


Algorithms for Computing Very Large BWTs: a Short Survey

Diego Díaz-Domínguez 


Department of Computer Science, University of Helsinki, Finland

Lavinia Egidi 

Computer Science Institute, DiSIT, University of Eastern Piedmont, Vercelli, Italy

Veronica Guerrini 

Department of Computer Science, University of Pisa, Italy

Felipe A. Louza 

Faculty of Electrical Engineering, Federal University of Uberlândia, Brazil

Giovanna Rosone¹ 

Department of Computer Science, University of Pisa, Italy

Abstract

The Burrows-Wheeler Transform (BWT) is a fundamental string transformation that, although initially introduced for data compression, has been extensively utilized across various domains, including text indexing and pattern matching within large datasets. Although the BWT construction is linear, the constants make the task impractical for large datasets, and as highlighted by Ferragina et al. [26], “to use it, one must first build it!”. Thus, the construction of the BWT remains a significant challenge. For these reasons, during the past three decades there has been a succession of new algorithms for its construction using techniques that work in external memory or that use text compression. In this survey, we revise some of the most important advancements and tools presented in the past years for computing large BWTs exploiting external memory or text compression approaches without using additional information about the data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Burrows-Wheeler transform, Extended Burrows-Wheeler transform, external memory, text compression, longest common prefix

Digital Object Identifier 10.4230/OASICS.Manzini.2025.7

Funding *Diego Díaz-Domínguez*: Supported by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011.

Veronica Guerrini: Supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 – “THE – Tuscany Health Ecosystem” – Spoke 6 “Precision medicine & personalized healthcare” CUP I53C22000780001.

Felipe A. Louza: Supported by Brazilian agencies CNPq (grants 408314/2023-0 and 311128/2025-4) and FAPEMIG (grant APQ-01217-22).

Giovanna Rosone: Partially supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 – “THE – Tuscany Health Ecosystem” – Spoke 6 “Precision medicine & personalized healthcare” CUP I53C22000780001, by the MUR PRIN 2022YRB97K PINC and by project “Hub multidisciplinare e interregionale di ricerca e sperimentazione clinica per il contrasto alle pandemie e all’antibioticoresistenza (PAN-HUB)” funded by the Italian Ministry of Health (POS 2014–2020, project ID: T4-AN-07, CUP: I53C22001300001).

¹ corresponding author



1 Introduction

The term “big data” has become widely used, encompassing a variety of data types. Genomic data is one of the most prominent forms of big data, due to its size, complexity, and impact [65]. The advancements in high-throughput sequencing technologies have resulted in an unprecedented expansion of genomic databases, which are expected to soon reach sizes of hundreds of terabytes. The scale and complexity of these datasets present significant challenges in terms of data storage, access, and processing.

One of the primary tasks when working with such large datasets is the construction of efficient indexes, which facilitate fast data retrieval and enable a variety of applications in genomics research. Until the early 2000s, the standard solutions for this task were the suffix array or the suffix tree. However, these approaches quickly become intractable as the text grows. Ferragina and Manzini solved this limitation with their FM-index [27], a succinct self-index² with suffix array functionality. Then, Lam *et al.* [41] noticed that the FM-index is efficient in performing pattern matching of short sequences with few mismatches over large genomes. This functionality led to the creation of popular bioinformatic tools such as **bwa** [46] and **bowtie** [42] based on the FM-index, and since then its use has been extended to other applications such as *de novo* assembly [62], variant detection [43, 60] and sequence compression [35, 61, 33, 31].

The main component of the FM-index is the Burrows-Wheeler transform [12], a string transformation introduced in the Data Compression field by Michael Burrows and David Wheeler in 1994. It is a reversible transformation that rearranges the symbols of a string w over an alphabet Σ , possibly increasing the runs of identical symbols. This structure enables compressing the original string using simple algorithms, such as Run-Length Encoding (RLE) followed by Move-to-Front (MTF) and Huffman coding (as employed in **bzip2** compressor), see [1] for further details.

Since the sequencing process generates a collection of sequences, each representing a read, it introduced challenges in how to represent, process, and analyze these sequence collections. Applying the BWT to each sequence individually could not capture the features among the different sequences, and hence, there emerged a necessity to extend the BWT to sets of strings. Moreover, the need to analyze multiple genomes together has recently emerged to study the complete genetic diversity within populations [18]. Here, we consider the extension of the Burrows-Wheeler transform to a string collection, in which a distinct end-marker symbol is appended to each string (see [14, 15] for a survey about the different BWT variants). When an end-marker symbol is appended to each string, there are no strings in the collection that are prefixed by other strings. Such a transform applied to a string collection is also known in the literature as multi-string BWT³ [23, 8]. Over the years, a multitude of strategies and tools have been introduced to build the BWT and the multi-string BWT, whose efficiency may be dictated by particular needs (e.g. input alphabet, redundant data, available resources, knowledge of a reference string).

In-memory algorithms to compute standard single-string BWTs are linear but do not scale to large inputs with multiple strings as, in those cases, they might not be able to fit everything in RAM (e.g., SAIS [56] and its BWT variant [57]). A possible solution is to keep large portions of data on disk and bring them back to RAM and caches whenever the BWT algorithm requires it. This strategy reduces the space usage in fast memory levels

² A data structure that encodes a text and an index for it in space proportional to the text size

³ In [15], the same transform is called Multidollar-EBWT.

considerably, but comes with a cost: While accessing caches requires nanoseconds in most modern computer architectures, accessing data from disks (aka I/O) requires milliseconds. This difference is the so-called I/O bottleneck, which amounts to a factor of 10^{5-6} [34, 25], and makes it necessary to adapt in-memory algorithms to minimize I/Os. An alternative solution is to keep the data in compressed form to fit more information in less space. This technique reduces the number of I/Os but requires extra time to compress and decompress. In addition, it is necessary to adapt the algorithm to make it compression-aware. In practice, disk-based and compression-aware algorithms are not incompatible and are often combined in the literature.

In this survey, we explore BWT construction algorithms that address the scalability problem by using techniques that work in external memory or by exploiting text compression approaches. We have selected some of the most representative algorithms from these two categories. Those that work in external memory (Section 3) must address the challenge of making the best possible use of the available RAM, minimizing and optimizing operations on external memory, avoiding random disk accesses. Optimal I/O volume when working with a set of strings of total size N is $\mathcal{O}(N)$ bytes which is never achieved to the best of our knowledge. The ordering of suffixes necessary to compute BWTs and suffix arrays is in general carried out with bucket sort algorithms that write data to disk in sequential order, and it is often implemented in a virtual way, ordering indices instead of the actual structures. Some of the algorithms approach the problem by computing partial BWTs, which could be either BWTs of substrings or BWTs of subcollections or BWTs of suffixes of fixed lengths or up to fixed lengths.

We first present **bwt-disk** [26] (Section 3.1), an external memory approach that works on a single input string. We then present **BCR** and **BCRext** [5] (Section 3.2), (semi-)external algorithms, which, although not competitive for collections of long strings stand out due to their simplicity, making them a good choice for sets of strings over any alphabet with limited lengths and for building BWTs with a reduced, possibly minimum, number of runs [13, 15]. In Section 3.3, we present **eGSA** [49], which works best with an amount of RAM significantly larger than the input size and it is extremely efficient, but requires a comparatively large amount of disk space, even though it has good I/O costs. In Section 3.4, we present **eGap** [23], which slightly improves on the I/O costs of **eGSA** and is faster when less RAM is available, also thanks to a specific heuristic. Finally, in Section 3.5, we present **BWT+LCP** [8], which has the same asymptotic complexity as **eGap** with no heuristic and fixed size alphabet.

In Section 4, we consider algorithms of the second class that leverage compression techniques that exploit the high repetitiveness of the input. This way more of the input fits in memory as close as possible to the CPU for faster processing, and repetitions in the input are eliminated. The risk, though, is introducing a significant overhead when strings must be decompressed for access. The art in designing these algorithms is finding compression schemes whose structure actually enhances the BWT computation process instead of hindering it.

We present in Section 4.1 **Big-BWT** [11], which takes as input a single string and performs a pre-processing step that encodes the input building a dictionary and a parse (*prefix-free parsing*). This strategy for building the BWT works well when the dictionary and the parse generated are much smaller than the input string. Finally, in Section 4.2, we present **grlBWT** [22], which joins the two worlds as an algorithm that uses both external memory and compression techniques. It maintains data in compressed form throughout the computation to reduce space usage and repetitions.

2 Preliminaries

Let Σ be a finite ordered alphabet of size σ . A *string* $S = S[1..n]$ over Σ of length $|S| = n$ is a sequence of n characters (or symbols) from Σ , where $S[i]$ denotes the i th character of S , and $S[i..j]$ denotes the substring $S[i] \cdots S[j]$, for $i \leq j$. By convention $S[i..j] = \epsilon$ if $i > j$, where ϵ is the empty string. Any substring of S of the form $S[1..i]$, $1 \leq i \leq n$, is called *prefix*, and any substring of the form $S[i..n]$, $1 \leq i \leq n$, is called *suffix*. Let S be a string of length $n - 1$ over Σ and $\$$ be a special character, which is not in Σ and is lexicographically smaller than any other character in Σ . Throughout this paper, we consider S with the special character (referred to as *end-marker*) appended to it, such that $S[n] = \$$.

For any two strings S and T , we define the *lexicographic order* $S \prec T$ if S is a proper prefix of T , or if there exists an index $1 \leq i \leq \min\{|S|, |T|\}$ such that $S[i] < T[i]$ and for all $j < i$, $S[j] = T[j]$.

We consider \mathcal{S} be a collection of k strings, $\mathcal{S} = \{S_1, \dots, S_k\}$, where $S_j[1..n_j - 1]$ is a string over Σ and $S_j[n_j]$ is the end-marker $\$j$, for any j . The total number of characters in \mathcal{S} is $N = \sum_{j=1}^k n_j$. By definition, the end-markers $\$j$ are not appearing elsewhere in S_1, \dots, S_k and are smaller than any other character in S_1, \dots, S_k . In addition, we assume that the end-markers $\$j$ are sorted according to the input strings⁴, i.e. $\$i < \j , if $i < j$.

We use the disk model of computation, which has two levels: a small but fast memory of M bits (i.e., RAM) and a slow but unbounded memory (i.e., disk). Given an input of size n , a procedure runs in RAM using words of $\Omega(\log n)$ bits that can be manipulated in constant time. Additionally, the procedure can transfer B bits between memories (I/O operation). We express the space complexity in bits and the I/O cost as the number of data transfers. Additionally, we express the logarithms of base two as \log .

Suffix array and LCP. Let S be a string of length $n - 1$ over Σ and $S[n] = \$$. The *suffix array* sa of S [50] stores the permutation of $\{1, \dots, n\}$ listing the starting positions of the suffixes of S in increasing lexicographic order, i.e., $S[\text{sa}[i]..n] \prec S[\text{sa}[i + 1]..n]$, for all $1 \leq i < n$.

The *generalized suffix array* (GSA) of the collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ is the array of N pairs of integers (t, j) , corresponding to the lexicographically sorted suffixes $S_j[t, n_j]$, where $1 \leq t \leq n_j$ and $1 \leq j \leq k$.

The *longest common prefix* array lcp stores the length of the longest common prefix (LCP) between lexicographically consecutive suffixes, i.e., $\text{lcp}[i] = \text{LCP}(S[\text{sa}[i - 1]..n], S[\text{sa}[i]..n])$, for all $1 < i \leq n$, and $\text{lcp}[1] = 0$ for convenience.

Burrows-Wheeler Transform (BWT) and Multi-string BWT. The *Burrows-Wheeler Transform* (BWT) [12] is a reversible text transformation that given as input a string S produces a permutation of its characters: the output string bwt is obtained by concatenating the characters that circularly precede the lexicographically sorted suffixes of S , i.e.

$$\text{bwt}[i] = \begin{cases} S[\text{sa}[i] - 1] & \text{if } \text{sa}[i] > 1, \\ S[n] & \text{if } \text{sa}[i] = 1. \end{cases}$$

The reversibility of the BWT is guaranteed by the following two properties [12]. Let F be the string formed by lexicographically sorting the characters of S ,

- for all $i = 1 \dots n$, the character $F[i]$ circularly follows the character $\text{bwt}[i]$ in the string S ;

⁴ In order not to increase the alphabet size, typically tools use the same character in the BWT output string, such as $\$$, to represent all distinct end-markers. Each end-marker's index can be stored separately.

- for each $c \in \Sigma$, the k th occurrence of c in \mathbf{bwt} corresponds to the k th occurrence of c in F . In particular, if $\mathbf{bwt}[i] = c$, then the position j in F corresponding to that c -occurrence is given by $C[\mathbf{bwt}[i]] + \text{rank}(\mathbf{bwt}[i], i)$, where $C[c]$ is the total number of characters in S that are smaller than c and $\text{rank}(c, i)$ is the number of occurrences of character c in the substring $\mathbf{bwt}[1..i]$.

The function described above that maps character occurrences from \mathbf{bwt} to F is known as *LF-mapping* [27]. The LF-mapping yields a permutation of the positions $1, 2, \dots, n$ which forms a single cycle over all positions.

► **Example 1.** Given the string $S = \text{CATGATGATA\$}$ of length 11, the BWT output string is $\mathbf{bwt} = \text{ATGGC\$TTAAA}$. The LF-mapping yields the following permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 9 & 7 & 8 & 6 & 1 & 10 & 11 & 3 & 4 & 5 \end{pmatrix}.$$

Historically, the BWT was extended to a collection of strings by Mantaci *et al.* [51] (EBWT), where cyclic rotations of the input strings are sorted⁵ without appending any end-marker to the input strings. Here, we consider the generalization that append to each input string in the collection a different end-marker and computes the transformed string by lexicographically sorting the suffixes of the input strings, as done for instance by [5, 24, 22]. Therefore, we define the BWT extended to a collection of strings $\mathbf{bwt}(\mathcal{S})$, *multi-string BWT*, as a permutation of the N characters in S_1, \dots, S_k obtained by concatenating the characters circularly preceding the lexicographically sorted suffixes of S_1, \dots, S_k .

Since each S_j is terminated by an end-marker symbol, the $\mathbf{bwt}(\mathcal{S})$ can be defined in terms of the generalized suffix array (GSA). In particular, $\mathbf{gsa}[q] = (t, j)$ if the q -th smallest suffix of the strings in \mathcal{S} is $S_j[t, n_j]$. So, for $i = 1, \dots, N$,

$$\mathbf{bwt}(\mathcal{S})[i] = \begin{cases} S_j[(t-1)] & \text{if } \mathbf{gsa}[i] = (t, j) \text{ with } t > 1, \\ \$_j & \text{if } \mathbf{gsa}[i] = (t, j) \text{ with } t = 1. \end{cases} \quad (1)$$

Equivalently, one can define $\mathbf{bwt}(\mathcal{S})$ in terms of suffix array $\mathbf{sa}_{1\dots k}$ of the concatenated string $S_1 \dots S_n$ (see [23, 24], where it is denoted $\mathbf{bwt}_{1\dots k}$) i.e.

$$\mathbf{bwt}(\mathcal{S})[i] = \begin{cases} S_j[\mathbf{sa}_{1\dots k}[i] - \sum_{h=1}^{j-1} n_h - 1] & \text{if } \sum_{h=1}^{j-1} n_h + 1 < \mathbf{sa}_{1\dots k}[i] \leq \sum_{h=1}^j n_h, \\ S_j[n_j] & \text{if } \mathbf{sa}_{1\dots k}[i] = \sum_{h=1}^{j-1} n_h + 1. \end{cases} \quad (2)$$

Note that both definitions in Equations (1) and (2), as well as the EBWT, are such that each input string cycles on itself. In fact, the two definitions above are equivalent, but we chose to present both of them because they imply different constructions. In presenting each algorithm we will refer to the definition that inspired the algorithm's strategy.

Finally, the multi-string BWT is reversible by definition: one can reconstruct the collection with the strings sorted according to their original order given the fact that the end-markers in $\mathbf{bwt}(\mathcal{S})$ are distinct [5, 6]. In addition, differently from the BWT applied to a single string, the LF-mapping associated with the multi-string BWT defines a permutation of the positions 1 through N that can be decomposed into k cycles, one for each string in the collection.

⁵ In this case, one needs to use the ω -order defined in [51].

► **Example 2.** Let \mathcal{S} be the collection $\{AGCGT\$1, TCAAC\$2, CGCAA\$3\}$. The multi-string BWT according to Equations (1) and (2) is given by $\text{bwt}(\mathcal{S}) = TCAACCA\$1AGT\$3GCACG\2 . The LF-mapping yields the following permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 17 & 9 & 4 & 5 & 10 & 11 & 6 & 1 & 7 & 14 & 18 & 3 & 15 & 12 & 8 & 13 & 16 & 2 \end{pmatrix},$$

which can be decomposed into 3 disjoint cycles: $(1\ 17\ 16\ 13\ 15\ 8)\ (2\ 9\ 7\ 6\ 11\ 18)\ (3\ 4\ 5\ 10\ 14\ 12)$.

Induced suffix sorting (ISS). Induced suffix sorting (ISS) [40, 37, 56] is a recursive technique that deduces the lexicographical order of suffixes in S starting from a subset of partially sorted suffixes. One of the most popular variants of this idea is SAIS [56], a linear-time algorithm to calculate the suffix array [56, 48, 55, 47] and the BWT [57, 3, 9, 22].

Each recursive level i of SAIS receives a string $S^i[1..n^i]$ ($S^i = S$ and $\Sigma^i = \Sigma$ when $i = 1$) and returns the suffix array sa^i of S^i . The algorithm classifies the positions of S^i into two types: *L-type* or *S-type*. A character $S^i[j]$ is *L-type* if $S^i[j] > S^i[j+1]$ or if $S^i[j] = S^i[j+1]$ and $S^i[j+1]$ is L-type. On the other hand, $S^i[j]$ is *S-type* if $S^i[j] < S^i[i+1]$ or if $S^i[i] = S^i[j+1]$ and $S^i[i+1]$ is S-type. By definition, $S[n] = \$$ is S-type. Furthermore, $S^i[j]$ is *LMS-type* (leftmost S-type) if $S^i[j]$ is S-type and $S^i[j-1]$ is L-type. A substring $S^i[j..j']$ is an *LMS substring* if $S^i[j]$ and $S^i[j']$ are LMS-type.

Every suffix $S^i[j..n]$ whose starting position $S^i[j]$ is LMS-type is also a local minimum $S[j-1..n] \succ S^i[j..n] \prec S^i[j+1..n]$. SAIS exploits this fact and performs three linear-time scans of S^i to induce from $S^i[j]$ a partial sort of the suffixes starting within $S^i[j'..j-1]$, $S^i[j']$ being the preceding LMS-type symbol. In this survey, this process is called *ISS pass*.

For any position $j' \in [1..n]$, let $\text{nextLMS}(j') > j'$ be the smallest integer such that $S^i[\text{nextLMS}(j')]$ is of LMS type, or $\text{nextLMS}(j') = n^i$ if $j' = n^i$ or there are no LMS-type positions after j' . Additionally, let \mathcal{T} be the set of strings over Σ^i that label the sequences $S^i[j'..\text{nextLMS}(j')]$. For two strings $X[1..n_x], Y[1..n_y] \in \mathcal{T}$, their LMS order \prec_{LMS} is

$$X \prec_{\text{LMS}} Y \iff \begin{cases} n_y < n_x \text{ and } X[1..n_y] = Y, \\ \text{or} \\ X \prec Y. \end{cases} \quad (3)$$

Let $\mathcal{O}_X \subset [1..n]$ be the set of integers such that each $j \in \mathcal{O}_X$ corresponds to some $S^i[j..\text{nextLMS}(j)] = X$. The relation $X \prec_{\text{LMS}} Y$ implies that for any pair $j \in \mathcal{O}_X$ and $j' \in \mathcal{O}_Y$, it holds $S^i[j..n^i] \prec S^i[j'..n^i]$. The ISS pass places the suffixes of S^i starting at positions in \mathcal{O}_X contiguously in the r th range $\text{sa}[s_X..e_X]$ of sa , where r is the \prec_{LMS} rank of X among the elements of \mathcal{S} . Thus, the suffixes in $\text{sa}^i[s_X..e_X]$ are lexicographically smaller than the suffixes in $\text{sa}^i[s_Y..e_Y]$ if $X \prec_{\text{LMS}} Y$. However, this sorting is partial because the relative order of the elements within $\text{sa}^i[s_X..e_X]$ and $\text{sa}^i[s_Y..e_Y]$ is still unknown.

SAIS now replaces the LMS substrings in S^i by their \prec_{LMS} ranks to create a new string $S^{i+1}[1..n^{i+1}]$, with $n^{i+1} \leq n^i/2$, which passes as input to the next recursive level $i+1$ to calculate sa^{i+1} . When the algorithm returns to level i from level $i+1$, it performs another ISS pass. However, this time it uses the information in sa^{i+1} as the starting point for the induction, and thus the output of the pass becomes sa^i . If $i > 1$, SAIS returns sa^i to the previous level sa^{i-1} , otherwise it finishes.

Each new S^{i+1} is at most half the size of S^i . Consequently, the total cost of the algorithm is $\mathcal{O}(n)$ time and $\mathcal{O}(n \log n)$ bits of working space. SAIS can also be easily adapted to compute the generalized suffix array gsa of a string collection \mathcal{S} [48].

Karp-Rabin fingerprint. The *Karp-Rabin (KR) fingerprints* method [38] is a rolling hash technique that associates strings with integers. The KR fingerprints for all the substrings of length w of a string S of length n can be computed in $\mathcal{O}(n)$ time.

Run-length encoding. A *run* in a string S is a maximal substring consisting of repetitions of only one character, i.e. c^e for some $c \in \Sigma$ and $e \geq 1$. The *run-length encoding (RLE)* of a string S encodes any run c^e in S by a pair (c, e) .

Grammar compression. This technique consists of computing a small context-free grammar that generates only the string $S[1..n]$ [39, 16]. A grammar $\mathcal{G} = (\Sigma, V, \mathcal{R}, I)$ is a tuple where Σ is the alphabet of S , V is a set of nonterminal symbols, \mathcal{R} is a relation $V \times (\Sigma \cup V)^*$ describing the set of rewriting rules, and $I \in V$ is the start symbol. Each nonterminal $b \in V$ has a replacement $X[1..x] \in (\Sigma \cup V)^*$ encoded by a rule $b \rightarrow X[1..x] \in \mathcal{R}$. Given two strings $w_a = A[1..q] \cdot b \cdot B[1..b]$, $w_b = A[1..a] \cdot X[1..x] \cdot B[1..b] \in (\Sigma \cup V)^*$, w_b rewrites w_a (denoted $w_a \Rightarrow w_b$) if $b \rightarrow X[1..x]$ exists in \mathcal{R} . The *expansion* $\text{exp}(b) \in \Sigma^*$ of $b \in V$ is the string resulting from successive rewrites $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_k$ with $w_1 = b$ and $w_k = \text{exp}(b)$. Expanding the start symbol I produces $\text{exp}(I) = S[1..n]$. Grammar compression algorithms create one rewriting rule $b \rightarrow X[1..x]$ for every nonterminal $b \in V$, so there is one possible sequence of rewrites that starts in b and ends in $\text{exp}(b)$.

3 Disk-based strategies

When the data to be indexed exceeds the capacity of internal memory, external-memory algorithms become necessary to handle inputs efficiently. These algorithms are specifically designed to minimize random access and optimize disk I/O operations, as accessing external storage is significantly slower than accessing RAM. They typically rely on techniques such as sequential scans, buffering, and external sorting to process data in manageable chunks while reducing disk seek times. Efficient external-memory algorithms are crucial for indexing massive datasets where internal-memory approaches would be impractical due to space constraints.

3.1 bwt-disk [Ferragina et al., 2012]

The algorithm **bwt-disk**⁶ [26] is an external memory algorithm that constructs the BWT for a (single) input string S of length n , without first building the suffix array for the complete string. It operates using only M words of RAM and n bits of disk space, in addition to the disk space needed to store the input and the output. Moreover, all data on disk is accessed exclusively through sequential scans.

The algorithm partitions the input string $S[1..n]$ into consecutive blocks of length $m = \Theta(M)$, such that $S = S^{[n/m]} S^{[n/m]-1} \dots S^2 S^1$. Each block S^i represents a substring of S with length m , except possibly the last substring $S^{[n/m]}$, which may be shorter if n is not a multiple of m . The **bwt**(S) is computed through $\lceil n/m \rceil$ passes, one for each block, which are processed backwards, from S^1 to $S^{[n/m]}$. The key idea is that, during each pass $h+1$, only the characters of S^{h+1} need to be inserted into **bwt**($S^h \dots S^2 S^1$) to obtain **bwt**($S^{h+1} \dots S^2 S^1$), and the relative order of suffixes from $S^h \dots S^2 S^1$ remains unchanged.

⁶ An implementation is available at <https://people.unipmn.it/manzini/bwtdisk/>

In Example 1, suppose $S = CATGATGATA\$$ is partitioned into blocks of length $m = 4$. Then, we have $S^1 = ATA\$$, $S^2 = GATG$ and $S^3 = CAT$. Initially, $\text{bwt}(S^1) = AT\$A$ is computed in internal memory, and $\text{bwt}(S^2S^1)$ will be obtained by sequentially inserting each character of S^2 into $\text{bwt}(S^1)$.

For each pass $h + 1$ (for $h = 1, 2, \dots, \lceil n/m \rceil - 1$), the algorithm loads from disk $S^{h+1}S^h$ into a string buffer $s[1..2m]$. Also, an auxiliary bit array, called \mathbf{gt} , is assumed to be stored on disk, indicating whether each suffix in $S^h \dots S^2S^1$ is greater than the suffix $S^h \dots S^2S^1$ itself. The part of \mathbf{gt} referring to the suffixes starting in S^h is loaded to the bit array $\mathbf{gt}_h[1..m]$ in internal memory.

Continuing from Example 1, at pass $h + 1 = 2$, we have the string buffer $s = S^2S^1 = GATGATGATA\$$ and $\mathbf{gt}_1 = [0, 1, 0, 0]$.

Then, the pass $h + 1$ proceeds through the following four steps.

In *Step 1*, the suffix array $\mathbf{sa}[1..m]$ containing the lexicographic ordering of the suffixes of $S[1..n]$ starting in block S^{h+1} (extending up to the last character $S[n]$) is computed in internal memory, without accessing any character of $S^{h-1} \dots S^2S^1$ on disk. To do that, the suffixes are compared using their prefixes stored in $s[1..2m]$, whenever the comparison of two suffixes exceeds position $s[m + 1]$, the algorithm uses the corresponding value in \mathbf{gt}_h to determine their relative order. For example, given two suffixes starting at positions i and j of S^{h+1} , with $i < j$, if $s[i..m]$ is lexicographically smaller (or greater) than $s[j..j + m - i]$, then the suffix starting at i is smaller (or greater) than the suffix starting at j . Otherwise, if $s[i..m] = s[j..j + m - i]$, their relative order is determined by the corresponding bit stored in $\mathbf{gt}_h[j - i + 1]$.

In Example 1, at pass $h + 1 = 2$, the resulting suffix array for S^2 , $\mathbf{sa} = [2, 4, 1, 3]$, is computed using only the string buffer. Therefore, the suffix ordering in S^2S^1 is: $ATGS^1, GS^1, GATGS^1, TGS^1$.

In *Step 2*, the array $\mathbf{bwt}_{int}[1..m]$ is computed given $\mathbf{sa}[1..m]$. For each $i = 1, \dots, m$, the value of $\mathbf{bwt}_{int}[i]$ is set to $S^{h+1}[\mathbf{sa}[i] - 1]$ if $\mathbf{sa}[i] \neq 1$; otherwise, $\mathbf{bwt}_{int}[i]$ is assigned the special character $\#$, which does not appear in S . Note that $\mathbf{bwt}_{int}[1..m]$ is not the actual BWT of the substring S^{h+1} , as the suffixes of S^{h+1} extend up to the last character $S[n]$ and their relative order may differ.

Continuing from Example 1, at pass $h + 1 = 2$, $\mathbf{bwt}_{int} = GT\#A$ is computed from $\mathbf{sa}[1..4]$ and $S^2 = GATG$.

In *Step 3*, the algorithm computes a counter array, $\mathbf{gap}[1..m]$, which gives in $\mathbf{gap}[j]$ how many suffixes of $S^h \dots S^2S^1$ lie lexicographically between the suffixes in positions $\mathbf{sa}[j - 1]$ and $\mathbf{sa}[j]$ from the current block S^{h+1} . This counter array is computed in $\mathcal{O}(n)$ time with a single scan over $S^h \dots S^2S^1$ using n extra bits of \mathbf{gt} and an $o(m)$ -bit data structure supporting constant time rank queries over \mathbf{bwt}_{int} . At *Step 3*, the algorithm also computes the content of \mathbf{gt}_{h+1} for the next pass as a by-product of the computation of \mathbf{gap} .

Again, in Example 1, at pass $h + 1 = 2$, we have $\mathbf{gap} = [3, 0, 0, 1]$, since three suffixes from S^1 (namely $\$, A\$, and ATA\$$) are smaller than the first suffix of S^2 , and the suffix $TA\$$ from S^1 is smaller than only the last suffix of S^2 .

Finally, in *Step 4*, the algorithm merges $\mathbf{bwt}(S^h \dots S^2S^1)$ with \mathbf{bwt}_{int} to create for the next pass $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$. To do that I/O-efficiently, the algorithm uses the counter array \mathbf{gap} , such that, for $i = 1, \dots, m$, $\mathbf{gap}[i]$ characters from $\mathbf{bwt}(S^h \dots S^2S^1)$ are copied to $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$ followed by the value $\mathbf{bwt}_{int}[i + 1]$. Whenever a character $\$$ or $\#$ is retrieved from $\mathbf{bwt}(S^h \dots S^2S^1)$ during this process, it is replaced by the last symbol $S^{h+1}[m]$ in $\mathbf{bwt}(S^{h+1}S^h \dots S^2S^1)$.

In Example 1, at the end of pass $h+1 = 2$, $\text{gap}[1] = 3$, then the first three values of $\text{bwt}(S^1)$ are retrieved (namely $AT\$$) and copied to $\text{bwt}(S^2S^1)$ as ATG directly to disk. Then, the value in bwt_{int} (namely G) is copied to $\text{bwt}(S^2S^1)$, together with the next two values in bwt_{int} (namely T and $\#$) as $\text{gap}[2] = 0$ and $\text{gap}[3] = 0$, resulting in $ATGGT\#$. Finally, $\text{gap}[4] = 1$ and the last value from $\text{bwt}(S^1)$ (namely A) is copied to $\text{bwt}(S^2S^1)$ and the last value from bwt_{int} (namely A) is copied as well, yielding the final result: $\text{bwt}(S^2S^1) = ATGGT\#AA$.

Theoretical costs. The running time of bwt-disk is $\mathcal{O}(n^2/M)$, for a (single) string $S[1..n]$, from a constant alphabet, using $M \lceil \log n \rceil$ bits of RAM. The algorithm executes $\mathcal{O}(n/M)$ passes over $\Theta(n)$ bits of disk data, using n bits of disk working space. The total number of I/Os is $\mathcal{O}(n^2/(MB \log n))$.

3.2 BCR and BCReXt [Bauer et al., 2013]

In this section we describe the core of a strategy for constructing the BWT of a string collection $\mathcal{S} = \{S_1, \dots, S_k\}$ according to the definition in Equation (1), which considers the strings in a circular way so that the symbol preceding the suffix $S_j[1..n_j]$ is the end-marker of the string S_j (i.e., $\$j$). The heart of this strategy is common to two algorithms, named BCR and BCReXt, introduced in [5] and part of the BEETL library⁷. BCR works in (semi-)external memory, whereas BCReXt works in external memory.

The common strategy does not require linearizing the strings in \mathcal{S} by concatenating them and does not require explicitly computing the GSA. Rather, it builds the BWT incrementally in m steps, where m is the length of the longest string in \mathcal{S} (including the end-markers) by exploiting the basic properties of the BWT and the LF-mapping. The basic idea is to left-align (or, equivalently, right-align) all the strings in \mathcal{S} and, by scanning all the strings in \mathcal{S} right-to-left at the same time, to incrementally construct the partial BWT of suffixes of \mathcal{S} with length at most $h+1 \leq m$, with $h = 0, \dots, m-1$.

For simplicity, in this description, we assume that all the strings have the same length m and they are right-aligned. So, at each iteration only suffixes of the same length are considered. Both algorithms build the $\text{bwt}(\mathcal{S})$ in m steps. At each step $h = 0, \dots, m-1$, they build a partial BWT, denoted $\text{bwt}_h(\mathcal{S})$, containing the BWT of suffixes in \mathcal{S} of length up to $h+1$.

Since we consider distinct end-markers and $\$i < \j if $i < j$, then it is easy to verify that $\text{bwt}_0(\mathcal{S})$ is obtained by concatenating the last symbol of each string S_j in the same order as the strings appear in the collection: $\text{bwt}_0(\mathcal{S}) = S_1[n_1-1]S_2[n_2-1] \dots S_k[n_k-1]$.

For each step $h = 1, \dots, m-2$, both algorithms simulate the insertion of all suffixes of length $h+1$ in the list of sorted suffixes, i.e., insert all symbols (circularly) preceding the suffixes of length $h+1$ at the correct positions into $\text{bwt}_{h-1}(\mathcal{S})$ in order to obtain $\text{bwt}_h(\mathcal{S})$.

In the last step $h = m-1$, the k end-markers are inserted at the correct positions into $\text{bwt}_{m-2}(\mathcal{S})$ and $\text{bwt}_{m-1}(\mathcal{S}) = \text{bwt}(\mathcal{S})$ is obtained.

At the iteration $h > 0$, let p_{j-1} be the position of the symbol $c = S_j[n_j-h]$ in $\text{bwt}_{h-1}(\mathcal{S})$. To insert the symbol $S_j[n_j-h-1]$ preceding the suffix $S_j[n_j-h, n_j]$ in $\text{bwt}_{h-1}(\mathcal{S})$, we compute the position p_j using the LF-mapping (we omit details for space reasons): $p_j = C[c] + \text{rank}(c, p_{j-1})$, where $C[c]$ is the number of symbols lexicographically smaller than c in $\text{bwt}_{h-1}(\mathcal{S})$, $\text{rank}(c, p_{j-1})$ is the function that returns the number of occurrences of c in the prefix $\text{bwt}_{h-1}(\mathcal{S})[1, p_{j-1}]$.

⁷ <https://github.com/BEETL/BEETL>

By using Example 2, the base case of BCR/BCRext consists in concatenating all symbols preceding the suffixes $\$1, \$2, \$3$ obtaining $\text{bwt}_0(\mathcal{S}) = TCA$ (because the partial sorted suffixes is: $[\$1, \$2, \$3]$ since $\$1 < \$2 < \$3$). Then, they simulate the insertion of the $T\$1, C\$2, A\$3$ suffixes into the partial sorted suffix list by inserting the symbols G, A, A preceding the $T\$1, C\$2, A\$3$ respectively suffixes into bwt_0 , resulting in $\text{bwt}_1 = TCAAAG$. The process continues by simulating the insertion of the suffixes $GT\$1, AC\$2, AA\$3$ into the sorted suffix list, then inserting the symbols C, A, C preceding the $GT\$1, AC\$2, AA\$3$ suffixes in order to obtain $\text{bwt}_2(\mathcal{S})$. And so on. In the last step, they simulate the inserting of the suffixes $AGCGT\$1, TCAAC\$2, CGCAA\$3$ and insert the symbols $\$1, \$2, \$3$ in the partial BWT $\text{bwt}_{m-1}(\mathcal{S}) = \text{bwt}(\mathcal{S})$.

If the strings have different lengths, an end-marker is inserted when the longest suffix of a string in \mathcal{S} is reached, and then no further symbols of that string will be inserted in the following iterations.

Note that, in practice, neither algorithm needs to explicitly append end-marker symbols to the strings, since they will only be inserted last and the symbols concatenated in bwt_0 follow the input order. Moreover, to avoid exponential growth of the alphabet, both algorithms write the same end-marker symbol in the BWT output string, but they can store and output the index associated with each end-marker in a separate file. This is particularly important for decoding. An interested reader can refer to [5, 6].

In order to reduce the I/O operations, both algorithms split each partial BWT $\text{bwt}_{h-1}(\mathcal{S})$ in $\sigma + 1$ files $B_{h-1}(t)$, where $B_{h-1}(0)$ contains the symbols preceding the suffixes consisting of only the end-marker symbols and $B_{h-1}(t)$ contains the symbols preceding the lexicographically sorted suffixes of \mathcal{S} of length at most $h - 1$ and starting with $c_t \in \Sigma$.

The difference between the two algorithms lies in the data structures they employ, and therefore in the strategy they adopt to insert symbols incrementally.

In fact, BCR needs a data structure to keep track of each symbol to be inserted, the index of the sequence to which it belongs, and the position where each symbol was inserted in the previous iteration. Once the positions at which the new symbols are to be inserted have been updated, BCR needs to sort this information so that, at iteration h , it can insert the symbols into each of the B_{h-1} files sequentially.

BCRext computes and updates the positions where new symbols are to be inserted in a similar way to BCR, but it keeps this information in auxiliary files and the strings themselves are sorted externally by using the least-significant-digit radix sort. So it uses a negligible workspace. At the start of iteration h , the suffixes of length $h - 1$ of \mathcal{S} are assumed to be ordered, this ordering being partitioned into external files $F_{h-1}(1), \dots, F_{h-1}(\sigma)$ according to the first symbols of the suffixes of length $h - 1$. Files $P_{h-1}(1), \dots, P_{h-1}(\sigma)$ are such that $P_{h-1}(s)$ contains the positions of the suffixes in $B_{h-1}(s)$, ordered the same way.

In [20], the authors show that BCR can be augmented in order to compute the LCP array and the generalized suffix array.

Moreover, in both algorithms the collection \mathcal{S} is considered ordered by using the input order, but one can modify them so that one can use a different ordering for the strings in \mathcal{S} to reduce the number of runs in the output of the BWT (see [19, 44, 6, 13, 14]).

Theoretical costs. BCR is performed in $\mathcal{O}((k + \sigma^2) \log N)$ bits of memory, with a worst-case time complexity of $\mathcal{O}(m(N + \text{sort}(k)))$, where $\text{sort}(k)$ is the time taken to sort k integers and $\mathcal{O}(Nm \log(\sigma))$ I/O (bits). Whereas BCRext uses $\mathcal{O}(\sigma^2 \log(N))$ bits of memory, with a worst-case time complexity of $\mathcal{O}(kN)$ and $\mathcal{O}(Nm \log(\sigma) + N \log(N))$ I/O (bits).

3.3 eGSA [Louza et al., 2017]

The algorithm eGSA⁸ [49] works in external memory and computes the (generalized) BWT for a string collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of total length N according to the definition in Equation (1), with each string $S_j \in \mathcal{S}$ having the same end-marker $\$$ for computational efficiency. The algorithm works in two phases, as follows.

Phase 1. The collection \mathcal{S} is partitioned into sub-collections $\mathcal{S}^1, \mathcal{S}^2, \dots, \mathcal{S}^m$ based on the available memory, such that each \mathcal{S}^x has size less than $\lfloor M/4 \rfloor$, where M denotes the number of available words in RAM. For each sub-collection \mathcal{S}^x , the arrays \mathbf{gsa}^x and \mathbf{lcp}^x are computed in internal memory and stored to disk. The authors used gSACA-K [48] combined with Φ -algorithm [36]. Together, these algorithms run in linear. As each value $\mathbf{gsa}^x[i] = (t, j)$ and $\mathbf{lcp}^x[i]$ are written to disk, the corresponding $\mathbf{bwt}^x(\mathcal{S}^x)[i]$ and a substring of length p of the suffix $S_j[t, n_j]$, denoted as $\mathbf{pre}^x[i]$, are computed and written to disk. In particular, $\mathbf{pre}^x[i]$ gives a prefix of $S_j[t, n_j]$ when combined with previous values and it is used to reduce external memory accesses in Phase 2.

In Example 2, suppose the input collection \mathcal{S} is partitioned into two sub-collections $\mathcal{S}^1 = \{AGCGT\$1, TCAAC\$2\}$ and $\mathcal{S}^2 = \{CGCAA\$3\}$, with prefix size $p = 2$. First, $\mathbf{gsa}^1 = [(1, 5), (2, 5), (2, 2), (2, 3), (1, 0), (2, 4), (2, 1), (1, 2), (1, 1), (1, 3), (1, 4), (2, 0)]$ and $\mathbf{lcp}^1 = [0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1]$ are computed in internal memory. While each pair $\mathbf{gsa}^1[i]$ and $\mathbf{lcp}^1[i]$ is written to disk, the corresponding values in $\mathbf{bwt}^1(\mathcal{S}^1) = TCCA\$ATGACG\$$ and $\mathbf{pre}^1 = [\$ \$, \$ \$, AA, AC, AG, C \$, CA, CG, GC, GT, T \$, TC]$ are obtained and stored on disk as well. Next, $\mathbf{gsa}^2 = [(3, 5), (3, 4), (3, 3), (3, 2), (3, 0), (3, 1)]$ and $\mathbf{lcp}^2 = [0, 0, 1, 0, 1, 0]$ are computed in internal memory, and the values of $\mathbf{bwt}^2(\mathcal{S}^2) = AACG\C and $\mathbf{pre}^2 = [\$ \$, A \$, A \$, CA, GC, GC]$ are obtained and stored on disk in the same way.

Phase 2. The computed arrays of $\mathcal{S}^1, \mathcal{S}^2, \dots, \mathcal{S}^m$ are merged with the help of a binary heap and internal memory buffers designed to reduce string comparisons on the disk. Let $\mathbf{gesa}^x = \langle \mathbf{gsa}^x, \mathbf{lcp}^x, \mathbf{bwt}^x, \mathbf{pre}^x \rangle$ be the computed arrays of \mathcal{S}^x . Each \mathbf{gesa}^x is split into r^x blocks of b elements, except possibly the last block, ensuring that one block of each $\mathbf{gesa}^1, \mathbf{gesa}^2, \dots, \mathbf{gesa}^m$ can simultaneously fit internal memory. Initially, the first block of each \mathcal{S}^x is loaded from disk to a buffer $B^x[1, b]$. The heading element of each B^x is inserted into a binary min-heap H . For $i = 1, 2, \dots, N$, the smallest suffix, say from B^x , is removed from H , and the value of $\mathbf{bwt}(\mathcal{S})[i]$ is written to an output buffer O of size d . Whenever O is full, it is written to the disk. Then, the next element from the same block buffer B^x is inserted to the heap. Whenever a buffer B^x is empty, the next corresponding block in \mathbf{gesa}^x is loaded from disk to B^x .

In Example 2, considering the block size $b = 3$, we have the following initial blocks in the buffers $B^1 = [\langle (1, 5), 0, T, \$ \$ \rangle, \langle (2, 5), 0, C, \$ \$ \rangle, \langle (2, 2), 0, C, AA \rangle]$ and $B^2 = [\langle (3, 5), 0, A, \$ \$ \rangle, \langle (3, 4), 0, A, A \$ \rangle, \langle (3, 3), 1, C, A \$ \rangle]$. We assume there is enough internal memory to hold B^1 and B^2 simultaneously. The heading elements $\langle (1, 5), 0, T, \$ \$ \rangle$ and $\langle (3, 5), 0, A, \$ \$ \rangle$ are inserted into the heap H .

Note, however, that each heap comparison during Phase 2 may require accessing strings in disk randomly. To reduce disk accesses, the following strategies are presented.

⁸ An implementation is available at <https://github.com/felipelouza/egsa>

For each sub-collection \mathcal{S}^x a string buffer W^x of maximum size s is used to store a prefix of the heading suffix from B^x in internal memory. Therefore, whenever a suffix from \mathcal{S}^x , say $\text{gesa}^x[i]$ with $\text{gsa}^x[i] = (t, j)$, is inserted into H , if $\text{lcp}[i] = 0$ then $W^x = \text{pre}^x[1, p] \cdot \#$, otherwise, $W^x = W^x[1, h_i] \cdot \text{pre}^x[1, p] \cdot \#$, where $h_i = \min(\text{lcp}^x[i], h_{i-1} + p)$ and $\#$ is an end-of-buffer marker. In this way, a starting portion of $S_j[t, n_j]$ is already loaded into W^x when it is compared with other suffixes in H . If $\#$ is reached, S_j is accessed in disk. Also, if $h_i + p > s$ the prefix is truncated and the next substrings are accessed from disk.

Continuing from Example 2, when the elements compared in H are $\langle(2, 2), 0, C, AA\rangle$ from B^1 and $\langle(3, 4), 0, A, A\$ \rangle$ from B^2 , their string buffers $W^1 = AA\#$ and $W^2 = A\#\#$ provide enough information to decide that the smallest suffix comes from B^2 (without accessing the disk) and $\text{bwt}(\mathcal{S})[i] = A$ is added to the output buffer O . The next element from B^2 , $\langle(3, 3), 1, C, A\$ \rangle$, is inserted into the heap H . The prefix W^2 is updated by considering that the current element from B^2 shared one symbol in common with the previous one, resulting in $W^2 = AA\#\#$.

Another strategy is the usage of the LCP between nodes in H to avoid string comparisons in heap insertions. Let X from B^x be the root of H and nodes Y and Z its children, if the LCP between X and the next element from B^x is larger than the LCP between X and Y , and the LCP between X and Z , the next smallest value of H comes from buffer B^x . Also, if the LCP between X and the next element from B^x is smaller or equal to the LCP between X and Y , the comparison between their corresponding suffixes can start from the minimum LCP value. The LCP values in H are updated as nodes are swapped and can be used to speed up suffix comparisons as well.

In Example 2, again when the elements $\langle(2, 2), 0, C, AA\rangle$ from B^1 and $\langle(3, 4), 0, A, A\$ \rangle$ from B^2 are compared in H , the LCP between the corresponding nodes in H is computed and stored. Subsequently, the node representing B^2 is removed from H , and the element $\langle(3, 3), 1, C, A\$ \rangle$ is inserted. Since the LCP between the new element from B^2 and the previous one is equal to 1 and the LCP between the corresponding nodes in H is also equal to 1, the next comparison in H between the elements from B^1 and B^2 may start from the second symbol of W^1 and W^2 . In particular, this suffix comparison will require accessing the disk, as W^1 is a prefix of W^2 .

Finally, eGSA also induces suffixes as each $\text{gesa}^x[i]$ is removed from H , with $\text{gsa}^x[i] = (t, j)$. Whenever $\text{bwt}^x[i] > S_j[t]$ the suffix $S_j[t - 1, n_j]$ will be induced. To do that, the value x is inserted into a (queue) induced buffer I_c , with $S_j[t - 1] = \text{bwt}^x[i] = c$. Each induced buffer I_c , with $c \in \Sigma$, is written to disk as it gets full. Then, when the first suffix starting with c is the smallest value in H , the induced values in disk and in I_c are used to decide which heading element of the blocks is the smallest in H and send it to the output buffer, with no string comparison. As these values are sent to O the corresponding nodes in H are consumed.

In Example 2, when the element $\langle(1, 5), 0, T, \$\$ \rangle$ from B^1 is the smallest in H , the suffix $S_1[4, 5] = T\$$ will be induced. To do that, $x = 1$ is inserted into the induced buffer I_T . Then, when the first suffix starting with T becomes the smallest in H , the heading element $\langle(1, 4), 0, G, T\$ \rangle$ from B^1 is send directly to the output buffer, without string comparison.

Theoretical costs. The running time of eGSA is $\mathcal{O}((N \log N/M) \text{maxlcp})$, where maxlcp is length of the longest common prefix between suffixes of the input strings. The total number of I/Os is $\mathcal{O}((N \log N/M) \max |S^i|)$, where $\max |S^i|$ is the length of the longest string in the collection.

3.4 eGap [Egidi et al., 2019]

The algorithm **eGap**⁹ [23] is an external memory algorithm for the computation of the BWT of a collection of strings $\mathcal{S} = \{S_1, \dots, S_k\}$, with $S_i \neq S_j$ for $i \neq j$. The algorithm works in two phases, first using the available RAM to compute in memory the BWTs of subcollections of \mathcal{S} , and then merging the results into the final BWT.

Phase 1. **eGap** uses the optimal linear time algorithm **gSACA-K** [48] for the first phase; the size of the subcollections is determined based on the available memory, taken in input. The **gSACA-K** algorithm is used to compute the suffix array for subcollections, from which, by Equation (2), the multi-string $\text{bwt}(\mathcal{S})$ is obtained.

Phase 2. The second phase is based on the **gap** algorithm [24], in turn derived from an earlier algorithm by Holt and McMillan [32]. The ideas from **gap** are used to merge the BWTs of the subcollections.

For simplicity of exposition, we describe here the algorithm when merging single-string BWTs, but the same algorithm can merge multi-string BWTs. The latter is important because it allows to adopt a multi-stage strategy in which the BWTs computed in Phase 1 are merged in successive rounds.

The merging phase does not need to access the original strings but requires only their BWTs $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$. It works iteratively sorting the bwt_{S_λ} symbols according to prefixes of increasing lengths of their contexts: after iteration h , they are sorted according to the length- h prefixes of their contexts. The original BWTs are not explicitly sorted during the iterations, but the algorithm produces a recipe for performing the merge: the final iteration yields a length n , k valued vector Z such that $Z[i] = \lambda$ iff the i -th entry of $\text{bwt}(\mathcal{S})$ is from bwt_{S_λ} ($i = 1, \dots, N$; $\lambda = 1, \dots, k$).

More formally, let $Z^{(h)}$ be the approximation of array Z after iteration h , with $Z^{(0)} = 1^{n_1} 2^{n_2} \dots k^{n_k}$. Then the following property holds:

Property Z. For $\lambda_1, \lambda_2 \in \{1, \dots, k\}$, $\lambda_1 < \lambda_2$, and $i = 1, \dots, n_{\lambda_1}$ and $j = 1, \dots, n_{\lambda_2}$ the i -th λ_1 precedes the j -th λ_2 in $Z^{(h)}$ iff $S_{\lambda_1}[\text{sa}_{\lambda_1}[i], \text{sa}_{\lambda_1}[i] + h - 1] \preceq S_{\lambda_2}[\text{sa}_{\lambda_2}[j], \text{sa}_{\lambda_2}[j] + h - 1]$. ◀

For instance, assume that we are merging the three BWTs of the strings of Example 2. The BWTs are $\text{bwt}_{S_1} = T\$1GACG$, $\text{bwt}_{S_2} = CCAAT\2 and $\text{bwt}_{S_3} = AACG\$3C$. Then, $Z^{(2)} = [1, 2, 3, 3, 2, 3, 2, 1, 2, 2, 3, 1, 3, 1, 3, 1, 1, 2]$ which corresponds to a partially merged **bwt** $TCAACCA\$1ATGG\$3ACCG\$2$, where the suffixes are ordered only by their length two prefixes. For instance, the twelfth and thirteenth character are $G\$3$ as opposed to $\$3G$ in the final $\text{bwt}(\mathcal{S})$ since their suffixes both start by CG and G comes from bwt_{S_2} whereas $\$3$ is from bwt_{S_3} .

Array $Z^{(h)}$ is obtained through a sequential scan of $Z^{(h-1)}$ together with $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$. The array $Z^{(h)}$ is partitioned into σ buckets, one for each character of the alphabet. This is achieved through the array $C[1, \sigma]$, where $C[c]$ ¹⁰ is defined as the number of occurrences of characters smaller than c in $\text{bwt}(\mathcal{S})$. Therefore $C[c]$ marks the beginning of the c -bucket in $Z^{(h)}$, whose first entry is $Z^{(h)}[C[c]]$. When scanning $Z^{(h-1)}$, if $Z^{(h-1)}[i]$ is λ , the algorithm looks at the next character in bwt_{S_λ} : if it is c , then it stores λ in the next free position of

⁹ An implementation is available at <https://github.com/felipelouza/egap/>

¹⁰ In [23] the array C is denoted F .

bucket c in $Z^{(h)}$. So, continuing the example above, the character C bucket of $Z^{(3)}$ will be 22331. Notice that now the twelfth character will be S_3 since its suffix is prefixed by CGC and the thirteenth character will be G , since its suffix starts with CGT .

From Property Z, it follows that $Z^{(h)}$ can be logically partitioned in blocks corresponding to sets of $\text{bwt}(S)$ symbols whose contexts share the same length- h prefixes. An additional bit array B keeps track of these blocks: $B[i] \neq 0$ iff a new block starts at $Z^{(h)}[i]$. When all entries of array B are non zero, $Z^{(h)}$ will not change in subsequent iterations and it is the required array Z . In our example, at the last iteration $Z = [1, 2, 3, 3, 3, 2, 2, 1, 2, 3, 2, 3, 1, 1, 1, 2]$.

Then, a sequential scan of Z and $\text{bwt}_{S_1}, \dots, \text{bwt}_{S_k}$ allows to write the characters of $\text{bwt}(S)$ in sequential order. Back to our example: $\text{bwt}(S)[0]$ is the first from $\text{bwt}_{S_{Z[0]}} = \text{bwt}_{S_1}$, so it's T ; the second is the first from $\text{bwt}_{S_{Z[1]}} = \text{bwt}_{S_2}$ (it's C); then we have an A from $\text{bwt}_{S_{Z[2]}} = \text{bwt}_{S_3}$; and then the second character from bwt_{S_3} (again an A) since $Z[3] = 3$.

The k input BWTs are read from disk and never moved to memory.

Phase 2 of **eGap** actually only uses two copies of array Z to store $Z^{(h)}$ for all values of h : namely, Z^{old} and Z^{new} , whose roles are swapped at each iteration. They can be maintained in external memory, since access to each bucket of Z^{old} and Z^{new} is sequential.

In [24, Lemma 5], it is proven that if $B[i]$ is set to 1 at iteration h , then $Z^{(g)}[i]$ will not change any more at any iteration $g \geq h + 2$. Since the roles of Z^{old} and Z^{new} are swapped, starting at iteration g , processing $Z^{\text{old}}[i]$ to compute $Z^{\text{new}}[i]$ will be useless. So we define an array B_x to keep track of entries that will not change again: if at iteration h the algorithm finds that $B[i] = 1$, then $B_x[i]$ will be set to 1 as well. Any sequence of 1s in B_x defines an *irrelevant range* that can be skipped in the subsequent processing. So, at each iteration, irrelevant ranges are sequentially read from file to skip them, and new irrelevant ranges are sequentially written to file. Since maintaining and skipping ranges has a cost, only ranges of a significant size (according to a configurable parameter) are considered.

Computing the LCP. The **eGap** algorithm can also compute the LCP of the input collection. This is done in two steps: during Phase 2, if at iteration h , $B[i] = 1$ and $B_x[i] = 0$, a pair $\langle i, h - 2 \rangle$ is written to file F_{h-2} to record that $\text{lcp}_{1..k}[i] = h - 2$ (then $B_x[i]$ is set to 1). Notice that in file F_{h-2} all entries are ordered according to their first components. In an additional third phase of the algorithm, the entries from all temporary files F_h are merged using a standard external memory multiway merge.

Theoretical costs. We analyze costs assuming a fixed size alphabet. Phase 1 of the algorithm runs in total time $\mathcal{O}(N)$, splitting the input into subcollections of size $\approx M/9$, if M is the available RAM.

Without skipping the irrelevant ranges, Phase 2 would require maxlcp (the maximum LCP value) sequential scans of $\mathcal{O}(N)$ items. By skipping irrelevant ranges, the overall amount of data *directly* read/written by the algorithm is $\mathcal{O}(N \text{ avelcp})$ items where avelcp is the arithmetic average of the entries in the LCP array. On the other hand, skipping irrelevant ranges causes an overhead in terms of blocks; so the overall cost of Phase 2 can be bound above by $\mathcal{O}(N \text{ maxlcp})$ sequential I/Os.

RAM usage is flexible. In Phase 1, the larger the memory used, the faster the algorithm. Phase 2 uses a negligible amount of RAM, but also a semi-external version that uses $\mathcal{O}(N)$ bytes of RAM was implemented.

The overall cost of Phase 3, to sort the LCP entries, is $\mathcal{O}(N \log_K \text{ maxlcp})$ sequential I/Os, since it requires $\mathcal{O}(\lceil \log_K \text{ maxlcp} \rceil)$ rounds (where K is a configurable parameter), each one merging K LCP files by sequentially reading and writing $\mathcal{O}(N)$ bytes of data.

3.5 BWT+LCP [Bonizzoni et al., 2021]

The BWT+LCP algorithm [8], implemented in a prototype called `bwt-lcp-em`¹¹, has similarities with BCR/BCRext [5] (Section 3.2) and uses ideas from [32], but proposes a different strategy.

The paper [8] uses a specific definition of suffixes that, contrary to the definition adopted in this survey, does not include the end-marker when counting the length of a suffix. So a suffix can have length 0 and in this case it is not the empty string but it consists of the end-marker. We chose here to use a notation that agrees with this choice, for the sake of the reader who wants to delve into the details of [8]. To avoid confusion, in the following we will call these suffixes *no\$-suffixes*.

The algorithm works in two distinct phases.

It first computes partial multi-string BWTs, and then merges the BWTs to the final BWT, making largely use of external memory.

The partial BWT¹² $\text{bwt}_{=j}$ only takes into account no\$-suffixes of length j . Notice the difference from the partial BWTs bwt_j of BCR (Section 3.2) that are relative to all suffixes of lengths up to j . Also notice the difference from eGap (Section 3.4) where the partial BWTs computed in the first phase are complete BWTs of subcollections.

Phase 1. The partial BWTs are computed with the support of positional representations of the input strings: these are arrays T_i such that element $T_i[j]$ is the i -th character from the end of string S_j , *without* the end-marker \$. The end-marker is added as a first character to each string. So T_0 contains all last characters of the strings.

The first partial BWT $\text{bwt}_{=0}$ contains the concatenation of the last characters of each string, since it is the BWT of length 0 no\$-suffixes (i.e. suffixes consisting only of the end-markers), and so is identical to T_0 .

To produce $\text{bwt}_{=j}$ from $\text{bwt}_{=j-1}$, for each $j = 1, \dots, \ell$ (where ℓ is the maximum length of the strings), a radix sort is used, ordering the strings from the rightmost character. In this phase array T_j must be kept in main memory since it is accessed in random order. The sorting is implemented using arrays N_i ($i = 1, \dots, \ell$). At the beginning of the j -th iteration, $N_j[i] = h$ if $\text{bwt}_{=j+1}[i]$ must be read from string S_h . As N_j is scanned, characters from the required strings are read: if character c is read from string S_h , it is appended to $\text{bwt}_{=j}$ which is under construction and h is appended to the bucket for character c . The concatenation of all buckets, ordered according to the alphabetical ordering of the characters, is the new N_{j+1} . The array N_0 is trivially initialized as $N_0 = \mathbf{12} \cdots \mathbf{k}$.

Let's consider again the strings of Example 2. To show how Phase 1 works, let's see how $\text{bwt}_{=4}$ is obtained from $N_3 = 231$ and T_4 . Notice that $T_4 = ATC$ since it contains the 5th character from the end of each string, not counting the end marker. According to N_3 , the first character must be read from string 2, so the first character of $\text{bwt}_{=4}$ is T . At the same time, the first index in the T bucket is 2. The second character must come from S_3 (it is C , and bucket C gets a 3), and the third from S_1 (it is an A and bucket A gets a 1). So $\text{bwt}_{=4} = TCA$ and $N_4 = 132$, as the concatenation of the buckets for A , C and T in alphabetical order.

¹¹<https://github.com/AlgoLab/bwt-lcp-em>

¹²In [8] the partial BWTs are denoted bwt_j , and the maximum length is k . We use here the “=” in $\text{bwt}_{=j}$ to explicitly distinguish these partial BWTs from the partial BWTs bwt_j computed in BCR.

Phase 2. When all partial BWTs $\text{bwt}_{=j}$ have been built, they are merged to the final BWT. Notice that the ordering of each partial BWT will be preserved in the merge, since length- j no\$-suffixes are already correctly ordered relative to each other.

The merge is first carried out virtually building an array I that describes how the merge must be carried out. Specifically $I[j] = h$ if the j -th element of the final BWT is the character preceding a length- h suffix. Therefore, to construct the output BWT, element j will be taken from $\text{bwt}_{=h}$.

To build I , the no\$-suffixes of different lengths must be ordered. I is approximated in successive iterations, and the no\$-suffixes are ordered starting from their last characters and using a radix sort algorithm.

Precisely, at iteration j , an approximation I_j is computed from I_{j-1} . The initial I_0 trivially proposes to concatenate all the partial BWTs in increasing order of the suffix lengths, that is $I_0 = \mathbf{0}^{|\text{bwt}_{=0}|} \mathbf{1}^{|\text{bwt}_{=1}|} \dots \ell^{|\text{bwt}_{=\ell}|}$.

Then, at iteration j , I_j is scanned sequentially, and for each i , if $I_j[i] = h$ then the next character c is read from $\text{bwt}_{=h}$. If it is not the end-marker, the length $h + 1$ is appended to the c -bucket. The \$-bucket is fixed as a sequence of 0s, since the suffixes starting with the end-markers are no\$-suffixes of length 0. Then I_{j+1} is the concatenation of all buckets, in alphabetical order of the characters.

Let us see this in our running example: the partial bwts are: $\text{bwt}_{=0} = TCA$, $\text{bwt}_{=1} = AAG$, $\text{bwt}_{=2} = CAC$, $\text{bwt}_{=3} = CGG$, $\text{bwt}_{=4} = TCA$ and $\text{bwt}_{=5} = \$_1\$_3\$_2$. Scanning $I_0 = \mathbf{0}^3 \mathbf{1}^3 \mathbf{2}^3 \mathbf{3}^3 \mathbf{4}^3 \mathbf{5}^3$, since $I_0[0] = 0$, the first character is from $\text{bwt}_{=0} = TCA$. Then the character is T , and so bucket T gets as first index $I_0[0] + 1 = 1$. The second character is again from $\text{bwt}_{=0}$, since $I_0[1] = 0$; it is a C and thus bucket C gets as first index $I_0[1] + 1 = 1$. All buckets are filled continuing in this fashion, except that when an end marker is encountered, it is disregarded: so, for instance, nothing is done for $I_0[15] = 5$, since the first character of $\text{bwt}_{=5}$ is $\$$. Finally, the concatenation of all buckets in alphabetical order yields $I_1 = [0, 0, 0, 1, 2, 2, 3, 5, 1, 3, 3, 4, 5, 2, 4, 4, 1, 5]$

From the final I , $\text{bwt}(S)$ is computed as it is done in **eGap** using the array Z , except that here the partial bwts $\text{bwt}_{=j}$ are used. In our example, $I = [0, 0, 0, 1, 2, 3, 2, 5, 1, 3, 4, 5, 3, 4, 4, 2, 1, 5]$. The first three characters of $\text{bwt}(S)$ are TCA from $\text{bwt}_{=0}$, since I starts with three 0s. Then we have an A which is the first character of $\text{bwt}_{=1}$, since $I[3] = 1$, and so on.

Computing the LCP. As implied by the name, the BWT+LCP algorithm [8] can also compute the LCP array of the input sequence collection. This is done during the second phase: as the array I is computed, at each iteration, the algorithm keeps track of the longest prefixes locally encountered at each location, by inspecting the corresponding BWT characters.

Theoretical costs. For a fixed size alphabet, the algorithm runs in $\mathcal{O}(N \text{maxlcp})$ time, where maxlcp is the maximum LCP value, when the alphabet is constant and memory addresses can be stored in a single memory word. It uses $\mathcal{O}(k + M + \log \text{maxlcp})$ main memory (where M is the maximum length of the input strings) and requires $\mathcal{O}(N \text{maxlcp})$ I/Os.

4 Strategies exploiting compressibility

Another way to scale the computation of large BWTs is to apply a lightweight compression scheme on top of the text to produce a small data representation from which we derive the BWT. This strategy has several advantages: (i) the compressed representation fits smaller

memories that are closer to the CPU, so the transmission of data is more efficient. We need fewer bits of satellite information on top of the text (i.e., less RAM and disk usage), and (ii) the deduplication of strings via compression avoids redundant BWT computations. Overall, if we use the correct tools, this approach reduces both time and space. However, it also has some pitfalls. A greedy compression scheme (like Lempel-Ziv or RePair) greatly reduces space usage, but also imposes considerable overhead, overcoming the whole purpose of compressing. Moreover, the compact data representation we produce must follow some structure that facilitates the computation of the BWT, otherwise decompressing the strings every time we require to operate over them makes things more inefficient.

In this section, we present algorithms that exploit the repetitiveness of the text to compute the BWT.

4.1 Big-BWT [Boucher et al., 2019]

Boucher *et al.* [11] introduced a pre-processing technique, called *prefix-free parsing*, to compute the BWT for a (single) highly repetitive input string S of length n in $\mathcal{O}(n)$ time. The rationale behind this technique is to apply a simple compression scheme as to exploit the string repetitiveness, and then to compute the BWT from the encoded string. The pre-processing step generates a dictionary and a parse of the input string, and it is effective if both the dictionary and the parse together are much smaller than the original input string. In this case, indeed, the BWT computation executed in internal memory is more resource-efficient. The Big-BWT tool¹³ implements this technique.

Experiments showed that the size of the parse is typically the most demanding component for very large and repetitive inputs. Thus, in order to reduce the memory requirements, *recursive prefix-free parsing* has been recently introduced in [58], where the prefix-free parse is applied to the parse generated by prefix-free parsing the input string.

In the following we outline the key phases of the BWT computation via prefix-free parsing and highlight the properties of this technique.

Parsing phase. The main idea of the parsing phase is to divide the input string S into overlapping phrases of variable length that will constitute the dictionary. More precisely, the string considered is $\#S\w , where S is the input string of length n over Σ , $\#$ and $\$$ are end-markers, and $w \geq 1$ is a fixed parameter.

In order to parse $\#S\w , let $E \subseteq \Sigma^w$ be a set of strings of fixed length w over Σ (called *trigger strings*) augmented with $\#$ and $\w . The *dictionary* \mathcal{D} comprises all the strings d such that (i) d is a substring of $\#S\w , (ii) exactly one prefix of d is in E , (iii) exactly one suffix of d is in E , (iv) no other substring of d is in E . The sequence of dictionary phrase occurrences that form $\#S\w is the *parse* \mathcal{P} , in which each phrase occurrence is encoded by a meta-character given by the lexicographic rank of that phrase in \mathcal{D} . Then, $\text{PFP}(S) = (\mathcal{D}, \mathcal{P})$, where \mathcal{D} is lexicographically sorted and \mathcal{P} is composed of characters that correspond to positions in \mathcal{D} . For instance, let S be the string in Example 1, $w = 2$, and trigger strings given by AT , $\#$ and $\2 . Then, $\mathcal{D} = \{\#CAT, ATA\$, ATGAT\}$. The parse of $\#S\2 is $\#CAT ATGAT ATGAT ATA\$,$ which results in $\mathcal{P} = [0, 2, 2, 1]$, when using the lexicographic rank to identify phrases in \mathcal{D} .

In Big-BWT [11], \mathcal{D} is iteratively built through a one-pass over $\#S\w using a Karp-Rabin hash function and a parameter p . The trigger strings are implicitly found by passing a sliding window of length w : wherever the KR fingerprint of the content of the current window is 0

¹³<https://gitlab.com/manzai/Big-BWT>

modulo p , a trigger string is found. Then, the current phrase terminates at the end of the window, and it is added to the dictionary (recording also the phrase frequency), and then, the next phrase starts at the beginning of the current window. Thus, the input string is decomposed into overlapping variable-length phrases, each starting and ending with a trigger string of length w . After sorting the dictionary \mathcal{D} , the final parse \mathcal{P} is generated.

The main property of the prefix-free parsing procedure is that none of the suffixes of length greater than w of the phrases in \mathcal{D} is a prefix of any other [11, Lemma 1]. This is a property fundamental for the next phase when building the BWT of S .

Building the BWT of the input string. To construct the BWT of S , we follow the definition that involves appending the end-marker $\$$ to S and then lexicographically sorting its suffixes.

Any suffix of $S\$$ corresponds to a unique suffix of $\#S\w of length greater than w , i.e., the suffix $y\$$ of $S\$$ is mapped to the suffix $y\w of $\#S\w , if y is not empty, otherwise $\$$ is mapped to $\#S\w . Then, the permutation that lexicographically sorts the suffixes of $\#S\w of length greater than w , also lexicographically sorts the corresponding suffixes of $S\$$ [11, Lemma 5].

Moreover, any suffix x of $\#S\w has exactly one prefix s that is a *phrase suffix*, i.e., s is a suffix of a phrase in the dictionary \mathcal{D} , [11, Lemma 3]. This implies that the order among phrase suffixes can be carried over to the suffixes of $\#S\w . Indeed, given two suffixes x and x' of $\#S\w (of length greater than w) and their unique corresponding prefixes s and s' that are phrase suffixes, $s \prec s'$ implies $x \prec x'$. However, if $s = s'$, some information about \mathcal{D} and \mathcal{P} must be taken into account.

More precisely, one can think of constructing the BWT string in two-passes according to the list of lexicographically sorted suffixes (of length greater than w) of phrases in \mathcal{D} , where each suffix is considered a number of times equal to its phrase frequency. In a first pass, one builds the sequence of characters preceding any suffix s of \mathcal{D} that is either a proper suffix of only one phrase dictionary $d \in \mathcal{D}$ or is an element of \mathcal{D} that occurs once in \mathcal{P} . Both cases have no ambiguity, since the preceding character is uniquely determined. In a second pass, one deals with the sequence of characters preceding suffixes that are either elements of \mathcal{D} occurring more than once in \mathcal{P} or proper suffixes of different dictionary phrases. In this second instance, one needs the list of lexicographically sorted suffixes of \mathcal{P} to put characters in sequence according to their correct order [11, Lemma 7]. Considering the string S of Example 1 and the PFP(S) defined above, the list of lexicographically sorted suffixes of \mathcal{D} of length greater than $w = 2$ is $\{\#CAT(1), A\$(1), ATA\$(1), ATGAT(2), CAT(1), GAT(2), TA\$(1), TGAT(2)\}$, where for each suffix, its phrase frequency is reported within brackets. Then, in the first pass, the sequence of characters of $\text{bwt}(S)$ built is $ATG - -\$TTAAA$, where dashes left at positions 4 and 5 correspond to $\{C, G\}$, which are the characters in S preceding the two occurrences of the phrase dictionary $ATGAT$. Finally, the $\text{bwt}(S) = ATGGC\$TTAAA$ is completed considering the lexicographic order of the suffixes of $\mathcal{P} = [0, 2, 2, 1]$ starting with 2, which is the meta-character corresponding to $ATGAT$.

In Big-BWT, the BWT of \mathcal{P} is performed in linear time using the suffix array construction algorithm SACA-K [54]. However, instead of the string $\text{BWT}(\mathcal{P})$, it stores an *inverted list* that associates to each dictionary phrase d the list of positions in the $\text{BWT}(\mathcal{P})$ where d occurs. The inverted list is a format more suitable for the next step, which consists in lexicographically sorting the suffixes in \mathcal{D} , and then by scanning them, placing the corresponding characters in the BWT string. The sorting of the phrase suffixes is accomplished by applying the gSACA-K [48] algorithm.

Theoretical costs. The BWT computation for an input string S of length n uses $\mathcal{O}(|\mathcal{D}| + |\mathcal{P}|)$ space, where $\text{PFP}(S) = (\mathcal{D}, \mathcal{P})$, $|\mathcal{D}|$ is the sum of the length of the dictionary phrases, and $|\mathcal{P}|$ the number of elements in \mathcal{P} , and it takes $\mathcal{O}(n)$ time when working in internal memory.

4.2 grlBWT [Díaz-Domínguez and Navarro, 2023]

Díaz-Domínguez and Navarro presented in [22] an external memory approach that computes the BWT of a collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ of k strings and N symbols according to Equation (1). Their method, called **grlBWT**¹⁴, builds on SAIS (Section 2) but maintains data in compressed form to reduce RAM usage and redundant calculations. We refer the reader to Section 2 for the notation used in this section.

Each recursive level i of **grlBWT** receives a collection \mathcal{S}^i and returns the BWT L^i of \mathcal{S}^i as defined in Equation (1). To do so, it computes the set \mathcal{F} of strings labelling the LMS substrings of \mathcal{S}^i and uses this set to fill in a preliminary BWT $pr(L^i)$. The output is preliminary because it has some incomplete areas that could not be inferred from \mathcal{F} . The algorithm then produces a grammar \mathcal{G}^i that generates strings in \mathcal{F} to finish $pr(L^i)$ later. Subsequently, **grlBWT** replaces the LMS substrings of \mathcal{S}^i by their corresponding nonterminals in \mathcal{G}^i to create another collection \mathcal{S}^{i+1} that passes as input to the recursive level $i+1$. The base case of the recursion is when all strings in \mathcal{S}^i have length one, in which case the BWT is the input itself. When **grlBWT** returns to level i , it combines the information from \mathcal{G}^i and L^{i+1} to fill the incomplete areas of $pr(L^i)$ and thus produce L^i .

In practice, **grlBWT** is an iterative method with two phases, the *parsing phase* and the *induction phase*. The former simulates the descend of the recursive levels and the second simulates the ascend.

Parsing phase. During each parsing round i , **grlBWT** *parses* \mathcal{S}^i to compute \mathcal{F} and then applies ISS over \mathcal{F} to *simulate* the partially sorted gsa^i of \mathcal{S}^i that SAIS obtains in an ISS pass. Recall from Section 2 that the ISS pass partitions gsa^i such that each block $\text{gsa}^i[s_X..e_X]$ contains the suffixes $S_u[j..n_u]$ of \mathcal{S}^i prefixed by the same $X = S_u[j..nextLMS(j)]$. The algorithm builds $pr(L^i)$ using this partition and the fact that each equal-symbol run $L^i[s_X..e_X] = (c, e_X - s_X + 1)$ does not need further processing of $\text{gsa}^i[s_X..e_X]$.

The parsing round i starts by calculating \mathcal{F} along with an array $W[1..|\mathcal{F}|]$ that stores in $W[u]$ the number of times $F_u \in \mathcal{F}$ appears as an LMS substring in \mathcal{S}^i . Subsequently, **grlBWT** runs ISS over \mathcal{F} to obtain an array A^i where the suffixes of \mathcal{F} are arranged in \prec_{LMS} order (Equation (3)), breaking ties for equal suffixes arbitrarily. Let $A^i[q] = (j, u)$ be the q -th smallest suffix $F_u[j..n_u]$ of \mathcal{F} in \prec_{LMS} order. The algorithm scans A^i from left to right to visit every block $A^i[y_X..z_X]$ of suffixes in \mathcal{F} with length > 1 labelled $X = F_u[j..n_u]$ and uses the information in W to compute the pair (s_X, e_X) for the block $\text{gsa}^i[s_X..e_X]$ storing the suffixes in \mathcal{S}^i labelled $X = S_{u'}[j'..nextLMS(j')] = F_u[j..n_u]$, with $S_{u'} \in \mathcal{S}^i$. The algorithm skips each suffix $X = F_u[j..n_u]$ of length one because it overlaps in \mathcal{S}^i the first symbol of some $F_{u'} \neq F_u \in \mathcal{F}$. When **grlBWT** reaches $A^i[y_X..z_X]$, it visits the symbol $F_u[j-1]$ that precedes $X = F_u[j..n_u]$ in each $A^i[q] = (j, u)$, with $q \in [y_X..z_X]$. If X is always a proper suffix in \mathcal{F} and is preceded by the same symbol c , **grlBWT** appends a run $(c, \ell = e_X - s_X + 1)$ to $pr(L^i)$. If X is a proper suffix, but is preceded by different symbols, then it appends an incomplete block $(\#, \ell)$ instead, where $\# \notin \Sigma^i$ is a special symbol. The third option is that X matches a full string $F_u[1..n_u] \in \mathcal{F}$. In that case, **grlBWT** appends $(*, \ell)$ to $pr(L^i)$, where $* \notin \Sigma^i$ is another special symbol that indicates another type of incomplete block.

¹⁴ An implementation is available at <https://github.com/ddiazdom/grlBWT>

The next step is to build a grammar $\mathcal{G}^i = (\Sigma^i, V, \mathcal{R}, I)$ (Section 2) that generates the set \mathcal{P} of different strings X producing incomplete blocks in $pr(L^i)$. Thus, for a phrase $X[1..x] \in \mathcal{P}$, grlBWT creates the rule $b \rightarrow X[1..x]$, where b is the \prec_{LMS} order of X in \mathcal{P} . In addition, it replaces $X[1..x]$ on the right-hand side with $b \rightarrow X[j_1 - 1]b_1$, where b_1 is the \prec_{LMS} order of the longest suffix $X[j_1..x]$ which is also a phrase in \mathcal{P} . Recursively, $b_1 \rightarrow X[j_2 - 1]b_2$ stores the second longest suffix $X[j_2..x]$ that also occurs in \mathcal{P} . This structure repeats o times to cover the o proper suffixes of X in \mathcal{P} , and the last two symbols $X[x - 1..x]$ form the rule $b_{o+1} \rightarrow X[x - 1]\epsilon$, where $\epsilon \notin \Sigma^i$ denotes the empty symbol. Let $\text{expseq}(b)$ be the sequence of pairs $(c_1, b_1), (c_2, b_2), \dots, (c_o, b_o), (c_{o+1}, \epsilon)$ in the recursive rewrites $b \Rightarrow c_1 b_1 \Rightarrow c_1 \cdot c_2 b_2 \Rightarrow \dots \Rightarrow c_1 c_2 c_{o+1}$ of $\text{exp}(b)$. Notice that every (c_u, b_u) encodes the nonterminal $b_u \in V$ expanding to the u -th suffix $X[j_u..x] \in \mathcal{P}$ of X (from left to right) and the symbol $c_u = X[j_u - 1] \in \Sigma^i$ that precedes it. Additionally, (c_{o+1}, ϵ) stores the rightmost symbol $c_{o+1} = X[x - 1]$ of X that does not overlap the following LMS substring in \mathcal{S}^i .

To depict the steps of the first parsing round, let us consider again the collection $\mathcal{S} = \{AGC\underline{GT}\$1, TCA\underline{AAC}\$2, CGC\underline{AA}\$3\}$ of Example 2. The underlined symbols are LMS-type positions. After computing the LMS-substrings from $\mathcal{S}^1 = \mathcal{S}$, the method obtains the set of phrases $\mathcal{F} = \{AGC, CGT\$1, TCA, AAC\$2, CGCAA\$3\}$ and the corresponding array $W[1..5] = 1, 1, 1, 1, 1$ with their frequencies in \mathcal{S}^1 . Then, it uses ISS to compute A^1 and $pr(L^1)$ from \mathcal{F} to produce:

$$\begin{array}{rcccccccccccccccccccc}
 & & \$1 & \$2 & \$3 & A\$3 & AA\$3 & AAC\$2 & AC\$2 & AGC & C\$2 & CAA\$3 & CA & CGCAA\$3 & CGT\$1 & GC AA\$3 & GC & GT\$1 & T\$1 & TCA \\
 A^1 & = & 4 & 4 & 6 & 5 & 4 & 1 & 2 & 1 & 3 & 3 & 3 & 4 & 4 & 2 & 2 & 2 & 3 & 1 \\
 & & 2 & 4 & 5 & 5 & 5 & 4 & 4 & 1 & 4 & 5 & 3 & 4 & 1 & 5 & 1 & 2 & 2 & 3
 \end{array}$$

$$pr(L^1) = \# \# \# A C * A * A G T * * C A C G *$$

Recall that $A^1[q] = (j, u)$ refers to the j -th suffix of the phrase u in \mathcal{F} (from left to right). The partition of A^1 according to the suffixes of \mathcal{F} becomes $A^1[1..3], A^1[4], A^1[5], \dots, A^1[18]$. After block $A^1[1..3]$, all the blocks have length one because their corresponding suffixes (in grey above) are unique in \mathcal{F} . Also note that the method regards $\$1 = \$2 = \$3 = \$$ as the same symbol. In $pr(L^1)$, the first block $pr(L^1)[1..3] = (\#, 3)$ is incomplete because the method cannot infer the relative order of the occurrences of $\$$ in \mathcal{S}^1 from A^1 and \mathcal{F} . Moreover, the blocks $pr(L^1)[6]$, $pr(L^1)[8]$, $pr(L^1)[12]$, $pr(L^1)[13]$, and $pr(L^1)[18]$ are also incomplete (with symbol $*$) because their corresponding suffixes match full phrases of \mathcal{F} . The next step is to construct the grammar \mathcal{G}^1 from the strings that produce incomplete blocks in $pr(L^i)$ (i.e., \mathcal{P}). The original rules are $\mathcal{R} = \{1 \rightarrow \$, 2 \rightarrow AAC\$2, 3 \rightarrow AGC, 4 \rightarrow CGCAA\$3, 5 \rightarrow CGT\$1, 6 \rightarrow TCA\}$. However, the method only keeps in the right-hand sides of \mathcal{R} the proper suffixes associated with incomplete blocks, so the rules become $\mathcal{R} = \{1 \rightarrow \$\epsilon, 2 \rightarrow C1, 3 \rightarrow G\epsilon, 4 \rightarrow A1, 5 \rightarrow T1, 6 \rightarrow C\epsilon\}$, with ϵ being the null symbol. Finally, the parsing round stores $pr(L)^1$ and \mathcal{G}^1 , computes the new collection $\mathcal{S}^2 = \{35, 62, 4\}$ and continues to the next parsing round.

Induction phase. Every iteration i induces the order of symbols in the incomplete blocks of $pr(L^i)$ using \mathcal{G}^i and L^{i+1} . The block $pr(L^i)[s_X..e_X] = (*, \ell)$ is associated with a string $X \in \mathcal{P}$ that only labels LMS substrings of \mathcal{S}^i , and if $pr(L^i)[s_X..e_X]$ is the b -th incomplete run of

$pr(L^i)$, then $b \in \Sigma^{i+1}$ is the nonterminal assigned to X in \mathcal{G}^i . Consider the partition of L^{i+1} according to their buckets in the fully sorted version of \mathbf{gsa}^i . Each block $L^{i+1}[s_b..e_b]$ stores the symbols that precede the suffixes in \mathcal{S}^{i+1} prefixed by b (i.e., those in $\mathbf{gsa}^{i+1}[s_b..e_b]$). There is a map from $\mathbf{gsa}^{i+1}[s_b..e_b]$ to the range $\mathbf{gsa}^i[s_X..e_X]$ with the suffixes $S_u[j..n_u]$ of \mathcal{S}^i prefixed by $S_u[j..nextLMS(j)] = X$. Consequently, filling in $pr(L^i)[s_X..e_X]$ reduces to scanning $L^{i+1}[s_b..e_b]$ from left to right, and for each $L^{i+1}[q]$ in $L^{i+1}[s_b..e_b]$, run $expseq(L^{i+1}[q]) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ and append c_{o+1} to $pr(L^i)[s_X..e_X]$.

On the other hand, each $pr(L^i)[s_Y..e_Y] = (\#, \ell)$ is associated with a string $Y \in \mathcal{P}$ that occurs as a proper suffix in different LMS substrings of \mathcal{S}^i , say $X[1..x]$ and $X'[1..x']$, and is preceded by different symbols. That is, $Y = X[j..x] = X'[j'..x']$ and $X[j-1] \neq X'[j'-1] \in \Sigma^i$. If Y has \prec_{LMS} order b_u in \mathcal{P} , then $pr(L^i)[s_Y..e_Y]$ is the b_u -th incomplete run of $pr(L^i)$ and ℓ is the number of times Y occurs as a suffix in the LMS substring of \mathcal{S}^i .

The encoding of \mathcal{G}^i is convenient to fill $pr(L^i)[s_Y..e_Y] = (\#, \ell)$ because $expseq(L^{i+1}[q]) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ tells that $c_1 \in \Sigma^i$ goes to the incomplete run number b_1 , $c_2 \in \Sigma^i$ goes to the incomplete run number b_2 , and so on. Thus, a symbol c_u associated with b_u in $expseq(L^{i+1}[q])$ goes to $L^{i+1}[s_Y..e_Y]$ after the symbols associated with b_u in $expseq(L^{i+1}[1]), expseq(L^{i+1}[2]), \dots, expseq(L^{i+1}[q-1])$. In other words, it is possible to fill all incomplete blocks $(\#, \ell)$ of $pr(L^i)$ in one linear decompression of L^{i+1} . Another important observation is that a run $L^{i+1}[q..q'] = (b, \ell')$ produces ℓ' copies of $expseq(b)$. Thus, `grlBWT` calls $expseq(b)$ once and copies the content ℓ' times in each incomplete block in $expseq(b)$.

The induction round i works as follows: Create an array P with m buckets, where m is the number of incomplete blocks of $pr(L^i)$ labelled $\#$. Then, scan L^{i+1} from left to right and, for each run (b, ℓ') in L^{i+1} , execute $expseq(b) = (c_1, b_1), (c_2, b_2), \dots, (c_{o+1}, \epsilon)$ and append ℓ' copies of every symbol c_u in the bucket b_u of P . In addition, replace (b, ℓ') by (c_{o+1}, ℓ') in L^{i+1} . Next, perform a sorted merge of P , L^{i+1} , and $pr(L^i)$ (observe that the three arrays over the alphabet $\Sigma^i \cup \{*, \#\}$), replacing each $(\#, \ell)$ of $pr(L^i)$ with the next ℓ symbols of P , and each $(*, \ell)$ with the next ℓ symbols of L^{i+1} .

Let us complete the preliminary BWT $pr(L^1)$ of the parsing round 1 from our current example using the multi-string BWT $L^2 = 65234$ and the grammar \mathcal{G}^1 . The rules of this grammar are $\mathcal{R} = \{1 \rightarrow \$\epsilon, 2 \rightarrow C1, 3 \rightarrow G\epsilon, 4 \rightarrow A1, 5 \rightarrow T1, 6 \rightarrow C\epsilon\}$. Observe that the alphabet of L^2 is a subset of the symbols on the left-hand side of \mathcal{R} . The method first creates the array P with one bucket as $pr(L^i)$ contains one block labelled $(\#, 3)$. The length of P is three because the cumulative length of the blocks in $pr(L^1)$ labelled $\#$ is three. The method scans L^2 from left to right to compute $expseq$ on the symbol of each run, and uses the output to fill the buckets of P . The combined information of $expseq(5) = T1 \Rightarrow T\$ \epsilon$, $expseq(2) = C1 \Rightarrow C\1 and $expseq(4) = A1 \Rightarrow A\$ \epsilon$ yields $P = TCA$. In turn, this information updates $pr(L^1)$ to $pr(L^1) = \underline{TCA}AC * A * AGT * * CACG*$, where the underlined symbols represent the replacement of $(\#, 3)$ by P . Note that $expseq(6)$ and $expseq(3)$ do not produce symbols for P because their suffixes do not produce incomplete blocks $(\#, \ell)$. The method also updates each $L^2[j]$ with the rightmost symbol in $expseq(L^2[j])$ that belongs to $\Sigma = \{ \$, A, C, G, T \}$, so L^2 becomes $C\$ \$ G \$$. Finally, the method merges L^2 and $pr(L^1)$ to produce the final multi-string BWT $L^1 = \underline{TCA}AC \underline{CA} \$ \underline{AGT} \$ \underline{GCACG} \$$ of \mathcal{S} in Example 2. In this case, the underlined symbols originally belonged to L^2 .

Theoretical costs. Under the RAM model, `grlBWT` runs in $\mathcal{O}(N + k \log n_{max})$ time and uses $\mathcal{O}((N + k \log n_{max}) \log N)$ bits of working memory, where n_{max} is the longest string in \mathcal{S}^i . However, this cost is reached only with adversarial inputs, with practical scenarios (e.g., $n_{max} = \mathcal{O}(N/k)$) running in $\mathcal{O}(N)$ time. On the other hand, `grlBWT` is a semi-external

approach that keeps \mathcal{S}^i , $pr(L^i)$, and L^i on disk and accesses them linearly, thus requiring $\mathcal{O}((N \log N)/B)$ I/Os. On the other hand, the RAM usage depends on the size of \mathcal{F} . This set is small when \mathcal{S} is repetitive, but there is no known theoretical bound for its size.

5 Future directions

There are several promising directions for future research. First, the efficient computation of auxiliary components in BWT-based compressed self indexes. Examples of such components are the $2r$ suffix array samples of the r -index [28], the samples of the subsampled r -index [17], or the LCP array samples of thresholds [2]. An important challenge in these examples is how to compute the data structures without building full versions of `sa` and `lcp`.

From a systems perspective, the design of disk-based algorithms optimized for Solid State Drives (SSDs) could lead to significant speedups, taking advantage of SSD-specific features such as high random access speeds and internal parallelism. Moreover, the acceleration of BWT construction via GPU architectures could offer significant performance improvements for massive datasets (see, for instance, the BWT-based compressor `libbssc`¹⁵ that uses NVIDIA GPU acceleration).

On the algorithmic side, the Prefix-Free Parsing (PFP) strategy should be further explored and adapted for the case of string collections. To our knowledge, the only PFP variant that works on collections is PFP-eBWT [10] that computes the extended BWT (EBWT) by Mantaci *et al.* [51]. On the other hand, designing new lightweight schemes for text compression that are compatible with the construction of the BWT could enhance the scalability for massive datasets. This idea is particularly relevant for `grlBWT`, where the compression of the input is the main bottleneck. Recently, Díaz-Domínguez *et al.* [21] proposed a parallel grammar algorithm that handles terabytes of data efficiently, and whose scheme shares some similarities with `grlBWT`. Adapting this technique could achieve considerable reductions in space and time.

Finally, there is growing interest in designing indexing¹⁶ data structures to encode BWTs with large alphabets (see [29, 4, 64] and references therein). The standard solution to this problem is the wavelet tree [30], which represents the BWT as a binary tree of height $\mathcal{O}(\log \sigma)$ and with σ leaves, σ being the BWT's alphabet size. However, when σ is large, the $\mathcal{O}(\log \sigma)$ factor that results from navigating the tree becomes a considerable overhead. Furthermore, the $\mathcal{O}(\sigma w)$ bits required to store the tree also make the wavelet tree impractical in this scenario. In this regard, Nishimoto *et al.* recently proposed a data structure [53] that encodes the BWT in $\mathcal{O}(r \log n)$ bits and reduces the $\mathcal{O}(\log \sigma)$ penalty to $\mathcal{O}(1)$ regardless of the alphabet. However, this solution is space-efficient only if $r \ll n$, which occurs when the text encoded by the BWT is highly repetitive. In practical scenarios where repetitive collections also have some noise, this solution grows rapidly in size [7].

6 Final Remarks

In this short survey, we highlight key algorithms for computing large BWTs by leveraging external memory or text compression techniques with no additional information about the data. It appears as a first step towards a more in-depth comparison of the corresponding tools, potentially including experimental results to highlight the differences among them.

¹⁵ <https://github.com/IlyaGrebnev/libbssc>

¹⁶ A representation encoding the BWT and that supports LF-mapping and backward search queries

■ **Table 1** Theoretical complexities of algorithms. For Big-BWT, we assume that the data structures fit into internal memory. We consider an (single) input string of length n for Big-BWT and bwt-disk, and a collection \mathcal{S} of k strings with total length N for the other algorithms. Let σ be the alphabet size, m the length of the longest string in \mathcal{S} , maxlcp the maximum LCP value, M the number of words available in RAM, and B the number of consecutive words on the disk.

Algorithm	Time Complexity	Memory Usage	I/O Complexity
Big-BWT [11]	$\mathcal{O}(n)$	$\mathcal{O}(\mathcal{D} + \mathcal{P})$	-
bwt-disk [26]	$\mathcal{O}(n^2/M)$	M words	$\mathcal{O}(n^2/(MB \log n))$
BCR [5]	$\mathcal{O}(m(N + \text{sort}(k)))$	$\mathcal{O}(k + \sigma^2)$ words	$\mathcal{O}(Nm \log \sigma)$
BCRext [5]	$\mathcal{O}(kN)$	$\mathcal{O}(\sigma^2)$ words	$\mathcal{O}(Nm \log \sigma + N \log N)$
eGSA [49]	$\mathcal{O}((N \log N/M) \cdot \text{maxlcp})$	M words	$\mathcal{O}((N \log N/M) \cdot m)$
eGap [23]	$\mathcal{O}(N \cdot \text{maxlcp})$	M words	$\mathcal{O}(N \cdot \text{maxlcp})$
BWT+LCP [8]	$\mathcal{O}(N \cdot \text{maxlcp})$	$\mathcal{O}(k + m + \log \text{maxlcp})$	$\mathcal{O}(N \cdot \text{maxlcp})$
griBWT [22]	$\mathcal{O}(N + k \log m)$	$\mathcal{O}((N + k \log m) \log N)$ bits	$\mathcal{O}((N \log N)/B)$

However, a direct comparison may not be entirely fair, given the differences in the intended use cases of the tools. Some are designed for a single string, others for string collections, and some work in external memory (semi- or fully), while others in internal memory. Additionally, some tools are prototypes, whereas others are engineered solutions. Table 1 summarizes the complexities of the algorithms at the basis of these tools.

In particular, bwt-disk was the first tool for constructing the BWT for a single string in external memory, without the need to first compute the suffix array. It optimizes disk access by using only sequential scans, which enables it to fully leverage modern caching systems. Additionally, bwt-disk stores all input, output, and intermediate files in compressed form, which allows it to use less total disk space than the uncompressed input for real-world datasets.

Specifically, BCR/BCRext was developed to build the BWT for huge collections of short strings over any alphabet. The experimental results in [5] showed that BCR is capable of computing the BWT for 1 billion strings, each 100 characters long (approximately 93 GB of data), using only 13 GB of RAM, while BCRext requires a negligible amount of RAM. Moreover, BCR can output auxiliary data structures (such as the Document Array, Generalized Suffix Array and Longest Common Prefix).

eGSA was the first external memory algorithm for constructing the generalized suffix arrays. It can also output the BWT for collections of strings with different sizes. The experimental results in [49] showed that eGSA tool can index up to 24 GB of data using only 2GB of RAM. One disadvantage of the eGSA is the large amount of disk working space. Also, it is observed in [49] that eGSA's running time degrades when the RAM is restricted to the input size.

eGap is very flexible about the use of RAM. The first phase adapts to the available memory, but also the second one has been implemented in an alternative semi-external version. Moreover, the algorithm can output additional information (such as the Document Array) to solve in a single sequential scan of the inputs three well known problems on collections of sequences: maximal repeats, all pairs suffix-prefix overlaps and succinct de Bruijn graphs.

BWT+LCP splits the problem of computing BWTs and LCPs into subproblems in a different way from other proposals, as we explained in Section 3.5, and it shows good performance in experiments.

In the context of tools that employ compression techniques, Big-BWT has been demonstrated to reduce the workspace when computing the BWT of highly repetitive inputs. The experiments in [11] showed that the prefix-free parsing procedure implemented in Big-BWT

allows to produce a dictionary \mathcal{D} and a parse \mathcal{P} that are considerably smaller than the input and can fit in internal memory, even in cases where the input is of considerable size. For example, on a dataset of 10,000 Salmonella genomes, the dictionary and the parse together took only about 7 GB, compared to the 50 GB of uncompressed input.

The current implementation of `grlBWT` can handle high volumes of repetitive data efficiently. In experiments in [22], the tool could process 400 human genomes (1.2 TB) in 41 hours, with a memory peak of 183 GB. The performance decreased in non-repetitive data (i.e., collections of DNA sequencing reads), although it remained one of the most efficient methods. In all inputs, the main bottleneck was text compression during the parsing phase, which took more than 90% of the running time. Currently, `grlBWT` only outputs the multi-string BWT. The construction of other data structures such as `lcp` or `sa` is not yet supported.

A comparison with almost all disk-based methods is available in [23]. More comparisons can be found in the original papers, such as in [22].

Nevertheless, there are a number of other tools in the literature, some of which are optimized to work in internal memory and/or on specific alphabet (such as, for instance, the DNA alphabet). For example, the recent `ropebwt3` [45] constructs the BWT of large DNA sequence sets and enables efficient searching via the FM-index. It builds on the result reported in [26], which has seen multiple implementations [63, 59]. It uses `libsais`¹⁷ to construct a partial multi-string BWT from a sequence subset, and then merges it into an existing BWT. The tool `ropebwt3`¹⁸ is particularly optimized for highly redundant data, such as pangenomes or high-coverage sequencing reads. It encodes the BWT using run-length encoding and a dynamic B^+ -tree (see also [44]). Another recent proposal [52] (called CMS-BWT¹⁹) introduces a method that takes as input a string collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ and a reference string R , and computes the BWT of the (single) concatenated string $S_1\$S_2\$ \dots S_k\$$ by exploiting the high repetitiveness of data through the compressed matching statistics introduced in [47].

References

- 1 Donald Adjero, Timothy Bell, and Amar Mukherjee. *The Burrows–Wheeler transform: data compression, suffix arrays, and pattern matching*. Springer, Boston, MA, 2008.
- 2 Hideo Bannai, Travis Gagie, et al. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020. doi:10.1016/J.TCS.2019.08.005.
- 3 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piątkowski. Constructing the Bijective and the Extended Burrows–Wheeler Transform in Linear Time. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:16, 2021. doi:10.4230/LIPIcs.CPM.2021.7.
- 4 Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st International Symposium on Algorithms and Computation (ISAAC)*, pages 315–326, 2010. doi:10.1007/978-3-642-17514-5_27.
- 5 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.
- 6 Gianmarco Bertola, Anthony J. Cox, Veronica Guerrini, and Giovanna Rosone. A Class of Heuristics for Reducing the Number of BWT-Runs in the String Ordering Problem. In *Proc. 35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:15, 2024. doi:10.4230/LIPIcs.CPM.2024.7.

¹⁷Source code: <https://github.com/IlyaGrebnev/libsais>

¹⁸<https://github.com/lh3/ropebwt3>

¹⁹<https://github.com/fmasillo/CMS-BWT>

- 7 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: optimizing the r-index. In *Proc. 22nd International Symposium on Experimental Algorithms (SEA)*, pages 1:1–1:19, 2024. doi:10.4230/LIPICS.SEA.2024.1.
- 8 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theoretical Computer Science*, 862:42–58, 2021. doi:10.1016/j.tcs.2020.11.041.
- 9 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 129–142, 2021. doi:10.1007/978-3-030-86692-1_11.
- 10 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proceedings of 28th International Symposium in String Processing and Information Retrieval SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142, 2021. doi:10.1007/978-3-030-86692-1_11.
- 11 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.
- 12 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 13 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Proc. 33rd Data Compression Conference (DCC)*, pages 71–80, 2023. doi:10.1109/DCC55655.2023.00015.
- 14 Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinformatics*, 40(7):btac333, 2024. doi:10.1093/bioinformatics/btac333.
- 15 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. submitted to Festschrift’s honoree Giovanni Manzini.
- 16 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.
- 17 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r-index. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, page article 13, 2021.
- 18 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2016. doi:10.1093/bib/bbw089.
- 19 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012. doi:10.1093/bioinformatics/bts173.
- 20 Anthony J. Cox, Fabio Garofalo, Giovanna Rosone, and Marinella Sciortino. Lightweight LCP construction for very large collections of strings. *Journal of Discrete Algorithms*, 37:17–33, 2016. doi:10.1016/J.JDA.2016.03.003.
- 21 Diego Diaz-Dominguez. Efficient terabyte-scale text compression via stable local consistency and parallel grammar processing. *arXiv preprint arXiv:2411.12439*, 2024. to appear in Proc. 23rd Symposium on Experimental Algorithms (SEA 2025).
- 22 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Information and Computation*, 294:105088, 2023. doi:10.1016/j.ic.2023.105088.
- 23 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019. doi:10.1186/S13015-019-0140-0.
- 24 Lavinia Egidi and Giovanni Manzini. Lightweight merging of compressed indices based on BWT variants. *Theoretical Computer Science*, 812:214–229, 2020. doi:10.1016/j.tcs.2019.11.001.

- 25 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
- 26 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012. doi:10.1007/S00453-011-9535-0.
- 27 Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001. doi:10.1016/S0020-0255(01)00098-6.
- 28 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020. doi:10.1145/3375890.
- 29 Alexander Golynski. Rank/select operations on large alphabets: a tool for text. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 122, page 368, 2006.
- 30 Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. High-order entropy-compressed text indexes. In *Proc. 14th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, volume 3, pages 841–850, 2003.
- 31 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC)*, pages 97–120, 2023. doi:10.1007/978-3-031-38854-5_6.
- 32 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014. doi:10.1093/bioinformatics/btu584.
- 33 Hongwei Huo, Pengfei Liu, Chenhui Wang, Hongbo Jiang, and Jeffrey Scott Vitter. CIndex: compressed indexes for fast retrieval of FASTQ files. *Bioinformatics*, 38(2):335–343, 2021. doi:10.1093/bioinformatics/btab655.
- 34 Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.
- 35 Lilian Janin, Giovanna Rosone, and Anthony J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, 30(1):24–30, 2014. doi:10.1093/bioinformatics/btt257.
- 36 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009. doi:10.1007/978-3-642-02441-2_17.
- 37 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 38 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.
- 39 John C. Kieffer and En Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000. doi:10.1109/18.841160.
- 40 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/j.jda.2004.08.002.
- 41 Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008. doi:10.1093/bioinformatics/btn032.
- 42 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10:1–10, 2009. doi:10.1186/gb-2009-10-3-r25.
- 43 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012. doi:10.1093/bioinformatics/bts280.
- 44 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. doi:10.1093/bioinformatics/btu541.
- 45 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, 2024. doi:10.1093/bioinformatics/btae717.

- 46 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.
- 47 Zsuzsanna Lipták, Francesco Masillo, and Simon J Puglisi. Suffix sorting via matching statistics. *Algorithms for Molecular Biology*, 19(1):11, 2024. doi:10.1186/s13015-023-00245-z.
- 48 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017. doi:10.1016/J.TCS.2017.03.039.
- 49 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms for Molecular Biology*, 12(1):26:1–26:16, 2017. doi:10.1186/S13015-017-0117-9.
- 50 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computation.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 51 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.
- 52 Francesco Masillo. Matching Statistics Speed up BWT Construction. In *Proc. 31st Annual European Symposium on Algorithms (ESA 2023)*, pages 83:1–83:15, 2023. doi:10.4230/LIPIcs.ESA.2023.83.
- 53 Takaaki Nishimoto and Yasuo Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In *Proc. 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 198, pages 101:1–101:15, 2021. doi:10.4230/LIPICS.ICALP.2021.101.
- 54 Ge Nong. Practical linear-time $o(1)$ -workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):15:1–15:15, 2013. doi:10.1145/2493175.2493180.
- 55 Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. Induced sorting suffixes in external memory. *ACM Transactions on Information Systems (TOIS)*, 33(3):1–15, 2015. doi:10.1145/2699665.
- 56 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.
- 57 Daisuke Okanohara and Kunihiko Sadakane. A linear-time Burrows–Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009. doi:10.1007/978-3-642-03784-9_9.
- 58 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big bwts. In *2023 Data Compression Conference (DCC)*, pages 62–70, 2023. doi:10.1109/DCC55655.2023.00014.
- 59 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *Proc. 31st Data Compression Conference (DCC)*, pages 203–212, 2021. doi:10.1109/DCC50243.2021.00028.
- 60 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Variable-order reference-free variant discovery with the Burrows–Wheeler Transform. *BMC Bioinformatics*, 21-S(8):260, 2020. doi:10.1186/S12859-020-03586-3.
- 61 Yoshihiro Shibuya and Matteo Comin. Better quality score compression through sequence-based quality smoothing. *BMC Bioinformatics*, 20-S(9):302:1–302:11, 2019. doi:10.1186/s12859-019-2883-5.
- 62 Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–56, 2012. doi:10.1101/gr.126953.111.
- 63 Jouni Sirén. Burrows–Wheeler Transform for Terabases. In *Proc. 16th Data Compression Conference (DCC)*, pages 211–220, 2016. doi:10.1109/DCC.2016.17.
- 64 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. doi:10.1093/BIOINFORMATICS/BTZ575.

- 65 Clare Turnbull, Richard H. Scott, Ellen Thomas, Louise Jones, Nirupa Murugaesu, Freya Boardman Pretty, Dina Halai, Emma Baple, Clare Craig, Angela Hamblin, Shirley Henderson, Christine Patch, Amanda O'Neill, Andrew Devereau, Katherine Smith, Antonio Rueda Martin, Alona Sosinsky, Ellen M. McDonagh, Razvan Sultana, Michael Mueller, Damian Smedley, Adam Toms, Lisa Dinh, Tom Fowler, Mark Bale, Tim Hubbard, Augusto Rendon, Sue Hill, and Mark J. Caulfield. The 100 000 genomes project: bringing whole genome sequencing to the nhs. *BMJ*, 361, 2018. doi:10.1136/bmj.k1687.