

Lightweight Data Indexing and Compression in External Memory^{*}

Paolo Ferragina¹, Travis Gagie², and Giovanni Manzini³

¹ Dipartimento di Informatica, Università di Pisa, Italy

² Departamento de Ciencias de Computación, Universidad de Chile, Chile

³ Dipartimento di Informatica, Università del Piemonte Orientale, Italy

Abstract. In this paper we describe algorithms for computing the BWT and for building (compressed) indexes in external memory. The innovative feature of our algorithms is that they are lightweight in the sense that, for an input of size n , they use only n bits of disk working space while all previous approaches use $\Theta(n \log n)$ bits of disk working space. Moreover, our algorithms access disk data only via sequential scans, thus they take full advantage of modern disk features that make sequential disk accesses much faster than random accesses.

We also present a scan-based algorithm for inverting the BWT that uses $\Theta(n)$ bits of working space, and a lightweight *internal-memory* algorithm for computing the BWT which is the fastest in the literature when the available working space is $o(n)$ bits.

Finally, we prove *lower* bounds on the complexity of computing and inverting the BWT via sequential scans 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 [24]. 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 about its generating source [11]. 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 [8], and some results on external memory compressed indexes have recently appeared in the literature [1, 4, 16, 26]. However, whichever is the index chosen (compressed or uncompressed), to use it one must first *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 [13, 18, 23] 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 first author has been partially supported by Yahoo! Research. The second and third authors have been supported by Italian MIUR grant “Italy-Israel FIRB Project”. This research was done while the second author was at the Università del Piemonte Orientale, Italy. Emails: ferragina@di.unipi.it, manzini@mf.n.unipmn.it, travis.gagie@gmail.com

(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. [6, 19]), but they all use $\Theta(n \log n)$ bits of working space. We found (see Section 3) that this working space can be up to 500 times larger than the final size of the compressed output that, for typical data, is three to five times 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 *lightweight* 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. This approach is motivated by the well known fact that sequential I/Os are much faster than random I/Os. Indeed, on modern disks sequential disk access rates are currently comparable to random access rates in internal memory [25]. Sequential access to data has the additional advantage of using modern caching architectures optimally, making the algorithm cache-oblivious. These facts are routinely exploited by expert programmers, and have motivated a large body of research, known as *Data Streaming* [22]. In this paper we investigate the problems of building (compressed) full-text indexes and compressing data using only sequential scans (i.e. streaming-like). We provide *upper* and *lower* bounds for them in terms of the product “internal-memory space \times passes over the disk data”.

In the following we consider the classical I/O model [28]: a fast internal memory with M words (i.e. $\Theta(M \log n)$ bits) and $O(1)$ disks of unbounded capacity. Disks are organized in pages consisting of B consecutive words (i.e. $\Theta(B \log n)$ bits overall). Since our algorithms access disk data only by sequential scans, we analyze them counting the number of disk passes as in the streaming models: From that number is straightforward also to derive the cost in terms of the number of I/Os (disk page accesses).

Our first contribution is a lightweight algorithm for computing the BWT — a basic ingredient of both compressors and compressed indexes — in $O(n/M)$ passes and n bits of disk working space. Note that the total space usage of the algorithm is $\Theta(n)$ bits and therefore proportional to the size of the input. Since at each pass we scan $\Theta(n)$ bits of disk data, each pass scans $\Theta(n/(B \log n))$ pages and the overall I/O complexity is $O(n^2/(MB \log n))$. We have implemented a prototype of this algorithm (available from <http://people.unipmn.it/manzini/bwtdisk>). The prototype takes advantage of the sequential disk access by storing all files (input, output, and intermediate) in compressed form, thus further reducing the disk usage and the total I/Os. Our tests show that our tool is the fastest currently available for the computation of the BWT in external memory, and that its disk working space is much smaller than the size of the input.

The second contribution of the paper is to show that from our algorithm we can derive: **(1)** a lightweight *internal-memory* algorithm for computing the BWT, which is the fastest in the literature when the amount of available working space is $o(n)$ bits (Theorem 2), and **(2)** lightweight algorithms for computing: the suffix array, the Ψ array, and a sampling of the suffix array, which are important ingredients of (compressed) indexes (see Theorems 3, 4, and 5).

Another contribution is a lightweight algorithm to invert the BWT which uses $O(n/M)$ passes with one disk or $O(\log^2 n)$ passes with two disks, and $\Theta(n)$ bits of disk working space (Theorem 6). This result is based on different techniques than the ones we used to derive our construction algorithms.

Finally, we try to assess to what extent we can improve our scan-based 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” [21]. For our BWT construction and inversion algorithms such product is $O(n \log n)$ bits and, by strengthening a lower bound from one of our previous papers [14], we prove that we cannot reduce it to $o(n)$ bits with a scan-based algorithm using a single disk (Theorem 7). Hence our algorithms are within an $O(\log n)$ factor of the optimal. We note that our lower bound is “best possible” because, if we have $\Omega(n)$ bits of memory, then we can read the input into internal memory with one pass over the disk and then compute the BWT there.

Related results. As we mentioned above, the problem of the lightweight computation of (compressed) indexes in internal memory has recently received much attention (see [13, 17, 18, 23, 27] and references therein). However, all the proposed algorithms perform many random memory-accesses so they cannot be easily transformed into external memory algorithms. To our knowledge no lightweight algorithms specific for external memory are known. The construction of most full-text indexes reduces to suffix-array construction, which in turn needs $\log n$ recursive sorting-levels [7]. In external memory this sort-based approach takes $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os [8] and is faster than our algorithms when $M = O(n/(\log n \log_{M/B} \frac{n}{B}))$. However, the sort-based approach is not lightweight since it uses $\Theta(n \log n)$ bits of disk working space.

2 Notation

We briefly recall some definitions related to compressed full-text indexes; for further details see [24]. 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 is 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 $\text{sa}[1, n]$ is the permutation of $[1, n]$ giving the lexicographic order of the suffixes of T , that is $T[\text{sa}[i], n] \prec T[\text{sa}[i+1], n]$ for $i = 1, \dots, n-1$. The inverse of the sa is the pos array, such that $\text{pos}[i]$ is the rank of suffix $T[i, n]$ in the suffix array. This way, $\text{sa}[\text{pos}[i]] = i$. We denote by pos_d the set of (n/d) values $\text{pos}[d], \text{pos}[2d], \dots, \text{pos}[n]$ that indicate the distribution of the positions of the d -spaced suffixes within sa .

The Burrows-Wheeler transform is an array of characters $\text{bwt}[1, n]$ defined as $\text{bwt}[i] = T[(\text{sa}[i] - 1) \bmod n]$. The array $\Psi[1, n]$ is the permutation of $[1, n]$ such that $\text{sa}[\Psi(i)] = \text{sa}[i] + 1 \bmod n$. 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 lightweight algorithms for the computation of all these three basic ingredients.

3 Lightweight Scan-Based BWT construction

In this section we describe the algorithm bwt-disk for the computation of the bwt of a text $T[1, n]$ when n is so large that the computation cannot be done in internal memory. Our algorithm is lightweight in the sense that it uses only M words of RAM and n bits of disk space — in addition to the disk space used for the input $T[1, n]$ and the output $\text{bwt}(T[1, n])$. Our algorithm is scan-based in the sense that all data on disk is accessed by sequential scans only. Note that in the description below our algorithm scans the input file right-to-left: in the actual implementation we scan the input rightward which means that we compute the bwt of T reversed. The bwt-disk algorithm is an

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

1. Compute in internal memory the array $\mathbf{sa}_{int}[1, m]$ which contains the lexicographic ordering of the suffixes starting in T_{h+1} and extending up to $T[n]$ (the end of T). This step uses T_{h+1}, T_h and the first $m - 1$ entries of \mathbf{gt} . 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 $\mathbf{bwt}_{int}[1, m]$ defined as $\mathbf{bwt}_{int}[i] = T_{h+1}[\mathbf{sa}_{int}[i] - 1]$, for $i = 1, \dots, m$. If $\mathbf{sa}_{int}[i] = 1$ set $\mathbf{bwt}_{int}[i] = \#$ where $\#$ is a character not appearing in T .
 3. Using \mathbf{bwt}_{int} and scanning both $T_h T_{h-1} \cdots T_1$ and \mathbf{gt} , compute how many old suffixes fall between two lexicographically consecutive new suffixes. At the same time update \mathbf{gt} so that it contains the correct information for the extended string $T_{h+1} T_h \cdots T_1$.
 4. Merge \mathbf{bwt}_{ext} and \mathbf{bwt}_{int} so that at the end of the step \mathbf{bwt}_{ext} contains the \mathbf{bwt} of $T_{h+1} T_h \cdots T_1$.
-

Fig. 1. Pass $h + 1$ of the \mathbf{bwt} -disk algorithm to compute $\mathbf{bwt}(T_{h+1} \cdots T_1)$ given $\mathbf{bwt}(T_h \cdots T_1)$.

evolution of a disk-based construction algorithm for suffix arrays first proposed in [15] and improved in [5]. However, our algorithm constructs the \mathbf{bwt} *directly* without passing through the \mathbf{sa} and uses some new ideas to reduce the working space from $\Theta(n \log n)$ to n bits.

The algorithm \mathbf{bwt} -disk 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 \mathbf{bwt} of T via n/m passes, one per block of T . Text blocks are examined right to left so that at pass $h + 1$ we compute and store on disk $\mathbf{bwt}(T_{h+1} \cdots T_1)$ given $\mathbf{bwt}(T_h \cdots T_1)$. The fundamental observation is that going from $\mathbf{bwt}(T_h \cdots T_1)$ to $\mathbf{bwt}(T_{h+1} \cdots T_1)$ requires only that we insert the characters of T_{h+1} in $\mathbf{bwt}(T_h \cdots T_1)$. In other words, adding T_{h+1} does not modify the relative order of the characters already in $\mathbf{bwt}(T_h \cdots T_1)$.

At the beginning of pass $h + 1$, in addition to the \mathbf{bwt} of $T_h \cdots T_1$ we assume we have on disk a bit array, called \mathbf{gt} , such that $\mathbf{gt}[i] = 1$ if and only if the suffix $T[i, n]$ starting in $T_h \cdots T_1$ is greater than the suffix $T_h \cdots T_1$ (hence at pass $h + 1$ this array takes exactly $hm - 1$ bits). For simplicity of exposition, we denote by $\mathbf{gt}_h[1, m - 1]$ the part of the array \mathbf{gt} referring to the text suffixes which start in T_h : namely, it is $\mathbf{gt}_h[i] = 1$ iff the suffix starting at $T_h[1 + i]$ is lexicographically greater than the suffix starting at $T_h[1]$, for $i = 1, \dots, m - 1$ (note that all these suffixes extend past T_h up to the last character of T).

The pseudo-code of the generic $(h + 1)$ -th pass is given in Figure 1. Step 1 reads into internal memory the substring $t[1, 2m] = T_{h+1} T_h$ and the binary array $\mathbf{gt}_h[1, m - 1]$. Then we build \mathbf{sa}_{int} by lexicographically sorting the suffixes starting in T_{h+1} and possibly extending up to $T[n]$ (the last character of T). 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 the strings $t[i, m]$ and $t[j, j + m - i]$, which have the same length and are completely contained in $t[1, 2m]$ (thus, they are in internal memory). If these strings differ we are done; otherwise, the order between the above two suffixes is determined by the order of the suffixes starting at $t[m + 1] \equiv T_h[1]$ and $t[j + m - i + 1] \equiv T_h[1 + j - i]$. This order is given by the bit stored in $\mathbf{gt}_h[j - i]$, also available in internal memory. This argument shows that $t[1, 2m]$ and \mathbf{gt}_h contain all the information we need to build \mathbf{sa}_{int} working in internal memory. The actual computation of \mathbf{sa}_{int} is done in $O(m)$ time as follows. First we compute the rank r_{m+1} of the suffix starting at $t[m + 1] \equiv T_h[1]$ among all suffixes starting in T_{h+1} ; that is, we compute for how many indices i with $1 \leq i \leq m$ the suffix starting at $t[i]$ is smaller than the suffix starting at $t[m + 1]$ (both extending up to $T[n]$). This can be done in $O(m)$ time using the above observation and Lemma 5 in [18]. At this point the problem of building \mathbf{sa}_{int} is equivalent

to the problem of building the suffix array of the string $t[1, m]\$, where \$ is a special end-of-string character that has rank precisely r_{m+1} (instead of being lexicographically smaller than all other suffixes, as is usually assumed). Thus, we can compute \mathbf{sa}_{int} in $O(m)$ time and $O(m \log m)$ bits of space with a straightforward modification of the algorithm DC3 [19].$

At Step 2 we build the array \mathbf{bwt}_{int} which is a sort of \mathbf{bwt} of the string T_{h+1} : it is not a *real* \mathbf{bwt} because it refers to suffixes which are not confined to T_{h+1} but start in this string and extend up to $T[n]$. The crucial point of the algorithm is then to compute some additional information that allows us to *merge* \mathbf{bwt}_{int} and \mathbf{bwt}_{ext} I/O-efficiently. This additional information consists of a counter array $\mathbf{gap}[0, m]$ which stores in $\mathbf{gap}[j]$ the number of (old) suffixes of the string $T_h \cdots T_1$ which lie lexicographically between the two new suffixes— i.e. $\mathbf{sa}_{int}[j-1]$ and $\mathbf{sa}_{int}[j]$ — starting in T_{h+1} . Note that the \mathbf{gap} array was used also in [5]. However in [5] \mathbf{gap} is computed in $O(n \log M)$ time using $\Theta(n \log n)$ extra bits; here we compute \mathbf{gap} in $O(n)$ time using only the n extra bits of \mathbf{gt} . The following lemma is the key to this improvement.

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

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

Now fix $c = T[k-1]$. Then, the old suffix $T[k-1, n] = cT[k, n]$ is lexicographically larger than precisely j new suffixes, that is, $T[\mathbf{sa}_{int}[j], n] \prec T[k-1, n] \prec T[\mathbf{sa}_{int}[j+1], n]$, where

$$j = \begin{cases} C[c] + \mathbf{Rank}(c, i) & \text{if } c \neq T_{h+1}[m]; \\ C[c] + \mathbf{Rank}(c, i) + \mathbf{gt}[k] & \text{if } c = T_{h+1}[m]. \end{cases}$$

Proof. Obviously $T[k-1, n]$ is larger than the new suffixes that start with a character smaller than c (they are $C[c]$), and is smaller than all new suffixes starting with a character greater than c . The crucial point is now to compute how many new suffixes $T[\ell, n]$ starting with $c = T[k-1]$ are smaller than $T[k-1, n]$. (Recall that $T[\ell, n]$ starts in T_{h+1} , and $T[k, n]$ starts in $T_h \cdots T_1$.)

Consider first the case $c \neq T_{h+1}[m]$. Since $T[\ell] = c \neq T_{h+1}[m]$, we have that $T[\ell+1, n]$ is also a new suffix (i.e. it lies in T_{h+1}) and $T[\ell, n] \prec T[k-1, n]$ iff $T[\ell+1, n] \prec T[k, n]$. Furthermore, $T[\ell] = T[k-1] = c$ so that the sorting of the rows in BWT implies that, counting how many new suffixes starting with c are smaller than $T[k-1, n]$ is equivalent to counting how many c 's occur in $\mathbf{bwt}_{int}[1, i]$. This is precisely $\mathbf{Rank}(T[k-1], i)$. Assume now that $c = T_{h+1}[m]$. Among the new suffixes starting with c 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 and thus not occurring in \mathbf{sa}_{int} . 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 $\mathbf{gt}[k] = 1$. \square

Step 3 uses the above lemma to compute the array \mathbf{gap} with a single right-to-left scan of the two arrays $T_h \cdots T_1$ and \mathbf{gt} available on disk. Step 3 takes $O(n)$ time because we can build a $o(m)$ -bit data structure supporting $O(1)$ time \mathbf{Rank} queries over \mathbf{bwt}_{int} [24]. Finally, Step 4 uses \mathbf{gap} to create the new array \mathbf{bwt}_{ext} by merging \mathbf{bwt}_{int} with the current \mathbf{bwt}_{ext} . The idea is very simple: for $i = 0, \dots, m-1$ we copy $\mathbf{gap}[i]$ old values in \mathbf{bwt}_{ext} followed by the value $\mathbf{bwt}_{int}[i+1]$.

Note that at Step 3 we also compute the content of \mathbf{gt} for the next pass: namely, $\mathbf{gt}[k] = 1$ iff $T_{h+1} \cdots T_1 \prec T[k, n]$. We know the lexicographic relation between $T_{h+1} \cdots T_1$ and all new suffixes

since it does exist r_1 such that $T[\text{sa}_{int}[r_1], n] = T_{h+1} \cdots T_1$ (the latter is a new suffix, indeed). The relation between $T_{h+1} \cdots T_1$ and any old suffix $T[k, n]$ is available during the construction of `gap`: when we find that $T[k, n]$ is larger than i new suffixes of `saint`, we know that $T_{h+1} \cdots T_1 \prec T[k, n]$ iff $r_1 \leq i$. So we can write the correct value for `gt[k]` to disk.

It is easy to see that our algorithm uses $O(m \log m)$ bits of internal memory. Hence, if the internal memory consists of M words, we can take $m = \Theta(M)$ and establish the following result.

Theorem 1. *We can compute the bwt of a text $T[1, n]$ in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $O(n^2/(MB \log n))$ and the CPU time is $O(n^2/M)$. \square*

Single-disk implementation. In the `bwt-disk` algorithm, and in its derivatives described below, we scan T and the `gt` array in parallel so we need at least *two disks*. However, in view of the lower bounds in Section 6, which hold for a single disk, it is important to point out that our algorithm (and its derivatives) can work via sequential scans using *only one* disk. This is possible by interleaving T and the `gt` 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 bits). These new bits together with the bits already interleaved in $T_{h-1} \cdots T_1$ allow us to store the portion of the `gt` array that is needed at the next pass. Note also that the merging of `bwtext` and `bwtint` at Step 4 can be done on a single disk. This requires that, at the beginning of the algorithm, we reserve on disk the space for the full output (n characters), and that we fill this space right-to-left (that is, at the end of pass h `bwt($T_h \cdots T_1$)` is stored in the rightmost mh characters of the reserved space).

Working with compressed files. Accessing files only by sequential scans makes it possible to store them on disk in compressed form. This is not particularly significant from a theoretical point of view — in the worst case the compressed files still take $\Theta(n)$ bits — but is a significant advantage in practice. If the input file $T[1, n]$ is large, it is likely that it will be given to us in compressed form. If the compression format allows for the scanning of a file without full decompression (as, for example, `gzip`, `bzip`, and `ppm`) our algorithm is able to work on the compressed input without additional overhead. An algorithm that accesses the input non-sequentially would require the additional space for an uncompressed image of $T[1, n]$. The same considerations apply to the output file `bwt(T)` and the intermediate files `bwt($T_h \cdots T_1$)`. Since they are `bwt`'s of (suffixes of) T they are likely to be highly compressible, so it is very convenient to be able to store them in compressed form: this makes our algorithm even more “lightweight”. It goes without saying that using compressed files also yields a reduction of the I/O transfer so this is advantageous also in terms of running time (see experimental results below).

Note that the use of compressed files is straightforward if we use two disks: in this way we can store T and `gt` separately and at Step 4 we can store on two different disks the compressed images of `bwt($T_h \cdots T_1$)` and `bwt($T_{h+1} \cdots T_1$)`. The use of compression in the single disk version is trickier and requires the use of ad-hoc compressors.

Experimental results. To test how `bwt-disk` works in practice, we have implemented a prototype in C (source code available at the page <http://people.unipmn.it/manzini/bwtdisk>). The main modification wrt the description of Fig. 1 is that, instead of storing the entire array `gt` on disk, we maintain a “reduced” version in RAM. In fact, Step 1 uses `gth[1, $m - 1$]` which can be stored in RAM. At Step 3 we need the entire `gt` to lexicographically compare all suffixes $T[k, n]$ of $T_h \cdots T_1$ with $T_h \cdots T_1$ itself (see proof of Lemma 1). Instead of storing the whole `gt`, we keep in internal

memory the length- ℓ prefix of $T_h \cdots T_1$, call it α_h , and the entries $\text{gt}[k]$ such that α_h is a prefix of $T[k, n]$. Unless T is a very pathological string, this “reduced” version of gt is much more succinct: by setting $\ell = 1024$ we were able to store it in internal memory in just 128KB. Using this “reduced” version, the comparison between $T[k, n]$ and $T_h \cdots T_1$, can be done by comparing $T[k, n]$ with α_h . If these two strings are different, we are done; otherwise, α_h is a prefix of $T[k, n]$ and thus the bit $\text{gt}[k]$ is available and provides the result of that suffix comparison. Hence, by using standard string-matching techniques, it is possible to compare all suffixes $T[k, n]$ with $T_h \cdots T_1$ in $O(n + \ell)$ time overall.

Our implementation can work with a block size m of up to 4GB and uses $8m$ bytes of internal memory for the storage (and computation) of sa_{int} , bwt_{int} , and the `gap` array. We ran our experiments with $m = 400\text{MB}$ on a Linux box with a 2.5Ghz AMD Phenom 9850 Quad Core processor (only one CPU was used for our tests) and 3.7GB of RAM. On the same machine we also tested the best competitor of our algorithm. Since all other known approaches for computing the BWT in external memory compute the suffix array first, we tested the DC3 tool [6] which is the current best algorithm for computing the suffix array in external memory. We ran DC3 using two disks for the storage of temporary files and setting the `ram_usage` parameter to 1500MB. With these settings the peak heap memory usage reported by `memusage` was between 3.2 and 3.3 Gigabytes for both `bwt-disk` and DC3.

In our implementation we store the files in compressed form: the input T is `gzip`-compressed, whereas the partial (and final) `bwt`’s are compressed by `Rle` followed by *range coding*: according to the experiments in [10] this combination offers the best compression/speed tradeoff for compressing the BWT. Our current implementation uses a single disk. Since at Step 4 we scan simultaneously two partial `bwt`’s (say $\text{bwt}(T_h \cdots T_1)$ and $\text{bwt}(T_{h+1} \cdots T_1)$) in that step the disk head has to move between the two files and the algorithm is not “scan-only”. We plan to support the use of two disks to remove this inefficiency in a future version.

Our algorithm stores on disk only the compressed input and at most two compressed partial `bwt`’s. Hence, the working space (the space used in addition to the input and the output) has the size of a single compressed partial `bwt`: in Fig. 2 we bound it with the size of the final compressed `bwt`. The results in Fig. 2 show that our algorithm is indeed lightweight: for all files the working space is (much) smaller than the size of the input text uncompressed; for most files even the *total* space usage is less than the size of the uncompressed input. The algorithm DC3 uses consistently a working space of more than 30 times the size of the uncompressed input. Comparing columns 3 and 9 we see that, for all files except `Random2`, DC3 working space is more than 100 times the size of the compressed `bwt`; for the file `Html-4`, which is highly compressible, DC3 working space is more than 500 times the size of the compressed `bwt`! (recall that `bwt-disk` working space is at most the size of the compressed `bwt`).

By comparing the running times (columns 8 and 9 in Fig. 2) we see that `bwt-disk` is always faster than DC3 (recall that DC3 only computes the suffix array so we are ignoring the additional cost of computing the BWT from the suffix array). The results show that the more compressible is the input, the faster is `bwt-disk`, while DC3’s running time is much less sensitive to the content of the input file. Another interesting data is the total I/O volume of the two algorithms (measured as the ratio between total I/Os and input size and not reported in Fig. 2). According to [6] for files up to 4GB for DC3 such ratio is between 200 and 300. For `bwt-disk` such ratio is less than 6 for all files except `Random2` for which the ratio is 14.76.

File Name	Size (GB)	Description
Proteins	1.10	Sequence of bare protein sequences from the Pizza&Chili corpus [12].
Swissprot	1.88	Annotated Swiss-prot Protein knowledge base (file uniprot_sprot.dat downloaded from ftp://ftp.ebi.ac.uk/pub/databases/swissprot/release on June 2009).
Genome	2.86	Human genome (May 2004 version) filtered in order to have a string over the alphabet A,C,G,T,N. This is the same file used in [6].
Gutenberg	3.05	Concatenation of English texts from Project Gutenberg. This is the same file used in [6].
Random2	4.00	Concatenation of two copies of a string of length 2GB with characters randomly generated over an alphabet of size 128 using the tool <code>gentext</code> [12].
Mice&Men	5.43	Concatenation of the Mouse (mm9) and Human (hg18) genomes filtered in order to have a string over the alphabet A,C,G,T,N.
Html	8.00	First 8 GB of file <code>Law03</code> from [2] consisting of a collection of html pages crawled from the UK-domain in 2006-07 (with the WARC headers removed).

Input file		bwt-disk space		bwt-disk time				DC3	
name	size	output/working	total	step 1	step 3	step 4	total	time	work space
Proteins	1.10	0.29	1.02	0.49	0.59	0.15	1.45	6.31	30.62
SwissProt	1.88	0.08	0.33	0.36	1.16	0.10	1.88	6.67	30.68
Genome	2.86	0.22	0.69	0.50	2.32	0.55	3.72	6.88	30.68
Gutenberg	3.05	0.18	0.74	0.85	2.19	0.36	3.76	7.14	30.58
Random2	4.00	0.56	2.00	0.80	3.28	2.32	6.90	7.48	30.66
Mice&Men-4	4.00	0.22	0.70	0.52	3.37	0.80	5.06	7.22	30.66
Html-4	4.00	0.06	0.33	0.58	2.55	0.19	3.60	7.49	30.66
Mice&Men	5.43	0.22	0.70	0.52	4.06	1.14	6.10	—	—
Html	8.00	0.05	0.32	0.49	4.90	0.33	6.01	—	—

Fig. 2. Dataset (top) and experimental results (bottom). Since DC3 cannot handle files larger than 4GB we considered also the files `Mice&Men` and `Html` truncated at 4GB (indicated by the suffix -4). In the bottom table, column 2 reports the size (in Gigabytes) of the uncompressed input file: the values in all other columns are normalized with respect to this size. Column 3 reports the size of the compressed `bwt` which is also an upper bound to the working space of `bwt-disk` (see text). Column 4 reports the total (working + input + output) disk space used by `bwt-disk`. Columns 5–9 report running (wallclock) times in microseconds per input byte. The last column reports the size of DC3 working space (again normalized with respect to the size of the input file).

The asymptotic analysis predicts that, if $M \ll n$, as the size of the input grows, our algorithm will eventually become slower than DC3 (our algorithm is designed to be lightweight, not to be fast!). However, the above results show that the use of compressed files and avoiding the construction of the suffix array make our algorithm, not only lightweight, but also faster than the available alternatives on real world inputs.

4 Other Lightweight Scan-Based Construction Algorithms

Internal Memory Lightweight BWT construction. Our `bwt-disk` 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 $2n$ bits of working space: n bits for the `gt` array and n bits for the M words that play the same role as the internal memory in `bwt-disk`. Setting $M = n / \log^{1+\epsilon} n$, with $\epsilon > 0$, the running time becomes $O(n \log^{1+\epsilon} n)$ and the working space is reduced to $n + o(n)$ bits. This algorithm still accesses the text and the partial `bwt`'s by sequential scans, hence it takes full advantage of the very fast caches available on modern CPU's.

We can further reduce the working space by replacing the n bits of the `gt` array with a $o(n)$ -bit data structure supporting $O(1)$ -time Rank queries over $\text{bwt}(T_h \cdots T_1)$. This data structure can provide in constant time the lexicographic rank of each suffix of $T_h \cdots T_1$ (in right-to-left order, see [24]) and therefore can emulate, without asymptotic slowdown, the scanning of `gt`.

If we no longer need the input text T , we can write the (partial) `bwt`'s over the already processed portion of text. That is, at the end of pass h , we store $\text{bwt}(T_h \cdots T_1)$ in the space originally used for $T_h \cdots T_1$. The right-to-left scan of $T_h \cdots T_1$ required at Step 3 can be emulated, without any asymptotic slowdown, using the same data structure used to replace `gt` (see again [24]). Note that overwriting T roughly doubles the size of the largest input that can be processed with a given amount of internal memory. Summing up, we have:

Theorem 2. *For any $\epsilon > 0$, we can compute the BWT in internal memory in $O(n \log^{1+\epsilon} n)$ time, using $o(n)$ bits of working space. The BWT can be stored in the space originally containing the input text. \square*

The only internal-memory BWT construction algorithm that can use such a small working space is [18] which—when restricted to using $o(n)$ bits of working space—runs in $\omega(n \log^2 n)$ time. Note, however, that the algorithm [18] has the advantage of working also for non constant alphabets and can use as little as $\Theta(n \log n / \sqrt{v})$ bits of working space with $v = O(n^{2/3})$, running in $O(n \log n + vn)$ worst case time.

The algorithms in [17, 23] build directly a compressed suffix array but, at least in their original formulation, they use $\Omega(n)$ bits of working space. The algorithm in [27] build a compressed suffix array of a collections of texts. Note that building a (compressed) suffix array for a collection of texts of total length n is a different (simpler) problem than building a (compressed) suffix array of single text of length n : in a collection each text is terminated by a unique eof symbol so there cannot be very long common prefixes. For a collection of $p = \Theta(\log n)$ texts of size n/p the algorithm in [27] runs in $O(n \log n)$ time using $O(n)$ bits of working space, storing the output in compressed form and overwriting the input. The algorithm in [27] is based on the same techniques from [15] that we use in this paper.¹ However we merge $\text{bwt}(T_h \cdots T_1)$ and $\text{bwt}(T_{h+1})$ following the original idea of [15] of locating the suffixes of $T_h \cdots T_1$ inside the (compressed) suffix array of T_{h+1} , while [27] does the opposite. This choice implies other differences: for example, instead of the `gap` array [27] builds an array of ranks which is later sorted to perform the merging.

Lightweight SA construction. We can transform our `bwt-disk` algorithm into a lightweight algorithm for computing the Suffix Array. The key observation is that the values stored in `bwtext` are never used in subsequent computations. Therefore, to compute the `sa`, we can simply replace `bwtext` with an array `saext` containing the `sa` entries (that is, at the end of pass h `saext` contains `sa(Th ⋯ T1)`). The only change in the algorithm is that, after the computation of the `gap` array, at Step 4 we update `saext` as follows: we copy `gap[i]` old `saext` entries followed by `saint[i + 1]`, for $i = 0, \dots, m - 1$. Summing up, we have the following result.

Theorem 3. *We can compute the suffix array in $O(n/M)$ passes over $\Theta(n \log n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $O(n^2/(MB))$ and the CPU time is $O(n^2/M)$. \square*

¹ Note that [27] was published after the first draft of this paper [9] was completed.

Note that compared to the algorithm in [5], which has a similar structure and similar features, our new proposal reduces the working space (and thus the amount of processed data), and the CPU time, by a logarithmic factor.

Lightweight Computation of the Ψ Array. We use the same framework as above and maintain an array Ψ_{ext} that, at the end of pass h , contains the Ψ values for the string $T_h \cdots T_1$. Since the value $\Psi[j]$ refers to the suffix of lexicographic rank j , at Step 4 Ψ_{ext} values are computed using the same scheme used for BWT and suffix array entries: for $i = 0, \dots, m - 1$, we first update $\text{gap}[i]$ values in Ψ_{ext} referring to old suffixes and then compute and write the Ψ value referring to $T[\text{sa}_{int}[i + 1], n]$. 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 $\text{gap}[0] + \text{gap}[1] + \dots + \text{gap}[k_j] < \Psi_h[j]$ (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 $\text{gap}[0] + \dots + \text{gap}[i]$, 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 $O((n^2 \log M)/M)$. \square*

To reduce the amount of processed data, we observe that although Ψ values are in the range $[1, n]$, it is well known [24] that the sequence $\Psi[1], \Psi[2] - \Psi[1], \Psi[3] - \Psi[2], \dots, \Psi[n] - \Psi[n - 1]$, 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 4. *We can compute the array Ψ in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $O(n^2/(MB \log n))$ and the CPU time is $O((n^2 \log M)/M)$. \square*

Lightweight Computation of pos_d . To compute the set pos_d with a sampling step $d = \Omega(\log n)$, we modify our bwt-disk algorithm as follows. At the end of pass h , instead of $\text{bwt}_{ext} = \text{bwt}(T_h \cdots T_1)$ we store on disk the pairs $\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle, \dots, \langle i_k, j_k \rangle$ such that $\text{sa}_h[i_\ell] = j_\ell$ is a multiple of d (here $\text{sa}_h = \text{sa}(T_h \cdots T_1)$). These pairs are sorted according to their first component and essentially represent $\text{pos}_d(T_h \cdots T_1)$. The update of this set of pairs 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 5. *We can compute pos_d in $O(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $O(n^2/(MB \log n))$ and the CPU time is $O(n^2/M)$. \square*

5 Lightweight Scan-Based BWT Inversion

The standard algorithm for inverting the BWT is based on the fact that the “successor” of character $\text{bwt}[i]$ in T is $\text{bwt}[\Psi[i]]$. Since we can set up a pointer from position i to position $\Psi[i]$ for $i = 1, \dots, n$ in linear time, to retrieve T we essentially need to solve a *list ranking* problem in which we have to restore a sequence given the first element and a pointer to each element’s successor. 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.* [3] 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. We now show that, still using Chiang *et al.*'s algorithm as a subroutine, we can invert **bwt** using a sorting primitive now applied on $O(n/\log n)$ items, for a total of $O(n)$ bits of disk space. In the full paper we will also show how we can similarly recover T from the array Ψ still using $O(n)$ bits of total disk space, and how to take advantage of the pos_d array.

To leave the discussion general we write $\text{sort}(x)$ to indicate the cost of sorting x items without detailing the underlying model of computation. Our algorithm for BWT-inversion works in $O(\log n)$ rounds, each working on two files. The first file contains a set \mathcal{S} of $n/\log n$ substrings of T . Each substring is prefaced by a header, which specifies (i) the position in **bwt** of the substring's first character, (ii) the position in **bwt** of the successor in T of the substring's last character, (iii) eventually, the character whose index is in (ii) and, (iv) eventually, the substring's position in a certain partial order that we will define later. These substrings are non-overlapping and their length increases as the algorithm proceeds with its rounds. The second file contains the **bwt** plus an n -bit array **bwtMark** which marks the characters of **bwt** already appended to some substring of \mathcal{S} . The overall space taken by both 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 to form one single, longer substring which is then inserted in \mathcal{S} , and those c constituting substrings are deleted. In each round, we use Chiang *et al.*'s list-ranking algorithm on the headers both to detect when substrings become adjacent and to determine the order in which we should merge adjacent substrings. Our 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 whenever a character is appended to a substring, its corresponding bit in **bwtMark** is set to 1.

Theorem 6. *We can invert the BWT in $O(n/M)$ passes on one disk and $O(\log^2 n)$ passes on two or more disks. If we allow random (i.e. non sequential) disk accesses we can invert the BWT in $O(\frac{n}{B} \log_{M/B} \frac{n}{B \log n})$ I/Os. For all algorithms the total disk usage is $\Theta(n)$ bits.*

Proof. A round of the algorithm is implemented as follows. We sort the substrings according to their headers' second components (i.e. positions in **bwt** of their following character in T), extract the headers into a separate file, and then fill in their third components. To do this with one disk, we need at most $O(n/(M \log n))$ passes over the headers and **bwt**; with more than one disk, we can do it with $O(1)$ passes. Whenever we reach a character in **bwt** which is pointed to by some header (i.e. follows the substring of \mathcal{S} corresponding to that header), then we copy this character in the (third component of the) header and then update its second component to make it point to the position in **bwt** of the new character's successor in T . (This is done by keeping track of distinct characters' frequencies in **bwt** as we go.) When we are finished filling in the third components, we

append the new (single) characters to the substrings of \mathcal{S} , update `bwtMark` by marking `bwtMark[i]` if the character in position i has just been appended, and reinsert the headers. All these steps have total cost $O(\text{sort}(n/\log n))$.

The two major difficulties we face 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 into the order in which they appear in T ; 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 extract the headers and apply list ranking to their first and second components, which has cost $O(\text{sort}(n/\log n))$. We store each header's rank as the header's third component, then reinsert the headers into the substrings. The headers' third components now tell us in what order the substrings appear in T , so we can sort the substrings by them to obtain T .

The second difficulty we face is in 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 three $\Theta(\log n)$ -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 easily be kept correctly. Again, these steps take a cost of $O(\text{sort}(n/\log n))$.

At each round, $n/\log n$ new characters are appended to the substrings of \mathcal{S} . Since these substrings are guaranteed not to overlap we are guaranteed that $O(\log n)$ rounds suffice to append all characters in T to \mathcal{S} 's substrings. The proof follows recalling that each round requires a constant number of sort/scan primitives over $O(n)$ items and that, in our model, $\text{sort}(x)$ takes $O(x/M)$ passes on one disk, $O(\log x)$ passes on two or more disks, or $O\left(\frac{x}{B} \log_{M/B} \frac{x}{M}\right)$ non-sequential I/Os. \square

6 Lower bounds

Our scan-based 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 $o(n)$ bits via any algorithm that uses only *one single disk* (accessed sequentially). Hence our algorithms are an $O(\log n)$ -factor from the optimal. We note that our lower bound is best-possible because, if we have $\Omega(n)$ bits of memory, then we can read the input into internal memory with one pass over the disk and then compute the BWT there using, e.g., Theorem 2.

In a recent paper [14] we observed that, if the repeated substring is larger than the product of the size of the memory and the number of passes, then an algorithm that uses multiple passes but only one disk still cannot take full advantage of the string’s periodicity. Using properties of De Bruijn sequences we proved that, with polylogarithmic memory and polylogarithmic passes over one disk, we cannot achieve entropy-only bounds and, therefore, we also cannot compute the BWT. In that paper, however, we were mostly concerned with low-entropy bounds, and only considered the BWT as a means to achieve them. Our new lower bound for the BWT alone is stronger, with

a simple and direct proof. Our previous lower bound was based on a technical lemma that we can restate as follows:

Lemma 3. *Consider an invertible function from strings to strings and a machine that computes (or inverts) that function using only one disk. We can compute any substring of an input string given 1) for each pass, the machine’s memory configurations when it reaches and leaves the part of the disk that initially (resp., eventually) holds that substring, and 2) the eventual (resp., initial) contents of that part of the disk. \square*

Our new lower bound is based on the same lemma but, instead of combining it with properties of De Bruijn sequences, we now combine it with a property of the BWT itself, demonstrated by Mantaci, Restivo and Sciortino [20]: it turns periodic strings with relatively short periods into strings consisting of relatively few runs.

Lemma 4. *If T is periodic and its minimum period r divides n , then $\text{bwt}(T)$ consists of r runs, each of length n/r and containing only one distinct character.*

Lemma 3 implies that, if the initial contents of some part of the disk are much more complex than its eventual contents (or vice versa), then the product of the memory’s size and the number of passes must be at least linear in the initial (resp., eventual) contents’ complexity. To see why, consider that we can compute the initial contents from the eventual contents (or vice versa) and two memory configurations for each pass; therefore, the product of the memory’s size and the number of passes must be at least the difference between the complexities. Lemma 4 implies that, if T is periodic, then short substrings of $\text{bwt}(T)$ are simple. Combining these ideas in a fairly obvious way gives us our lower bound.

Theorem 7. *In the worst case, we can neither compute nor invert the BWT using only one disk when the product of the memory’s size in bits and the number of passes is $o(n)$.*

Proof. Suppose T is periodic with minimum period r , where r is sublinear in n but still larger than the product of the memory’s size and the number of passes, and consider any algorithm A that computes $\text{bwt}(T)$ using only one disk. Without loss of generality, we can assume A completely overwrites T . Therefore, by Lemma 4, A replaces each copy of T ’s repeated substring, which has length r , by a substring of $\text{bwt}(T)$ consisting of runs of length n/r (except possibly for the first and last). Notice each new substring consists of at most $r/(n/r) + 1 = o(r)$ runs, so we can store it in $o(r)$ bits, whereas storing T ’s repeated substring takes $\Omega(r)$ bits in the worst case. Lemma 3 says we can compute T ’s repeated substring from one of these new substrings and two memory configurations for each pass A makes; it follows that two times the memory’s size times the number of passes must be $\Omega(r)$ bits in the worst case. Since r can be any integer-valued function in $o(n)$, it follows that we cannot compute the BWT using only one disk when the product of the memory’s size and the number of passes is $o(n)$.

Now consider any algorithm A' that inverts $\text{bwt}(T)$ using only one disk. Again without loss of generality, we can assume A' completely overwrites $\text{bwt}(T)$. Therefore, by Lemma 4, with each copy of T ’s repeated substring, A' replaces on the disk a substring of $\text{bwt}(T)$ consisting of $o(r)$ runs. It follows, by the same arguments as above, that we cannot invert the BWT using only one disk when the product of the memory’s size and the number of passes is $o(n)$. \square

References

1. M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *ACM PODS*, pages 233–242, 2006.
2. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
3. 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.
4. 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.
5. 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.
6. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12, 2008.
7. 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.
8. P. Ferragina. String search in external memory: Data structures and algorithms. In S. Aluru, editor, *Handbook of Computational Molecular Biology*. Chapman and Hall, 2005.
9. P. Ferragina, T. Gagie, and G. Manzini. Space-conscious data indexing and compression in a streaming model. Technical Report TR-INF-2008-02-01, Dipartimento di Informatica, Università Piemonte Orientale, <http://www.di.unipmn.it/TechnicalReports/TR-INF-2008-02-01-UNIPMN.pdf>, 2008.
10. 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)*, Lecture Notes in Computer Science vol. 4168, pages 756–767. Springer, 2006.
11. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.
12. P. Ferragina and G. Navarro. The Pizza&Chili corpus home page. <http://pizzachili.dcc.uchile.cl/> or <http://pizzachili.di.unipi.it/>, 2007.
13. G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 533–545, 2007.
14. T. Gagie. On the value of multiple read/write streams for data compression. In *Proceedings of the 20th Symposium on Combinatorial Pattern Matching*, LNCS n. 5577, pages 68–77, 2009.
15. 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.
16. 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.
17. W.-K. Hon, T. W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.
18. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387:249–257, 2007.
19. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.
20. S. Mantaci, A. Restivo, and M. Sciortino. Burrows-Wheeler transform and Sturmian words. *Information Processing Letters*, 86(5):241–246, 2003.
21. J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.
22. S. Muthukrishnan. *Data Streams: Algorithms and Applications*, volume 1:2. Foundations and Trends in Theoretical Computer Science, NOW, 2005.
23. 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.
24. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
25. M. Ruhl. *Efficient algorithms for new computational models*. PhD thesis, Massachusetts Institute of Technology, 2003.
26. R. Sinha, S. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *SIGMOD '08*, pages 661–672. ACM, 2008.

27. Jouni Sirén. Compressed suffix arrays for massive data. In *Proc. 16th Int. Symp. on String Processing and Information Retrieval (SPIRE '09)*, pages 63–74. Springer Verlag LNCS n. 5721, 2009.
28. J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.