BWT for String Collections

Davide Cenzato
Ca' Foscari University of Venice, Italy

Zsuzsanna Lipták
University of Verona, Italy

Nadia Pisanti
University of Pisa, Italy

Giovanna Rosone
University of Pisa, Italy

Marinella Sciortino
University of Palermo, Italy

Abstract

We survey the different methods used for extending the BWT to collections of strings, following largely [Cenzato and Lipták, CPM 2022, Bioinformatics 2024]. We analyze the specific aspects and combinatorial properties of the resulting BWT variants and give a categorization of publicly available tools for computing the BWT of string collections. We show how the specific method used impacts on the resulting transform, including the number of runs, and on the dynamicity of the transform with respect to adding or removing strings from the collection.

We then focus on the number of runs of these BWT variants and present the optimal BWT introduced in [Cenzato et al., DCC 2023], which implements an algorithm originally proposed by [Bentley et al., ESA 2020] to minimize the number of BWT-runs. We also discuss several recent heuristics and study their impact on the compression of biological sequences.

We conclude with an overview of the applications and the impact of the BWT of string collections in bioinformatics.

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

Keywords and phrases Burrows-Wheeler transform, Extended Burrows-Wheeler transform, compressed text indexes, text compression, string collections, bioinformatics

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.3

Funding Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. GR and NP are partially supported by PNRR – M4C2 – Investimento 1.5, Ecosistema dell'Innovazione ECS00000017 – "THE – Tuscany Health Ecosystem" – Spoke 6 "Precision medicine & personalized healthcare", funded by the European Commission under the NextGeneration EU programme. ZsL, NP, GR, MS, are partially supported by MUR PRIN 2022 YRB97K "PINC" funded by Next Generation EU PNRR M4 C2 Inv. 1.1. ZsL, GR, and MS are also partially supported by the INdAM - GNCS Project CUP_E53C24001950001.
Davide Cenzato: DC is funded by ERC StG "REGINDEX: Compressed indexes for regular languages with applications to computational pan-genomics" grant nr 101039208.
Nadia Pisanti: NP is also supported by the PANGAIA and ALPACA projects that received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No. 872539 and 956229, respectively.
Marinella Sciortino: MS is partially supported by Project "ACoMPA" (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR MUR M4 C2 Inv. 1.5 – Project ECS00000017 Tuscany Health Ecosystem (Spoke 6), CUP B63C22000680007.

1 Introduction

With the advent of high-throughput sequencing technologies, the amount of genomic data available in public databases has undergone an exponential increase [43]. Similar data explosion can be witnessed in other areas based on textual data, such as webpages, versioned texts (such as Wikipedia), versioned software repositories (such as Github), digital books, and many others (for more details, see [93]).

However, not only has the amount or size of textual data increased, but it has also changed character. Most importantly, nowadays most data of interest consists of *string collections* rather than of individual strings. For the purpose of this survey, a string collection is any multiset of strings (or sequences, two terms we use interchangeably). Early examples include the famous 1000 Genomes Project [116], in the course of which almost 2000 human genomes were sequenced, followed by the 10,000 Genomes Project [56], the 100,000 Human Genome Project [120], the 1001 Arabidopsis Project [117], the 3,000 Rice Genomes Project (3K RGP) [115], and others. Projects of the last few years include the COVID-19 Genomics UK Consortium's [38] and the H3Africa Consortium's work [34].

The Burrows-Wheeler Transform (BWT) [23] is one of the most fundamental methods for string processing and compressed data storage [48, 83, 54]; the tools built upon these BWT-based compressed data structures are among the most used in bionformatics [71, 76, 69, 78]. It is therefore natural that researchers and practicioners have gone ahead and applied BWT-based methods to this new kind of data. The BWT, however, was originally defined for *one* string, and it is not immediately clear how to extend it to several strings.

As it turns out, several different methods have been used to generalize the BWT to string collections. Some of these methods were carefully designed and consciously chosen, while others appear to have been simply implementation choices. In fact, in many publications it is not explicitly specified how the BWT of a string collection was computed. Underlying this is the implicit assumption – incorrect, as we will see – that all methods are equivalent.

The classic method, paralleling *generalized suffix trees* and *generalized suffix arrays* (see e.g., [65] or [95]), is to concatenate the strings in the collection, using distinct end-of-string symbols to separate consecutive strings. In effect, this means turning the string collection into one string, at which point one can apply the classic BWT in the usual way. Following established terminology, we will refer to these end-of-string-symbols as 'dollars'; so we have a distinct dollar ($\$_i$) for each input string. In actual implementations, of course, the same dollar-symbol is used, because otherwise the alphabet would be blown up enormously, not a viable choice. However, the dollars remain distinct conceptually; one common way to handle this is to use the position of the dollars for breaking ties. This method, concatenating the strings with distinct dollars, is what we will refer to in this paper as *multidollarBWT*.

Another method that is commonly used is to concatenate the input strings but drop the distinction between the dollars; this is a much simpler choice on an implementation level because one does not have to take care of distinguishing between the different dollars. In order to ensure correctness (of BWT construction, LF-mapping, backward search, etc.), it is necessary to append one more symbol at the very end of the concatenated string (often termed 'hash' and denoted #), which is smaller than all other characters, including the dollar. We will call this method *concatBWT*.

In 2007, a mathematically clean extension of the BWT to string collections was proposed by Mantaci et al. [84], which the authors called *extended BWT* (EBWT). The EBWT is essentially the inverse of the Gessel-Reutenauer bijection [57], known in combinatorics on words, but hitherto unknown in the research community working on textual data structures

Table 1 The five BWT variants and the three BWTs using specific string orders, on the multiset $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. Double listings due to options of the software. We note also that `lFGSACA` is a library, while the others are dedicated command-line tools for BWT computation.

	result on example	*tools*
BWT variants (Sections 3 and 4)		
EBWT	`GGGCTACTCACACCTCTAGCG`	`PFP-eBWT` [101], `cais` [27], `lFGSACA` [73]
dolEBWT	`ACACAGGGCGCCTAT$$$TCTC$$G$C`	`G2BWT` [49], `PFP-eBWT` [101], `cais` [27], `msbwt` [88]
mdolEBWT and mdolBWT	`AGCACAGCGGCCTTA$$$TTCC$$G$C`	`BEETL` [8], `BCR_LCP_GSA` [7], `ropeBWT2` [107], `ropeBWT3` [108], `Merge-BWT` [87], `gsufsort` [61], `grlBWT` [60], `eGAP` [45], `eGSA` [47], `bwt-lcp-parallel` [24], `bwt-lcp-em` [25]
concatBWT	`$ACAGCAGCGGCCTAT$$#TCTC$$G$C`	`BigBWT` [15], `r-pfbwt` [105], `CMS-BWT` [37], tools for single-string BWT
specific string orders (Section 5)		
colexBWT	`AAACCGCGGGCCTAT$$$TCTC$$G$C`	`BCR_LCP_GSA` [7], `ropeBWT2` [107], `ropeBWT3` [108]
plusBWT	`AAACCGGGCGCCTTA$$$TTCC$$G$C`	`BCR_LCP_GSA` [7]
optBWT	`AAAGCCCGGGCCTTA$$$TTCC$$G$C`	`optimalBWT` [99]

and string processing. The EBWT is a generalization of the BWT from one string to a multiset of primitive strings. Similarly to the BWT, it is a permutation of the characters of the strings in input and therefore has length equal to the total length of the multiset in input. It also allows backward search and has the same 'clustering effect' as the BWT, i.e., it tends to produce long same-character runs on repetitive inputs. We refer to this method simply as EBWT.

Applying the classic BWT to a string which terminates with a dollar is different in many ways from applying it to one without. (For example, BWT(`banana$`) = `annb$aa`, while BWT(`banana`) = `nnbaaa`.) Similarly, appending a dollar to the ends of the input strings changes the properties of the EBWT. Again, one possibility is to append the same dollar to every input string; this is what happens, for example, reading in strings line-by-line, since each line is terminated by a newline symbol. We will call this method *dollarEBWT*.

Finally, if one appends different dollars to the input strings (or the same dollar but considering them as different), then we get what we call *multidollarEBWT*. Regarding the resulting transform, this method is similar to the multidollarBWT method (concatenate the strings with distinct dollars separating them), except that the indices of the dollars in the final transform are shifted by one.

In Table 1, we give a classification of existing BWT construction tools, geared specifically to string collections, according to the BWT variant they produce. As it is not always possible to distinguish the underlying method, our categorization is based on a combination of the information given in the accompanying papers and on reverse engineering the final output of the tools.

In this paper, we survey the five different methods sketched above, giving their exact definitions and combinatorial properties. As we will see, which method is chosen has important consequences for the resulting transform. This includes issues such as input order dependence (giving the string collection in a different order may or may not result in different outputs), dynamicity (how does adding or removing a string from the collection impact on

the transform), and number of runs of the BWT. This last parameter, often denoted r, is central in BWT-based data structure design, since modern compressed data structures such as the r-index [54] require storage space linear in r.

We will in fact show that if the goal is to reduce the number of runs, then either the multidollarEBWT or the multidollarBWT should be chosen, because methods leading to few runs – the exact method, called *optimalBWT*, or heuristics like *colexBWT* or *plusBWT* – only work with these variants. Moreover, both allow adding or removing strings from the collection without having to recompute the BWT. On the other hand, if input order independence is desired, then the EBWT or the dollarEBWT are the right choice.

We will also see that the concatBWT, the variant where the strings are concatenated using the same dollar, has several undesirable properties.

The number r of runs of the BWT appears increasingly as a parameter characterizing the input data: r, or equivalently, the average runlength n/r of the BWT, is often used as a measure of how repetitive the data is. This is well justified in the case of one string, as many recent publications on the theory of r as a repetitiveness measure attest (see [93] for a recent survey). However, for string collections, the parameter r is not well defined, given that different methods on the same input collection can give different r's, and even the same method on the same collection, if presented in a different order, can give very different r's (up to a multiplicative factor of over 31 on real data [30]). We argue that this parameter should be standardized, to r_{opt}, the minimum number of runs among all possible transforms on a given string collection.

We round off the paper with a section about applications in bionformatics that make use of string collections. These are highly repetitive data sets that are the ideal object for data structures based on the BWT. We give an overview of the different areas and problems using highly repetitive sequence collections, which are at the center of today's bioinformatics research.

1.1 Overview of paper

The paper is organized as follows. We provide necessary background and terminology in Section 2. In Section 3, we give the precise definitions and examples for the five BWT variants, distinguishing between EBWT-based variants (Section 3.1) and concatenation-based variants (Section 3.2). We study their combinatorial properties in Section 4, followed by a section on reducing the number of runs of the BWT (Section 5). This section also contains experimental results on real data, showing how much the number of runs can be reduced. Section 6 contains an overview of areas in bioinformatics where string collections play an important role. We close with a summary and some suggestions in Section 7.

The classification of BWT variants for string collections presented in this paper is based largely on that contained in [32, 33] but with some additions and modifications. In particular, Section 3, containing the definitions, has been extended, and the classification has been further developed[1]. All examples contained in this paper are new. The combinatorial properties of the BWT variants of Section 4 are partially from [32, 33], with updated presentation; Section 4.3, on dynamicity of the BWT variants, has been newly added and Section 4.4, on the output order of the BWT variants, generalizes results presented in [32]. Finally, Section 5 contains mostly results from [30, 14], while the survey of bioinformatics applications contained in Section 6 is not contained elsewhere.

[1] We now group the BWT variants in EBWT-based versus concatenation-based, and we subsume colexBWT and optBWT under multidollarEBWT and multidollarBWT.

2 Basics

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet Σ with $a_1 < a_2 < \ldots < a_\sigma$. A string (or text) $T = T[1..n]$ is a sequence of symbols $T[1] \cdots T[n]$ over the alphabet Σ. We denote by ϵ the empty string and by $|T|$ the length of string T. For $1 \leq i \leq n$, the substring $T[1..i]$ is called the ith *prefix* of T and $T[i..n]$ the ith *suffix* of T. The string $T^{\text{rev}} = T[n] \cdots T[1]$ is the *reverse* string of T. Let $<_{\text{lex}}$ denote the standard lexicographic order. The *colexicographic* order (sometimes also referred to as *reverse lexicographic order*) is defined as $S <_{\text{colex}} T$ if and only if $S^{\text{rev}} <_{\text{lex}} T^{\text{rev}}$.

Given a string T, the string $T^k = TT \cdots T$ is the string obtained by concatenating k copies of T. A string T is called *primitive* if $T = U^k$ implies $T = U$ and $k = 1$. For every string T, there exists a unique primitive string U and a unique integer k such that $T = U^k$. The string U is called $\text{root}(T)$ and k is called $\exp(T)$; thus, $T = \text{root}(T)^{\exp(T)}$.

For a string T, we denote by $T^\omega = TT \cdots$ the infinite string obtained by concatenating an infinite number of copies of T. The *omega-order* on Σ^* is defined as follows: $S \prec_\omega T$ if (a) $S^\omega <_{\text{lex}} T^\omega$, or (b) $S^\omega = T^\omega$ and $\exp(S) < \exp(T)$ (note that $S^\omega = T^\omega$ implies $\text{root}(S) = \text{root}(T)$). It can be verified that the ω-order relation coincides with the lexicographic order if neither of the two strings is a proper prefix of the other. Otherwise, the two orders can be different. For instance, `CGA` $<_{\text{lex}}$ `CGACC` but `CGACC` $\prec_\omega$ `CGA`.

Given string T, the *alphabet of* T is defined as $\text{alph}(T) = \{T_i \mid 1 \leq i \leq |T|\}$. A *run* in a string T is a maximal substring U such that $\text{alph}(U)$ is a singleton. Given a string T, we denote by $\rho(T)$ the number of runs of T, e.g., $\rho(\texttt{baaabbc}) = 4$. Moreover, we denote by $r(T)$ the number of runs of the BWT of T (defined below).

The string S is a *conjugate* (or *cyclic rotation*, or simply *rotation*) of the string T if $S = T[i..n]T[1..i-1]$, for some $i \in \{1, \ldots, n\}$. In this case, S is also called the ith conjugate of T, denoted $\text{conj}_i(T)$. The *conjugate array* CA of a string T of length n is the permutation of $\{1, \ldots, n\}$ such that $\text{CA}[j] = i$ if $\text{conj}_i(T)$ is the jth conjugate of T with respect to the lexicographic order, with ties broken according to string order, i.e. if $\text{CA}[j] = i$ and $\text{CA}[j'] = i'$ for some $j < j'$, then either $\text{conj}_i(T) <_{\text{lex}} \text{conj}_{i'}(T)$, or $\text{conj}_i(T) = \text{conj}_{i'}(T)$ and $i < i'$. If T is a primitive string, a conjugate S of T is called *Lyndon rotation* of T, denoted as $Lynd(T)$, if it is lexicographically the smallest among all conjugates of T.

To define the *suffix array* SA, it is common to assume that the string has an end-of-string symbol \$ appended at the end, such that \$ is smaller than all other characters and does not occur elsewhere in the string. Then, the SA is a permutation of the indices $\{1, 2, \ldots, n+1\}$ such that $\text{SA}[j] = i$ if suffix $T[i..n+1]$ is the jth suffix of all suffixes in lexicographic order. Note that in this case, there can be no ties, since two distinct suffixes necessarily have different lengths. Since the string $T\$$ has exactly one occurrence of \$, the conjugate array and the suffix array of $T\$$ coincide.

The *Burrows-Wheeler Transform* (BWT) [23] is a reversible transformation that permutes the characters of the input string. Given a string T, the *BW-matrix* of T is defined as the matrix whose rows are the conjugates of T, given in lexicographic order. We denote by $\text{BWT}(T)$ the last column of the matrix. The transformation also outputs an integer i, which is the position of T within the matrix. It follows from the definition that if a string S is a conjugate of T, then $\text{BWT}(S) = \text{BWT}(T)$. The index i is fundamental for recovering T from $\text{BWT}(T)$, otherwise the original string can be recovered only up to conjugates. When the reconstruction of the original string is not relevant, the index is omitted.

The *standard permutation* of a word $S = S[1..n]$ over Σ is the permutation π_S of $\{1, 2, \ldots, n\}$ such that $\pi_S(i) < \pi_S(j)$ if and only if $S[i] < S[j]$ or $S[i] = S[j]$ and $i < j$. The standard permutation π_S of a string S is also called LF-mapping if $S = \text{BWT}(T)$, for some

string T. It is known that T is a primitive word if and only if π_S is a cycle of length n. In this case, given S, π_S and an index i, the ith conjugate U of T in lexicographic order can be recovered as follows: $U[n] = S[i]$ and $U[n-j] = S[\pi_S^j(i)]$, for $j = 1, \ldots, n-1$. The conjugate $Lynd(T)$ can be recovered with $i = 1$.

The permutation BWT(T) can be computed from the conjugate array CA, since for all $j = 1, \ldots, n$, BWT(T)[j] = T[CA[j] − 1], assuming that $T[0] = T[n]$. Alternatively, if the input string T terminates with an end-of-string symbol \$, BWT(T\$) can be computed from the suffix array SA. Moreover, recovering the first row of the BW-matrix will yield \$$T$, because \$ is smaller than all other symbols, and since \$ occurs exactly once, unique recovery is ensured also in this case.

Appending an end-of-string symbol to the string T has traditionally allowed BWT(T\$) to be computed using linear-time algorithms for suffix array construction. However, it is possible to compute BWT(T) in linear time without appending any end-of-string symbol \$ to T. This can be done by computing the suffix array of $U = Lynd(\text{root}(T))$ [58] and then obtaining BWT(T) = BWT(U^k), where $k = \exp(T)$, from BWT(U), applying a simple relationship between the two transforms: BWT(U^k) can be obtained by concatenating k copies of each character of BWT(U) [85]. Alternatively, it is also possible to compute BWT(T) directly in linear time with the `cais` algorithm, which is based on induced sorting [18].

Now let $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ be a multiset of strings (often referred to as a *string collection*), with $|T_d| = n_d$ for $1 \leq d \leq m$. We denote by $|\mathcal{M}| = m$ the number of strings in $\mathcal{M}$ and by $||\mathcal{M}|| = \sum_{d=1}^{m} n_d$ their total length. Further, we will denote by $\mathcal{M}_\$ = \{T_1\$_1, T_2\$_2, \ldots, T_m\$_m\}$ the set of strings with end-of-string markers appended to all strings in $\mathcal{M}$, of total length $m + \sum_{d=1}^{m} n_d$.

The *generalized conjugate array* GCA of the collection $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ is an array of total length $||\mathcal{M}||$, where $GCA[j] = (d, i)$ if $conj_i(T_d)$ is the jth string in the ω-sorted list of the conjugates of the strings in $\mathcal{M}$, with ties broken first w.r.t. the index of the string (in case of identical strings), and then w.r.t. the position in the string (in case of identical conjugates of the same string).

The *generalized suffix array* GSA of the collection $\mathcal{M}_\$$ is an array of length $||\mathcal{M}_\$|| = m + \sum_{d=1}^{m} n_d$, containing pairs of integers (d, i), corresponding to the lexicographically sorted suffixes of the strings in $\mathcal{M}_\$$. In particular, GSA[j] = (d, i) is the pair corresponding to the jth smallest suffix of the strings in $\mathcal{M}_\$$, with ties broken according to $\$_1 < \$_2 < \ldots < \$_m$.

3 BWT variants for string collections

For string collections, several variants of the BWT have been proposed. Each of these variants aims to extend the features of the BWT for single strings to collections of strings. In this section, we review the most significant variants, which form the foundation of the most widely used tools for indexing and compressing string collections based on the BWT. We use the classification introduced in [32, 33], and in addition group these variants into two families, depending on the strategy used to process the strings of the collection.[2] The first family includes transformations that explicitly or implicitly apply the Extended BWT (EBWT) introduced in [84] to the multiset of strings, to which end-of-string symbols may or may not have been appended. The second family, on the other hand, consists of those

[2] In difference to [32, 33], here we include a new separator-based BWT variant, the multidollarEBWT, which was previously subsumed under multidollarBWT.

transformations that involve concatenating the strings of the collection, and then applying the traditional BWT to the concatenated string. We will see that these BWT variants, while preserving the general functionality of the BWT, can produce significantly different outputs.

Let us denote by $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ the input string collection, i.e., a multiset of m strings, with $|T_d| = n_d$ for $1 \leq d \leq m$. Even though the input is a *multiset* of strings, in real applications, this input is always given in some order. As we will see, the input order plays an important role in the resulting transform for several of the BWT variants we will review. For this reason, in slight abuse of notation, whenever the definition depends on the input order, we will treat $\mathcal{M}$ in the following as an ordered collection and will write accordingly $\mathcal{M} = (T_1, T_2, \ldots, T_m)$. When it does not play a role, in the sense that the output is always the same whatever the input order, in those cases we will stick to the multiset notation.

In Table 1 we list, for each BWT variant, the tools that compute it (up to renaming of dollars).

3.1 EBWT-based transforms

In this section, we present the three EBWT-based transforms: EBWT, dolEBWT, and mdolEBWT. See Table 2 for an example.

Extended BWT. The *Extended Burrows–Wheeler Transform* [84] (EBWT) is a generalization of the BWT to a multiset of strings. Given the collection of strings $\mathcal{M} = \{T_1, \ldots, T_m\}$, EBWT($\mathcal{M}$) is a permutation of the characters of the strings in $\mathcal{M}$, obtained by concatenating the last characters of all conjugates of the strings of $\mathcal{M}$, listed according to the ω-order. Note that EBWT is a transform on the string collections treated as multisets. Indeed, the output EBWT($\mathcal{M}$) is independent of the order in which the strings appear in the collection $\mathcal{M}$. For this reason, we denote its input as a multiset.

Similarly to BWT, the permutation EBWT($\mathcal{M}$) can be computed from the generalized conjugate array of $\mathcal{M}$ in linear time, since

$$\text{EBWT}(\mathcal{M})[i] = \begin{cases} T_d[j-1] & \text{if GCA}[i] = (d, j) \text{ with } j > 1, \\ T_d[n_d] & \text{if GCA}[i] = (d, j) \text{ with } j = 1. \end{cases} \tag{1}$$

The transformation also outputs the m-tuple $I = (i_1, \ldots, i_m)$ listing the positions of the strings of the collection within this ω-sorted list. The m-tuple $I = (i_1, \ldots, i_m)$ is used to recover the multiset $\mathcal{M}$ from EBWT($\mathcal{M}$). In fact, if π_U is the standard permutation of the word $U = \text{EBWT}(\mathcal{M})$, then π_U decomposes into m disjoint cycles $\pi_1, \pi_2, \ldots, \pi_m$ such that the index i_d appears in the cycle π_d. The strings of the collection $\mathcal{M}$ are recovered as follows: for every $d = 1, \ldots, m$, $T_d[l_d - t] = \text{EBWT}[{\pi_d}^t[i_d]]$, where l_d is the length of the cycle π_d and t ranges from 0 to $l_d - 1$. Moreover, if the set of indices is considered in increasing order, then the strings of the collection are recovered in ω-order. When the reconstruction of the original multiset is not relevant, the m-tuple is omitted.

▶ **Example 1.** Given the collection $\mathcal{M} = \{\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA}\}$, the output of EBWT is the string `GGGCTACTCACACCTCTAGCG` together with the indices $(9, 10, 12, 16, 18, 21)$, as shown in Table 2. The LF-mapping decomposes into the cycles (1 13 9 7 6) (2 14 10) (3 15 20 12) (4 5 18) (8 19 16 11) and (17 21). Starting from the indices in the set 9, 10, 12, 16, 18 and 21, the strings `CGACC`, `CGA`, `CTGA`, `GTCC`, `TCA` and `TG` are reconstructed.

Dollar-EBWT. This transformation, here denoted dolEBWT, uses a single end-of-string symbol appended to each string in the collection. More precisely, given the collection $\mathcal{M} = \{T_1, \ldots, T_m\}$,

$$\text{dolEBWT}(\mathcal{M}) = \text{EBWT}(\{T_d\$ \mid d = 1, \ldots, m\}).$$

Note that in this case, to recover the set of strings in the collection, it is sufficient to know the positions of the \$-symbols in dolEBWT($\mathcal{M}$). Since none of the strings in the collection is a prefix of another, the ω-order coincides with the lexicographic order. For this reason, starting from the positions of the \$ symbols in increasing order, the strings of $\mathcal{M}$ can be recovered in lexicographic order.

Similarly to EBWT, the permutation dolEBWT($\mathcal{M}$) can be computed from the generalized conjugate array GCA of the multiset $\{T_d\$ \mid T_d \in \mathcal{M}\}$ in linear time, since

$$\text{dolEBWT}(\mathcal{M})[i] = \begin{cases} T_d[j-1] & \text{if GCA}[i] = (d, j) \text{ with } j > 1, \\ \$ & \text{if GCA}[i] = (d, j) \text{ with } j = 1. \end{cases} \tag{2}$$

▶ **Example 2.** Let us consider the collection $\mathcal{M} = \{\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA}\}$, then dolEBWT($\mathcal{M}$) = ACACAGGGCGCCTAT\$\$\$TCTC\$\$G\$C, as shown in Table 2. The LF-mapping decomposes into 6 cycles (1 7 20 16) (2 11 14 10 22 17) (3 8 21 27 18) (4 12 15 25 23) (5 9 13 24) and (6 19 26).

Multidollar-EBWT. This transformation, here denoted mdolEBWT, uses distinct end-of-string symbols appended to the strings in the collection. Given the collection $\mathcal{M} = (T_1, \ldots, T_m)$ and the end-of-strings symbols $\$_1 < \$_2 < \cdots < \$_m$,

$$\text{mdolEBWT}(\mathcal{M}) = \text{EBWT}(\{T_d\$_d \mid d = 1, \ldots, m\}).$$

Note that, unlike the previous transformations, the output of mdolEBWT($\mathcal{M}$) is influenced by the order in which the strings appear in the collection, because the distinct end-of-string symbols result in different orders among the strings in the multiset. (For more on this, see Section 3.) Further, the position of each $\$_d$ allows for the reconstruction of the corresponding string T_d of the collection.

Since end-marker symbols are appended to each string in the collection, mdolEBWT($\mathcal{M}$) can be constructed using the lexicographic order of the suffixes of the strings in $\mathcal{M}$ as follows:

$$\text{mdolEBWT}(\mathcal{M})[i] = \begin{cases} T_d\$_d[(j-1)] & \text{if GSA}[i] = (d, j) \text{ with } j > 1 \\ \$_d & \text{if GSA}[i] = (d, j) \text{ with } j = 1 \end{cases} \tag{3}$$

▶ **Example 3.** Let us consider the collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. Then mdolEBWT($\mathcal{M}$) = AGCACAGCGGCCTTA\$$_6$\$$_5$\$$_1$TTCC\$$_3$\$$_4$G\$$_2$C, as shown in Table 2. The LF-mapping decomposes into 6 cycles (1 7 20 27 18) (2 19 26) (3 11 14 25 23) (4 8 13 24) (5 12 15 10 22 17) and (6 9 21 16). For $\mathcal{M}' = (\texttt{TCA}, \texttt{CTGA}, \texttt{CGA}, \texttt{TG}, \texttt{GTCC}, \texttt{CGACC})$, where the same strings are given in another order, mdolEBWT($\mathcal{M}'$) = AAAGCCCGGGCCTTA\$$_3$\$$_6$\$$_2$TTCC\$$_5$\$$_1$G\$$_4$C. For more details, see Section 5.

3.2 Concatenation-based transforms

In this section, we present the concatenation-based transforms, mdolBWT and concatBWT. See Table 3 for an example.

Table 2 EBWT-based BWT variants: from left to right, we show the extended BWT (EBWT), the dollar-EBWT, and the multidollar-EBWT (using different string orderings) of the string collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. We use different colors to identify the characters preceding different identical suffixes. See Sections 3.1 and 5 for more details.

	EBWT		
i	GCA	rotation	
1	(5,3)	ACCCG	G
2	(6,3)	ACG	G
3	(1,4)	ACTG	G
4	(4,3)	ATC	C
5	(4,2)	CAT	T
6	(5,4)	CCCGA	A
7	(5,5)	CCGAC	C
8	(3,3)	CCGT	T
9	(5,1)	CGACC	C
10	(6,1)	CGA	A
11	(3,4)	CGTC	C
12	(1,1)	CTGA	A
13	(5,2)	GACCC	C
14	(6,2)	GAC	C
15	(1,3)	GACT	T
16	(3,1)	GTCC	C
17	(2,2)	GT	T
18	(4,1)	TCA	A
19	(3,2)	TCCG	G
20	(1,2)	TGAC	C
21	(2,1)	TG	G

	dollar-EBWT			multidollar-EBWT					
i	GCA	rotation		opt	plus	colex	input	rotation	GSA
1	(6,4)	$CGA	A	A	A	A	A	$_1CTGA	(1,5)
2	(5,6)	$CGACC	C	A	A	A	G	$_2TG	(2,3)
3	(1,5)	$CTGA	A	A	A	A	C	$_3GTCC	(3,5)
4	(3,5)	$GTCC	C	G	C	C	A	$_4TCA	(4,4)
5	(4,4)	$TCA	A	C	C	C	C	$_5CGACC	(5,6)
6	(2,3)	$TG	G	C	G	G	A	$_6CGA	(6,4)
7	(6,3)	A$CG	G	C	G	C	G	A$_1CTG	(1,4)
8	(1,4)	A$CTG	G	G	G	G	C	A$_4TC	(4,3)
9	(4,3)	A$TC	C	G	C	G	G	A$_6CG	(6,3)
10	(5,3)	ACC$CG	G	G	G	G	G	ACC$_5CG	(5,3)
11	(5,5)	C$CGAC	C	C	C	C	C	C$_3GTC	(3,4)
12	(3,4)	C$GTC	C	C	C	C	C	C$_5CGAC	(5,5)
13	(4,2)	CA$T	T	T	T	T	T	CA$_4T	(4,2)
14	(5,4)	CC$CGA	A	T	T	A	T	CC$_3GT	(3,3)
15	(3,3)	CC$GT	T	A	A	T	A	CC$_5CGA	(5,4)
16	(6,1)	CGA$	$	$_3	$_2	$_2	$_6	CGA$_6	(6,1)
17	(5,1)	CGACC$	$	$_6	$_5	$_4	$_5	CGACC$_5	(5,1)
18	(1,1)	CTGA$	$	$_2	$_1	$_3	$_1	CTGA$_1	(1,1)
19	(2,2)	G$T	T	T	T	T	T	G$_2T	(2,2)
20	(6,2)	GA$C	C	T	T	C	T	GA$_1CT	(1,3)
21	(1,3)	GA$CT	T	C	C	T	C	GA$_6C	(6,2)
22	(5,2)	GACC$C	C	C	C	C	C	GACC$_5C	(5,2)
23	(3,1)	GTCC$	$	$_5	$_4	$_5	$_3	GTCC$_3	(3,1)
24	(4,1)	TCA$	$	$_1	$_3	$_1	$_4	TCA$_4	(4,1)
25	(3,2)	TCC$G	G	G	G	G	G	TCC$_3G	(3,2)
26	(2,1)	TG$	$	$_4	$_6	$_6	$_2	TG$_2	(2,1)
27	(1,2)	TGA$C	C	C	C	C	C	TGA$_1C	(1,2)

Multidollar-BWT. In this transformation, denoted mdolBWT, given the string collection $\mathcal{M} = (T_1, \ldots, T_m)$, the strings are concatenated in the order in which they appear in the m-tuple, using $\$_1 < \$_2 < \ldots < \$_m$ as separators, and then the BWT is applied to the concatenated string. More formally,

$$\text{mdolBWT}(T_1, \ldots, T_m) = \text{BWT}(T_1\$_1T_2\$_2 \cdots T_m\$_m).$$

Unlike the EBWT-based transforms, if π_L is the standard permutation of the word $L = \text{dolEBWT}(\mathcal{M})$, then π_L consists of a single cycle. Each string T_d in the collection can be reconstructed, using the LF mapping, starting from the symbol $\$_d$ and terminating the reconstruction when the symbol $\$_{d-1}$ (for $d > 1$) or $\$_m$ (for $d = 1$) is reached.

The order in which the strings appear in the collection can significantly affect the output, giving rise to specific transforms. See Section 5 for more details.

Note that sometimes the separators are added only implicitly when concatenating the strings, for example by maintaining a bitvector that marks the beginning of a new string. Moreover, some implementations of mdolBWT compute the BWT after appending an additional end-of-string symbol #, which is smaller than all the characters of the alphabet

and the dollar symbols, to the concatenation of the strings. The resulting output differs from mdolBWT only in that the symbol $\$_m$ is added in the first position, and the symbol # replaces $\$_m$. The mdolBWT($\mathcal{M}$) is constructed using the suffix array SA of the concatenated string $T = T_1\$_1T_2\$_2\cdots T_m\$_m$ as follows:

$$\text{mdolBWT}(\mathcal{M})[i] = \begin{cases} T[\text{SA}[i]-1] & \text{if SA}[i] > 1, \\ \$_m & \text{if SA}[i] = 1. \end{cases} \tag{4}$$

▶ **Example 4.** Let us consider the collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. Then mdolBWT($\mathcal{M}$) = $\texttt{AGCACAGCGGCCTTA}\$_5\$_4\$_6\texttt{TTCC}\$_2\$_3\texttt{G}\$_1\texttt{C}$, as shown in Table 3. The LF-mapping consists of a single cycle, which is (1 7 20 27 18 6 9 21 16 5 12 15 10 22 17 4 8 13 24 3 11 14 25 23 2 19 26). If $\mathcal{M}''$ contains the same strings in colexicographic order, i.e., $\mathcal{M}'' = (\texttt{TCA}, \texttt{CGA}, \texttt{CTGA}, \texttt{CGACC}, \texttt{GTCC}, \texttt{TG})$, then mdolBWT($\mathcal{M}''$) = $\texttt{AAACCGCGGGCCTAT}\$_1\$_3\$_2\texttt{TCTC}\$_4\$_6\texttt{G}\$_5\texttt{C}$. If the end-of-string symbol # is appended to the concatenated strings, then the output of the transformation is $\$_6\texttt{AGCACAGCGGCCTTA}\$_5\$_4\#\texttt{TTCC}\$_2\$_3\texttt{G}\$_1\texttt{C}$.

Concatenated-BWT. This transformation, denoted concatBWT, concatenates all the strings into a single string using a single end-of-string symbol \$ as separator and appending one additional character, the final end-of-string symbol, here denoted by #, which is smaller than all other characters. Then the classic BWT is applied to this string. This is the method implicitly applied when using a tool for classic BWT construction, so probably this is the most commonly used method. Given the collection $\mathcal{M} = (T_1, T_2, \ldots, T_m)$,

$$\text{concatBWT}(T_1, \ldots, T_m) = \text{BWT}(T_1\$T_2\$\cdots T_m\$\#).$$

The concatBWT can be computed using the SA of the concatenation of the input strings, similarly to Eq. (4), as follows:

$$\text{concatBWT}(\mathcal{M})[i] = \begin{cases} T[\text{SA}[i]-1] & \text{if SA}[i] > 1, \\ \# & \text{if SA}[i] = 1. \end{cases} \tag{5}$$

Because of the extra end-of-string symbol, the additional suffix starting with # is the smallest, and therefore, we haver an additional dollar in the first position of the transform. To facilitate the comparison with the other transforms, we denote by *adapted concatBWT*, the transform obtained by removing the first symbol from concatBWT and replacing the # by \$.

▶ **Example 5.** Let us consider the collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. Then concatBWT($\mathcal{M}$) = $\texttt{\$ACAGCAGCGGCCTAT\$\$\#TCTC\$\$G\$C}$, as shown in Table 3. The LF-mapping consists of a single cycle, which is (1 2 8 21 17 3 12 15 11 23 18 4 9 14 25 6 13 16 26 24 5 20 27 7 10 22 28 19). The adapted concatBWT is $\texttt{ACAGCAGCGGCCTAT\$\$\$TCTC\$\$G\$C}$.

Note that the #-symbol at the end of the string is a necessary addition, as without it, the LF-mapping, and thus, backward search, may not work correctly. This is because in that case, a suffix of the concatenated string may be a proper prefix of another suffix. This is the BWT variant treated in [29, 80].

3.3 A short literature overview of BWT variants

Until a few years ago, all methods and algorithms extending the Burrows-Wheeler Transform (BWT) to string collections were considered equivalent. In the previous subsections, we described different types of approaches and the specific characteristics of each BWT variant. In this subsection, we survey the main papers where the BWT variants have been treated, often using different algorithmic strategies. The corresponding tools are listed in Table 1.

Table 3 Concatenation-based BWT variants: the multidollar-BWT (left) and the concat-BWT (right) of the string collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. We use different colors to identify the characters preceding different identical suffixes. See Section 3.2 for more details.

	multidollar-BWT			concatenated-BWT			
i	SA	rotation			rotation	SA	i
				$	#	28	1
1	5	$\$_1TG\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	A	A	$#	27	2
2	8	$\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	G	C	CGA#	23	3
3	13	$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	C	A	$CGACC$CGA$#	17	4
4	17	$\$_4$CGACC$\$_5CGA\$_6$	A	G	$GTCC$TCA$CGACC$CGA$#	8	5
5	23	$\$_5CGA\$_6$	C	C	TCACGACCCGA#	13	6
6	27	$\$_6$	A	A	TGGTCCTCACGACCCGA#	5	7
7	4	A$\$_1TG\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	G	G	A$#	26	8
8	16	A$\$_4$CGACC$\$_5CGA\$_6$	C	C	A$CGACC$CGA$#	16	9
9	26	A$\$_6$	G	G	ATGGTCCTCACGACCCGA#	4	10
10	20	ACC$\$_5CGA\$_6$	G	G	ACCCGA#	20	11
11	12	C$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	C	C	CCGA#	22	12
12	22	C$\$_5CGA\$_6$	C	C	CTCACGACCCGA#	12	13
13	15	CA$\$_4$CGACC$\$_5CGA\$_6$	T	T	CA$CGACC$CGA$#	15	14
14	11	CC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	T	A	CCCGA#	21	15
15	21	CC$\$_5CGA\$_6$	A	T	CCTCACGACCCGA#	11	16
16	24	CGA$\$_6$	$\$_5$	$	CGA$#	24	17
17	18	CGACC$\$_5CGA\$_6$	$\$_4$	$	CGACCCGA#	18	18
18	1	CTGA$\$_1TG\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	$\$_6$	#	CTGATGGTCCTCACGACCCGA#	1	19
19	7	G$\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	T	T	G$GTCC$TCA$CGACC$CGA$#	7	20
20	3	GA$\$_1TG\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	T	C	GA$#	25	21
21	25	GA$\$_6$	C	T	GATGGTCCTCACGACCCGA#	3	22
22	19	GACC$\$_5CGA\$_6$	C	C	GACCCGA#	19	23
23	9	GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	$\$_2$	$	GTCCTCACGACCCGA#	9	24
24	14	TCA$\$_4$CGACC$\$_5CGA\$_6$	$\$_3$	$	TCA$CGACC$CGA$#	14	25
25	10	TCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	G	G	TCCTCACGACCCGA#	10	26
26	6	TG$\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	$\$_1$	$	TG$GTCC$TCA$CGACC$CGA$#	6	27
27	2	TGA$\$_1TG\$_2$GTCC$\$_3TCA\$_4$CGACC$\$_5CGA\$_6$	C	C	TGATGGTCCTCACGACCCGA#	2	28

Linear-time algorithms for the computation of the original EBWT variant were recently presented in [18, 96, 5]. In particular, in [18], an adaptation of the well-known Suffix Array Induced Sorting (SAIS) algorithm of Nong et al. [94] is used. In [5], the computation is done via computing the Bijective BWT (BBWT) of a string which is obtained from the input multiset. In [96], a non-recursive suffix array construction is used.

The dolEBWT variant is studied in [41, 66]. In [41], the computation is based on the grammar compression of the text. In [66], a merging strategy is used to compute the BWT of a collection by successively merging the BWTs of smaller subsets. The mdolEBWT and mdolBWT variants, although based on different constructions – via the EBWT and via concatenation, respectively – produce essentially the same output string when the indices associated with the dollar symbols are ignored. Indeed, their output strings coincide up to the renaming of the dollar symbols, as detailed in Section 4. Notably, the LF-mapping associated with mdolEBWT decomposes into a number of cycles equal to the number of input strings, whereas for mdolBWT it forms a single cycle. However, the two transforms are easily related: one can be obtained from the other by shifting by one the indices associated with the dollar symbols, as stated in Lemma 6. Therefore, any tool computing one of these

two BWT variants can be easily adapted to compute the other. For this reason, in Table 1 where the indices associated to dollar symbols are ignored, we grouped together all tools that compute mdolEBWT and/or mdolBWT.

More in detail, the computation of mdolEBWT is explicitly treated in [6, 74, 112, 82, 46, 17, 81, 16, 42, 75]. Different algorithmic strategies are described in these papers: some algorithms implicitly sort the suffixes or build the generalized suffix array incrementally; others compute the transform by merging partial BWTs obtained from subsets of the collection; and some – such as the approach described in [81] – adopt a hybrid strategy, constructing the GSA starting from a single concatenated string and then adapting it to the collection setting. Instead, the mdolBWT variant is explicitly used in [9] to address the problem of finding the optimal order of concatenation of the strings, aiming at minimizing the number of runs in the resulting BWT.

The concatBWT variant, which uses a single dollar symbol as a separator and appends a final hash character, is adopted in [22, 97, 86]. This is also the variant implicitly computed by tools originally designed for single-string BWT construction.

4 Combinatorial properties of the different BWT variants

In this section, we study combinatorial properties of the five BWT variants introduced in Section 3. We give an overview in Table 4. Full proofs for Sections 4.1 and 4.2 can be found in the Supplementary Material of [33]. Section 4.3 is new, and Section 4.4 is a generalization of the respective results in [32].

First notice that all BWT variants except the EBWT append an end-of-string symbol (resp. separator-symbol) at the end of each input string. For this reason, we will in the following refer to these BWT variants as *separator-based* BWT variants.

Table 4 Overview of some properties of the BWT variants considered in this paper on the string collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. The colors in the example BWTs correspond to SAP intervals in separator-based variants, while the same characters are highlighted in the EBWT for showing their positions, see Section 4.2. Dynamicity is w.r.t. removal and addition of strings to the collection, see Section 4.3. The order of shared suffixes is discussed in Section 4.4.

BWT variant	example	order of shared suffixes	input order dependence	dynamicity
EBWT-*based*				
EBWT($\mathcal{M}$)	GGGCTACTCACACCTCTAGCG	omega-order	no	yes
dolEBWT($\mathcal{M}$)	ACACAGGGCGCCTAT$$$TCTC$$G$C	lexicographic order	no	yes
mdolEBWT($\mathcal{M}$)	AGCACAGCGGCCTTA$_6$$_5$$_1$TTCC$$_3$$_4$G$$_2$C	input order	yes	yes
concatenation-based				
mdolBWT($\mathcal{M}$)	AGCACAGCGGCCTTA$_5$$_4$$_6$TTCC$$_2$$_3$G$$_1$C	input order	yes	yes
concatBWT($\mathcal{M}$)	$ACAGCAGCGGCCTAT$$#TCTC$$G$C	lex. order of subseq. string in input	yes	no

4.1 Differences between EBWT and separator-based variants

In all separator-based BWT variants, the m-length prefix of the resulting transform consists of a permutation of the last characters of the input strings T_d, since the smallest rotations are those starting with the separators. In the EBWT, on the other hand, these m characters appear interspersed with the other characters (see Table 4). In general, adding separators introduces a distinction, not present in the EBWT, between occurrences of a substring as

suffix and otherwise: in the EBWT, the left-context of a substring appears according to the omega-order of the corresponding rotations, while in the separator-based variants, the occurrences that are suffixes appear in one block at the start.

To demonstrate this, consider the substring CC in our running example. As can be seen in Table 2, CC occurs twice as suffix and once not as a suffix (in this case, this is a circular occurrence). The two preceding characters of the suffix occurrences, A and T, form one block in all separator-based BWTs, while in the EBWT, they are intermixed with the preceding C of the non-suffix occurrence. A similar observation can be made about the two occurrences of C as suffix, whose preceding characters (both C) appear in once block in the separator-based BWT variants, while they are intermixed with all other occurrences of C in the EBWT.

Moreover, observe that adding separator-symbols changes the order of the strings. In particular, appending \$'s to the strings has the consequence that no string is a proper prefix of another, and therefore, the omega-order and the lexicographic order on these strings coincide. This is true whether the same separator-symbol or different symbols are used. An example is given in Figure 1, where we also show the difference between mdolEBWT($\mathcal{M}$) and mdolBWT($\mathcal{M}$).

EBWT({GTC, GT})		EBWT({GTC\$, GT\$})		EBWT({GTC\$$_1$, GT\$$_2$})		BWT(GTC\$$_1$GT\$$_2$)	
CGT	T	\$GT	T	\$$_1$	C	\$$_1$GT\$$_2$GTC	C
GTC	C	\$GTC	C	\$$_2$	T	\$$_2$GTC\$$_1$GT	T
GT	T	C\$GT	T	C\$$_1$	T	C\$$_1$GT\$$_2$GT	T
TCG	G	GT\$	\$	GT\$$_2$	\$$_2$	GT\$$_2$GTC\$$_1$	\$$_1$
TG	G	GTC\$	\$	GTC\$$_1$	\$$_1$	GTC\$$_1$GT\$$_2$	\$$_2$
		T\$G	G	T\$$_2$	G	T\$$_2$GTC\$$_1$G	G
		TC\$G	G	TC\$$_1$	G	TC\$$_1$GT\$$_2$G	G
EBWT($\mathcal{M}$)		dolEBWT($\mathcal{M}$)		mdolEBWT($\mathcal{M}$)		mdolBWT($\mathcal{M}$)	

Figure 1 The EBWT of the three sets {GTC, GT}, {GTC\$, GT\$}, and {GTC\$$_1$, GT\$$_2$}, and the BWT of the concatenated string GTC\$$_1$GT\$$_2$, corresponding to EBWT, dolEBWT, mdolEBWT, and mdolBWT of the same input. As can be seen, the result is different in each case. See Section 4.1 for a discussion.

Let us denote by $\hat{=}$ that two transforms are equal up to renaming the dollar symbols. The following lemma states that the transformation mdolBWT and mdolEBWT produce the same string, up to renaming dollars, and that dolEBWT is equivalent to first lexicographically sorting the input strings, and then applying either the mdolBWT or the mdolEBWT.

▶ **Lemma 6.** *Let $\mathcal{M} = (T_1, T_2, \ldots, T_m)$ be a string collection. Then*

1. *mdolBWT($\mathcal{M}$) $\hat{=}$ mdolEBWT($\mathcal{M}$). Moreover, \$$_d$ in mdolEBWT($\mathcal{M}$) is replaced by \$$_{d-1}$ in mdolBWT($\mathcal{M}$), and \$$_1$ by \$$_m$. Formally, mdolBWT($\mathcal{M}$) = EBWT($\{T_d\$_{d-1} \mid d = 1, \ldots, m,$ with $T_0 = T_m\}$).*
2. *dolEBWT($\mathcal{M}$) $\hat{=}$ mdolBWT(lex($\mathcal{M}$)) $\hat{=}$ mdolEBWT(lex($\mathcal{M}$)), where lex($\mathcal{M}$) denotes the lexicographic order of the strings in $\mathcal{M}$.*

In the following section, we will concentrate on separator-based BWT variants only and show how they differ from each other.

4.2 Interesting intervals and SAP-intervals

Let us call a string U a *shared suffix* if there exist at least two strings $T_d, T_{d'} \in \mathcal{M}$, $d \neq d'$, such that U is a suffix of both T_d and $T_{d'}$. Let b denote the lexicographic rank of the smallest rotation beginning with $U\$$ (where \$ now stands for *any* separator-symbol) and e that of the

largest. In other words, $[b, e]$ is the SA-interval of $U\$$. We refer to $[b, e]$ as an *SAP-interval*, following [39][3]. If there exist $d \neq d'$ such that U is a suffix of both T_i and T_j and the preceding character in T_d is distinct from the preceding character in $T_{d'}$, then $[b, e]$ is called an *interesting interval* [32, 33].

The next lemma follows from the fact that no two distinct substrings ending in \$ can be one the prefix of the other.

▶ **Lemma 7.** *Any two distinct interesting intervals are disjoint. Moreover, any two distinct SAP-intervals are disjoint.*

Interesting intervals allow us to locate precisely how and where two separator-based BWT variants of the same multiset can differ, which we state formally in the next proposition. For the proof, see the Supplemental Material of [33].

▶ **Proposition 8.** *Let L_1 and L_2 be two separator-based BWTs of the same multiset $\mathcal{M}$.*

1. *If $L_1[i] \neq L_2[i]$ then $i \in [b, e]$ for some interesting interval $[b, e]$.*
2. *Let $\mathcal{I}_1$ resp. $\mathcal{I}_2$ be the sets of the positions of the dollars in L_1 resp. L_2. If $\mathcal{I}_1 \neq \mathcal{I}_2$ then there exist $d \neq d'$ such that T_d is a proper suffix of $T_{d'}$.*

Thus, the differences between two separator-based BWT variants of the same multiset can be found only in interesting intervals (Property 1). To see Property 2, consider the following example:

▶ **Example 9.** Consider the mdolEBWT of the same multiset $\{\texttt{GCA}, \texttt{CA}\}$, given in two different orders: $\text{mdolEBWT}(\texttt{GCA}, \texttt{CA}) = \texttt{AACCG\$}_2\texttt{\$}_1$, while $\text{mdolEBWT}(\texttt{CA}, \texttt{GCA}) = \texttt{AACC\$}_1\texttt{G\$}_2$. This is because the interesting interval $[5, 6]$ associated with the shared suffix CA contains a dollar and a G, which change place according to the input order.

Proposition 8 implies that the differences between two separator-based BWT variants of the same multiset can be fully explained based on what rule is used to break ties for shared suffixes. We will see in Section 4.4 how the different BWT variants determine this tie-breaking rule.

4.3 Dynamicity of BWT variants

In this subsection, we study the dynamicity of the five BWT variants presented: what happens to the transforms if one or more of the strings are removed from (or, symmetrically, added to) the collection? Are the transforms robust to these operations? In other words, is the relative order of characters preserved when strings are added or deleted? These characteristics are particularly important for adaptive indexing structures, as they could allow efficient updates when the input changes dynamically.

We will see that all except the concatBWT are robust w.r.t. removal or addition of strings.

We recall a string S is a *subsequence* of a string T if S can be obtained from T by deleting zero or more characters without changing the order of the remaining characters. More formally, given the string $T = T[1..|T|]$, a string $S = S[1..|S|]$ is a subsequence of T, denoted $S \sqsubseteq T$, if there exists a sequence of integers $i_1, i_2, \ldots, i_{|S|}$, such that $1 \leq i_1 < i_2 < \ldots < i_{|S|} \leq |T|$ and $S[j] = T[i_j]$ for every $j = 1, \ldots, |S|$.

[3] SAP stands for "same as previous"

The following lemma follows from the fact that, for $T_d \in \mathcal{M}$, any two cyclic rotations (resp. suffixes) of string T_d preserve their relative lexicographic order when they are considered in the ω-sorted (resp. lexicographically sorted) list of all cyclic rotations (resp. suffixes) of the string collection $\mathcal{M}$.

▶ **Lemma 10.** *Given a multiset $\mathcal{M}$, for every string $T_d \in \mathcal{M}$,*

1. $\text{BWT}(T_d) \sqsubseteq \mathit{EBWT}(\mathcal{M})$,
2. $\text{BWT}(T_d\$) \sqsubseteq \mathit{dolEBWT}(\mathcal{M})$,
3. $\text{BWT}(T_d\$) \sqsubseteq \mathit{mdolEBWT}(\mathcal{M})$, *up to renaming the character* \$,
4. $\text{BWT}(T_d\$) \sqsubseteq \mathit{mdolBWT}(\mathcal{M})$, *up to renaming the character* \$,
5. $\text{BWT}(T_d\$) \sqsubseteq \mathit{concatBWT}(\mathcal{M})$, *up to renaming the character* \$.

When subsets of the string collection are considered, rather than a single string, the following proposition shows that Lemma 10 can be generalized for EBWT and dolEBWT.

▶ **Proposition 11.** *Let $\mathcal{M}$ be a multiset of strings and let $\mathcal{S} \subseteq \mathcal{M}$. Then:*

1. $\mathit{EBWT}(\mathcal{S}) \sqsubseteq \mathit{EBWT}(\mathcal{M})$, *and*
2. $\mathit{dolEBWT}(\mathcal{S}) \sqsubseteq \mathit{dolEBWT}(\mathcal{M})$.

In order to study the behavior of the input order dependent BWT variants when sub-collections of strings are considered, we use the following definition. Given an ordered collection $\mathcal{M} = (T_1, T_2, \ldots, T_m)$, the ordered collection $\mathcal{S} = (S_1, S_2, \ldots, S_p)$ is a *sub-collection* of $\mathcal{M}$ (denoted $\mathcal{S} \preceq \mathcal{M}$) if there exists a sequence of integers $i_1, i_2, \ldots, i_p$, such that $1 \leq i_1 < i_2 < \ldots < i_p \leq m$ and $S_j = T_{i_j}$ for every $j = 1, \ldots, p$.

In the following proposition, we consider mdolEBWT and mdolBWT. In both of these transformations, the relative order of two identical suffixes is determined by the order of the distinct dollar symbols. Therefore, if the sub-collection of strings maintains the same relative order, then the output of each transformation applied to the sub-collection is a subsequence of the output obtained with the entire string collection.

▶ **Proposition 12.** *Let $\mathcal{M}$ be an ordered string collection and let $\mathcal{S} \preceq \mathcal{M}$. Then:*

1. $\mathit{mdolEBWT}(\mathcal{S}) \sqsubseteq \mathit{mdolEBWT}(\mathcal{M})$, *up to renaming the dollar symbols used for the strings in $\mathcal{S}$, and*
2. $\mathit{mdolBWT}(\mathcal{S}) \sqsubseteq \mathit{mdolBWT}(\mathcal{M})$, *up to renaming the dollar symbols used for the strings in $\mathcal{S}$.*

The previous two propositions show that all the considered BWT-variants can be used as basis of dynamic compressed indexing structures, since they allow the removal of a string from the collection simply by deleting the characters of the string from the output of the transformation on the full string collection. Similarly, they enable the use of the merge of the transforms on sub-collections to obtain the output of a collection containing all the strings.

The following example shows that, on the other hand, the BWT-variant we refer to as concatBWT, i.e., concatenating the input strings using the same dollar-symbol to separate them, does not satisfy this property.

▶ **Example 13.** Let us consider the collection $\mathcal{M} = (\texttt{CCA}, \texttt{ACA}, \texttt{TCA})$. Then $\text{concatBWT}(\mathcal{M}) =$ \$AAACCC\$TCA#\$. When the sub-collection $\mathcal{S} = (\texttt{CCA}, \texttt{ACA})$ is considered, then $\text{concatBWT}(\mathcal{S})$ = \$AACC\$AC#, which is not a subsequence of $\text{concatBWT}(\mathcal{M})$.

4.4 Input order and output order

It is clear that the first m characters of a separator-based BWT variant constitute an SAP-interval, namely the one corresponding to $U = \epsilon$. Since this m-length prefix contains every string T_d exactly once, it determines a permutation of the input strings. This is the permutation of the indices $1 \leq d \leq m$ found in the m first entries of GCA, i.e., in GCA$[1..m]$.[4] This permutation then determines the order in which the respective suffixes appear in the interesting intervals, which, by Proposition 8, determines the final transform.

As we have a multiset in input, the relative order of equal strings $T_d = T_{d'}$, where $d \neq d'$, does not matter. We can thus use a meta-string to model the input and output permutations of $\mathcal{M}$, as follows. Let $m' \leq m$ denote the number of distinct strings in $\mathcal{M}$. We define a meta-string t of length m, with characters from an ordered alphabet of size m', to model the *input permutation* of $\mathcal{M} = (T_1, T_2, \ldots, T_m)$, by replacing every string T_d by a meta-character according to its lexicographic rank among all distinct input strings. For example, for $\mathcal{M} = (\texttt{ACA}, \texttt{TGA}, \texttt{ACA}, \texttt{GAA}, \texttt{TGA}, \texttt{TGA})$, we get $t = \texttt{acabcc}$. Now the m-length prefix of the BWT, corresponding to the *output permutation*, can be seen as another meta-string, which is, in fact, a permutation of t. As this string depends on the BWT variant employed, each of these variants can be viewed as a string transform, acting on the input meta-string t. We will denote these transforms as $\pi_{dolE}, \pi_{mdolE}, \pi_{mdol}$, and π_{concat}, respectively.

▶ **Example 14.** Let $\mathcal{M} = (\texttt{ACA}, \texttt{TGA}, \texttt{ACA}, \texttt{GAA}, \texttt{TGA}, \texttt{TGA})$. Then $t = \texttt{acabcc}$ and $\pi_{mdolE}(t) = \pi_{mdol}(t) = \texttt{acabcc}$, $\pi_{dolE}(t) = \texttt{aabccc}$, and $\pi_{conc}(t) = \texttt{ccacab}$.

Figure 2 The first m rotations or suffixes and the output permutations for all separator-based BWTs of $\mathcal{M} = (\texttt{ACA}, \texttt{TGA}, \texttt{ACA}, \texttt{GAA}, \texttt{TGA}, \texttt{TGA})$.

mdolEBWT	π_{mdolE}	mdolBWT	π_{mdol}	dolEBWT	π_{dolE}	concatBWT	π_{conc}
$\$_1$ACA	a	$\$_1TGA\$_2$ACA..	a	$ACA	a	$#	c
$\$_2$TGA	c	$\$_2ACA\$_3$GAA..	c	$ACA	a	ACAGAA$..	c
$\$_3$ACA	a	$\$_3GAA\$_4$TGA..	a	$GAA	b	GAATGA$..	a
$\$_4$GAA	b	$\$_4TGA\$_5$TGA..	b	$TGA	c	TGA#	c
$\$_5$TGA	c	$\$_5TGA\$_6$	c	$TGA	c	TGAACA$..	a
$\$_6$TGA	c	$\$_6$	c	$TGA	c	TGATGA$#	b

It is easy to see that both $\pi_{mdolE}(t)$ and $\pi_{mdol}(t)$ are equal to t, since the dollar-symbols are ordered according to the input order. For the dolEBWT, the rank of $\$T_d$ equals the lexicographic rank of T_d among all input strings (Lemma 6), i.e., $\pi_{dolE}(t)$ is the lexicographically sorted permutation of the input string t. The situation is more complex for the concatBWT. Since # is the smallest character and it is in the final position of the concatenated string $T_1\$T_2\$\cdots T_m\$\#$, the last string T_m of the input will be the first, while for the other strings T_d, the lexicographic rank *of the following string* T_{d+1} decides the order. Moreover, if T_d and $T_{d'}$ are followed by two equal strings $T_{d+1} = T_{d'+1}$, then the lexicographic order of the strings following the following strings will break the tie, and so on. In other words, π_{concat} is essentially the BWT of t. (We thank Massimiliano Rossi for this insight.)

[4] Or, equivalently, in the document array DA.

▶ **Lemma 15.** *Let t be the input order meta-string of $\mathcal{M} = (T_1, \ldots, T_m)$. Then $\pi_{concat}(t) =$ BWT$^*(t)$, where for a string u, BWT$^*(u)$ is defined as* BWT$(u\$)$ *from which the end-of-string symbol* `$` *has been removed.*

▶ **Example 16.** Continuing Example 14, BWT$(t\$)$ = BWT(`acabcc$`) = `cc$acab`, and therefore, BWT$^*(t)$ = `ccacab`.

Lemma 15 has important consequences for the BWT variant concatBWT. First, it means that in most cases, the transforms mdolBWT and concatBWT will produce different outputs, given the same input. This is because $\pi_{concat}(t)[1] = t[m] = \pi_{mdol}(t)[m]$, and therefore, if the strings in $\mathcal{M}$ are all distinct, then for any input order, the output permutations induced by mdolBWT and concatBWT are distinct: in this case, $t[1] \neq t[m]$, and thus $\pi_{mdol}(t) \neq \pi_{concat}(t)$ holds for all t. This means that, in whatever order the strings are given, on most string sets the resulting transforms mdolBWT and concatBWT will differ.

Another consequence is that concatBWT cannot produce all possible transforms. Already for $m = 3$, on an input collection of three distinct strings, the mapping corresponding to concatBWT is not surjective, since the mapping π_{concat} for $m' = m = 3$ is as follows: `abc` $\mapsto$ `cab`, `acb` $\mapsto$ `bca`, `cab` $\mapsto$ `bca`, `bac` $\mapsto$ `cba`, `bca` $\mapsto$ `acb`, and `cba` $\mapsto$ `abc`. In particular, no t maps to `bac`.

It has been shown experimentally that more than half of binary and ternary strings of length between 10 and 20 do not lie in the image of the function BWT*, with the percentage of those not in the image increasing with increasing length [59]. These results seem to indicate that, in fact, the majority of multi-permutations cannot be produced by concatBWT (for a concrete example, see Section 5).

On the other hand, it follows from Lemmas 6 and 15 that every separator-based transform can be simulated by both mdolBWT and mdolEBWT, in the sense that there exists an input order of $\mathcal{M}$ such that the resulting mdolBWT and mdolEBWT will be equal to the transform, up to possibly renaming dollars. We summarize:

▶ **Theorem 17.** *Given a string collection $\mathcal{M} = (T_1, T_2, \ldots, T_m)$ and L the output of a separator-based BWT variant on $\mathcal{M}$, there exists a permutation τ of $\mathcal{M}$ such that*

$$L \mathrel{\hat{=}} mdolEBWT(T_{\tau(1)}, T_{\tau(2)}, \ldots, T_{\tau(m)}) \mathrel{\hat{=}} mdolBWT(T_{\tau(1)}, T_{\tau(2)}, \ldots, T_{\tau(m)}).$$

We will use the insight of Theorem 17 in the next section.

5 Reducing the number of runs

Recall that $L \mathrel{\hat{=}} L'$ if the L and L' are equal up to renaming the dollar symbols. In Section 4, we showed that all separator-based BWT variants can be simulated by mdolEBWT or by mdolBWT (Thm. 17): if L is the output of one of these variants, then the input collection can be permuted in such a way that mdolEBWT and mdolBWT will produce a transform which differs from L only in the denomination of the dollar signs. This justifies the following definition.

Given a string collection $\mathcal{M}$, let $\mathfrak{S}_{\mathcal{M}}$ denote the string family obtained by applying mdolEBWT to all possible orders of the strings in $\mathcal{M}$. Formally, for $\mathcal{M} = (T_1, \ldots, T_m)$,

$$\mathfrak{S}_{\mathcal{M}} = \{\text{mdolEBWT}(\{T_{\tau(d)}\$_d \mid 1 \leq d \leq m\}) \mid \tau \text{ is permutation of } \{1, \ldots, m\}\}.$$

This string family coincides with the one introduced in [14] (albeit there given with a different definition).

In this section, we focus on reducing the number of runs of the BWT, commonly referred to as r. This is a fundamental parameter, since the BWT is primarily used for compressed text indexes, such as the FM-index [48], the r-index [54], or the extended r-index [19, 20], whose storage space depends on r. Moreover, the value n/r, the average runlength of the BWT, is increasingly being used as a parameter describing the input data. This is because it is well known that the more repetitive the input data, the fewer runs the BWT tends to have.

Therefore, we are interested in the following:

▶ **Definition 18.** *Given a string collection $\mathcal{M}$, we let $r_{\text{opt}}(\mathcal{M}) = \min\{\rho(L) \mid L \in \mathfrak{S}_{\mathcal{M}}\}$, and call a string $L \in \mathfrak{S}_{\mathcal{M}}$* optimal *if $\rho(L) = r_{\text{opt}}(\mathcal{M})$. When $\mathcal{M}$ is clear from the context, we just write r_{opt} for $r_{\text{opt}}(\mathcal{M})$.*

As we saw in Section 4.2, any differences between two transforms L and L' on the same collection $\mathcal{M}$, in whatever order, necessarily lie within interesting intervals. This ignores the naming of the separators, which is justified, given that no tool outputs the transform with *different dollar-symbols*, even if it computes the mdolBWT or mdolEBWT, which means that internally it handles the dollar-symbols as different symbols.[5] Thus, for the sake of compression, it is appropriate to treat the output independently of the names of the dollar-symbols.

Bentley et al. [9] gave a linear-time algorithm for minimizing the number of runs but they gave no implementation. An implementation called `optimalBWT` [99] was presented by Cenzato et al. [30] which computes an optimal transform L. To understand how the algorithm works, let us consider an interesting interval $[b, e]$. It is clear that within this interval, the number of runs can be minimized if all occurrences of same characters are grouped together. Therefore, if there are k distinct characters in $L[b..e]$, then k is the minimum number of runs achievable in this interval. Now let's say we have grouped all characters into k runs and $L[b..e] = a_1^{r_1} a_2^{r_2} \cdots a_k^{r_k}$. If the previous character $L[b-1] \neq a_1$ then it may be possible to reduce the number of runs by swapping some other run, say $a_i^{r_i}$ with the first run $a_1^{r_1}$, namely if $L[b-1] = a_i$. Similarly, one may be able to reduce the number of runs by swapping some other run with the last one $a_k^{r_k}$. The problem is that this choice may not be unique, and when there are consecutive interesting intervals then it may not be possible to decide until these interesting intervals end. Bentley et al. give an algorithm for making these choices, modeling the problem as what they refer to as a *tuple ordering problem*, which can be turned into a shortest path problem on a DAG and solved optimally in linear time.

Now consider our toy example $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$. In Table 5, we give different transforms from $\mathfrak{S}_{\mathcal{M}}$, with the number of runs between 19 (top row, input order as given above), and 14 (bottom row), which is the minimum r_{opt}. In the middle rows we give two heuristics, which we explain next.

Recall that the colex order (also referred to as *reverse lexicographic order (rlo)*) is defined as $S <_{\text{colex}} T$ if $S^{\text{rev}} <_{\text{lex}} T^{\text{rev}}$. It is easy to see that if the input strings are in colex order, then every interesting interval will have the minimum number of runs. On our example, the colex results in 18 runs, one less than the input order. In particular, several existing tools (`BCR` [6, 7], `ropeBWT2` [107, 74], `ropeBWT3` [108, 75]) have the option to output the BWT with respect to the colex order. This was already shown in [39] to significantly reduce the number of runs.

[5] Some tools are able to output the indices corresponding to the dollars, such as `BCR` and `BEETL` [6, 7, 8], but this is done in a separate file, or the indices can be extracted from the DA or the GSA, such as `eGAP` [46, 45].

Table 5 For the input collection $\mathcal{M} = (\texttt{CTGA}, \texttt{TG}, \texttt{GTCC}, \texttt{TCA}, \texttt{CGACC}, \texttt{CGA})$, we show the resulting transform for the input order, the optimal order, and two heuristics (details see text). Note that in the count of the number of runs, all dollars are considered equal.

	transform	number of runs r
mdolEBWT($\mathcal{M}$)	AGCACAGCGGCCTTA$\$_6\$_5\$_1$TTCC$\$_3\$_4G\$_2$C	19
colexBWT($\mathcal{M}$)	AAACCGCGGGCCTAT$\$_2\$_4\$_3$TCTC$\$_5\$_1G\$_6$C	18
plusBWT($\mathcal{M}$)	AAACCGGGCGCCTTA$\$_2\$_5\$_1$TTCC$\$_4\$_3G\$_6$C	15
optBWT($\mathcal{M}$)	AAAGCCCGGGCCTTA$\$_3\$_6\$_2$TTCC$\$_5\$_1G\$_4$C	14

Several other heuristics were introduced in [14], all based on the idea of grouping all occurrences in interesting intervals into one single run and possibly swapping the runs within the interesting interval. The one that performed best was plusBWT, which during the construction greedily puts as first run the character immediately before the interesting interval and as last run the character immediately after, if these characters are present. In Table 7, we report experimental results, over benchmarks that are described in Table 6, that show that both of these heuristics get very close to the optBWT. Interestingly, even randomly permuting the runs within interesting intervals results in a slight improvement over the colex order. For details, see [14, 6].

What is also remarkable is the improvement that can be achieved by the optimalBWT as compared to the input order, namely up to a multiplicative factor of over 31, on real biological data [30, 14], see Table 7. In other words, a resulting data structure built on top of the BWT would be far smaller if the optBWT was used, rather than just any BWT transform. Indeed, in [31], first experimental results on the size of the resulting r-index on real datasets are presented.

Another interesting aspect is the following: in order to reduce the number of runs, a necessary step is grouping all equal characters together within the interesting intervals. The larger the interesting intervals, the fewer the fraction of permutations of the string collection that do this. This implies that the probability that some arbitrary input order – as given by the order in which the collection comes, e.g. as downloaded from a database – will group these characters together is very small. This explains why the potential gain from applying the optBWT (or, indeed, one of the heuristics given above) is highest on very large sets of very similar short strings: many suffixes will be shared by many input strings, which produces long interesting intervals. (See [32, 33] for a more detailed analysis.)

Now consider the following example, which shows that with the concatBWT, it can be impossible to minimize the number of runs of the BWT. In other words, on some data sets, whatever the input order, concatBWT($\mathcal{M}$) will always have more runs than necessary.

▶ **Example 19.** Let $\mathcal{M} = \{\texttt{ACA}, \texttt{TGA}, \texttt{GAA}\}$, then $t = \texttt{acb}$, and the output order corresponding to the colexicographic order of the strings is `bac`. As it happens, this order yields the minimum number of runs among all strings in $\mathfrak{S}_{\mathcal{M}}$, with colexBWT($\mathcal{M}$) = optBWT($\mathcal{M}$) = `AAAACGG$AT$$`, which has 7 runs. However, as we saw in Section 4.4, the string `bac` does not lie in the image of π_{concat}, so no permutation of the strings in $\mathcal{M}$ will yield this order for concatBWT. In particular, the colexBWT($\mathcal{M}$) = `AAAACGG$AT$$` has 7 runs, while all concatBWTs have at least 8: `AAAGACG$AT$$`, `AAACGAG$AT$$`, `AAAAGCG$AT$$`, `AAAGCAG$AT$$`, `AAACAGG$AT$$`. (Here we are using the adapted variant of the concatBWT, for better comparison.)

Table 6 Features of the datasets included in our experiments. From left to right, we report the dataset code and description, the BWT length, the maximum string length, the number of sequences, and the ratio between the BWT length and the number of runs of the optBWT.

Dataset	Description	BWT length	Max length	Number of sequences	n/r_{opt}
SRR7494928–30	*Epstein Barr Virus*	3,225,480,720	101	9,648,932	79.25
ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	116.62
SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	113.71
ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	32.23
SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	50.75
SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	15.30
SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	151.62
ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	11.37
pdb_seqres	*proteins*	241,121,574	16,181	865,773	14.33

Table 7 Number of runs of the mdolEBWT (resp. mdolBWT) with sequences presented in input order (inputBWT) and in two heuristic orders, colexBWT and plusBWT, compared to the optBWT for all datasets in Table 6. In the rightmost column we report the ratio between the number of runs of the inputBWT (r_{input}) and the optBWT (r_{opt}). Results reported from [14].

Dataset	inputBWT	Heuristic string orders		optBWT	r_{input}/r_{opt}
		colexBWT	plusBWT		
SRR7494928–30	254,663,327	41,730,649	41,372,530	40,700,607	**6.26**
ERR732065–70	48,727,709	11,941,093	11,766,827	11,539,661	**4.22**
SRR12038540	209,136,502	17,026,009	15,226,766	14,864,523	**14.07**
ERR022075_1	259,821,570	75,846,202	74,529,428	71,203,469	**3.65**
SRR059298	249,873,376	50,495,777	49,619,150	48,376,632	**5.17**
SRR065389–90	2,251,887,226	968,098,124	954,489,749	921,561,895	**2.44**
SRR2990914_1	3,313,966,937	109,772,697	108,466,351	105,250,120	**31.49**
ERR1019034	23,084,021,291	11,312,737,256	11,179,873,104	10,860,229,434	**2.13**
pdb_seqres	17,971,532	16,862,960	16,848,496	16,829,629	**1.07**

6 BWT for string collections in bioinformatics

Recent advances in DNA and RNA sequencing technologies led to the explosion of the amount of *in silico* genomic data to be assembled, investigated, stored and compared. In this final section we will overview the applications in bioinformatics that benefit from the efficiency of BWT-based indexing of string collections.

Both DNA and RNA sequences can be seen as strings over an alphabet of size four: the set of nucleotides that compose them. Furthermore, as we will see below, in all the applications we will mention, sequences are highly repetitive and the different strings of the collection encode redundant data: an ideal target for BWT based tools that are specifically designed for string collections.

When the string collection consists of *raw sequencing data*, that is, a large amount of fragments that directly result from the sequencing process, the most requested tasks can be:

Assembling. Size and accuracy details have changed across the time and also change from a technology to another, but all sequencing machines are not able to process molecules of arbitrary size. As a consequence, the typical procedure is to: first, create a redundant number of copies of the sequence while still *in vitro*; second, cut (this can be done using specific enzymes) the sequences into small enough fragments randomly enough to ensure

that distinct copies are cut at different sites; third, perform the actual sequencing that turn the sequence into *in silico* data; fourth, exploiting the redundancy of the original copies and hence fragments overlap, build a layout for the fragments that enables to reconstruct the original sequence. This last task is named *Fragment Assembly*, and its efficient solution played a crucial role in the historical human genome project. For decades the problem of fragment assembly has been the *Holy Grail* of algorithmic tasks for bioinformatics, having to deal with millions of DNA fragments, each of a few hundreds of length, that had to be jointly analysed, looking for local similarities among them [111].

Mapping. When the newly sequenced genome belongs to species for which a fully assembled *reference* genome already exists, then one does not need to perform a *de novo* assembly out of the fragments, as the reference can be exploited. In this case, the strings of the collection are, in general, not as many as in the case of the assembly task, because redundancy is not required as much as there. However, a huge amount of strings must be mapped onto the reference genome, dealing with sequencing errors, repetitions inside the genome that make the mapping loci ambiguous, as well as the physiological differences among the query genome and the reference [71, 76].

Haplotype Matching. A Haplotype is a set of genetic markers or DNA variations that are inherited together from a single parent. Haplotype Matching asks for finding long matches between sequences within a given large collection of aligned genetic sequences (that are, indeed, the haplotypes). The task is relevant because these long matches are candidates to be regions that are identical by descent (IBD) from a common ancestor. Even for this task BWT based methods have been designed: the positional BWT [44, 113, 114, 110] for linear strings, and more recently the GBWT for Pangenome Graphs [109].

Assembly free (and reference free) variant calling. Variant calling is the fundamental task that has to be performed with a genome, and it consists of detecting individual traits and genetic diversity of the query genome to discover genetic based diseases, perform personalised medicine, as well as to understand evolution (when variants are annotated at population scale). When a reference genome is not available, this task is performed directly on raw data without a preliminary assembling phase, and hence this requires the investigation and comparison of (pairs of) large collections of strings [104, 102, 103, 70].

When, instead, the string collections are multiple genomes, or several RNA isoforms, or belong to possibly many different genomes, we have a (possibly large) set of strings that are as long as billions of letters. In these cases, the downstream analyses tasks are different, and they include:

Comparative genomics and variant calling. Here two or more (whole) genomes must be compared, one of them possibly being *the* reference. Like for the above mentioned tasks, criticalities arise because of the highly repetitive nature of genomes, and by the different nature of variants that one needs to detect: the so called *point mutations* or the *SNPs* that involve single letters, or the short *INDELs* that are the insertion or the deletion of a short fragment, up to the fragment level mutations: the deletions or insertion or duplication of a possibly large DNA fragment, or the translocation (that is, its position in the genome changes), or the copy number variations (fragments that are commonly present in multiple copies show a different number of them in different individuals). Moreover, when a DNA mutation event involves a whole fragment, there is in general a 50% probability that a copy is *inverted*, that is it occurs in the reversed direction. The comparison tasks are very complex and cannot be made by means of global alignments due to the fragment level mutations. Hence efficient indices of the strings are a basic toolkit for the various possible algorithmic strategies [20, 44].

Transcriptomics. With the so called new generation sequencing technologies, also *RNA-Seq* became a common task, that is to move to *in silico* also RNA the sequences. In particular, mRNA (messenger RNA) isoforms are very interesting to investigate as they are RNA fragments that encode the information that leads a coding part of the DNA (that is, a gene) to the synthesis of a protein. The transcription of this coding part of DNA into RNA, called *gene expression*, is a very interesting topic for molecular biologists and genetists because the same gene can express in different ways according to the cell it is (say, in the liver rather than in the skin), or according to external conditions (say, in a skin cell whether it is cold or it is being burned). The set of RNA isoforms that are in a certain cell under specific conditions is called the *transcriptome*, and this is a sort of a picture of gene expression. Investigating or comparing transcriptome(s) requires counting the abundance of each kind, being each of them a different concatenation of a subset of the fragments of the gene: again several strings, and again highly redundant, and hence a perfect target for a string collection index based on the BWT [119, 18, 20].

Metagenomics. Metagenomics is the (comparative) study of genetic material found in environmental samples (say, into the soil, or in sea water, or in an animal's gut), with the purpose of detecting and investigating (the distribution of) genomes of different species of microorganisms such as bacteria, viruses, and fungi. The ultimate goals of metagenomic is the study of microbial diversity, and the interactions: both mutual and with the ecosystem it belongs to. This requires various tasks that include profiling, assembling, comparing and classifying data consisting of a huge amount of fragments belonging to different individuals and different species. As such, the redundancy will be less than in the other applications; however, indexing these string collections with BWT is still fundamental to make efficient compression and analyses [67, 64, 14, 62, 63, 40].

Pangenomics. Using a single genome as *the* reference clearly introduces a bias for the discovery of genomic variations. Therefore, in the literature there came a new challenge that simultaneously deals with, and exploits, the widespread availability of sequencing data: using a *pangenome* rather than a genome. In [118], a pangenome was defined as "any collection of genomic sequences to be analyzed jointly or to be used as a reference". In contrast to a *linear* reference, a pangenome reference aims to compactly represent the variation within a population by encoding the commonalities and differences among the underlying sequences. In the literature there are many suggestions for pangenome representations [77, 4, 28, 106, 55, 100, 79, 13, 12, 92, 26, 72] whose construction may benefit from efficient indexing of data as a starting step [81, 46, 53, 89, 98, 110, 18, 109, 35]. Some of these representations support efficient and accurate analyses tools such as comparison [1, 2, 51, 52, 50] or alignments [90, 11, 91, 10, 3, 36], that could be sped up with string collections indices. However, by now none of these graph structures has been acknowledged as the standard yet. The computational challenges posed by pangenomes often result in a trade-off between the efficiency and accuracy of the methods and the information content of the chosen representation. In this view, a valid option for certain tasks turns out to be to just index and investigate the collection of genomes, disregarding loci and alignment information and rather choosing speed and low memory usage [21, 68, 22, 44].

7 Conclusion

In this paper, we studied different methods for constructing the BWT of string collections (as opposed to individual strings). We showed that the methods in use are non-equivalent, in the sense that they produce different transforms, which differ not only in the denomination

of separator-symbols ("dollars"). We analyzed precisely where and how these differences occur ("interesting intervals"). The differences between the transforms extend to the number of runs r of the BWT, a central parameter in compressed text indexing.

We showed that several of these methods are input-order dependent, and that this can be exploited to significantly reduce the number of runs: by up to a factor of over 31, on real biological data. This is the multiplicative gain of the optBWT as compared to computing the transform on the collection in the order it comes in. We also saw that several simple heuristics come close to this optimum.

On the downside, we saw that one of the most common methods, which we refer to as concatBWT (concatenating the input strings and separating them with the same dollar), has several undesirable properties. First, it cannot produce all possible transforms, and can therefore in general not be used as a method for run minimization. Second, it is not robust with respect to addition or removal of strings from the collection (dynamicity). Finally, it is input-order dependent but in a fairly opaque manner (its order being in essence the BWT of the input order), which makes it difficult to control the output in any way.

Based on our findings, we make the following suggestions:

1. We advise to exercise care when using the method we term concatBWT, due to the issues listed above. It is a good method because it is simple and efficient, but the user (or programmer) should be aware of its shortcomings.
2. We believe that the definition of the number of runs of a string collection should be standardized to r_{opt}, the minimum number of runs among all possible separator-based transforms. This number is also fairly easy to approximate with heuristics, the simplest of which is colexBWT.

References

1 Mai Alzamel, Lorraine A.K.Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.

2 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate string comparison and applications. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 21:1–21:14, 2018. doi:10.4230/LIPICS.WABI.2018.21.

3 Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster online elastic degenerate string matching. In *29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 105 of *LIPIcs*, pages 9:1–9:10, 2018. doi:10.4230/LIPICS.CPM.2018.9.

4 Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.

5 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and indexing the bijective and extended Burrows-Wheeler transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.

6 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.

7 BCR_LCP_GSA. URL: https://github.com/giovannarosone/BCR_LCP_GSA.

8 BEETL. URL: https://github.com/BEETL/BEETL.

9 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the Complexity of BWT-Runs Minimization via Alphabet Reordering. In *28th Annual European Symposium on*

Algorithms (ESA), volume 173 of *LIPIcs*, pages 15:1–15:13, 2020. doi:10.4230/LIPIcs.ESA.2020.15.

10 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *46th International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 132 of *LIPIcs*, pages 21:1–21:15, 2019. doi:10.4230/LIPICS.ICALP.2019.21.

11 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20M1368033.

12 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In *24th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. doi:10.1007/978-3-319-67428-5_7.

13 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/J.TCS.2019.08.012.

14 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 *35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 296 of *LIPIcs*, pages 7:1–7:15, 2024. doi:10.4230/LIPICS.CPM.2024.7.

15 BigBWT. URL: https://github.com/alshai/Big-BWT.

16 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theor. Comput. Sci.*, 862:42–58, 2021. doi:10.1016/j.tcs.2020.11.041.

17 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Multithread multistring Burrows-Wheeler Transform and Longest Common Prefix array. *J. Comput. Biol.*, 26(9):948–961, 2019. doi:10.1089/cmb.2018.0230.

18 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the Original eBWT Faster, Simpler, and with Less Memory. In *28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.

19 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. In *28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2021. doi:10.1007/978-3-030-86692-1_1.

20 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.

21 Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: streamed matching statistics with multi-genome references. In *31st Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021. doi:10.1109/DCC50243.2021.00027.

22 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.

23 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

24 bwt-lcp-parallel. URL: https://github.com/AlgoLab/bwt-lcp-parallel.

25 bwt-lcp-parallel. URL: https://github.com/AlgoLab/bwt-lcp-em/.

26 Thomas Büchler and Enno Ohlebusch. An improved encoding of genetic variation in a Burrows–Wheeler transform. *Bioinform.*, 36(5):1413–1419, 2019. doi:10.1093/bioinformatics/btz782.

27 `cais`. URL: https://github.com/davidecenzato/cais.

28 Vincenzo Carletti, Pasquale Foggia, Erik Garrison, Luca Greco, Pierluigi Ritrovato, and Mario Vento. Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In *12th IAPR-TC-15 International Workshop Graph-Based Representations in Pattern Recognition (GbRPR)*, pages 237–246, 2019. doi:10.1007/978-3-030-20081-7_23.

29 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 128 of *LIPIcs*, pages 24:1–24:20, 2019. doi:10.4230/LIPIcs.CPM.2019.24.

30 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Data Compression Conference (DCC)*, pages 71–80. IEEE, 2023. doi:10.1109/DCC55655.2023.00015.

31 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT and r-index of very large read collections. Submitted, 2025.

32 Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In *33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223 of *LIPIcs*, pages 25:1–25:18, 2022. doi:10.4230/LIPICS.CPM.2022.25.

33 Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinform.*, page btae333, 2024. doi:10.1093/bioinformatics/btae333.

34 Ananyo Choudhury, Shaun Aron, and Laura R. Botigué et al. High-depth African genomes inform human migration and health. *Nature*, 586:741–748, 2020. doi:10.1038/s41586-020-2859-7.

35 Lapo Cioni, Veronica Guerrini, and Giovanna Rosone. The Burrows-Wheeler Transform of an Elastic-Degenerate String. In *25th Italian Conference on Theoretical Computer Science (ICTCS)*, volume 3811 of *CEUR Workshop Proceedings*, pages 66–80, 2024. URL: https://ceur-ws.org/Vol-3811/paper240.pdf.

36 Aleksander Cislak, Szymon Grabowski, and Jan Holub. SOPanG: online text searching over a pan-genome. *Bioinform.*, 34(24):4290–4292, 2018. doi:10.1093/BIOINFORMATICS/BTY506.

37 `CMS-BWT`. URL: https://github.com/fmasillo/CMS-BWT.

38 The COVID-19 Genomics UK (COG-UK) consortium. An integrated national scale SARS-CoV-2 genomic surveillance network. *The Lancet Microbe*, 1(3):e99–e100, 2020. doi:10.1016/S2666-5247(20)30054-9.

39 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012. doi:10.1093/bioinformatics/bts173.

40 Kim Daehwan, Li Song, Florian P.Breitwieser, and Steven L. Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome Research*, 26(12):1721–1729, 2016. doi:10.1101/gr.210641.116.

41 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the extended BWT from grammar-compressed DNA sequencing reads. *CoRR*, abs/2102.03961, 2021. doi:10.48550/arXiv.2102.03961.

42 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Inf. Comput.*, 294:105088, 2023. doi:10.1016/j.ic.2023.105088.

43 Zachary D.Stephens, Skylar Y.Lee, Faraz Faghri, Roy H.Campbell, Chengxiang Zhai, and Miles J.Efron et al. Big data: Astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015. doi:10.1371/journal.pbio.1002195.

44 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinform.*, 30(9):1266–1272, 2014. doi:10.1093/BIOINFORMATICS/BTU014.

45 eGAP. URL: https://github.com/felipelouza/egap.

46 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6:1–6:15, 2019. doi:10.1186/S13015-019-0140-0.

47 eGSA. URL: https://github.com/felipelouza/egsa.

48 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.

49 G2BWT. URL: https://bitbucket.org/DiegoDiazDominguez/lms_grammar/src/bwt_imp2.

50 Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string comparison. *Inf. Comput.*, 304:105296, May 2025. doi:10.1016/j.ic.2025.105296.

51 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 259 of *LIPIcs*, pages 11:1–11:20, 2023. doi:10.4230/LIPICS.CPM.2023.11.

52 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Pangenome Comparison via ED Strings. *Frontiers in Bioinformatics*, 4:11:1–11:20, 2024. doi:10.3389/fbinf.2024.1397036.

53 Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *21st International Conference on Algorithms in Bioinformatics (WABI)*, volume 201 of *LIPIcs*, pages 13:1–13:21, 2021. doi:10.4230/LIPICS.WABI.2021.13.

54 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1), 2020. doi:10.1145/3375890.

55 Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, David G. Ashbrook, Kaisa Thorell, Rachel L. Rusholme-Pilcher, Gianni Liti, Emilio Rudbeck, Agnieszka A. Golicz, Sven Nahnsen, Zuyu Yang, Moses Njagi Mwaniki, Franklin L. Nobrega, Yi Wu, Hao Chen, Joep de Ligt, Peter H. Sudmant, Sanwen Huang, Detlef Weigel, Nicole Soranzo, Vincenza Colonna, Robert W. Williams, and Pjotr Prins. Building pangenome graphs. *Nature Methods*, 21:2008–2012, 2024. doi:10.1038/s41592-024-02430-3.

56 Genome 10K Community of Scientists. A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, 100:659-674, 2009. doi:10.1093/jhered/esp086.

57 Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory A*, 64(2):189–215, 1993. doi:10.1016/0097-3165(93)90095-P.

58 Raffaele Giancarlo, Antonio Restivo, and Marinella Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387:236–248, 2007. doi:10.1016/J.TCS.2007.07.019.

59 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello. Novel results on the number of runs of the Burrows-Wheeler-Transform. In *47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 12607 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2021. doi:10.1007/978-3-030-67731-2_18.

60 grlBWT. URL: https://github.com/ddiazdom/grlBWT.

61 gsufsort. URL: https://github.com/felipelouza/gsufsort.

62 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. doi:10.1186/S13015-023-00232-4.

63 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinform.*, 21-S(8):299, 2020. doi:10.1186/S12859-020-03628-W.

64 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *Biomedical Engineering Systems and Technologies*, pages 97–120, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-38854-5_6.

65 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

66 James Holt and Leonard McMillan. Merging of multi-string bwts with applications. *Bioinform.*, 30(24):3524–3531, 2014. doi:10.1093/bioinformatics/btu584.

67 Lilian Janin, Giovanna Rosone, and Anthony J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinform.*, 30(1):24–30, 2014. doi:10.1093/BIOINFORMATICS/BTT257.

68 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient Construction of a Complete Index for Pan-Genomics Read Alignment. *J. Comput. Biol.*, 27(4):500–513, 2020. doi:10.1089/CMB.2019.0309.

69 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.

70 Ben Langmead, Michael C. Schatz, Jimmy Lin, Mihai Pop, and Steven L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10:R134, 2009. doi:10.1186/gb-2009-10-11-r134.

71 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:R25, 2009. doi:10.1186/gb-2009-10-3-r25.

72 Brice Letcher, Martin Hunt, and Zamin Iqbal. Gramtools enables multiscale variation analysis with genome graphs. *Genome Biology*, 22, 2021. doi:10.1186/s13059-021-02474-0.

73 lFGSACA. URL: https://gitlab.com/qwerzuiop/lfgsaca.

74 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014. doi:10.1093/BIOINFORMATICS/BTU541.

75 Heng Li. BWT construction and search at the terabase scale. *Bioinform.*, 40(12):btae717, 2024. doi:10.1093/bioinformatics/btae717.

76 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.

77 Heng Li, Xiaowen Feng, and Chong Chu. The Design and Construction of Reference Pangenome Graphs with Minigraph. *Genome Biology*, 21(265), 2020. doi:10.1186/s13059-020-02168-z.

78 Ruiqiang Li et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272, 2010. doi:10.1101/gr.097261.109.

79 Wen-Wei Liao et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.

80 Bo Liu, Dixian Zhu, and Yadong Wang. deBWT: parallel construction of Burrows-Wheeler Transform for large collection of genomes with de Bruijn-branch encoding. *Bioinform.*, 32(12):174–182, 2016. doi:10.1093/BIOINFORMATICS/BTW266.

81 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15, 2020. doi:10.1186/s13015-020-00177-y.

82 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26:1–26:16, 2017. doi:10.1186/s13015-017-0117-9.

83 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J of Comput.*, 12(1):40–66, 2005.

84 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

85 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.

86 Francesco Masillo. Matching statistics speed up BWT construction. In *31st Annual European Symposium on Algorithms (ESA)*, volume 274 of *LIPIcs*, pages 83:1–83:15, 2023. doi:10.4230/LIPICS.ESA.2023.83.

87 `Merge-BWT`. URL: https://github.com/jltsiren/bwt-merge.

88 `msbwt`. URL: https://github.com/holtjma/msbwt.

89 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching Reads to Many Genomes with the r-Index. *J. Comput. Biol.*, 27(4):514–518, 2020. doi:10.1089/CMB.2019.0316.

90 Njagi Moses Mwaniki, Erik Garrison, and Nadia Pisanti. Fast exact string to d-texts alignments. In *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC)*, pages 70–79. SCITEPRESS, 2023. doi:10.5220/0011666900003414.

91 Njagi Moses Mwaniki and Nadia Pisanti. Optimal sequence alignment to ED-strings. In *18th International Symposium on Bioinformatics Research and Applications (ISBRA)*, volume 13760 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2022. doi:10.1007/978-3-031-23198-8_19.

92 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment with gaps. *Theor. Comput. Sci.*, 710:148–157, 2018. doi:10.1016/j.tcs.2017.02.020.

93 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

94 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.

95 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. URL: http://www.oldenbusch-verlag.de/.

96 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. Generic non-recursive suffix array construction. *ACM Trans. Algorithms*, 20(2), 2024. doi:10.1145/3641854.

97 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big BWTs. In *33rd Data Compression Conference (DCC)*, pages 62–70, 2023. doi:10.1109/DCC55655.2023.00014.

98 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *31st Data Compression Conference (DCC)*, pages 203–212. IEEE, 2021. doi:10.1109/DCC50243.2021.00028.

99 `optimalBWT`. URL: https://github.com/davidecenzato/optimalBWT.

100 Benedict Paten, Adam M. Novak, Jordan M. Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665–676, 2017. doi:10.1101/gr.214155.116.

101 `PFP-eBWT`. URL: https://github.com/davidecenzato/PFP-eBWT.

102 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Detecting Mutations by eBWT. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 3:1–3:15, 2018. doi:10.4230/LIPICS.WABI.2018.3.

103 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. SNPs detection by eBWT positional clustering. *Algorithms Mol. Biol.*, 14(1):3:1–3:13, 2019. doi:10.1186/S13015-019-0137-8.

104 Nicola Prezza, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler Transform. *BMC Bioinform.*, 21-S(8):260, 2020. doi:10.1186/S12859-020-03586-3.

105 `r-pfbwt`. URL: https://github.com/marco-oliva/r-pfbwt.

106 Goran Rakocevic et al. Fast and accurate genomic analyses using genome graphs. *Nature Genetics*, 51:354–362, 2019. doi:10.1038/s41588-018-0316-4.

107 ropeBWT2. URL: https://github.com/lh3/ropebwt2.

108 ropeBWT3. URL: https://github.com/lh3/ropebwt3.

109 Ahsan Sanaullah, Seba Villalobos, Degui Zhi, and Shaojie Zhang. Haplotype matching with GBWT for pangenome graphs. *bioRxiv*, 2025. doi:10.1101/2025.02.03.634410.

110 Pramesh Shakya, Ardalan Naseri, Degui Zhi, and Shaojie Zhang. mcPBWT: Space-Efficient Multi-column PBWT Scanning Algorithm for Composite Haplotype Matching. In *11th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, volume 13254 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2021. doi:10.1007/978-3-031-17531-2_10.

111 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinform.*, 26(12):367–373, 2010. doi:10.1093/BIOINFORMATICS/BTQ217.

112 Jouni Sirén. Burrows-Wheeler Transform for terabases. In *26th Data Compression Conference (DCC)*, pages 211–220, 2016. doi:10.1109/DCC.2016.17.

113 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 4:1–4:13, 2018. doi:10.4230/LIPICS.WABI.2018.4.

114 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinform.*, 36(2):400–407, 2020. doi:10.1093/BIOINFORMATICS/BTZ575.

115 Chen Sun et al. RPAN: rice pan-genome browser for 3000 rice genomes. *Nucleic Acids Res*, 45(2):597–605, 2017. doi:10.1093/nar/gkw958.

116 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015. doi:10.1038/nature15393.

117 The 1001 Genomes Consortium. Epigenomic Diversity in a Global Collection of Arabidopsis thaliana Accessions. *Cell*, 166(2):492–505, 2016. doi:10.1016/j.cell.2016.06.044.

118 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings Bioinform.*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.

119 Cole Trapnell, Lior Pachter, and Steven L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinform.*, 25(9):1105–1111, 2009. doi:10.1093/BIOINFORMATICS/BTP120.

120 Clare Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the NHS. *Br. Med. J.*, 361, 2018. doi:10.1136/bmj.k1687.