Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

Building genomic data structures from compressed representations using prefix-free parsing

Rahul Varki and Christina Boucher

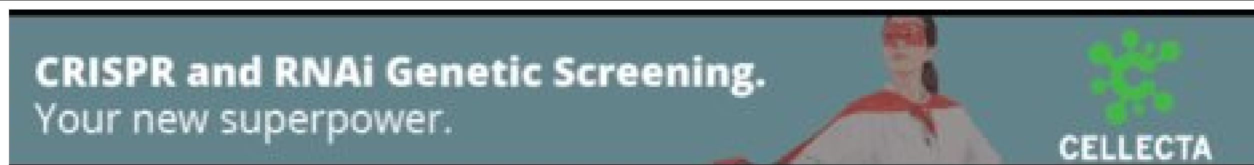
Genome Res. published online May 15, 2026
Access the most recent version at doi:10.1101/gr.281245.125

P<P Published online May 15, 2026 in advance of the print journal.

Open Access Freely available online through the *Genome Research* Open Access option.

Creative Commons License This article, published in *Genome Research*, is available under a Creative Commons License (Attribution 4.0 International), as described at http://creativecommons.org/licenses/by/4.0/.

Email Alerting Service Receive free email alerts when new articles cite this article - sign up in the box at the top right corner of the article or **click here.**



To subscribe to *Genome Research* go to:
https://genome.cshlp.org/subscriptions

© 2026 Varki and Boucher; Published by Cold Spring Harbor Laboratory Press

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

Review

Building genomic data structures from compressed representations using prefix-free parsing

Rahul Varki and Christina Boucher

Department of Computer and Information Science and Engineering, Herbert Wertheim College of Engineering, University of Florida, Gainesville, Florida 32611, USA

Advances in high-throughput sequencing have lowered the cost and complexity of genome sequencing, making it possible for the first time to assemble large pangenomic data sets for many species. These data sets, comprising thousands of individuals, already span from hundreds of gigabytes to petabytes, far exceeding the memory capacity of most machines, and are expected to continue growing in scale over time. Already, many traditional bioinformatics tools fail on inputs at this scale because they cannot construct their necessary data structures within memory limits. There is a growing need for methods that can construct these structures directly from compressed representations. Prefix-free parsing (PFP) addresses this challenge. PFP serves as a preprocessing step that compresses sufficiently repetitive text, yet still permits building important data structures for the original data set from its compressed output. This survey offers an overview of PFP, covering its core principles, the primary data structures it enables, current applications, and future research directions.

Before the advent of high-throughput sequencing, obtaining the first complete genome of a species was a costly and time-consuming endeavor. However, advances in sequencing technology have reduced the cost and resources required, enabling the sequencing of thousands of individuals from the same species. This technological leap has led to the generation of large pangenomic data sets across various species. For instance, the 1000 Genomes Project produced over 6 terabytes of assembled genomes (Clarke et al. 2012; The 1000 Genomes Project Consortium 2015), whereas the 100,000 Genomes Project (Turnbull et al. 2018), which focused on sequencing individuals with rare diseases, generated a staggering 21 petabytes worth of data (Weka 2020). The scope of these efforts has extended beyond humans to other species, as seen in the 1001 Genomes Project, which sequenced over a thousand cultivars of the plant *Arabidopsis thaliana* (Weigel and Mott 2009). As a result, the concept of a reference genome has evolved, shifting from a single representative genome per species to a collection of genomes—a pangenome—that provides a more comprehensive and accurate view of the species' genetic diversity. This shift underscores the need for advanced techniques to efficiently store and process these vast data sets, which has become crucial as pangenomes continue to expand in size and complexity.

Meeting these demands has proven difficult with existing technologies, as most traditional bioinformatics tools cannot scale to handle the volume of pangenomic data. These tools were originally developed for static single-reference genomes, which were the standard at the time. As a result, the underlying methodologies of these tools were not meant to scale to the size of the rapidly growing pangenomic data sets we see today. The lack of scalable, efficient algorithms and data structures remains a critical bottleneck in genomic research.

Among emerging methods, prefix-free parsing (PFP) stands out as a promising algorithm for efficient processing of increasingly large pangenomic data sets. Introduced in Boucher et al. (2019), PFP at a high-level is a compression algorithm that works by segmenting a genomic sequence into overlapping phrases, which can then be used to build data structures in a space-efficient manner. It achieves compression in practice by identifying identical phrases both within and between sequences, exploiting the repetitive nature of pangenomic data sets. Algorithms for building useful data structures from PFP's output have been developed and continue to evolve. Of particular interest are the PFP-based construction algorithms for the data structures that comprise the FM-index, namely the Burrows–Wheeler transform (BWT) (Burrows and Wheeler 1994) and the suffix array (SA) (Manber and Myers 1993). The FM-index underlies the widely used traditional read aligners BWA (Li 2013) and Bowtie (Langmead et al. 2009; Langmead and Salzberg 2012), but the BWT and SA are also of independent interest. Additionally, the longest common prefix (LCP) array (Manber and Myers 1993) is of interest as it is often built alongside these other data structures as an auxiliary structure. These data structures can be built in space proportional to the output of PFP which in practice is typically smaller than the size of the input when the data are sufficiently repetitive. Although PFP is a relatively recent development, it has already been applied extensively across many applications (see the PFP applications section). PFP has allowed these applications to process some of the largest pangenomic data sets currently available, such as those from the 1000 Genomes Project and similar efforts.

In this survey, we provide a comprehensive review of PFP, focusing on its conceptual foundation, key construction algorithms for important data structures, applications, and future directions. We begin by explaining the core principles of PFP, particularly its use in compressing and indexing repetitive genomic data. We then discuss the BWT, SA, and LCP, along with their corresponding PFP-based construction algorithms. This is followed by a brief overview of other PFP applications, and concludes with potential directions for future PFP-based research. As the size and complexity of genomic data sets continue to grow, methods like PFP offer promising approaches for managing these challenges. Although there is still room for optimization and broader application, PFP provides a solid framework for efficient data processing in genomics. Future

Corresponding author: rvarki@ufl.edu
Article published online before print. Article and publication date are at https://www.genome.org/cgi/doi/10.1101/gr.281245.125. Freely available online through the *Genome Research* Open Access option.

© 2026 Varki and Boucher This article, published in *Genome Research*, is available under a Creative Commons License (Attribution 4.0 International), as described at http://creativecommons.org/licenses/by/4.0/.

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

innovations in this area will likely further enhance our ability to analyze and interpret pangenomic data, contributing to advances in understanding genetic diversity and disease.

Overview of prefix-free parsing

In this section, we provide a detailed overview of prefix-free parsing (PFP), which was originally motivated by rsync (Tridgell and Mackerras 1996). Before delving into the specifics, we first present the necessary preliminaries.

Basic definitions

In this paper, we use 0-based indexing for all strings and arrays. The := symbol denotes an assignment or update operation, the = symbol represents mathematical equality or definition, and the $\cdots$ symbol indicates a range of values. Let $T = T[0 \cdots n-1] = T[0] \cdots T[n-1]$ be a finite string over an alphabet Σ of size σ whose symbols can be unambiguously ordered. The symbol $T[i]$ denotes the character at index i of T, and $T[i \cdots j]$ denotes the substring $T[i] \cdots T[j]$ if $0 \le i \le j < n$ or the empty string ϵ otherwise. For $0 \le i < n$, the substring $T[0 \cdots i]$ is the $(i+1)$th prefix of T, and $T[i \cdots n-1]$ is the $(n-i)$th suffix of T. A prefix $T[0 \cdots i]$ is a *proper prefix* if $0 \le i < n-1$, whereas a suffix $T[i \cdots n-1]$ is a *proper suffix* if $0 < i \le n-1$. If $T_2[0 \cdots m-1]$ is lexicographically smaller than $T_1[0 \cdots n-1]$, then either T_2 is a proper prefix of T_1, or there exists an index $0 \le i < \min(n,m)$ such that $T_2[0 \cdots i-1] = T_1[0 \cdots i-1]$ and $T_2[i] < T_1[i]$. Let $\text{rank}_c(T, i)$ be a function that returns the number of times c appears in $T[0 \cdots i-1]$, where $c \in \Sigma$. Let $\text{RMQ}_A(i, j)$ denote a range minimum query (RMQ) that returns the minimum value in the subarray $A[i \cdots j]$, for $0 \le i \le j < |A|$, where $|A|$ is the size of A.

The concept of prefix-free is crucial for understanding how the output of prefix-free parsing can be used to build other data structures, so a formal definition is provided.

Definition 1: ***A multiset of strings $\mathcal{S}$ is* prefix-free *if no string in $\mathcal{S}$ is a proper prefix of another string in $\mathcal{S}$.***

For example, the multiset $\mathcal{S}$ = {ACCA, GCC, ACCA, TCCAG} is a prefix-free multiset because no string $s \in \mathcal{S}$ is a proper prefix of any other string $s' \in \mathcal{S}$, as the first character of all *unique* strings is lexicographically different.

Prefix-free parsing

Prefix-free parsing (PFP) is a lossless preprocessing compression algorithm designed for large, repetitive texts such as genomic data sets (Boucher et al. 2019). It achieves effective compression by decomposing the text into variable-length overlapping phrases that often occur. The main advantage of the algorithm is that it enables the scalable construction of other data structures from its compressed output. To understand how the algorithm works in practice, we now turn to its details.

PFP(T, w, p) takes as input a text T of length n and two user-defined integers greater than 1, denoted w and p. It first transforms T into T' where $T' := \$T\#^w$ by prepending a \$ and appending w consecutive copies of #. Both symbols are arbitrary, provided that they are lexicographically smaller than every symbol in Σ. A rolling hash is then computed over all consecutive w-length substrings of T' in a single pass by sliding a window of length w and hashing each window. Any hash function may be used, provided collisions are rare; in practice, a Karp-Rabin hash (Karp and Rabin 1987) is preferred, as it allows for efficient updates between adjacent windows. This hash is denoted as $H_p(t_w) = H(t_w) \bmod p$, where H is the hash function, p is the modulus, and t_w is the current substring within the window. Substrings whose hash value is zero are referred to as trigger strings. The following definition succinctly summarizes the set of w-length substrings that yield trigger strings.

Definition 2: ***Let E be the set of all w-length substrings t_w of T such that $H_p(t_w) \equiv 0$. We refer to this set of strings as* trigger strings. *In addition, treat $t_w = \#^w$ as a trigger string, and also consider \$ as a trigger string even though it is not of length w. We use this set to parse T' into overlapping phrases.***

Trigger strings identified by the algorithm mark phrase boundaries. As the algorithm scans T', each trigger string signals the end of a phrase (besides the starting \$). Each phrase begins at the start of the previous trigger string and ends at the end of the current one; it cannot span any other trigger string. The prepended \$ and appended $\#^w$ characters are always treated as trigger strings, ensuring that the first and last phrase begin and end with a trigger string.

PFP(T, w, p) outputs a dictionary D_T and parse P_T. The dictionary stores only the unique phrases encountered, while the parse records the order that those phrases occur in the text. During parsing, when a trigger string is identified, the algorithm adds the newly terminated phrase to D_T if it is not already present, but always appends a reference to the corresponding dictionary entry to P_T. After parsing, PFP lexicographically sorts the phrases in D_T and updates the references in P_T with the corresponding dictionary integer ranks before writing the files to disk. The combined output size depends on the repetitiveness of the text and the selection of w and p. Lucà et al. (2025) extensively tested PFP with various configurations of w and p on different biological data sets and found that, for most data sets, fixing w while increasing p leads to decreased output size, reaching a minimum when $p = 50$ or $p = 100$. Beyond that, they observed a slight increase in output size, likely due to longer dictionary phrases caused by fewer observed trigger strings. Reconstructing T from D_T and P_T is straightforward. The D_T phrases need to be appended in the order specified by P_T, trimming the last w characters of each phrase to account for the overlap.

The name of the algorithm, prefix-free parsing, stems from the observation that all dictionary phrase suffixes longer than w are prefix-free with respect to one another (see Definition 1). This is formalized in Lemma 3, which we now prove. Let $\mathcal{S}$ be the multiset of all dictionary phrase suffixes that are longer than w. Each $s \in \mathcal{S}$ must span the end trigger string of its respective phrase, and this trigger string cannot be a prefix of s, because all suffixes are longer than w. If the suffixes in $\mathcal{S}$ were not prefix-free with respect to each other, then some $s \in \mathcal{S}$ would be a proper prefix of another $s' \in \mathcal{S}$. This would imply that the phrase from which s' originates spans a trigger string that is neither a proper prefix nor a suffix of the phrase, contradicting the definition of a PFP phrase. Therefore, $\mathcal{S}$ must be prefix-free. Note that although $\mathcal{S}$ is prefix-free, it does not follow that each $s \in \mathcal{S}$ is necessarily unique, as distinct phrases may share the same suffix. An overview of the entire PFP workflow is provided in Figure 1.

Lemma 3 (Boucher et al. 2019): ***Given a text T and PFP(T, w, p) = D_T, P_T, we denote the multiset of all phrase suffixes in D_T that have a length greater than w as $\mathcal{S}$. It follows that $\mathcal{S}$ is a prefix-free multiset, which means that no string in $\mathcal{S}$ is a proper prefix of any other string in $\mathcal{S}$.***

We illustrate PFP using a small example. Instead of the one-pass algorithm presented earlier, we present it as a two-pass approach in this example for the sake of clarity. Let $w = 2$ and

$$T' := \$T\#\# = \texttt{\$GATTACAT!GATACAT!GATTAGATA\#\#}.$$

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

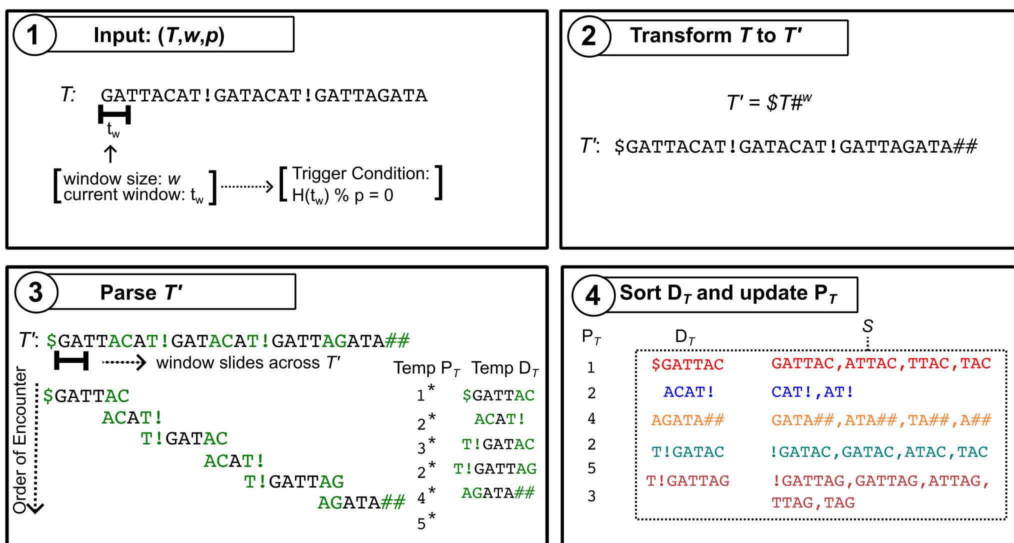


Figure 1. Overview of the PFP workflow. (1) PFP takes as input a text T and two integer parameters, the window size w and the hash modulus p. In the example, $w = 2$ and $p \in Z^+$, where Z^+ represents the set of positive integers. (2) It internally prepends the $ symbol and appends w consecutive copies of # to T. (3) The modified text is parsed by sliding a window of length w across the text; when a trigger string is encountered (highlighted in green), the current phrase terminates and the next phrase begins. During parsing, the dictionary D_T and parse P_T are constructed, with D_T initially unsorted and P_T containing temporary phrase records (*). (4) Finally, D_T is sorted and the records in P_T are replaced by the ranks of the corresponding dictionary phrases; the dotted box shows the multiset S of suffixes longer than w derived from phrases in D_T.

Next, imagine we do one pass through T' and we use a Karp-Rabin hash that defines the set E of trigger strings in T' as $E = \{$AC, AG, T!, $, ##$\}$. In the second pass, knowing these trigger strings, we would see that the dictionary D is

{$GATTAC, ACAT!, AGATA##, T!GATAC, T!GATTAG}

and the parse P is [1, 2, 4, 2, 5, 3].

The multiset of suffixes $\mathcal{S}$ as specified in Lemma 3 would be

{$GATTAC, GATTAC, ATTAC, TTAC, TAC, ACAT!, CAT!, AT!, AGATA##, GATA##, ATA##, TA##, A##, T!GATAC, !GATAC, GATAC, ATAC, TAC, T!GATTAG, !GATTAG, GATTAG, ATTAG, TTAG, TAG}.

PFP is designed for large repetitive texts, so this small example does not fully demonstrate the algorithm's compression capabilities. In practice, the combined representation of $D_T + P_T$ is usually much smaller than T when T is large and repetitive. However, even in this small example, PFP achieves a level of compression. The second dictionary index appears twice in the parse, indicating that the algorithm effectively eliminated storing one instance of the second dictionary phrase without penalty. In practice, PFP is fairly efficient. The algorithm uses $O(|D_T| + |P_T|)$ memory and runs in $O(n)$ time, where $|D_T|$ represents the length of all the phrases in D_T, $|P_T|$ is the length of P_T, and n is the length of T. PFP can be used as a preprocessing step to build data structures such as BWT_T, SA_T, and LCP_T. In the following sections, we will go over the construction algorithms of these data structures.

Recursive prefix-free parsing

Recursive PFP provides an additional layer of compression on the PFP output. For large repetitive data sets, the dictionary D_T has been observed to grow sublinearly relative to the parse P_T (Oliva et al. 2023). Recall from the Prefix-free parsing section that D_T grows only when a unique phrase is encountered, whereas P_T grows with every phrase. Although the output of PFP is usually much smaller than the input, it can still be too large for memory-constrained applications, prompting the development of recursive PFP.

Recursive PFP applies PFP to the parse P_T, producing a new dictionary D_P and parse P_P that fully represent it. Combined with D_T, these components fully represent T and form the output of recursive PFP. The key idea behind recursive PFP is that although P_T grows linearly, it remains highly repetitive, as phrases typically occur in the same order in repetitive texts, making it suitable for another round of PFP. Once PFP is applied to P_T, it can be safely deleted. If P_T is larger than D_T and is compressible, then recursive PFP will produce a noticeably smaller output than PFP. The idea of recursive PFP was briefly mentioned in Gagie et al. (2019) and more recently in Oliva et al. (2023). In theory, any application using PFP outputs can also be modified to use recursive PFP outputs. See Table 1 for applications where the recursive PFP has been successfully applied. Recursive PFP enables PFP to scale to arbitrarily large repetitive data sets by applying it recursively to each successive parse, provided the parse remains sufficiently compressible.

Building classic bioinformatics data structures from PFP

This section first introduces the BWT, SA, and LCP array data structures, followed by a description of how to construct them using the PFP components.

Burrows–Wheeler transform (BWT)

The Burrows–Wheeler transform (BWT) was introduced in Burrows and Wheeler (1994) as a reversible preprocessing technique that rearranges text to make it more compressible. The BWT_T is a special reversible permutation of the text $T$$, where $T[0 \cdots n-1]$ is the original text and $ is a unique sentinel character

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

appended to T that is lexicographically smaller than all other characters. Conceptually, the BWT_T can be constructed by first building the BWT_T matrix by sorting all rotations of $T\$$ lexicographically. The last column of the BWT_T matrix is referred to as the BWT_T. To illustrate this construction, Figure 2A shows a visualization of BWT_T for the example introduced in the Prefix-free parsing section. However, in practice, the BWT_T matrix is never explicitly constructed due to its quadratic space requirements.

The key observation of Burrows and Wheeler is that identical characters tend to group together and create *runs* in the BWT_T. For example, the text used in Figure 2A contains 27 characters, yet its BWT_T has only 13 maximal character runs. This demonstrates how the BWT_T reorders $T\$$ to increase its local regularity by exploiting higher-order contexts. The effect arises because the initial lexicographic sort groups together rotations with similar or identical prefixes. In particular, the start of each rotation corresponds to a suffix of $T\$$, which explains why the prefixes of consecutive rotations reveal repeated substrings in the text. Because texts are structured for comprehensibility, only a few distinct characters typically precede any given substring, and this number decreases as the length of the substring increases. This structural property gives rise to the runs observed in the BWT_T. The presence of runs in the BWT_T does not compress the text on its own, but the resulting structure is well-suited for compression by other algorithms. The authors suggest applying a move-to-front transformation (Bentley et al. 1986) to the BWT_T to convert these runs into a skewed distribution of small integers, followed by Huffman encoding (Huffman 1952). This compression scheme forms the core of the Bzip2 algorithm (Seward 1996), a widely used compressor today.

Another important property of the BWT_T is that it is reversible, allowing the original text to be recovered. This reversibility is guaranteed by the sentinel character \$ appended to the text, which establishes a unique total order among the rotations of T by breaking ties that might otherwise arise during lexicographic comparison. Because all rotations of $T\$$ are used to construct the BWT_T, every row and column of the conceptual BWT_T matrix is a permutation of $T\$$, as illustrated in Figure 2A. In particular, the relative order of the characters in $T\$$ is preserved between the first and last columns because the rotations are lexicographically sorted. Moving from the last column to the first effectively steps back one character in $T\$$. This relationship between the two columns is known as the last-to-first (LF) mapping, formally defined below.

Definition 4: ***Let C be an integer array where C[c] stores the number of characters in T\$ lexicographically smaller than c. Then, the LF-mapping property is defined as*** $LF(i, c) := C[c] + rank_c(BWT_T, i)$ ***and*** $LF(i) := LF(i, BWT_T[i])$.

From Definition 4, $LF(i)$ returns the index in the first column of the BWT_T matrix that corresponds to the character at index i in the BWT_T. The C integer array is a highly compressed representation of the first column of the BWT_T matrix, a column whose characters are lexicographically sorted. Both the first column (C) and the last column (BWT_T) are essential for LF-mapping. However, compared to the size of the BWT_T which is the size of $|T\$|$, the C array is significantly smaller and requires only $|\Sigma| + 1$ entries. For DNA, this means only four entries for the different nucleotides, plus one entry for the special \$ character appended to the text.

The text $T\$$ can be reconstructed from the BWT_T by repeatedly applying $LF(i)$, starting from the last character, \$. Let T' represent the reconstructed text. Initialize $T'[n] := \$$ and $s := 0$ where n is the length of T and s is an iterator where $0 \leq s \leq n$. For $i = n-1, \ldots, 0$, assign $T'[i] := BWT_T[s]$ and update $s := LF(s)$. At the end of the loop, we will get $T' = T\$$, recovering the text. In other words, $T\$$ was reconstructed in reverse using the LF-mapping property.

Ferragina and Manzini (2000) showed that the BWT_T can be used for more than just compression purposes. In particular, they demonstrated that the BWT_T enables efficient exact backward pattern matching within the text. This process is similar to how the original text is reconstructed. However, instead of tracking individual positions, the algorithm maintains a range in the BWT_T corresponding to all suffixes of T that match the current query suffix. Given a pattern $P[0 \cdots p-1]$, backward matching consists of p steps that preserve the following property: after matching the last i characters of P, the start of the search range sp corresponds to the first row of the BWT_T matrix prefixed by $P[p-i \cdots p-1]$ and the end of the range ep corresponds to the last such row. To extend the match to the next preceding character, LF-mapping is applied to both sp and ep, computing the new range as:

$$sp := C[c] + \text{rank}_c(BWT_T, sp),$$
$$ep := C[c] + \text{rank}_c(BWT_T, ep + 1) - 1,$$

where c is the character at $P[p-i-1]$. Note that if the rank queries of both sp and ep return the same value during the matching, the pattern does not occur in the text. An example of pattern matching is shown in Figure 2A. Also, observe that we can only determine the number of occurrences of a pattern with just the BWT_T. To find their locations within the text requires augmenting the BWT_T with another data structure called the suffix array.

Suffix array

The suffix array (SA) was formally introduced in Manber and Myers (1993) as a space-efficient alternative to suffix trees (Weiner 1973; McCreight 1976; Apostolico 1985) for online string searches, demonstrating three to five times lower space usage. Conceptually simple, the SA_T is an array of length $|T\$|$ where $SA_T[i]$ gives the starting index in the text of the $(i+1)$th lexicographically smallest suffix of $T\$$, for $0 \leq i < |T\$|$. That is, the SA_T stores the starting indices of the lexicographically sorted suffixes of $T\$$. Figure 2B shows the SA_T for the example introduced in the Prefix-free parsing section. Although the SA_T is more space-efficient than the suffix tree, it still requires $O(|T\$|)$ space.

The SA_T and BWT_T are intrinsically linked, as both are constructed by lexicographically sorting the suffixes of the text $T\$$. More formally, the relationship between the two data structures is as follows:

$$BWT_T[i] := T\$[(SA_T[i] - 1) \bmod (n+1)],$$

where $T\$[n] = \$$. Most implementations of the BWT_T require building the SA_T first. This relationship between the two data structures was already known when Burrows and Wheeler published their seminal paper on the BWT (Burrows and Wheeler 1994). Later, Ferragina and Manzini showed that augmenting the BWT_T with the SA_T allows efficient retrieval of pattern occurrence indices (Ferragina and Manzini 2000). However, rather than storing the full SA_T, they proposed a sampled version that stores only a subset of suffix indices. In practice, this saves memory at the cost of slower locate queries. If the index of the matched suffix is stored in the sampled SA_T, it can be returned immediately. Otherwise, LF-mapping is applied iteratively until a stored value is reached, from which the index of the matched suffix can be recovered.

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

Let s be the index of the suffix prefixed by the pattern whose value is not stored, and let s' be the index of the next suffix with a stored value, found after the v LF-mapping steps. Then,

$$SA_T[s] := (SA_T[s'] + v) \bmod (n+1).$$

The combination of the BWT_T and a sampled SA_T is known as the FM-index (Ferragina and Manzini 2000, 2005). This index is used by popular read aligners such as Bowtie (Langmead et al. 2009; Langmead and Salzberg 2012) and BWA (Li 2013).

The r-index

The FM-index, which consists of the BWT_T and SA_T, has been a foundational data structure in bioinformatics. However, a notable drawback is that the size of both the BWT_T and SA_T is proportional to the size of $T\$$ (see the BWT and SA sections). For very large data sets, these data structures become too large to be practical. As a result, considerable research has focused on compressing the FM-index. The use of run-length encoding the BWT_T has been established for some time (Mäkinen and Navarro 2005; Mäkinen et al. 2010), but sampling the SA_T within similar space remained an open problem for much longer. The problem remained open until 2017, when Gagie et al. (2020) introduced the r-index in their preprint, solving the problem by storing SA_T values only at the run boundaries of the BWT_T. With a few small auxiliary data structures, the missing SA_T can be efficiently computed. This insight completed the design of the r-index, enabling it to be stored in $O(r)$ space, where r denotes the number of maximal character runs in the BWT_T. In practice, $r \ll |T\$|$. Despite the shift toward the r-index in recent literature, its construction is conceptually rooted in the FM-index, which serves as the architectural basis for its compressed successor.

Building the FM-index (BWT and SA) from PFP

Boucher et al. (2019) introduced the PFP algorithm as a preprocessing step for the scalable construction of large BWTs. Building on this work, Kuhnle et al. (2020) showed how to construct the SA_T sample alongside the BWT_T. In doing so, they effectively showed how to construct the FM-index of the text from the PFP output, namely D_T and P_T. We illustrate this process through a detailed walkthrough of the algorithm by continuing the example introduced at the end of the Prefix-free parsing section, which serves as the basis for the following discussion. Figure 3A shows the PFP output.

In the Prefix-free parsing section, we described how PFP is applied to the text $\$T\#^w$. The prepended \$ and appended $\#^w$ ensure that the text begins and ends with a trigger string. Although \$ is shorter than the standard w-length trigger string, it still functions as a valid trigger because it is guaranteed to be unique within the text. This uniqueness preserves the prefix-free property (see Lemma 3). Furthermore, this use of \$ is consistent with its role in the standard BWT and SA constructions, where \$ is the lexicographically smallest sentinel appended to T. Establishing this consistent role of \$ is crucial, as it allows us to align the lexicographically sorted rotations of the PFP modified text $\$T\#^w$ with those of the standard *FM-index* text $T\$$.

Structurally, $T\$$ and $\$T\#^w$ differ in only two respects: the placement of the sentinel symbol \$ and the addition of the appended $\#^w$ characters. The placement of the sentinel, whether at

$T\$$: GATTACAT!GATACAT!GATTAGATA$

index: 0 ... 4 ... 9 ... 14 ... 19 ... 26

Pattern: GAT

C array	F (C)	BWT_T Matrix	L (BWT_T)	runs	i	$SA_T[i]$	$LCP_T[i]$
C[0] = 0	$\$_0$	GATTACAT!GATACAT!GATTAGAT	A_0	$r_A = 1$	0	26	0
C[1] = 1	$!_0$	GATACAT!GATTAGATA$GATTACA	T_0	$r_T = 6$	1	8	0
	$!_1$	GATTAGATA$GATTACAT!GATACA	T_1		2	16	4
C[2] = 3	A_0	$GATTACAT!GATACAT!GATTAGA	T_2		3	25	0
	A_1	CAT!GATACAT!GATTAGATA$GAT	T_3		4	4	1
	A_2	CAT!GATTAGATA$GATTACAT!GA	T_4		5	12	8
	A_3	GATA$GATTACAT!GATACAT!GAT	T_5		6	21	1
	A_4	T!GATACAT!GATTAGATA$GATTA	C_0	$r_C = 2$	7	6	1
	A_5	T!GATTAGATA$GATTACAT!GATA	C_1		8	14	6
	A_6	TA$GATTACAT!GATACAT!GATTA	G_0	$r_G = 4$	9	23	2
	A_7	TACAT!GATTAGATA$GATTACAT!	G_1		10	10	3
	A_8	TTACAT!GATACAT!GATTAGATA$	G_2		11	1	2
	A_9	TTAGATA$GATTACAT!GATACAT!	G_3		12	18	4
C[3] = 13	C_0	AT!GATACAT!GATTAGATA$GATT	A_1	$r_A = 3$	13	5	0
	C_1	AT!GATTAGATA$GATTACAT!GAT	A_2		14	13	7
C[4] = 15	G_0	ATA$GATTACAT!GATACAT!GATT	A_3		15	22	0
	G_1	ATACAT!GATTAGATA$GATTACAT	$!_0$	$r_! = 1$	16	9	4
	G_2	ATTACAT!GATACAT!GATTAGATA	$\$_0$	$r_\$ = 1$	17	0	3
	G_3	ATTAGATA$GATTACAT!GATACAT	$!_1$	$r_! = 1$	18	17	5
C[5] = 19	T_0	!GATACAT!GATTAGATA$GATTAC	A_4	$r_A = 3$	19	7	0
	T_1	!GATTAGATA$GATTACAT!GATAC	A_5		20	15	5
	T_2	A$GATTACAT!GATACAT!GATTAG	A_6		21	24	1
	T_3	ACAT!GATACAT!GATTAGATA$GA	T_6	$r_T = 1$	22	3	2
	T_4	ACAT!GATTAGATA$GATTACAT!G	A_7	$r_A = 1$	23	11	9
	T_5	AGATA$GATTACAT!GATACAT!GA	T_7	$r_T = 1$	24	20	2
	T_6	TACAT!GATACAT!GATTAGATA$G	A_8	$r_A = 2$	25	2	1
	T_7	TAGATA$GATTACAT!GATACAT!G	A_9		26	19	3

Figure 2. Visualization of the BWT_T, SA_T, and LCP_T for the example introduced in the Prefix-free parsing section. For clarity, the BWT_T matrix is shown in each subfigure to highlight the information captured by each data structure. (*A*) The BWT_T. In the BWT_T matrix, only the first (*C*) and last (BWT_T) columns are highlighted; these columns are annotated with character ranks as subscripts, although in practice these ranks are computed using an auxiliary data structure. To the *left* of the first column, we illustrate how the *C* array is populated, and to the *right* of the last column we show the maximal character runs in the BWT_T. Also shown is how pattern matching is performed using the *C* array and BWT_T. (*B*) The SA_T. In the BWT_T matrix, the suffixes of $T\$$ that prefix each rotation are highlighted. (*C*) The LCP_T. In the BWT_T matrix, the longest common prefix between consecutive rotations of $T\$$ is highlighted.

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

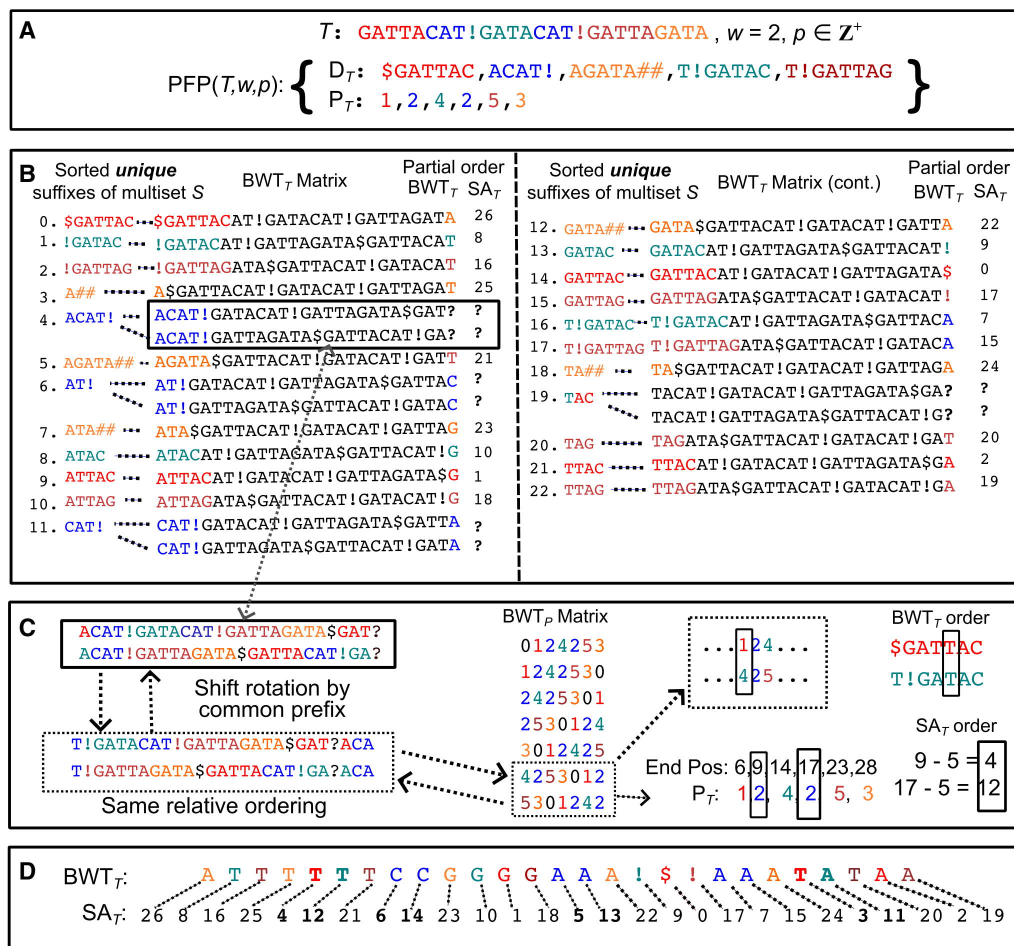


Figure 3. PFP-based construction of the FM-index for the example introduced in the Prefix-free parsing section. Color denotes phrase membership in all subfigures. (*A*) The input text and the PFP output after parsing. See the Prefix-free parsing section for more details. (*B*) The partial order BWT_T and SA_T constructed from D_T and the occurrence frequencies of P_T, where each unique suffix in S (derived from D_T) prefixes at least one BWT_T rotation and appears in sorted order in the matrix. Suffixes that prefix multiple rotations create ambiguous entries, denoted by a bold "?" symbol. (*C*) Ambiguous entries are resolved using the positions, in the BWT_P, of the phrases from which those suffixes originate. (*D*) The total order BWT_T and SA_T computed after resolving all ambiguous entries. Entries that were previously ambiguous are bolded.

the beginning or end, does not affect the lexicographic ordering of rotations, as the text is treated circularly; this is consistent with the definition of the BWT and SA, which are constructed from all cyclic rotations of the text. Similarly, appending $\#^w$ symbols does not alter the ordering because these symbols are defined to be lexicographically smallest, ensuring the lexicographical order of the rotations is preserved. Taken together, these observations imply that the lexicographical ordering of the rotations for both texts is identical. Furthermore, all suffixes of $T\$$ act as prefixes to rotations of $\$T\#^w$ when the sentinel is conceptually moved to the front. Therefore, the ability to lexicographically sort $\$T\#^w$ implies that $T\$$ can also be sorted, effectively bridging the two texts.

The lexicographically sorted rotations of $\$T\#^w$ can be obtained by sorting the dictionary phrase suffixes of the multiset S. From Lemma 3, we know that the suffixes in this multiset have length greater than w and are prefix-free, meaning that the *unique* suffixes in the multiset can be unambiguously ordered lexicographically. Because the initial PFP segmentation is lossless and divides the text into w-length overlapping phrases, this means that each suffix in S must be a prefix to at least one rotation of the text $\$T\#^w$, as illustrated in Figure 3B. By transitivity, lexicographically ordering the unique suffixes in S induces the corresponding lexicographic ordering of $T\$$. However, Figure 3B shows only 23 unique suffixes, whereas Figure 2B shows that $T\$$ contains 27 suffixes, revealing a discrepancy of four suffixes. There are two reasons for the discrepancy in the counts. First, the second dictionary phrase appears twice in the parse, so each of its suffixes longer than w is a prefix to twice as many rotations as it would if the phrase appeared only once. This accounts for three of four missing suffixes. Second, as shown in Figure 3B, the 20th lexicographically smallest suffix (index 19), TAC, is shared by the first and fourth dictionary phrases. Although TAC prefixes two distinct rotations in the text, these occurrences are represented by a single entry in the sorted list of unique suffixes, accounting for the final missing suffix. This information can be derived solely from the dictionary and the frequency of each phrase in the parse. However, relying solely on this information yields only a partial ordering of the rotations, resulting in an incomplete BWT_T and SA_T, as shown in Figure 3B. If a phrase appears multiple times, we cannot determine which phrase precedes each occurrence, making it

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

difficult to fully resolve the BWT_T and SA_T. Similarly, if multiple phrases share a common suffix, the BWT_T character and SA_T value for the rotations prefixed by that suffix become ambiguous. Computing the total order BWT_T and SA_T requires positional information from the parse.

Ambiguities in the partial order BWT_T and SA_T can be resolved by examining the BWT_P, as shown in Figure 3C. Recall that the parse lists the lexicographic ranks of dictionary phrases in textual order. Consequently, the sorted rotations of the parse correspond to the relative order of the phrases within the sorted rotations of the text. Because ambiguities in the partial order arise from identical phrase suffixes, they can be resolved by comparing the text immediately following those suffixes (the remainders of the rotations). Because PFP phrases always end at trigger strings, the remainders of the rotations correspond to the start of the *next* phrase in the sequence. Therefore, the lexicographical ordering of the parse rotations (captured by the BWT_P) provides the necessary information to resolve these ambiguities.

Continuing from the partial order in our running example, the first ambiguity arises because the second phrase, ACAT#, is preceded by both the first and fourth dictionary phrases. Therefore, to determine the ordering, we must identify which occurrence of the second phrase appears earliest in the BWT_P. In the parse sequence, these occurrences are followed by the fourth and fifth phrases, respectively. A visual inspection of the first column of the BWT_P matrix confirms that the fourth phrase appears earlier than the fifth. This implies that the second phrase occurrence preceded by the first phrase is lexicographically smaller than the one preceded by the fourth phrase. In the context of resolving the BWT_T, it coincidentally does not matter because both preceding phrases end with T (after accounting for the w-length overlap); however, it does matter in resolving the SA_T at these indices. The second ambiguity arises because the suffix TAC is shared by both the first and fourth dictionary phrases. Because it is preceded by different characters in each phrase, the order in which the phrases appear in the BWT_P becomes significant in constructing the BWT_T. To resolve the ambiguity, we follow the same steps as before. It turns out that both phrases are followed by occurrences of the second phrase, whose order we previously determined. By transitivity, this implies that the first phrase precedes the fourth, meaning that T should precede A in the BWT_T for these specific rotations. Finally, once the correct ordering is determined, calculating the SA_T values is straightforward: the suffix length is subtracted from the corresponding phrase's end position in the text to obtain the suffix's start position. The total order BWT_T and SA_T can be seen in Figure 3D.

To efficiently implement these concepts, two technical optimizations are required. First, the ordering of the suffixes in S in practice is achieved by constructing the suffix array of the dictionary, denoted as SA_D. The dictionary phrases are concatenated using a unique end-of-word separator symbol [EOW], forming the string $D_T = d_1[\text{EOW}]d_2[\text{EOW}]\cdots d_{|D|-1}[\text{EOW}]d_{|D|}[\text{EOW}]$ where d_i is the ith lexicographically sorted dictionary phrase. In SA_D, any suffix containing [EOW] within its first $w+1$ characters is ignored because these suffixes are shorter than or equal to w and therefore do not belong to multiset S. The [EOW] symbol is defined to be lexicographically smaller than all other characters, including the $ and # symbols added during the PFP parsing. Therefore, the first —D_T—suffixes in SA_D will correspond to rotations that start with [EOW]. Notably, the [EOW] symbols will appear in the exact positional order of their occurrence in the concatenated D_T, because the dictionary phrases were previously sorted. This structure is beneficial, as it allows for binary search in this subsection of the SA_D to efficiently identify the originating phrase of any suffix and calculate its length. Second, explicitly scanning the BWT_P is inefficient. Instead of explicitly scanning the BWT_P, an auxiliary data structure called the inverted list (IL) is built. This is implemented as a single array of size $|P_T|+1$ (accounting for the parse length $|P_T|$ plus the sentinel). The array stores the BWT_P indices of all phrase occurrences, grouped by phrase and ordered lexicographically. In this capacity, the IL array effectively acts as an interface between the first column (sorted phrases) and the last column (BWT_P) of the BWT_P matrix, enabling the efficient retrieval of phrase occurrences in the BWT_P without the overhead of a full scan. These indices can then be used to index auxiliary data structures of size $|P_T|+1$ generated by PFP, which store the BWT_T character and ending position of each phrase to efficiently resolve ambiguous entries in the BWT_T and SA_T.

Longest common prefix array

The longest common prefix (LCP) array was introduced in Manber and Myers (1993) as an auxiliary structure to accelerate pattern search with the SA. For two arbitrary strings $T_1[0\cdots n-1]$ and $T_2[0\cdots m-1]$, the value of $LCP(T_1, T_2)$ falls into one of the three cases: (1) 0, if $T_1[0]\neq T_2[0]$, (2) $\min(n, m)$, if T_1 or T_2 prefixes the other, or (3) the unique i such that $T_1[0\cdots i-1]=T_2[0\cdots i-1]$ and $T_1[i]\neq T_2[i]$. The LCP_T array stores the LCP values between consecutive suffixes of T in lexicographic order. Formally, $LCP_T[0\cdots n]$ is defined by $LCP_T[0]:=0$ and for $1\leq i\leq n$:

$$LCP_T[i] := LCP(T\$[SA_T[i-1]\cdots n],\ T\$[SA_T[i]\cdots n]),$$

where $T\$[n]=\$$. The LCP_T array for the example introduced in the Prefix-free parsing section is shown in Figure 2C. One useful property of the LCP_T array is that the longest common prefix of *any* two suffixes can be computed by performing a range minimum query (RMQ) over the LCP_T values between their indices in the suffix array. More formally, for two indices $i<j$, computing $LCP(i, j)$ is equivalent to finding the minimum value in the range $LCP_T[i+1\cdots j]$. This query is often referred to as computing the longest common extension (LCE) between two suffixes of T. For example, in Figure 2C, the LCP value between $SA_T[7]$ and $SA_T[12]$ can be found by computing $RMQ_{LCP_T}(8, 12)=\min(LCP_T[8], LCP_T[9], LCP_T[10], LCP_T[11], LCP_T[12])=2$, which corresponds to the common prefix AT. There exist data structures that support constant $O(1)$ time RMQ queries that can be built in linear time (Fischer 2010). Precomputing the LCP_T array helps avoid performing redundant character comparisons, making it especially useful in string matching problems.

Building the LCP array from PFP

Rossi et al. (2022) introduced a PFP-based FM-index construction that simultaneously computes the LCP array, allowing for a wider range of string queries that can be performed with the index. To construct the complete LCP_T array, they first build an auxiliary structure called the SLCP, a variant of the LCP_P, where the LCP values are computed over the uncompressed form of P_T0, excluding the w-length overlap between phrases. Note that the 0 is a sentinel appended to P_T, which is why the dictionary ranks in the parse start at 1 rather than 0. More formally, $SLCP[0\cdots|P_T|]$ is defined by $SLCP[0\cdots 2]=0$ and for $2<i\leq|P_T|$:

$$SLCP[i] := LCP(T\$[p_{i-1}\cdots n],\ T\$[p_i\cdots n]),$$

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

where $T\$ [n] = \$$ and p_{i-1}, p_i are the starting indices in $T\$$ of the rotations that are prefixed by the $(i-1)$th and ith lexicographically smallest suffixes of the uncompressed P_T. The initialization accounts for the two sentinels: the text sentinel \$ and the parse sentinel 0, which occupy the initial lexicographical ranks. This data structure can be computed in $O(|P_T|)$ time with a slight modification of the algorithm described in Kasai et al. (2001). Figure 4 shows an image of the SLCP computed from the parse of the example text introduced in the Prefix-free parsing section. Additionally, they build a RMQ data structure over the SLCP.

During the FM-index construction, the LCP_T can start to be built once the lexicographic ordering of the rotations is determined. As discussed in the Building the FM-index from PFP section, the problem of sorting the rotations reduces to correctly sorting the suffixes of the D_T phrases with length greater than w, which prefix all rotations. Now let α and β be the D_T suffixes that prefix the rotations of $SA_T[i-1]$ and $SA_T[i]$ and let p_1, p_2 be the indices in the BWT_P of the phrases from which α and β are derived from and let $p_1 < p_2$. Then, the LCP_T can be computed as follows:

$$LCP_T[i] := \begin{cases} \text{LCP}(\alpha, \beta) & \text{if } \alpha \neq \beta, \\ |\alpha| - w + h & \text{otherwise,} \end{cases}$$

where $h = RMQ_{SLCP}(p_1 + 1, p_2) = \min(SLCP[p_1 + 1 \cdots p_2])$. If the suffixes that prefix two consecutive sorted rotations differ, their LCP value can be computed directly by comparing the suffixes, as the difference must be within their length due to the prefix-free property of the multiset S. However, if the suffixes that prefix both rotations are identical, then additional information is needed to compute the LCP value. Recall that P_T is effectively a compressed representation of $T\$$ and that the rotations of P_T represent a subset of the rotations of $T\$$. Because P_T encodes the lexicographic ranks of phrases in D_T, its sorted rotations appear in the same relative order as in $T\$$. If two rotations share the same D_T suffix as a prefix, then the remainder of each rotation must begin with a full dictionary phrase, implying that the remainders of these rotations correspond to a rotation of P_T. As discussed in the LCP section, we can compute the LCP value between these rotations by performing

$T\$ \leftrightarrow \$T$: \$GATTACAT!GATACAT!GATTAGATA

P_T: 1, 2, 4, 2, 5, 3, 0 ← Sentinel Added

i	SLCP[i]	BWT_P Matrix (uncompressed)
0	0	0\$GATTACAT!GATACAT!GATTAGATA
1	0	\$GATTACAT!GATACAT!GATTAGATA0
2	0	ACAT!GATACAT!GATTAGATA0\$GATT
3	8	**ACAT!GAT**TAGATA0\$GATTACAT!GAT
4	1	**A**GATA0\$GATTACAT!GATACAT!GATT
5	0	T!GATACAT!GATTAGATA0\$GATTACA
6	5	**T!GAT**TAGATA0\$GATTACAT!GATACA

Figure 4. Visualization of the SLCP for the example introduced in the Prefix-free parsing section. The SLCP represents the LCP_P with P_T decompressed and the last w characters trimmed from each phrase, effectively creating rotations starting at phrase boundaries of T. In the accompanying BWT_P matrix, colors are used to highlight phrases within each rotation, and bolded characters emphasize the SLCP values. Because the built-in overlap is removed, the last w characters of each phrase are split and colored half by the preceding phrase and half by the following phrase.

an RMQ on the SLCP between indices p_1 and p_2. This is equivalent to performing an RMQ on the LCP_T array between the SA_T indices of these rotations. This enables the computation of the remaining match length between the rotations.

Suppose that we want to calculate $LCP_T[23]$ in our example. Recall that in the Prefix-free parsing section that this rotation was initially ambiguous in the partial order BWT_T because it was prefixed by the suffix TAC, which is shared by both the first and fourth dictionary phrases. We now know that the rotation of the first phrase precedes the rotation from the fourth phrase. Because these two rotations are prefixed by the common suffix TAC, we know that the LCP value between these two rotations is at least 3. From Figure 4, we find that their remainders correspond to the third (index 2) and fourth (index 3) rotations of the BWT_P matrix, so we need to perform an RMQ on the SLCP between the range of $[2+1 \cdots 3]$ and this returns 8, the only value within the range. Therefore, $LCP_T[23] = 3 - 2 + 8 = 9$, which matches the value shown in Figure 2C. Note that w must be subtracted from the suffix length in this case, because the minimum SLCP value already accounts for the w-length overlap between the common suffix prefixing the rotations and the start of their remainders.

PFP applications

PFP has been applied in a variety of contexts to overcome the limitations of traditional indexing on large highly repetitive data. Table 1 presents a selected subset of works that use PFP, broadly categorized by their contribution.

Building fundamental bioinformatic data structures

As previously discussed, the BWT_T (Boucher et al. 2019), SA_T (Kuhnle et al. 2020), and LCP_T (Rossi et al. 2022) have been built from the PFP output. Boucher et al. (2019) showed that PFP enables substantially more resource-efficient construction of the run-length FM-index (RLFM-index) than Bowtie's (Langmead et al. 2009) FM-index-based approach on large collections of human Chromosome 19 (Chr 19) sequences. Across collections of up to 1000 Chr 19 sequences, the RLFM-index was consistently at least an order of magnitude faster to build than Bowtie's index and, once the collection exceeded 50 sequences, required an order of magnitude less peak memory. In both cases, the performance gap widened as additional sequences were added. On a 512 GB server, Bowtie exhausted all available memory after indexing just 250 sequences, whereas the PFP-based RLFM-index was successfully constructed for all 1000 sequences within the same memory constraints. Additional evidence supporting these claims is presented in the Experimental results section. Subsequently, both Kuhnle et al. (2020) and Rossi et al. (2022) further demonstrated the efficiency of PFP-based construction algorithms for large, repetitive data sets.

To further reduce both the time and memory requirements of constructing the aforementioned data structures for large data sets, recursive PFP has been applied (Oliva et al. 2023; Ferro et al. 2024) (see the Recursive prefix-free parsing section for details). Oliva et al. (2023) showed that, for extremely large collections of sequences, constructing the BWT_T from the output of recursive PFP is more computationally efficient than the construction from standard PFP. They repeated the experiment of Boucher et al. (2019), this time scaling up to 2400 Chr 19 sequences, more than double the amount used previously. Compared with

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

Table 1. Applications of prefix-free parsing

Category	Subcategory	Paper	Contribution
Data structure	BWT, SA, LCP	Boucher et al. (2019), Kuhnle et al. (2020), and Rossi et al. (2022)	Builds the BWT, SA, and LCP using PFP
	BWT, SA, LCP	Oliva et al. (2023) and Ferro et al. (2024)	Builds the BWT, SA, and LCP using recursive PFP
	eBWT	Boucher et al. (2021a, 2024)	Computes the extended BWT (eBWT) for a collection of strings using PFP
	BWT	Díaz-Domínguez et al. (2025)	Shows how to merge BWTs built with PFP
	Compressed suffix tree	Boucher et al. (2021b) and Oliva et al. (2022)	Shows how PFP output can be modified to support full suffix tree functionality
String matching	MEMs	Rossi et al. (2022) and Boucher et al. (2024)	Computes maximal exact matches (MEMs) using data structures built with PFP
	MEMs, MUMs	Shivakumar and Langmead (2025)	Computes maximal exact (MEM) or unique (MUM) matches across sequences using data structures built with PFP
	MEMs, full read alignment	Varki et al. (2025)	MEM-based short-read aligner built on index constructed with PFP
	MS	Boucher et al. (2021c) and Rossi et al. (2022)	Computes matching statistics (MS) with data structures built with PFP
	PMLs	Ahmed et al. (2021, 2023)	Computes an approximation of MS called pseudomatching lengths (PMLs) using data structures built from PFP
	LF-mapping, BWT	Hong et al. (2024)	Shows that leveraging PFP phrases for LF-mapping yields better performance than standard character-level LF-mapping
Compression	LZ77	Hong et al. (2023)	Constructs LZ77 factorization using PFP
	RePair	Gagie et al. (2019)	Builds RePair grammar from PFP output
	RePair	Kim et al. (2024)	Builds RePair grammar from recursive PFP output

The table highlights a *subset* of works that use PFP, categorized by their primary contribution. Although grouped by category, many of these papers build on one another, resulting in significant overlap.

the standard PFP-based construction, the recursive PFP-based approach reduced memory usage from approximately 40 to 15 GB and halved the running time from roughly 3000 to 1500 s, demonstrating substantial improvements in both speed and memory efficiency. However, this improvement only started to become noticeable after around 500–1000 sequences. For smaller collections, the performance difference between the two methods was negligible. These empirical results are supported by the theory of recursive PFP. Ferro et al. (2024) extended this work and showed how to build the SA_T and LCP_T alongside the BWT_T with recursive PFP.

Extensions and specialized indexing structures

Beyond the classical bioinformatics data structures discussed in this paper, PFP has also been used to build extensions of these structures, or components thereof, including the compressed suffix tree (CST) (Boucher et al. 2021b; Oliva et al. 2022), the extended BWT (eBWT) (Boucher et al. 2021a, 2024), the suffixient set (Cenzato et al. 2024), and the move structure (Zakeri et al. 2024, 2025). Rather than directly building the CST, Boucher et al. (2021b) show that the output of PFP can be augmented to support full suffix tree functionality. This is particularly important because the suffix tree can be viewed as a unifying conceptual framework for many string data structures, including those considered in this paper, and therefore plays a central role in bioinformatics, at least from a historical point of view. Suffix trees have a large space overhead relative to the input, and although CSTs reduce this overhead, they remain expensive to construct. Notably, the PFP-based CST construction by Boucher et al. (2021b) was shown to be more time and memory efficient than the state-of-the-art implementations in the Succinct Data Structure Library (SDSL) (Gog et al. 2014), a widely used benchmark library for compressed data structures. The eBWT extends the BWT to string collections by grouping related subsequences across different strings (Mantaci et al. 2007). The eBWT enables similarity detection and has been used as an index in metagenomic applications, helping to determine the taxonomic origin of sequencing reads (Ander et al. 2013; Guerrini et al. 2020). The suffixient set (Depuydt et al. 2023; Cenzato et al. 2024; Navarro et al. 2025) is a novel data structure that stores a small subset of integers capturing all branching points of a suffix tree. Although being new, it holds promise as a compressed index for string matching tasks in bioinformatics. The move structure (Nishimoto et al. 2022) is another new compressed data structure, based on the BWT. It has the unique beneficial properties of being stored in $O(r)$ space while still supporting $O(1)$ LF queries. Zakeri et al. (2024, 2025) implemented the data structure and showed that, in practice, its favorable locality of reference limits cache misses, leading to significant speed gains for string matching queries over other pangenomic data structures, such as the r-index.

String matching queries

Another subset of PFP research focuses on string queries enabled by data structures constructed from PFP. Many useful string matching queries can be calculated from matching statistics. Briefly, the matching statistics $MS[0 \cdots m-1]$ describe how a read $R[0 \cdots m-1]$ aligns to a text $T[0 \cdots n-1]$. For each position i in R, $MS[i]$ returns a pair: the length of the longest prefix starting at $R[i]$ that occurs in T, and the position of one such occurrence

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

in T. Matching statistics have been used in PFP applications to support various string queries, including maximal exact matches (MEMs) (Rossi et al. 2022), which cannot be extended in either direction; maximal unique matches (MUMs) (Giuliani et al. 2022; Shivakumar and Langmead 2025), which occur only once in T; and pseudo matching lengths (PMLs) (Ahmed et al. 2021, 2023), an efficient approximation of matching statistics. Varki et al. (2025) used MEMs as seeds for their r-index based short-read aligner, adopting the approach popularized by BWA-MEM (Li 2013). Shivakumar and Langmead (2025) computed multi-MUMs across 300+ human assemblies using under a terabyte of memory, facilitating the visualization of novel structural insights. Ahmed et al. (2021, 2023) demonstrated that PMLs can be computed in a streaming manner fast enough to process Nanopore sequencing reads in real time, allowing for direct metagenomic classification.

Text compression and grammar induction

A further line of PFP research investigates its use as a preprocessing step to construct classical compressed representations, building on its origins in text compression. Hong et al. (2023) showed that PFP can be used to construct the LZ77 factorization, a compression algorithm that divides the text into the longest substrings matching earlier occurrences (Ziv and Lempel 1977). LZ77 is a foundational algorithm in string processing and forms the basis of many modern compression algorithms, including LZMA2 and Deflate (core algorithm of gzip). Similarly, Gagie et al. (2019) showed that PFP (called CTPH at the time) can be used to build a variant of the RePair grammar. RePair is a grammar-based compression algorithm that recursively replaces the most frequent adjacent pair of characters with a new previously unseen character until no repeats remain (Larsson and Moffat 2000). Although conceptually simple, it achieves good compression in practice. However, it requires memory proportional to the input size, making it impractical for large data sets. Gagie et al. (2019) showed that their PFP-based RePair grammars can be constructed for inputs several orders of magnitude larger than those handled by standard RePair, using a fraction of the time and memory. Kim et al. (2024) extended this work and showed that both time and memory usage could be further reduced when the grammar was built from recursive PFP, noting that these benefits were observed only for large collections of sequences, consistent with the findings of Oliva et al. (2023).

Experimental results

We benchmarked PFP[1] against a grammar-based compressor (RePair[2]) and general-purpose compressors[3] (Bzip2, Deflate, Deflate64, LZMA2, and PPMd) on an increasing number of human Chromosome 19 (Chr 19) sequences from the 1000 Genomes Project (The 1000 Genomes Project Consortium 2015). These files were used without any preprocessing, retaining their headers and whitespace. As discussed in the Prefix-free parsing section, PFP was designed as a preprocessing algorithm for building other data structures in a memory-efficient manner. As a result, its output (the dictionary and parse) does not natively exploit the benefits of advanced compression techniques such as entropy coding or bit-packing, unlike the general-purpose compressors whose primary goal is compression. To evaluate the compressibility of the PFP output, we archived and compressed its dictionary and parse using tar and gzip (denoted as PFP (GZipped)). We applied the same procedure to the RePair output. PFP was run with its default parameter settings ($w = 10$, $p = 100$). All the general-purpose compressors were run at compression level 6 (on a scale of 1–9, where 1 denotes minimal compression and 9 maximal), which is the default or near-default setting. All files were compressed using one thread. All experiments were run and benchmarked with Snakemake (v7.32.4) (Mölder et al. 2025) on a server with an 2.95 GHz AMD EPYC 75F3 32-core processor. All tasks were allocated up to 100 GB of memory and a 24-h runtime, and were terminated if either limit was exceeded. The benchmarking results can be seen in Figure 5.

The experimental results confirm that PFP is a highly efficient compressor when run on large repetitive data sets. As shown in Figure 5A, PFP compressed all 2048 Chr 19 sequences (117.4 GB) to the smallest size (12.3 GB) among the tested compressors. This is reflected in Figure 5B as PFP has the smallest compression ratio among all the compressors at 0.1 for the full set of sequences. For context, compression ratio is defined as compressed size divided by uncompressed size. These numbers further improved when the PFP output was gzipped, reducing the file size to 4.6 GB and the compression ratio to 0.04. For comparison, Deflate achieved a compression ratio of 0.24 on the full set. Because Deflate is the underlying algorithm used by gzip, the fact that PFP (Gzipped) achieved over an 80% reduction in file size suggests that the PFP output is significantly more compressible than the original text. The only compressors that appeared to rival PFP on the full set were RePair and LZMA2; however, neither was able to compress the entire data set under the given constraints as LZMA2 timed out and RePair ran out of memory (see Fig. 5C,D). The motivation for using PFP on specifically large data sets is evident from Figure 5B. For the first 16 sequences, PFP had a compression ratio above 1.0, meaning its output was larger than the input. This occurs when there is little phrase repetition among the sequences, causing both the dictionary and parse to grow proportionally with the number of input sequences. However, with enough sequences, the dictionary size was observed to grow sublinearly, causing the compression ratio to decrease.

The computational performance benchmarks show that PFP performs favorably compared to the other compressors, with increasing advantages as the data set size grows. As shown in Figure 5C, PFP was the fastest method, compressing all 2048 Chr 19 sequences in 2612 s (1.28 s/sequence). The only method with comparable speed was PPMd, which compressed the full set in 4638 s (2.26 s/sequence), nearly twice as long as PFP. PFP is fast because it processes the input in a single pass using a sliding window with rolling hashes, avoiding expensive backtracking or repeated scans of the data. In terms of memory usage, PFP was efficient, though not the most memory efficient among the compressors. As shown in Figure 5D, PFP used 12.8 GB to compress all the Chr 19 sequences, representing only a small fraction (0.11) of the uncompressed data set size. Moreover, PFP required significantly less memory than LZMA2 and RePair, which used 27.6 and 100 GB, respectively, on their last completed data set. By contrast, Deflate, Deflate64, BZip2, and PPMd used very little memory (<1 GB). The reason the memory usage increases in PFP is because

[1]https://github.com/marco-oliva/pfp
[2]http://www.dcc.uchile.cl/gnavarro/software/repair.tgz
[3]All provided by 7-Zip, a popular open-source archiver tool (https://www.7-zip.org).

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

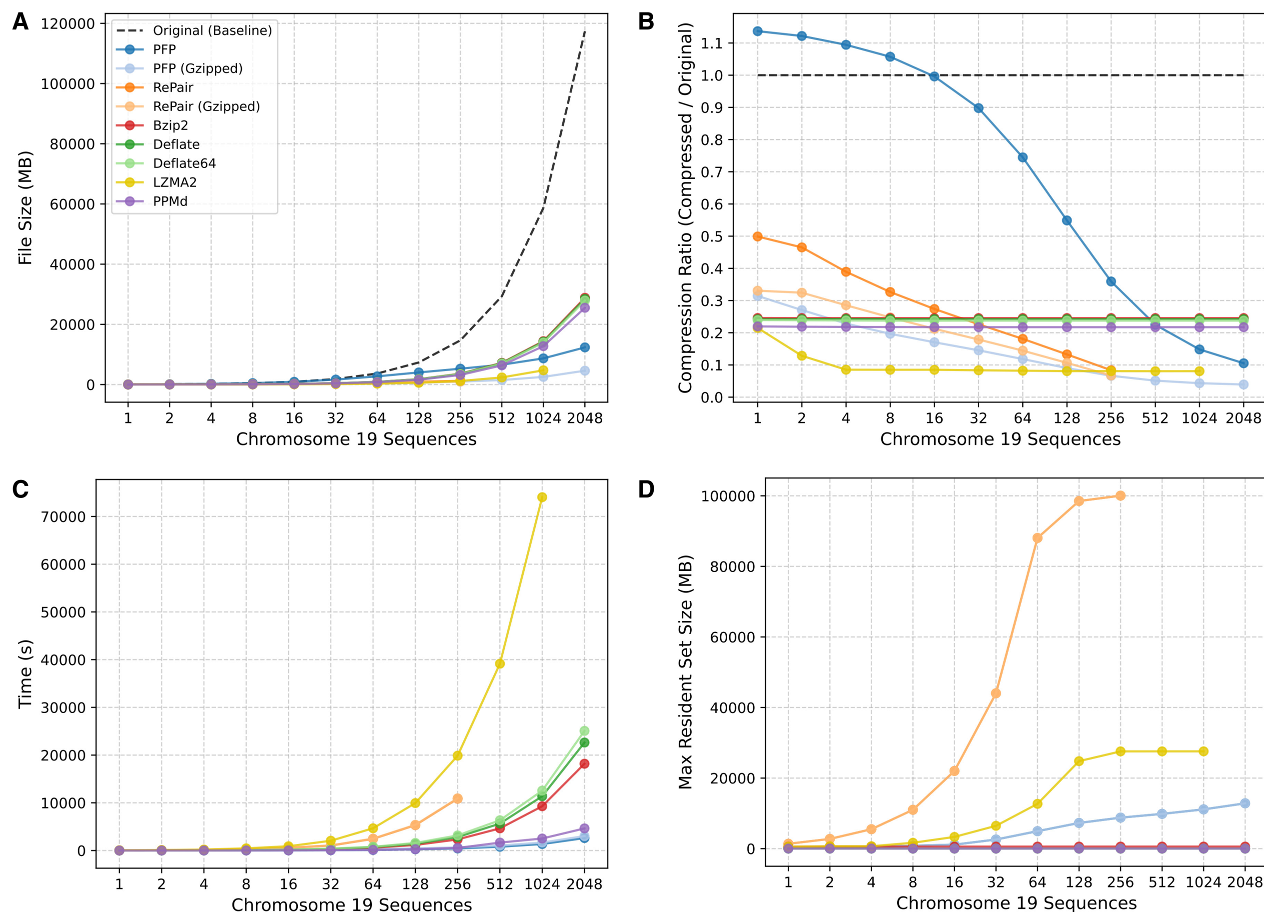


Figure 5. Performance comparison of PFP against standard compressors on an increasing number of human Chromosome 19 sequences. The subplots show: (*A*) compressed file size (MB), (*B*) compression ratio (defined as compressed/original size), (*C*) wall-clock time (s), and (*D*) peak memory usage (MB). All files were compressed using a single thread.

the dictionary and parse are kept in memory throughout the parsing.

The compression and computational benchmarks together demonstrate that PFP is a practical and competitive choice for compressing large, repetitive data sets like biological data sets.

Future directions

As described in the PFP applications section, PFP has been used in a wide range of applications. Despite its adoption, there are still opportunities to refine PFP and develop new applications that leverage its capabilities. One potential improvement to PFP lies in the way the dictionary is represented. PFP achieves its best compression on large collections of highly similar sequences, as repeated identical phrases cause the dictionary to grow sublinearly compared to the parse. However, when a variation is encountered, the entire phrase containing it must be added to the dictionary, even if it differs only by one character from another existing phrase (Oliva et al. 2022). Seemingly, new phrases caused by variations to existing phrases could be represented more space-efficiently in the dictionary.

A recent paper by Díaz-Domínguez et al. (2025) presents a method to merge BWTs constructed with PFP. This is significant because it potentially enables parallel BWT construction via multithreading. Their key insight is that if trigger strings in each partition are unique across partitions, then phrase suffixes remain unique to their respective partitions. As a result, the relative order of the characters in the BWT of each partition is preserved in the merged BWT, because no ties occur between suffixes of different partitions. The merged BWT is then constructed by lexicographically comparing the suffixes between partitions and writing the corresponding characters accordingly. The challenge is to find unique trigger strings that only appear in specific partitions that are valid throughout the text. To find this set of trigger strings, they parse each partition and remove any triggers that appear in other partitions. This approach seems impractical for collections of similar texts, where trigger strings are likely shared. The authors acknowledge this limitation and note that it performs better on diverse text collections. However, the compression benefits of PFP are reduced with more diverse texts, highlighting the need for a better solution.

In addition to building data structures, PFP may help improve the performance of large language models (LLMs), which are increasingly being studied and applied in bioinformatics (Ji et al. 2021; Nguyen et al. 2023; Zhang et al. 2023; Sanabria et al. 2024; Zhou et al. 2024). An important yet understudied aspect of these

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

models is how their input tokens are defined. Tokens are small units of data that collectively form the input of the model. They are essential for reducing the input space, which would be prohibitively large if the model processed the data directly. The choice in tokenizer has been shown to directly influence model performance (Dotan et al. 2024; Lindsey et al. 2025). Currently, *k*-mer and byte-pair encoding (BPE) (Gage 1994; Sennrich et al. 2016) are the most widely used tokenization schemes in bioinformatics-specific LLMs, but both have limitations. *k*-mer tokenization divides the input sequence into overlapping subsequences, each consisting of *k* consecutive characters. A key limitation is that individual tokens often lack a meaningful representation of the underlying data. Moreover, *k* cannot be set too large due to the limited vocabulary capacity of these models. In contrast, BPE is a dynamic tokenization method that is a variation of the RePair algorithm (briefly discussed in the PFP applications section), but it merges the most frequent adjacent characters or existing tokens until a target vocabulary size is reached. However, BPE can introduce intermediate scaffold tokens that become part of the vocabulary but are rarely used. PFP has the potential to address both issues. By treating PFP-generated phrases as tokens, it produces a minimal set of variable-length tokens that occur at least once in the input, which can be fine-tuned by changing the *w* and *p* parameters.

Conclusion

PFP has emerged as a valuable preprocessing technique for scalable construction of key bioinformatics data structures. As biological data sets continue to grow in size and complexity, PFP is poised to play an increasingly critical role due to its ability to scale efficiently with the size of the data. It has already demonstrated the capability to process hundreds of gigabytes of data. Beyond its practical utility, PFP provides a robust theoretical framework that enables researchers to extend its use, enabling a wide range of applications built on it. Although its primary adoption has been within bioinformatics, PFP is a generalizable algorithm that performs effectively on highly repetitive data sets. Its versatility and scalability position PFP as a foundational approach to the efficient construction of data structures on repetitive data sets.

Competing interest statement

The authors declare no competing interests.

Acknowledgments

R.V. and C.B. were supported by "Fully Realizing Pangenomics Alignment" (National Institutes of Health [NIH] Award No. 1R56HG013865-01; Contact PI: Christina Boucher) and "Personal and Panel References for Improved Alignment" (NIH Award No. 2R01HG011392-06; formerly 5R01HG011392-05; Contact PI: Ben Langmead).

Author contributions: R.V. and C.B. conceptualized the paper. R.V. drafted the manuscript, conducted the experiments, and produced the figures. C.B. provided feedback and supervised all aspects of the work. Both authors reviewed and edited the manuscript.

References

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

Ahmed O, Rossi M, Kovaka S, Schatz MC, Gagie T, Boucher C, Langmead B. 2021. Pan-genomic matching statistics for targeted nanopore sequencing. *iScience* **24:** 102696. doi:10.1016/j.isci.2021.102696

Ahmed O, Rossi M, Gagie T, Boucher C, Langmead B. 2023. SPUMONI 2: Improved pangenome classification using a compressed index of minimizer digests. *Genome Biol* **24:** 122. doi:10.1186/s13059-023-02958-1

Ander C, Schulz-Trieglaff OB, Stoye J, Cox AJ. 2013. metaBEETL: High-throughput analysis of heterogeneous microbial populations from shotgun DNA sequences. *BMC Bioinform* **14**(Suppl. 5)**:** S2. doi:10.1186/1471-2105-14-S5-S2

Apostolico A. 1985. The myriad virtues of subword trees. In *Proceedings of the international conference on combinatorial algorithms on words (WORDS)*, Maratea, Italy, pp. 85–96.

Bentley JL, Sleator DD, Tarjan RE, Wei VK. 1986. A locally adaptive data compression scheme. *Commun ACM* **29:** 320–330. doi:10.1145/5684.5688

Boucher C, Gagie T, Kuhnle A, Langmead B, Manzini G, Mun T. 2019. Prefix-free parsing for building big BWTs. *Algorithms Mol Biol* **14:** 13:1–13:15. doi:10.1186/s13015-019-0148-5

Boucher C, Cenzato D, Lipták Z, Rossi M, Sciortino M. 2021a. Computing the original eBWT faster, simpler, and with less memory. In *Proceedings of the international symposium on string processing and information retrieval (SPIRE)*, Lille, France, pp. 129–142.

Boucher C, Cvacho O, Gagie T, Holub J, Manzini G, Navarro G, Rossi M. 2021b. PFP compressed suffix trees. In *Proceedings of the workshop on algorithm engineering and experiments (ALENEX)*, pp. 60–72. SIAM. Virtual conference.

Boucher C, Gagie T, Tomohiro I, Koppl D, Langmead B, Manzini G, Navarro G, Pacheco A, Rossi M. 2021c. PHONI: Streamed matching statistics with multi-genome references. In *Proceedings of the data compression conference (DCC)*, pp. 193–202. Virtual conference.

Boucher C, Cenzato D, Lipták Z, Rossi M, Sciortino M. 2024. r-indexing the eBWT. *Inf Comput* **298:** 105155. doi:10.1016/j.ic.2024.105155

Burrows M, Wheeler DJ. 1994. A block-sorting lossless data compression algorithm. Technical Report 0769518966, Digital Equipment Corporation, Systems Research Center. Publication Title: SRC Research Report.

Cenzato D, Olivares F, Prezza N. 2024. On computing the smallest suffixient set. In *Proceedings of the international symposium on string processing and information retrieval (SPIRE)*, Puerto Varas, Chile, pp. 73–87.

Clarke L, Zheng-Bradley X, Smith R, Kulesha E, Xiao C, Toneva I, Vaughan B, Preuss D, Leinonen R, Shumway M, et al. 2012. The 1000 genomes project: Data management and community access. *Nat Methods* **9:** 459–462. doi:10.1038/nmeth.1974

Depuydt L, Gagie T, Langmead B, Manzini G, Prezza N. 2023. Suffixient sets. arXiv:2312.01359 [cs.DS]. doi:10.48550/arXiv.2312.01359

Díaz-Domínguez D, Gagie T, Guerrini V, Langmead B, Lipták Z, Manzini G, Masillo F, Shivakumar V. 2025. Prefix-free parsing for merging big BWTs. In *International symposium on string processing and information retrieval (SPIRE)*, pp. 54–63. Springer.

Dotan E, Jaschek G, Pupko T, Belinkov Y. 2024. Effect of tokenization on transformers for biological sequences. *Bioinformatics* **40:** btae196. doi:10.1093/bioinformatics/btae196

Ferragina P, Manzini G. 2000. Opportunistic data structures with applications. In *The proceedings of the foundations of computer science (FOCS)*, Redondo Beach, CA, pp. 390–398.

Ferragina P, Manzini G. 2005. Indexing compressed text. *J ACM* **52:** 552–581. doi:10.1145/1082036.1082039

Ferro E, Oliva M, Gagie T, Boucher C. 2024. Building a pangenome alignment index via recursive prefix-free parsing. *iScience* **27:** 110933. doi:10.1016/j.isci.2024.110933

Fischer J. 2010. Optimal succinctness for range minimum queries. In *Proceedings of the Latin American symposium on theoretical informatics (LATIN)*, Oaxaca, Mexico, pp. 158–169.

Gage P. 1994. A new algorithm for data compression. *C Users J* **12:** 23–38.

Gagie T, Navarro G, Sakamoto H, Takabatake Y. 2019. Rpair: Rescaling rePair with Rsync. In *The proceedings of the symposium string processing and information retrieval (SPIRE)*, Segovia, Spain, pp. 35–44.

Gagie T, Navarro G, Prezza N. 2020. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J ACM* **67:** 1–54. doi:10.1145/3375890

Giuliani S, Romana G, Rossi M. 2022. Computing maximal unique matches with the r-index. In *Proceedings of the international symposium on experimental algorithms (SEA 2022)*, Heidelberg, Germany, pp. 22:1–22:16.

Gog S, Beller T, Moffat A, Petri M. 2014. From theory to practice: Plug and play with succinct data structures. In *International symposium on experimental algorithms*, pp. 326–337. Springer.

Guerrini V, Louza FA, Rosone G. 2020. Metagenomic analysis through the extended Burrows–Wheeler transform. *BMC Bioinform* **21**(Suppl. 8)**:** 299. doi:10.1186/s12859-020-03628-w

Downloaded from genome.cshlp.org on June 11, 2026 . Published by Cold Spring Harbor Laboratory Press

Hong A, Rossi M, Boucher C. 2023. LZ77 via prefix-free parsing. In *The proceedings of the symposium on algorithm engineering and experiments (ALENEX)*, Florence, Italy, pp. 123–134. SIAM.

Hong A, Oliva M, Köppl D, Bannai H, Boucher C, Gagie T. 2024. PFP-FM: An accelerated FM-index. *Algorithms Mol Biol* **19:** 15. doi:10.1186/s13015-024-00260-8

Huffman DA. 1952. A method for the construction of minimum-redundancy codes. *Proc IRE* **40:** 1098–1101. doi:10.1109/JRPROC.1952.273898

Ji Y, Zhou Z, Liu H, Davuluri RV. 2021. DNABERT: Pre-trained bidirectional encoder representations from transformers model for DNA-language in genome. *Bioinformatics* **37:** 2112–2120. doi:10.1093/bioinformatics/btab083

Karp RM, Rabin MO. 1987. Efficient randomized pattern-matching algorithms. *IBM J Res Dev* **31:** 249–260. doi:10.1147/rd.312.0249

Kasai T, Lee G, Arimura H, Arikawa S, Park K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the symposium on combinatorial pattern matching (CPM)*, Jerusalem, Israel, pp. 181–192.

Kim J, Varki R, Oliva M, Boucher C. 2024. Re2Pair: Increasing the scalability of repair by decreasing memory usage. In *Proceedings of the 32nd annual European symposium on algorithms (ESA)*, London, pp. 78:1–78:15.

Kuhnle A, Mun T, Boucher C, Gagie T, Langmead B, Manzini G. 2020. Efficient construction of a complete index for pan-genomics read alignment. *J Comput Biol* **27:** 500–513. doi:10.1089/cmb.2019.0309

Langmead B, Salzberg SL. 2012. Fast gapped-read alignment with Bowtie 2. *Nat Methods* **9:** 357–359. doi:10.1038/nmeth.1923

Langmead B, Trapnell C, Pop M, Salzberg SL. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* **10:** R25. doi:10.1186/gb-2009-10-3-r25

Larsson NJ, Moffat A. 2000. Off-line dictionary-based compression. *Proc IEEE* **88:** 1722–1732. doi:10.1109/5.892708

Li H. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997 [q-bio.GN]. doi:10.48550/arXiv.1303.3997

Lindsey LM, Pershing NL, Habib A, Dufault-Thompson K, Stephens WZ, Blaschke AJ, Jiang X, Sundar H. 2025. The impact of tokenizer selection in genomic language models. *Bioinformatics* **41:** btaf456. doi:10.1093/bioinformatics/btaf456

Lucà S, Masillo F, Lipták Z. 2025. Measuring genomic data with prefix-free parsing. *Comput Biol Chem* **122:** 108870. doi:10.1016/j.compbiolchem.2025.108870

Mäkinen V, Navarro G. 2005. Succinct suffix arrays based on run-length encoding. In *Annual symposium on combinatorial pattern matching*, pp. 45–56. Springer.

Mäkinen V, Navarro G, Sirén J, Välimäki N. 2010. Storage and retrieval of highly repetitive sequence collections. *J Comput Biol* **17:** 281–308. doi:10.1089/cmb.2009.0169

Manber U, Myers G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J Comput* **22:** 935–948. doi:10.1137/0222058

Mantaci S, Restivo A, Rosone G, Sciortino M. 2007. An extension of the Burrows–Wheeler transform. *Theor Comput Sci* **387:** 298–312. doi:10.1016/j.tcs.2007.07.014

McCreight EM. 1976. A space-economical suffix tree construction algorithm. *J ACM* **23:** 262–272. doi:10.1145/321941.321946

Mölder F, Jablonski KP, Letcher B, Hall MB, van Dyken PC, Tomkins-Tinch CH, Sochat V, Forster J, Vieira FG, Meesters C, et al. 2025. Sustainable data analysis with Snakemake. *F1000Research* **10:** 33. doi:10.12688/f1000research.29032.3

Navarro G, Romana G, Urbina C. 2025. Smallest suffixient sets as a repetitiveness measure. In *International symposium on string processing and information retrieval*, pp. 217–232. Springer.

Nguyen E, Poli M, Faizi M, Thomas A, Wornow M, Birch-Sykes C, Massaroli S, Patel A, Rabideau C, Bengio Y, et al. 2023. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution. *Adv Neural Inf Process Syst* **36:** 43177–43201.

Nishimoto T, Kanda S, Tabei Y. 2022. An optimal-time RLBWT construction in BWT-runs bounded space. In *The proceedings of the international colloquium on automata, languages, and programming (ICALP)*, Paris, pp. 99:1–99:20.

Oliva M, Cenzato D, Rossi M, Lipták Z, Gagie T, Boucher C. 2022. CSTs for terabyte-sized data. In *2022 data compression conference (DCC)*, pp. 93–102. IEEE, Piscataway, NJ.

Oliva M, Gagie T, Boucher C. 2023. Recursive prefix-free parsing for building big BWTs. In *2023 data compression conference (DCC)*, pp. 62–70. IEEE, Piscataway, NJ.

Rossi M, Oliva M, Langmead B, Gagie T, Boucher C. 2022. MONI: A pangenomic index for finding maximal exact matches. *J Comput Biol* **29:** 169–187. doi:10.1089/cmb.2021.0290

Sanabria M, Hirsch J, Joubert PM, Poetsch AR. 2024. DNA language model GROVER learns sequence context in the human genome. *Nat Mach Intell* **6:** 911–923. doi:10.1038/s42256-024-00872-0

Sennrich R, Haddow B, Birch A. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th annual meeting of the association for computational linguistics (Volume 1: Long papers)*, Berlin, pp. 1715–1725.

Seward J. 1996. *bzip2 and libbzip2*. http://www.bzip.org/.

Shivakumar VS, Langmead B. 2025. Mumemto: Efficient maximal matching across pangenomes. *Genome Biol* **26:** 169. doi:10.1186/s13059-025-03644-0

Tridgell A, Mackerras P. 1996. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University.

Turnbull C, Scott RH, Thomas E, Jones L, Murugaesu N, Pretty FB, Halai D, Baple E, Craig C, Hamblin A, et al. 2018. The 100 000 genomes project: Bringing whole genome sequencing to the NHS. *BMJ* **361:** k1687. doi:10.1136/bmj.k1687

Varki R, Rossi M, Ferro E, Oliva M, Garrison E, Langmead B, Boucher C. 2025. Accurate short-read alignment through *r*-index-based pangenome indexing. *Genome Res* **35:** 1609–1620. doi:10.1101/gr.279858.124

Weigel D, Mott R. 2009. The 1001 genomes project for *Arabidopsis thaliana*. *Genome Biol* **10:** 107. doi:10.1186/gb-2009-10-5-107

Weiner P. 1973. Linear pattern matching algorithms. In *The proceedings of the annual symposium on switching and automata theory (SWAT)*, Iowa City, IA, pp. 1–11. IEEE, Piscataway, NJ.

Weka. 2020. Genomics England: Improved scale and performance for on-premises cluster. Case study. https://www.weka.io/wp-content/uploads/files/2026/01/weka-genomics-england-case-study.pdf

Zakeri M, Brown NK, Ahmed OY, Gagie T, Langmead B. 2024. Movi: A fast and cache-efficient full-text pangenome index. *iScience* **27:** 111464. doi:10.1016/j.isci.2024.111464

Zakeri M, Brown NK, Gagie T, Langmead B. 2025. Movi 2: Fast and space-efficient queries on pangenomes. bioRxiv doi:10.1101/2025.10.16.682873

Zhang D, Zhang W, Zhao Y, Zhang J, He B, Qin C, Yao J. 2023. DNAGPT: A generalized pre-trained tool for versatile DNA sequence analysis tasks. arXiv:2307.05628 [q-bio.GN]. doi:10.48550/arXiv.2307.05628

Zhou Z, Ji Y, Li W, Dutta P, Davuluri R, Liu H. 2024. DNABERT-2: Efficient foundation model and benchmark for multi-species genome. In *Proceedings of the 12th international conference on learning representations (ICLR)*, Vienna.

Ziv J, Lempel A. 1977. A universal algorithm for sequential data compression. *IEEE Trans Inf Theory* **23:** 337–343. doi:10.1109/TIT.1977.1055714

Received July 25, 2025; accepted in revised form April 15, 2026.