5 and 6); and a lightweight *internal-memory* algorithm for computing the BWT, which is the fastest in the literature when the amount of working space is $n + o(n)$ bits (Theorem 4).

Finally, we try to assess to what extent we can improve our algorithms for computing/inverting the BWT with only one disk. In this setting, lower bounds are often established considering the product "internal-memory space $\times$ passes" [24]. For our BWT construction and inversion algorithms such product is $O(n \log n)$ bits, and we prove that we cannot reduce them to $o(n/\log n)$ with a streaming algorithm using a single disk (Theorem 8). Hence our algorithms are a factor $O(\log^2 n)$ from the optimal. To our knowledge, this is the first lower bound of this kind established in the field of data compression and indexing.

2 Notation

We briefly recall some definitions related to compressed full-text indexes; for further details see [27]. Let $T[1, n]$ denote a text drawn from a constant size alphabet Σ. As is usual, we assume that $T[n]$ is a character not appearing elsewhere in T and lexicographically smaller than all other characters. Given two strings s, t we write $s \prec t$ to denote that s precedes t lexicographically. The suffix array $\mathsf{sa}[1, n]$ is the permutation of $[1, n]$ giving the lexicographic order of the suffixes of T, that is $T[\mathsf{sa}[i], n] \prec T[\mathsf{sa}[i + 1], n]$ for $i = 1, \ldots, n - 1$. The inverse of the sa is the pos array, such that $\mathsf{pos}[i]$ is the rank of suffix $T[i, n]$ in the suffix array. This way, $\mathsf{sa}[\mathsf{pos}[i]] = i$. We denote by pos_d the set of (n/d) values $\mathsf{pos}[d], \mathsf{pos}[2d], \ldots, \mathsf{pos}[n]$ that indicate the distribution of the positions of the d-spaced suffixes within sa.

The Burrows-Wheeler transform $\mathsf{bwt}[1, n]$ is defined as $\mathsf{bwt}[i] = T[(\mathsf{sa}[i] - 1) \bmod n]$. The array $\Psi[1, n]$ is the permutation of $[1, n]$ such that $\mathsf{sa}[\Psi(i)] = \mathsf{sa}[i] + 1 \bmod n$. The value $\Psi[i]$ is the lexicographic rank of the suffix which is one character shorter than the suffix of rank i. The basic ingredients of most compressed indexes are either the bwt or the Ψ array, optionally combined with the set pos_d for some $d = \Omega(\log n)$. In this paper we describe external memory algorithms for the computation of all these three basic ingredients.

3 Construction algorithms

The Crauser-Ferragina's algorithm. In [6] Crauser and Ferragina proposed an external-memory suffix array construction algorithm which is an evolution of a previous algorithm by

At the beginning of pass $h+1$, we assume that sa_{ext} contains the suffix array of $T_h T_{h-1} \cdots T_1$, and bits_{ext} is defined as described in the text. Both arrays are stored on disk.

1. Compute in internal memory the array $\mathsf{sa}_{int}[1,m]$ which contains the lexicographic ordering of the suffixes starting in T_{h+1}. This step uses T_{h+1}, T_h and the first $m-1$ entries of bits_{ext}. Let us call the suffixes starting in T_{h+1} *new* suffixes, and the ones starting in $T_h \cdots T_1$ *old* suffixes.
2. Compute in internal memory the array $\mathsf{bwt}[1,m]$ defined as $\mathsf{bwt}[i] = T_{h+1}[\mathsf{sa}_{int}[i]-1]$, for $i = 1,\ldots,m$. If $\mathsf{sa}_{int}[i] = 1$ set $\mathsf{bwt}[i] = \$$ where $\$$ is a character not appearing in T.
3. Scanning bwt, $T_h T_{h-1} \cdots T_1$, and bits_{ext} compute how many old suffixes fall between two lexicographically consecutive new suffixes.
4. Update sa_{ext} and bits_{ext} so that they contain the correct information for the extended string $T_{h+1} T_h \cdots T_1$.

Fig. 2. Pass $h+1$ of the new CF-algorithm.

Gonnet *et al.* [16]. Like [16], the algorithm logically partitions the input text $T[1,n]$ into blocks of size $m = \Theta(M)$ characters each, i.e. $T = T_{n/m} T_{n/m-1} \cdots T_2 T_1$, and computes incrementally the suffix array sa of T via n/m passes, one per block of T. Unlike [16], these text blocks are examined from the right to the left and auxiliary data structures are introduced to guarantee improved I/O-bounds in the worst case. The pseudo-code of the generic $(h+1)$-th pass is given in Figure 1.

At the beginning of pass h, CF-algorithm maintains on disk the suffix array of the string $T_h \cdots T_1$, and its inverse array pos_{ext}. After the last pass $h = n/m$, so we have $\mathsf{sa}_{ext} = \mathsf{sa}$ and $\mathsf{pos}_{ext} = \mathsf{pos}$. The generic pass $(h+1)$ works as follows. We load the text block $t = T_{h+1} T_h$ into memory, and compute the lexicographic ordering sa_{int} of the (new) text suffixes starting in T_{h+1} (and extending up to $T[n]$); then merge sa_{int} with the (old) text suffixes in sa_{ext} in order to produce the new arrays sa_{ext} and pos_{ext} for the extended string $T_{h+1} T_h \cdots T_1$. The merging process is implemented in [6] with the help of a counter array $\mathsf{gap}[0,m]$: which stores in $\mathsf{gap}[j]$ the number of (old) suffixes of the string $T_h \cdots T_1$ which lie lexicographically between the $\mathsf{sa}_{int}[j-1]$-th and the $\mathsf{sa}_{int}[j]$-th new text suffix starting in T_{h+1}. The computation of gap is performed by scanning rightward the string $T_h \cdots T_1$ and by binary-searching sa_{int} for the lexicographic position of each old suffix. Finally, gap is used to quickly merge sa_{int} with sa_{ext}, and update pos_{ext} correspondingly, via one disk scan: entry $\mathsf{gap}[j]$ indicates how many consecutive (old) suffixes in sa_{ext} follow the $\mathsf{sa}_{int}[j-1]$-st new suffix and precede the $\mathsf{sa}_{int}[j]$-th new suffix.

Theorem 1 ([6, Th. 1]). *The CF-algorithm computes the suffix array in $O(n/M)$ passes over $\Theta(n \log n)$ bits of disk data, using $\Theta(n \log n)$ bits of working space. The CPU time is $O\big((n^2/M) \log M\big)$.* □

The CF-algorithm scans the input text leftwards; we can turn any of its disk scans into a rightward scan by simply *reversing* T. Keeping this in mind, and for simplicity of description, our algorithms below will also be described as algorithms that execute *right-to-left* disk scans.

The new CF-algorithm. We now describe a version of the CF-algorithm that introduces two main improvements: working space (and consequently amount of processed data), and CPU time. The new algorithm still splits T into n/m blocks $T = T_{n/m} \cdots T_1$, but it keeps on disk a bit array bits_{ext}, in place of pos_{ext}, defined as follows: at the beginning of pass $h+1$ it is $\mathsf{bits}_{ext}[i] = 1$ if and only if the suffix $T[i,n]$ starting in $T_h \cdots T_1$ is larger than the first

suffix $T_h \cdots T_1$. We notice that at pass $(h+1)$, the first $j_h = n - hm + 1$ positions of bits_{ext} are not used, and that $T[j_h]$ is the first character of $T_h \cdots T_1$. The pseudo-code of pass $h+1$ of the new CF algorithm is illustrated in Fig. 2.

The use of bits_{ext} instead of pos_{ext} gains a twofold advantage over the original CF-algorithm: reduction of the working space on disk (n bits instead of $\Theta(n \log n)$), and logarithmic reduction in the number of passes (and in the total amount) of processed data. Precisely, at the beginning we read into internal memory the substring $t[1, 2m] = T_{h+1}T_h$ and the binary array $b[1, m-1] = \mathsf{bits}_{ext}[j_h + 1, j_h + m - 1]$.[2] Note that, by definition of bits_{ext}, it is $b[i] = 1$ iff the suffix starting at $T_h[1+i]$ is lexicographically greater than the suffix starting at $T_h[1]$, for $i = 1, \ldots, m-1$. To build sa_{int} we need to lexicographically sort the suffixes starting in T_{h+1} and possibly extending up to $T[n]$. Observe that, given two such suffixes starting at positions i and j of T_{h+1}, with $i < j$, we can compare them lexicographically by comparing (in internal memory) the strings $t[i, m]$ and $t[j, j + m - i]$. If these strings differ we are done. If $t[i, m] = t[j, j + m - i]$ the order of the suffixes is determined by the relative order of the suffixes starting at $t[m+1] \equiv T_h[1]$ and $t[j + m - i + 1] \equiv T_h[j - i + 1]$ which is given by the bit stored in $b[j - i]$, available in internal memory.

This argument shows that $t[1, 2m]$ and $b[1, m-1]$ contain all the information we need to build sa_{int} by just working in internal memory. The actual computation of sa_{int} is done in $O(m)$ time as follows. First we compute the rank r_m of the suffix starting at $t[m]$; that is, we compute for how many indices i with $1 \leq i < m$ the suffix starting at $t[i]$—and extending up to $T[n]$—is smaller than the suffix starting at $t[m]$. This can be done in $O(m)$ time using, for example, Lemma 5 in [19]. At this point the problem of building sa_{int} is equivalent to the problem of building the suffix array of the string $t[1, m-1]\$$ where \$ is a special end-of-string character that has rank precisely r_m (instead of being lexicographically smaller than all other suffixes, as is usually assumed). Thus, we can compute sa_{int} in $O(m)$ time and $O(m \log m)$ bits of space with a straightforward modification of the algorithm DC3 [21].

Then we build the array bwt which is a sort of BWT of T_{h+1} (it is not the usual BWT since the order of the suffixes in sa_{int} takes into account the whole string $T_{h+1} \cdots T_1$). The crucial point of the algorithm is now to use bwt to compute efficiently how many old suffixes (of $T_h \cdots T_1$) lie between two lexicographically consecutive new suffixes (starting in T_{h+1}). This is done by deploying the following lemma.

Lemma 1. *For any character $\alpha \in \Sigma$, let $C[\alpha]$ denote the number of characters in bwt that are smaller than α, and let $\mathsf{Rank}(\alpha, i)$ denote the number of occurrences of α in the prefix $\mathsf{bwt}[1, i]$. Assume that the old suffix $T[k, n]$ is lexicographically larger than precisely i new suffixes, that is,*

$$T[\mathsf{sa}_{int}[i], n] \prec T[k, n] \prec T[\mathsf{sa}_{int}[i+1], n].$$

Then, the (one-character longer) old suffix $T[k-1, n]$ is lexicographically larger than precisely j new suffixes, that is, $T[\mathsf{sa}_{int}[j], n] \prec T[k-1, n] \prec T[\mathsf{sa}_{int}[j+1], n]$, where

$$j = \begin{cases} C[T[k-1]] + \mathsf{Rank}(T[k-1], i) & \text{if } T[k-1] \neq T_{h+1}[m]; \\ C[T[k-1]] + \mathsf{Rank}(T[k-1], i) + \mathsf{bits}_{ext}[k] & \text{if } T[k-1] = T_{h+1}[m]. \end{cases}$$

Proof. Let $\alpha = T[k-1]$. Obviously $T[k-1, n]$ is larger than the new suffixes that start with a character smaller than α (they are $C[\alpha]$), and is smaller than all new suffixes starting with a

[2] It suffice to read T_{h+1} since, as we shall see, T_h and $\mathsf{bits}_{ext}[j_h + 1, j_h + m - 1]$ are used to do the binary-search at pass h so we just need to keep them in internal memory.

character greater than α. The crucial point is now to compute how many new suffixes $T[\ell, n]$ starting with $\alpha = T[k-1]$ are smaller than $T[k-1, n]$.

Consider first the case $\alpha \neq T_{h+1}[m]$. Since $T[\ell] = \alpha \neq T_{h+1}[m]$ we have that $T[\ell+1, n]$ is also a new suffix and $T[\ell, n] \prec T[k-1, n]$ iff $T[\ell+1, n] \prec T[k, n]$. Furthermore, $T[\ell] = T[k-1] = \alpha$ so that the sorting of the rows in BWT imply that counting how many new suffixes starting with α are smaller than $T[k-1, n]$ is equivalent to counting how many α's occur in $\mathsf{bwt}[1, i]$. This is precisely $\mathsf{Rank}(T[k-1], i)$.

Assume now that $\alpha = T_{h+1}[m]$. Among the new suffixes starting with α there is also the one starting at position $T_{h+1}[m]$, call it $T[\ell', n]$. We cannot use the above trick to compare $T[k-1, n]$ with $T[\ell', n]$ since $T[\ell'+1, n]$ coincides with $T_h \cdots T_1$ and is therefore an old suffix, not a new one. However, it is still true that $T[\ell', n] \prec T[k-1, n]$ iff $T[\ell'+1, n] \prec T[k, n]$ and since $T[\ell'+1, n] = T_h \cdots T_1$ we know that this holds iff $\mathsf{bits}_{ext}[k] = 1$. □

Using the above lemma—and the fact that we can build a $o(m)$-bit data structure supporting $O(1)$ time Rank queries over bwt [27]—with a single right-to-left scan of $T_h \cdots T_1$ and bits_{ext} we can compute in $O(n)$ time the same array $\mathsf{gap}[0, m]$ of the CF-algorithm, and then use it to update sa_{ext}: for $i = 0, \ldots, m-1$ we simply copy $\mathsf{gap}[i]$ old sa_{ext} values followed by $\mathsf{sa}_{int}[i+1]$. It is a simple algorithmic exercise to show that we can update sa_{ext} in place, without using any additional working space.

During this step we also compute the content of bits_{ext} for the next pass. Recall that at pass $h+2$ we need that $\mathsf{bits}_{ext}[k] = 1$ iff $T_{h+1} \cdots T_1 \prec T[k, n]$. To update bits_{ext}, we first compute the index r_1 such that $T[\mathsf{sa}_{int}[r_1], n] = T_{h+1} \cdots T_1$ (r_1 is the rank of $T_{h+1} \cdots T_1$ among the new suffixes). When we find that there are exactly i new suffixes smaller than $T[k, n]$ we know that $T_{h+1} \cdots T_1 \prec T[k, n]$ iff $r_1 \leq i$ so we can write $\mathsf{bits}_{ext}[k]$ to disk.

Theorem 2. *The new CF-algorithm computes the suffix array in $O(n/M)$ passes over $\Theta(n \log n)$ bits of disk data, using n bits of working space. The CPU time is $O(n^2/M)$.* □

Compared to the original CF-algorithm (Theorem 1), our new proposal reduces the working space (and thus the amount of processed data), and the CPU time, by a logarithmic factor. We also notice that in the new CF algorithm, and in the derivatives described below, we scan T and the bits_{ext} array in parallel so we need at least *two disks*. However, in view of the lower bounds in Section 5, which hold for a single disk, it is important to point out that our new CF-algorithm (and its derivatives) can work via sequential scans using *only one* disk. This is possible by interleaving T and the bits_{ext} array in a single file. At pass h we interleave m new bits within the segment T_h (so that the portion $T_{n/m} \cdots T_{h+1}$ is shifted by $m/\log n$ words). These new bits together with the bits already interleaved in $T_{h-1} \cdots T_1$ allow us to store the portion of the bits_{ext} array that is needed at the next pass.

Finally, we point out that in practice it is not necessary to write the whole bits_{ext} array to disk. Indeed, fix $\ell > 0$ and let w denote the length-ℓ prefix of $T_h \cdots T_1$: It is straightforward to modify the algorithm so that we write the bit $\mathsf{bits}_{ext}[k]$ only for those indices k such that the length-ℓ prefix of $T[k, n]$ coincides with w. This is possible since, when we need to compare $T_h \cdots T_1$ with $T[k, n]$, we have just read the leftmost characters of the latter suffix. Hence, we can determine the lexicographic ordering of the two suffixes by first comparing w with the length-ℓ prefix of $T[k, n]$ and only for ties do we need to look at $\mathsf{bits}_{ext}[k]$. Using this trick with, say, $\ell = 100$ is likely to significantly reduce the amount of data written to disk in practice, even if we may still end up in writing n bits in the worst case.

BWT computation. Because of its very simple structure, it is straightforward to transform the new CF algorithm into an algorithm for computing the BWT. The key observation is that the values stored in sa_{ext} are never used in subsequent computations. Therefore, if we are interested in the BWT, we can simply replace sa_{ext} with an array bwt_{ext} containing the BWT characters. Because of the relationship $\mathsf{bwt}[i] = T[\mathsf{sa}[i] - 1]$, the only change in the algorithm is that, after the computation of the gap array, we update bwt_{ext} as follows: we copy $\mathsf{gap}[i]$ old BWT characters followed by the character $T[\mathsf{sa}_{int}[i+1] - 1]$, for $i = 0, \ldots, m-1$.

Theorem 3. *We can compute the BWT in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of working space. The CPU time is $O\big(n^2/M\big)$.* □

We remark that if we had used the suffix-array construction as an intermediate step to derive the BWT, then we would have ended up with $\Theta(n \log n)$ bits of working space, whereas the input (text) and the output (BWT) now sum up to $O(n)$ bits. Hence Theorem 3 is a key improvement because now the working space is of the order of input/output space, so in total $O(n)$ bits instead of the $O(n \log n)$ bits required by the suffix-array based approach. In addition to that, since we access T and bwt_{ext} sequentially, we can save further space—and sequential I/Os— by storing them in compressed form. In fact, at the beginning of pass $h+1$, bwt_{ext} contains the BWT of $T_h \cdots T_1$ so we can expect it too to be highly compressible with simple algorithms such as MTF and RLE.

Finally, we observe that this I/O-conscious algorithm can be turned into a lightweight *internal memory* algorithm with interesting time-space tradeoffs. For example, setting $M = n/\log n$ we get an internal memory algorithm that runs in $O(n \log n)$ time and uses $\Theta(n)$ bits of working space. Setting $M = n/\log^{1+\epsilon} n$, with $\epsilon > 0$, the running time becomes $O\big(n \log^{1+\epsilon} n\big)$ and the working space goes down to $n + o(n)$ bits. Note that if we no longer need the input text T, we can save extra space writing the (partial) BWT over the already processed portion of text. That is, at the end of pass h, we store $\mathsf{bwt}_h = \mathsf{bwt}(T_h \cdots T_1)$ in the space originally used for $T_h \cdots T_1$. The right-to-left scan of $T_h \cdots T_1$ required for the binary-search steps on sa_{int} can be done without any asymptotic slowdown using bwt_h and a $o(n)$-space data structure supporting $O(1)$ Rank queries over bwt_h (see again [27]). Summing up, we proved:

Theorem 4. *For any $\epsilon > 0$, we can compute the BWT in internal memory in $O\big(n \log^{1+\epsilon} n\big)$ time, using $n + o(n)$ bits of working space. The BWT can be stored in the space originally containing the input text.* □

The only internal-memory BWT algorithm known in the literature using $n + o(n)$ bits of working space is [19] which—when restricted to use $\Theta(n)$ bits of working space—runs in $O\big(n \log^2 n\big)$ time. The algorithm [19] works also for non constant alphabets and can use as little as $\Theta(n \log n/\sqrt{v})$ bits of working space with $v = O\big(n^{2/3}\big)$, running in $O(n \log n + vn)$ worst case time.

Computation of the array Ψ. We use the same framework as above and maintain an array Ψ_{ext} that, at the end of pass $h+1$, contains the Ψ values for the string $T_{h+1} \cdots T_1$. Since the value $\Psi[j]$ refers to the suffix of lexicographic rank j, Ψ values are computed using the same scheme used for suffix array and BWT values: for $i = 0, \ldots, m-1$, we first update $\mathsf{gap}[i]$ values in Ψ referring to old suffixes and then compute and write the Ψ value referring to $T[\mathsf{sa}_{int}[i], n]$. We can compute Ψ values for the new suffixes using information available in internal memory, while for old suffixes we make use of the relationship $\Psi_{h+1}[j] = \Psi_h[j] + k_j$

where k_j is the largest integer such that $\mathsf{gap}[0] + \mathsf{gap}[1] + \cdots + \mathsf{gap}[k_j] < \Psi_h[j]$ (details in the full paper). Since each value k_j can be computed in $O(\log M)$ time with a binary search over the array whose i-th element is $\mathsf{gap}[0] + \cdots + \mathsf{gap}[i]$, we have the following result.

Lemma 2. *We can compute the array Ψ in $O(n/M)$ passes over $\Theta(n \log n)$ bits of disk data, using n bits of working space. The CPU time is $O\big((n^2 \log M)/M\big)$.* □

To reduce the amount of processed data, we observe that although Ψ values are in the range $[1, n]$, it is well known [27] that the sequence $\Psi[1], \Psi[2] - \Psi[1], \Psi[3] - \Psi[2], \ldots, \Psi[n] - \Psi[n-1]$, can be represented in $\Theta(n)$ bits. Thus, by storing an appropriate encoding of the differences $\Psi[i] - \Psi[i-1]$ we can obtain an algorithm that works over a total of $O(n)$ bits.

Theorem 5. *We can compute the array Ψ in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using $\Theta(n)$ bits of working space. The CPU time is $O\big((n^2 \log M)/M\big)$.* □

Computation of pos_d. To compute the set pos_d with a sampling step $d = \Omega(\log n)$, we modify our new CF algorithm as follows: instead of storing sa_{ext} on disk, we store pairs $\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle, \ldots \langle i_k, j_k \rangle$ such that $\mathsf{sa}_{ext}[i_\ell] = j_\ell$ is a multiple of d. The update of the pairs $\langle i_\ell, j_\ell \rangle$ at pass $h+1$ is straightforward: the second component does not change, whereas the value i_ℓ must be increased by the number of new suffixes which are lexicographically smaller than i_ℓ old suffixes. This can be done via a sequential scan of the already computed set of pairs and of the gap array. Since the set pos_d contains $n/d = O(n/\log n)$ pairs, we have

Theorem 6. *We can compute pos_d in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of working space. The CPU time is $O\big(n^2/M\big)$.* □

4 Inversion algorithms

Rather surprisingly, an efficient method to invert the BWT on disk was known long before the BWT itself was discovered. According to Knuth [22], in about 1967 Hardy proposed a method to restore the original order of a sequence that has been permuted but whose elements point to their original successors. This problem is now called *list ranking*, and inverting the BWT reduces to it easily: the "successor" of character $\mathsf{bwt}[i]$ in T is $\mathsf{bwt}[\Psi[i]]$, so we set a pointer from position i to position $\Psi[i]$. It is well-known [3] how to do this calculation by deploying the frequency of each distinct character both in bwt as a whole and in its prefix $\mathsf{bwt}[1, i]$. Therefore, setting up the n "list pointers" takes only two passes over bwt. Then solving the list-ranking problem on that list, we get the positions in T where each $\mathsf{bwt}[i]$ maps to.

The naïve algorithm for list ranking — follow each pointer in turn — is optimal when the permuted sequence and its pointers fit in memory, but very slow when they do not. List ranking in external memory has been extensively studied, and Chiang *et al.* [4] showed how to reduce this problem to sorting a set of n items (recursively), each of size $\Theta(\log n)$ bits. If we invert bwt by turning it into an instance of the list-ranking problem and solve that by using Chiang *et al.*'s algorithm, then we end up with a solution requiring $\Theta(n \log n)$ bits of disk space. In this section we show how, still using Chiang *et al.*'s algorithm as a subroutine, we can invert bwt using sorting and scanning primitives now applied on $O(n/\log n)$ items, for a total of $O(n)$ bits of disk space. Because no other I/O-conscious approach for BWT inversion was previously known, we prefer to leave the discussion general enough, and write $\mathsf{sort}(x)$ (resp. $\mathsf{scan}(x)$) to indicate the *cost* of a sorting (resp. scanning) primitive applied over

x items without detailing the underlying model of computation. This way the reader can then choose her preferred framework of implementation for our BWT-inversion algorithm, and so get the corresponding bounds in terms of number of I/Os or passes. We finally point out that in the full paper we will also show how we can similarly recover T from the array Ψ or from a compressed index, still taking $O(n)$ bits of total disk space.

Let us now concentrate on BWT-inversion. Our algorithm works in $O(\log n)$ rounds, each working on three files: the first file contains a set $\mathcal{S}$ of $n/\log n$ substrings of T, non-overlapping and whose length increases as the algorithm proceeds with its rounds; the second file contains one triple per substring of $\mathcal{S}$, each consisting of (i) a pointer to that substring, (ii) a pointer to a position of bwt whose character follows that substring in T, and (iii) one single character (eventually the one of (ii)); finally, the third file contains the bwt plus a n-bit array bwtMark which marks the characters of bwt already appended to some substring of $\mathcal{S}$. The overall space taken by these three files is $O(n)$ bits.

The main idea underlying our algorithm is to cover T by the substrings of $\mathcal{S}$, avoiding their overlapping. The substrings of $\mathcal{S}$ consist initially of the characters which occupy the first $n/\log n$ positions of bwt; then, they are extended one character after the other along the $O(\log n)$ rounds, always taking care that they do not overlap. If, at some round, c of those substrings become adjacent in T, they are merged together to form one single longer substring which is then inserted in $\mathcal{S}$, and those c constituting substrings are deleted. The algorithm preserves the condition $|\mathcal{S}| = n/\log n$, by selecting $(c-1)$ new substrings which are inserted in $\mathcal{S}$ and consist of one single character not already belonging to any substring of $\mathcal{S}$. This is easily done by scanning bwt and bwtMark and taking the first $(c-1)$ characters of bwt which result *unmarked* in bwtMark. Keep in mind that any time a character is appended to a substring, its corresponding bit in bwtMark is set to 1.

It is not difficult to maintain the invariant at each round by using the above three files and by adopting a constant number of sort/scan primitives over $O(n)$ bits, and thus $n/\log n$ words. Precisely, a round of the algorithm is implemented as follows. We sort the triples according to their second component (i.e. position in bwt of their following character in T) and then scan them and bwt simultaneously. Whenever we reach a character in bwt which is pointed by some triple (i.e. follows the substring of $\mathcal{S}$ corresponding to that triple), then we copy this character in the (third component of the) triple and then update its second component to make it point to the position in bwt of the new character's successor in T.[3] When we are finished scanning bwt, we sort the triples by their first component (substring), and then scan $\mathcal{S}$ and the triples simultaneously in order to append the new (single) characters to the substrings of $\mathcal{S}$. We then re-sort the triples by their second component (i.e., the positions in bwt to which they point), and scan them and bwtMark simultaneously to mark bwtMark$[i]$ if position i is pointed to by some triple. All these steps have cost $O(\mathsf{scan}(n/\log n) + \mathsf{sort}(n/\log n))$. The pseudo-code of a round is given in Figure 3.

The two major difficulties we face now are, first, that the starting position of a substring in bwt does not tell anything about its position in T; second, that the first $n/\log n$ characters in bwt will usually not be spread evenly throughout T. Therefore, we will eventually need to sort the substrings and, in the meantime, we need to prevent them overlapping. The first of these problems is easier, and will help us with the second. Assume that, after the last round, the substrings cover T and do not overlap. Because the first character in each substring is the successor in T of the last character in some other substring (we consider $T[1]$ to be $T[n]$'s successor), we can sort the substrings by list ranking, as follows. We store with each of the

[3] This can be done by keeping track of distinct characters' frequencies in bwt as we go.

At the beginning of the round:

- $\mathcal{S}$ contains $n/\log n$ non-overlapping substrings of T;
- each triple points to a substring and the character in bwt that follows that substring in T;
- each bit in the bit-array is 1 if the corresponding character is already in a substring or pointed to by a triple.

1. Simultaneously scan bwt and triples to copy characters to be appended into triples, and set triples' pointers to characters' successors.
2. Sort the triples by their pointers to substrings. Simultaneously scan bwt and $\mathcal{S}$ to append characters.
3. Re-sort the triples by their pointers into bwt. Simultaneously scan bwtMark and the triples to set the bits corresponding to characters that will be appended in the next round.
4. Extract the $n/\log n$ pairs of pointers and sort the substrings by list ranking. Merge adjacent substrings. Insert each merged substring into $\mathcal{S}$ and delete its c constituent substrings; choose $c-1$ characters unmarked in bwtMark and start $c-1$ new substrings and triples.

Fig. 3. A round of the inversion algorithm.

$n/\log n$ substrings the positions in bwt of its first character and its last character's successor in T. We use these $n/\log n$ pairs of pointers to set up a list, then solve the list-ranking problem on them in order to find the substrings' order in T taking cost $O(\mathsf{scan}(n/\log n) + \mathsf{sort}(n/\log n))$.[4]

The second difficulty we face is preventing the substrings overlapping during the rounds: if we simply stop appending to some substrings, because the characters we would append are already in other substrings, then the number of characters we append per round decreases and we may use more than $O(\log n)$ rounds; on the other hand, if we start new substrings without reducing the number of old ones, then we may store more than $O(n/\log n)$ substrings and so, because each has two $\Theta(\log n)$-bit pointers, use more than $O(n)$ bits of disk space. Our solution is to sort the substrings by list ranking (as described above) during every round, to find maximal sequences of adjacent substrings; we merge adjacent substrings into one longer substring, which is inserted in $\mathcal{S}$, and delete the others; pointers can be easily kept correctly. Again, these steps take a cost of $O(\mathsf{scan}(n/\log n) + \mathsf{sort}(n/\log n))$. We notice that if $c-1$ substrings have been dropped, then we need to start $c-1$ new substrings (of one character each), so that we again have $|\mathcal{S}| = n/\log n$. To achieve this, we choose $c-1$ characters in bwt that are not marked in bwtMark and thus do not belong yet to any substring of $\mathcal{S}$. Each of these characters will form a new substring of $\mathcal{S}$. This completes the round.

Theorem 7. *Given* bwt*, we recover the original text T with a cost $O((\log n)\,\mathsf{sort}(n/\log n))$ and using $\Theta(n)$ bits of total disk space.*

Proof. At each round, $n/\log n$ new characters are appended to the substrings of $\mathcal{S}$. These substrings are guaranteed not to overlap. Since T consists of n characters, we are guaranteed that $O(\log n)$ rounds suffice to append all characters in T to $\mathcal{S}$'s substrings. □

[4] In the full paper we will explain how we use Chiang *et al.*'s algorithm to actually sort the substrings, rather than just determine their order. Our idea is to break each substring after every $\log n$ characters, so each piece fits in $O(1)$ words, and add temporary pointers between consecutive pieces of each substring (notice there are still $O(n/\log n)$ pieces).

Recall that, in our model, $\mathsf{sort}(x)$ takes $O(x/M)$ passes on one disk and $O(\log x)$ passes on two or more disks. Hence our BWT inversion algorithm takes $O(n/M)$ passes on one disk and $O(\log^2 n)$ passes on two or more disks.

5 Lower bounds

Our algorithms to compute or invert the bwt have a product "memory's size $\times$ number of passes" which is $O(n \log n)$ bits. We prove in this section that we cannot reduce them to $o(n/\log n)$ bits via any algorithm that uses only *one single disk* (accessed sequentially). Hence our algorithms are an $O(\log^2 n)$-factor from the optimal.

We prove the lower bound *indirectly* by considering the compression possible with the BWT. [23] shows that, by applying MTF+RLE and Arithmetic coding to bwt, we can encode T in $(5+\epsilon)nH_k^*(T) + O(\sigma^k)$ bits for all k simultaneously and $\epsilon \approx 0.03$, where $H_k^*(T)$ is the kth-order modified empirical entropy of T and $\sigma = |\Sigma|$ is the alphabet size. We can encode and decode with MTF+RLE and Arithmetic coding using one disk, $O(\log n)$ bits of memory and one scan. Therefore, a lower bound on what we need to encode T in, or decode it from, $O(nH_k^*(T) + \sigma^k)$ bits, implies the same lower bound on the what we need to compute or invert bwt.

Let us denote by A a P-pass lossless compression algorithm that uses $\widehat{M}$ bits of memory. We first give a simple proof that, if $P \times \widehat{M} = o(n)$, there exist periodic strings with period $o(n)$ that A does not compress well. We then refine our argument to show there exist strings with low empirical entropy that A does not compress well. A symmetric argument, which we will include in the full paper, shows that, for any lossless decompression algorithm, there are strings we cannot decode from any encoding whose length is $O(nH_k^*(T) + \sigma^k)$. Without loss of generality, we restrict our attention to binary strings, i.e., $\sigma = 2$.

Consider the simple case in which $P = 1$; let $\mathsf{Conf}(T[1,i])$ denote A's memory configuration after reading $T[1,i]$, and let $\mathsf{Out}(T[i,j])$ denote A's output while reading $T[i,j]$.

Lemma 3. *We can compute $T[i,j]$ from $\langle \mathsf{Conf}(T[1,i-1]), \mathsf{Out}(T[i,j]), \mathsf{Conf}(T[1,j]) \rangle$.*

Proof. By contradiction. Assume there is another string T' that takes A between $\mathsf{Conf}(T[1,i-1])$ and $\mathsf{Conf}(T[1,j])$ while causing it to output $\mathsf{Out}(T[i,j])$. Then $\mathsf{A}(T[1,i-1]\,T'\,T[j+1,n]) = \mathsf{A}(T[1,n])$, contradicting the assumption that A is lossless. □

Let $\mathsf{Kolm}(T[i,j])$ denote the Kolmogorov complexity of $T[i,j]$, i.e., the minimum number of bits needed to store $T[i,j]$. Notice Lemma 3 has the following corollary.

Corollary 1. $|\mathsf{Out}(T[i,j])| \geq \mathsf{Kolm}(T[i,j]) - 2\widehat{M} - O(1)$. □

Now suppose $P > 1$; let $\mathsf{Conf}_p(T[1,i])$ denote A's memory configuration after reading $T[1,i]$ in the pth pass, and let $\mathsf{Out}_p(T[i,j])$ denote A's output while reading $T[i,j]$ in the pth pass. Generalizing our arguments, we obtain the results (which do *not* hold for more than one disk).

Lemma 4. *We can compute $T[i,j]$ from $\langle \mathsf{Conf}_1(T[1,i-1]), \mathsf{Out}_1(T[i,j]), \mathsf{Conf}_1(T[1,j]), \ldots, \mathsf{Conf}_P(T[1,i-1]), \mathsf{Out}_P(T[i,j]), \mathsf{Conf}_P(T[1,j]) \rangle$.* □

Corollary 2. *If $P \times \widehat{M} = o(\mathsf{Kolm}(T[i,j]))$, then $\sum_{i=1}^{P} |\mathsf{Out}_i(T[i,j])| = \Omega(\mathsf{Kolm}(T[i,j]))$.* □

Lemma 5. *If $P \times \widehat{M} = o(n)$ then, for any P-pass algorithm A that uses $\widehat{M}$ bits of memory, there exist periodic strings with period $o(n)$ that A does not compress well.*

Proof. Suppose T consists of repetitions of a randomly chosen string s whose length is in $[\omega(P \times \widehat{M}), o(n)]$. Since s is randomly chosen, with high probability $\mathsf{Kolm}(s) = \Theta(|s|) = \omega(P \times \widehat{M})$. Therefore, by Corollary 2, with high probability we output $\Omega(|s|)$ bits for every copy of s in T, that is, $\Omega(n)$ bits in total. On the other hand, since T is periodic, we can store it as an encoding of n plus one explicit copy of s, which takes $o(n)$ bits. □

To refine our argument, we use properties of kth-order De Bruijn sequences, which are binary sequences containing every possible k-tuple exactly once. By definition, such a sequence has length $2^k + k - 1$, the same first and last $k-1$ bits, and kth-order (unmodified) empirical entropy equal to 0.[5] Therefore, if T consists of repetitions of the first 2^k bits of such a sequence, then occurrences of the same k-tuple are always followed by the same bit and so, by the definition of (modified) empirical entropy $H_k^*(T)$ [23], we have that $nH_k^*(T) + O(2^k) = nH_k(T) + O(2^k \log(n/2^k)) = O(2^k \log n)$. There are $2^{2^{k-1}}$ possible kth-order De Bruijn sequences, so with high probability a randomly chosen one has Kolmogorov complexity $\Theta\big(2^k\big)$, i.e., proportional to its length.

Lemma 6. *If $P \times \widehat{M} = o(n/\log n)$, for any P-pass algorithm A using $\widehat{M}$ bits of memory does exist a string T that A encodes in $\Omega(n)$ bits even though $nH_k^*(T) + O(2^k) = o(n)$ for some k.*

Proof. Let k be a function of n such that $2^k \in [\omega(P\widehat{M}), o(n/\log n)]$. Suppose T consists of repetitions of the first 2^k bits s of a randomly chosen De Bruijn sequence. By Corollary 2 and the properties of De Bruijn sequences, with high probability we output $\Omega\big(2^k\big)$ bits for every copy of s in T, that is, $\Omega(n)$ bits in total. On the other hand, $nH_k^*(T) + O\big(2^k\big) = O\big(2^k \log n\big) = o(n)$. □

The proof of Lemma 6 says that we must output many bits while encoding each copy of s in T. In the full paper we will turn the proof around and show that we must also read many bits while decoding one, which yields the following lemma.

Lemma 7. *If $P \times \widehat{M} = o(n/\log n)$, for any P-pass decompression algorithm using $\widehat{M}$ bits of memory, there exist a string T and a value k such that this algorithm cannot recover T from any encoding of at most $O\big(nH_k^*(T) + 2^k\big)$ bits.* □

As we noted above, these two lemmas immediately give us the following theorem.

Theorem 8. *In the worst case, we can neither compute nor invert* bwt *when the product of the memory's size in bits and the number of passes is $o(n/\log n)$.* □

References

1. G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *IEEE FOCS*, pages 540–549, 2004.
2. M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *ACM PODS*, pages 233–242, 2006.
3. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] Since every possible k-tuple occurs exactly once in a kth-order De Bruijn sequence, knowing the k preceding characters tells us the next one.

4. Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External-memory graph algorithms. In *ACM-SIAM SODA*, pages 139–149, 1995.
5. Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *IEEE DCC*, 2008.
6. A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.
7. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*. To appear.
8. C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *ACM-SIAM SODA*, pages 714–723, 2006.
9. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
10. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *ACM-SIAM SODA*, pages 745–754, 2005.
11. P. Ferragina. String search in external memory: Data structures and algorithms. In S. Aluru, editor, *Handbook of Computational Molecular Biology*. Chapman and Hall, 2005.
12. P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. In *Proc. 14th European Symposium on Algorithms (ESA '06)*, pages 756–767. Springer Verlag LNCS n. 4168, 2006.
13. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.
14. L. Foschini, R. Grossi, A. Gupta, and J. Vitter. Fast compression with a static model in high-order entropy. In *Procs of IEEE Data Compression Conference (DCC)*, pages 62–71, 2004.
15. G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 533–545, 2007.
16. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.
17. R. González and G. Navarro. A compressed text index on secondary memory. In *Proceedings of the 18th International Workshop on Combinatorial Algorithms (IWOCA 2007)*, pages 80–91. College Publications, UK, 2007.
18. W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. of the 44th IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.
19. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387:249–257, 2007.
20. J. Kärkkäinen and S. Rao. Full-text indexes in external memory. In *Algorithms for Memory Hierarchies: Advanced Lectures*, pages 149–170. Springer-Verlag LNCS n. 2625, 2003.
21. Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.
22. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.
23. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
24. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.
25. S. Muthukrishnan. *Data Streams: Algorithms and Applications*, volume 1:2. Foundations and Trends in Theoretical Computer Science, NOW, 2005.
26. J. C. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space. *Theoretical Computer Science*, 386:127–136, 2007.
27. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

28. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007.
29. J. Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2):209–271, 2001.