Acceleration of FM-Index Queries Through Prefix-Free Parsing

Aaron Hong[1]
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Marco Oliva
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Dominik Köppl
Institut für Informatik, Univeristät Münster, Germany

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University, Japan

Christina Boucher
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

FM-indexes are a crucial data structure in DNA alignment, but searching with them usually takes at least one random access per character in the query pattern. Ferragina and Fischer [6] observed in 2007 that word-based indexes often use fewer random accesses than character-based indexes, and thus support faster searches. Since DNA lacks natural word-boundaries, however, it is necessary to parse it somehow before applying word-based FM-indexing. Last year, Deng et al. [4] proposed parsing genomic data by induced suffix sorting, and showed the resulting word-based FM-indexes support faster counting queries than standard FM-indexes when patterns are a few thousand characters or longer. In this paper we show that using prefix-free parsing – which takes parameters that let us tune the average length of the phrases – instead of induced suffix sorting, gives a significant speedup for patterns of only a few hundred characters. We implement our method and demonstrate it is between 3 and 18 times faster than competing methods on queries to GRCh38. And was consistently faster on queries made to 25,000, 50,000 and 100,000 SARS-CoV-2 genomes. Hence, it is very clear that our method accelerates the performance of count over all state-of-the-art methods with a minor increase in the memory.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases FM-index, pangenomics, scalability, word-based indexing, random access

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.13

Supplementary Material *Software (Source Code of PFP-FM)*: `https://github.com/marco-oliva/afm`

Funding *Aaron Hong*: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher
Marco Oliva: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher
Dominik Köppl: JSPS KAKENHI Grant Number JP21K17701, JP22H03551, and JP23H04378
Hideo Bannai: JSPS KAKENHI Grant Number JP20H04141

[1] Corresponding author

Christina Boucher: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher, NSF/SCH grant INT-2013998 to Christina Boucher, and NIH/NIAID grant R01AI14180 to Christina Boucher
Travis Gagie: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSERC grant RGPIN-07185-2020 to Travis Gagie, NSF/BIO grant DBI-2029552 to Christina Boucher

1 Introduction

The FM-index [5] is one of the most famous data structures in bioinformatics as it has been applied to countless applications in the analysis of biological data. Due to the long-term impact of this data structure, Burrows, Ferragina, and Manzini earned the 2022 ACM Paris Kanellakis Theory and Practice Award[2]. It is the data structure behind important read aligners – e.g., Bowtie [10] and BWA [11] – which take one or more reference genomes and build the FM-index for these genomes and use the resulting index to find short exact alignments between a set of reads and the reference(s) which then can be extended to approximate matches [10, 11]. Briefly, the FM-index consists of a sample of the suffix array (denoted as SA) and the Burrows–Wheeler transform (BWT) array. Given an input string S and a query pattern Q, `count` queries that answer the number of times the longest match of Q appears in S, can be efficiently supported using the BWT. To locate all of these occurrences the SA sample is needed. Hence, together the FM-index efficiently supports both `count` and `locate` queries. We mathematically define the SA and BWT in the next section.

There has been a plethora of research papers on reducing the size of the FM-index (see, e.g., [13, 9, 7]) and on speeding up queries. The basic query, `count`, returns the number of times a pattern Q appears in the indexed text S, but usually requires at least $|Q|$ random accesses to the BWT of S, which are usually much slower than the subsequent computations we perform on the information those accesses return. More specifically, a `count` query for Q use rank queries at $|Q|$ positions in the BWT; if we answer these using a single wavelet tree for the whole BWT, then we may use a random access for every level we descend in the wavelet tree, or $\Omega(|Q| \log \sigma)$ random access in all, where σ is the size of the alphabet; if we break the BWT into blocks and use a separate wavelet tree for each block [9], we may need only one or a few random accesses per rank query, but the total number of random accesses is still likely to be $\Omega(|Q|)$. As far back as 2007, Ferragina and Fischer [6] addressed compressed indexes' reliance on random access and demonstrated that word-based indexes perform fewer random accesses than character-based indexes: *"The space reduction of the final word-based suffix array impacts also in their query time (i.e. less random access binary-search steps!), being faster by a factor of up to 3."*

Thus, one possibility of accelerating the random access to genomic data – where it is widely used – is to break up the sequences into words or phrases. In light of this insight, Deng et al. [4] in 2022 applied a grammar [18] that factorizes S into phrases based on the leftmost S-type suffixes (LMS) [17]. Unfortunately, one round of that LMS parsing leads to phrases that are generally too short, so they obtained speedup only when Q was thousands of characters. The open problem was how to control the length of phrases with respect to the input to get longer phrases that would enable larger advances in the acceleration of the random access.

Here, we apply the concept of prefix-free parsing to the problem of accelerating `count` in the FM-index. Prefix-free parsing uses a rolling hash to first select a set of strings (referred to as *trigger strings*) that are used to define a parse of the input string S; i.e., the prefix-free

[2] `https://awards.acm.org/kanellakis`

parse is a parsing of S into phrases that begin and end at a trigger string and contain no other trigger string. All unique phrases are lexicographically sorted and stored in the dictionary of the prefix-free parse, which we denote as D. The prefix-free parse can be stored as an ordered list of the phrases' ranks in D. Hence, prefix-free parsing breaks up the input sequence into phrases, whose size is more controllable by the selection of the trigger strings. This leads to a more flexible acceleration than Deng et al. [4] obtained.

We assume that we have an input string S of length n. Now suppose we build an FM-index S, an FM-index for the parse P, and a bitvector B of length n with `1`'s marking characters in the BWT of S that immediately precede phrase boundaries in S, i.e., that immediately precede a trigger string. We note that all the 1s are bunched into at most as many runs as there are distinct trigger strings in S. Also, as long as the ranks of the phrases are in the same lexicographic order as the phrases themselves, we can use the bitvector to map from the interval in the BWT of S for any pattern starting with a trigger string to the corresponding interval in the BWT of P, and vice versa. This means that, given a query pattern Q, we can backward search for Q character by character in the FM-index for S until we hit the left end of the rightmost trigger string in Q, then map into the BWT of P and backward search for Q phrase by phrase until we hit the left end of the leftmost trigger string in Q, then map back into the BWT of S and finish backward searching character by characters again.

We implement this method, which we refer to as `PFP-FM`, and extensively compare against the FM-index implementation in `sdsl` [8], `RLCSA` [19], `RLFM` [13, 12], and `FIGISS` [4] using sets of SARS-CoV-2 genomes taken from the NCBI website, and the Genome Reference Consortium Human Build 38 with varying query string lengths. When we compare `PFP-FM` to FM-index in `sdsl` using 100,000 SARS-CoV-2 genomes, we witnessed that `PFP-FM` was able to perform between 2.1 and 2.8 more queries. In addition, `PFP-FM` was between 64.38% and 74.12%, 59.22% and 78.23%, and 49.10% and 90.70% faster than `FIGISS`, `RLCSA`, and `RLFM`, respectively on 100,000 SARS-CoV-2 genomes. We evaluated the performance of `PFP-FM` on the Genome Reference Consortium Human Build 38, and witnessed that it was between 3.86 and 7.07, 2.92 and 18.07, and 10.14 and 25.46 times faster than `RLCSA`, `RLFM`, and `FIGISS`, respectively. With respect to construction time, `PFP-FM` had the most efficient construction time for all SARS-CoV-2 datasets and was the second fastest for Genome Reference Consortium Human Build 38. All methods used less than 60 GB for memory for construction on the SARS-CoV-2 datasets, making the construction feasible on any entry level commodity server – even the build for the 100,000 SARS-CoV-2 dataset. Construction for the Genome Reference Consortium Human Build 38 required between 26 GB and 71 GB for all methods, with our method using the most memory. In summary, we develop and implement a method for accelerating the FM-index, and achieve an acceleration between 2 and 25 times, with the greatest acceleration witnessed with longer patterns. Thus, accelerated FM-index methods – such as the one developed in this paper – are highly applicable to finding very long matches (125 to 1,000 in length) between query sequences and reference databases. As reads get longer and more accurate (i.e., Nanopore data), we will soon be prepared align long reads to reference databases with efficiency that surpasses traditional FM-index based alignment methods. The source code is publicly available at `https://github.com/marco-oliva/afm`.

2 Preliminaries

2.1 Basic Definitions

A string S of length n is a finite sequence of symbols $S = S[0..n-1] = S[0] \cdots S[n-1]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$. We assume that the symbols can be unambiguously ordered. We denote by ε the empty string, and the length of S as $|S|$. Given a string S, we denote the reverse of S as $rev(S)$, i.e., $rev(S) = S[n-1] \cdots S[0]$.

We denote by $S[i..j]$ the substring $S[i] \cdots S[j]$ of S starting in position i and ending in position j, with $S[i..j] = \varepsilon$ if $i > j$. For a string S and $0 \leq i < n$, $S[0..i]$ is called the i-th prefix of S, and $S[i..n-1]$ is called the i-th suffix of S. We call a prefix $S[0..i]$ of S a *proper prefix* if $0 \leq i < n-1$. Similarly, we call a suffix $S[i..n-1]$ of S a *proper suffix* if $0 < i < n$.

Given a string S, a symbol $c \in \Sigma$, and an integer i, we define $S.\mathsf{rank}_c(i)$ (or simply `rank` if the context is clear) as the number of occurrences of c in $S[0..i-1]$. We also define $S.\mathsf{select}_c(i)$ as $\min(\{j-1 \mid S.\mathsf{rank}_c(j) = i\} \cup \{n\})$, i.e., the position in S of the i-th occurrence of c in S if it exists, and n otherwise. For a bitvector $B[0..n-1]$, that is a string over $\Sigma = \{0,1\}$, to ease the notation we will refer to $B.\mathsf{rank}_1(i)$ and $B.\mathsf{select}_1(i)$ as $B.\mathsf{rank}(i)$ and $B.\mathsf{select}(i)$, respectively.

2.2 SA, BWT, and Backward Search

We denote the *suffix array* [14] of a given a string $S[0..n-1]$ as SA_S, and define it to be the permutation of $\{0,\ldots,n-1\}$ such that $S[\mathsf{SA}_S[i]..n-1]$ is the i-th lexicographical smallest suffix of S. We refer to SA_S as SA when it is clear from the context. For technical reasons, we assume that the last symbol of the input string is $S[n-1] = \$$, which does not occur anywhere else in the string and is smaller than any other symbol.

We consider the matrix W containing all sorted rotations of S, called the BWT matrix of S, and let F and L be the first and the last column of the matrix. The last column defines the BWT array, i.e., $\mathsf{BWT} = \mathsf{L}$. Now let $\mathsf{C}[c]$ be the number of suffixes starting with a character smaller than c. We define the LF-mapping as $\mathsf{LF}(i,c) = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(i)$ and $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. With the LF-mapping, it is possible to reconstruct the string S from its BWT. It is in fact sufficient to set an iterator $s = 0$ and $S[n-1] = \$$ and for each $i = n-2,\ldots,0$ do $S[i] = \mathsf{BWT}[s]$ and $s = \mathsf{LF}(s)$. The LF-mapping can also be used to support `count` by performing the backward search, which we now describe.

Given a query pattern Q of length m, the *backward search* algorithm consists of m steps that preserve the following invariant: at the i-th step, p stores the position of the first row of W prefixed by $Q[i,m]$ while q stores the position of the last row of W prefixed by $Q[i,m]$. To advance from i to $i-1$, we use the LF-mapping on p and q, $p = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(p)$ and $q = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(q+1) - 1$.

2.3 FM-index and count Queries

Given a query string $Q[0..m-1]$ and an input string $S[0..n-1]$, two fundamental queries are: (1) `count` which counts the number of of occurrences of Q in S; (2) `locate` which finds the location of each of these matches in S. Ferragina and Manzini [5] showed that, by combining SA with the BWT, both `count` and `locate` can be efficiently supported. Briefly, backward search on the BWT is used to find the lexicographical range of the occurrences of Q in S; the size of this range is equal to `count`. The SA positions within this range are the positions where these occurrences are in S.

2.4 Prefix-Free Parsing

As we previously mentioned, the *Prefix-Free Parsing* (PFP) takes as input a string $S[0..n-1]$, and positive integers w and p, and produces a parse of S (denoted as P) and a dictionary (denoted as D) of all the unique substrings (or phrases) of the parse. We note that w defines the length of the trigger strings and p is used in the rolling-hash. We briefly go over the algorithm for producing this dictionary and parse. First, we assume there exists two symbols,

say # and \$, which are not contained in Σ and are lexicographically smaller than any symbol in Σ. Next, we let T be an arbitrary set of w-length strings over Σ and call it the set of *trigger strings*. As mentioned before, we assume that $S[n-1] = \$$ and consider S to be cyclic, i.e., for all i, $S[i] = S[i \bmod n]$. Furthermore, we assume that $\$S[0..w-2] = S[n-1..n+w-2] \in T$, i.e., the substring of length w that begins with \$ is a trigger string.

We let the dictionary $\mathsf{D} = \{d_1, .., d_{|\mathsf{D}|}\}$ be a (lexicographically sorted) maximum set of substrings of S such that the following holds for each d_i: i) exactly one proper prefix of d_i is contained in T, ii) exactly one proper suffix of d_i is contained in T, iii) and no other substring of d_i is in T. These properties allow for the SA and BWT to be constructed since the lexicographical placement of each rotation of the input string can be indentified unambiguously from D and P [2, 3, 16]. An important consequence of the definition is that D is prefix-free, i.e., for any $i \neq j$, d_i cannot be a prefix of d_j.

Since we assumed $S[n-1..n+w-2] \in T$, we can construct D by scanning $S' = \$S[0..n-2]S[n-1..n+w-2]$ to find all occurrences of T and adding to D each substring of S' that starts and ends at a trigger string being inclusive of the starting and ending trigger string. We can also construct the list of occurrences of D in S', which defines the parse P.

We choose T by a Karp-Rabin fingerprint f of strings of length w. We slide a window of length w over S', and for each length w substring r of S', include r in T if and only if $f(r) \equiv 0 \pmod p$ or $r = S[n-1..n+w-2]$. Let $0 = s_0 < \cdots < s_{k-1}$ be the positions in S' such that for any $0 \leq i < k$, $S'[s_i..s_i+w-1] \in T$. The dictionary is $\mathsf{D} = \{S'[s_i..s_{i+1}+w-1] \mid i = 0, \ldots, k-1\}$, and the parse is defined to be the sequence of lexicographic ranks in D of the substrings $S'[s_0..s_1+w-1], \ldots, S'[s_{k-2}..s_{k-1}+w-1]$.

As an example, suppose we have $S' =$ \$AGACGACT#AGATACT#AGATTCGAGACGAC\$A, where the trigger strings are highlighted in red, blue, or green. It follows that we have $\mathsf{D} =$ {\$AGAC, AC\$A, ACGAC, ACT#AGATAC, ACT#AGATTC, TCGAGAC} and $\mathsf{P} = 0, 2, 3, 4, 5, 2, 1$.

3 Methods

As we previously mentioned, we will use prefix-free parsing to build a word-based FM-index in a manner in which the length of the phrases can be controlled via the parameters w and p. To explain our data structure, we first describe the various components of our data structure, and then follow with describing how to support `count` queries in a manner that is more efficient than the standard FM-index.

3.1 Data Structure Design

It is easiest to explain our two-level design with an example, so consider a text

$$S[0..n-1] = \texttt{TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT\$}$$

of length $n = 41$ that is terminated by a special end-of-string character `$` lexicographically less than the rest of the alphabet. Suppose we parse S using $w = 2$ and a Karp-Rabin hash function such that the normal trigger strings occurring in S are `AA`, `CG` and `TA`. We consider S as cyclic, and we have $\$S[0..w-2] =$ \$T as a special trigger string, so the the dictionary D is

$$\mathsf{D}[0..5] = \{\texttt{\$TCCAGAA}, \texttt{AAGACATA}, \texttt{AAGAGTA}, \texttt{CGACATGTTGAA}, \texttt{TATCTCCTCG}, \texttt{TATGAT\$T}\},$$

with the phrases sorted in lexicographic order. (Recall that phrases consecutive in S overlap by $w = 2$ characters.) If we start parsing at the `$`, then the prefix-free parse for S is

$$\mathsf{P}[0..5] = (0, 2, 4, 3, 1, 5),$$

where each element (or phrase ID) in P is the lexicographic rank of the phrase in D.

0	2	4	3	1	5	$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT
1	5	0	2	4	3	AAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTG
2	4	3	1	5	0	AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG
3	1	5	0	2	4	CGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCT
4	3	1	5	0	2	TATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAG
5	0	2	4	3	1	TATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA

Figure 1 The BWT matrix for our prefix-free parse P **(left)** and the cyclic shifts of S that start with a trigger string **(right)**, in lexicographic order.

Next, we consider the BWT matrix for P. Figure 1 illustrates the BWT matrix of P for our example. We note that since there is only one $ in S, it follows that there is only one 0 in P; we can regard this 0 as the end-of-string character for (a suitable rotation of) P corresponding to $ in S. If we take the i-th row of this matrix and replace the phrase IDs by the phrases themselves, collapsing overlaps, then we get the lexicographically i-th cyclic shift of S that start with a trigger string, as shown on the right of the figure. This is one of the key insights that we will use later on.

▶ **Lemma 1.** *The lexicographic order of rotations of* P *correspond to the lexicographic order of their corresponding rotations of* S*.*

Proof. The characters of P are the phrase IDs that act as meta-characters. Since the meta-characters inherit the lexicographic rank of their underlying characters, and due to the prefix-freeness of the phrases, the suffix array of P permutes the meta-characters of P in the same way as the suffix array of S permutes the phrases of S. This means that the order of the phrases in the BWT of S is the same as the order of the phrase IDs in P. ◀

Next, we let $B[0..n-1]$ be a bitvector marking these cyclic shifts' lexicographic rank among all cyclic shifts of S, i.e., where they are among the rows of the BWT matrix of S. Figure 2 shows the SA, BWT matrix and BWT of S, together with B; we highlight the BWT – the last column of the matrix – and the cyclic shifts from Figure 1 in red. We note that B contains at most one run of 1's for each distinct trigger string in S so it is usually highly run-length compressible in practice.

In addition to the bitvector, we store a hash function h on phrases and a map M from the hashes of the phrases in D to those phrases' lexicographic ranks, which are their phrase IDs; M returns NULL when given any other key. Therefore, in total, we build the FM-index for S, the FM-index for P, the bitvector B marking the cyclic rotations, the hash function h on the phrases and the map M. For our example, suppose

$$\begin{aligned} h(\texttt{\$TCCAGAA}) &= 91785 & M(91785) &= 0 \\ h(\texttt{AAGACATA}) &= 34865 & M(34865) &= 1 \\ h(\texttt{AAGAGTA}) &= 49428 & M(49428) &= 2 \\ h(\texttt{CGACATGTTGAA}) &= 98759 & M(98759) &= 3 \\ h(\texttt{TATCTCCTCG}) &= 37298 & M(37298) &= 4 \\ h(\texttt{TATGAT\$T}) &= 68764 & M(68764) &= 5 \end{aligned}$$

and $M(x) = \text{NULL}$ for any other value of x.

If we choose the range of h to be reasonably large then we can still store M in space proportional to the number of phrases in D with a reasonably constant coefficient and evaluate $M(h(\cdot))$ in constant time with high probability, but the probability is negligible that $M(h(\gamma)) \neq \text{NULL}$ for any particular string γ not in D. This means that in practice we can use $M(h(\cdot))$ as a membership dictionary for D, and not store D itself.

i	SA$[i]$	$B[i]$	T[SA$[i]$..(SA$[i]-1$) mod n]	BWT$[i]$
0	40	1	$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT	T
1	28	1	AAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTG	G
2	5	1	AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG	G
3	31	0	ACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAG	G
4	20	0	ACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCG	G
5	3	0	AGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCC	C
6	29	0	AGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGA	A
7	6	0	AGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGA	A
8	8	0	AGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAG	G
9	38	0	AT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATG	G
10	33	0	ATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGAC	C
11	11	0	ATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGT	T
12	35	0	ATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACAT	T
13	22	0	ATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGAC	C
14	2	0	CAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TC	C
15	32	0	CATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGA	A
16	21	0	CATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGA	A
17	1	0	CCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$T	T
18	15	0	CCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCT	T
19	18	1	CGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCT	T
20	13	0	CTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTAT	T
21	16	0	CTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTC	C
22	27	0	GAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTT	T
23	4	0	GAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCA	A
24	30	0	GACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAA	A
25	19	0	GACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTC	C
26	7	0	GAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAA	A
27	37	0	GAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATAT	T
28	9	0	GTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGA	A
29	24	0	GTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACAT	T
30	39	0	T$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGA	A
31	10	1	TATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAG	G
32	34	1	TATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA	A
33	0	0	TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$	$
34	14	0	TCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATC	C
35	17	0	TCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCC	C
36	12	0	TCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTA	A
37	26	0	TGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGT	T
38	36	0	TGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATA	A
39	23	0	TGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACA	A
40	25	0	TTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATG	G

Figure 2 The SA, BWT matrix and BWT of T, together with the bitvector B in which 1s indicate rows of the matrix starting with trigger strings. The BWT is highlighted in red, as are the columns marked by 1s.

3.2 Query Support

Next, given the data structure that we define above, we describe how to support `count` queries for a given pattern Q. We begin by parsing Q using the same Karp-Rabin hash we used to parse S, implying that we will have all the same trigger strings as we did before and possibly additional ones that did not occur in S. However, we will not consider Q to be cyclic nor assume an end-of-string symbol that would assure that Q starts and ends with a trigger string.

If Q is a substring of S, then, since Q contains the same trigger strings as its corresponding occurrence in S, the sequence of phrases induced by the trigger strings in Q must be a substring of the sequence of phrases of S. Together with the prefix and suffix of Q that are a suffix and prefix of the phrases in S to the left and right of the shared phrases, we call this the partial encoding of Q, defined formally as follows.

▶ **Definition 2** (partial encoding). *Given a substring $S[i..j]$ of S, the* partial encoding *of $S[i..j]$ is defined as follows: If no trigger string occurs in $S[i..j]$, then the partial encoding of $S[i..j]$ is simply $S[i..j]$ itself. Otherwise, the partial encoding of $S[i..j]$ is the concatenation of: (1) the shortest prefix α of $S[i..j]$ that does not start with a trigger string and ends with a trigger string, followed by (2) the sequence of phrase IDs of phrases completely contained in $S[i..j]$, followed by (3) the shortest suffix β of $S[i..j]$ that begins with a trigger string and does not end with a trigger string.*

So the partial encoding partitions $S[i..j]$ into a prefix α, a list of phrase IDs, and a suffix β. If $S[i..j]$ begins (respectively ends) with a trigger string, then α (respectively β) is the empty string.

Parsing Q can be done in time linear in the length of Q.

▶ **Lemma 3.** *We can represent M with a data structure taking space (in words) proportional to the number of distinct phrases in D. Given a query pattern Q, this data structure returns NULL with high probability if Q contains a complete phrase that does not occur in S. Otherwise (complete phrases of Q occur in S), it returns the partial encoding of Q. In either case, this query takes $O(|Q|)$ time.*

Proof. We keep the Karp-Rabin (KR) hashes of the phrases in D, with the range of the KR hash function mapping to $[1..n^3]$ so the hashes each fit in $O(\log n)$ bits. We also keep a constant-time map (implemented as a hash table with a hash function that's perfect for the phrases in D) from the KR hashes of the phrases in D to their IDs, that returns NULL given any value that is not a KR hash of a phrase in D. We set M to be the map composed with the KR hash function.

Given Q, we scan it to find the trigger strings in it, and convert it into a sequence of substrings consisting of: (a) the prefix α of Q ending at the right end of the first trigger string in Q; (b) a sequence of PFP phrases, each starting and ending with a trigger string with no trigger string in between; and (c) the suffix β of Q starting at the left end of the last trigger string in Q.

We apply M to every complete phrase in (b). If M returns NULL for any complete phrase in (b), then that phrase does not occur in S, so we return NULL; otherwise, we return α, the sequence of phrase IDs M returned for the phrases in (b), and β.

Notice that, if a phrase in Q is in S, then M will map it to its lexicographic rank in D; otherwise, the probability the KR hash of any particular phrase in Q but not in D collides with the KR hash of a phrase in D, is at most $n/n^3 = 1/n^2$. It follows that, if Q contains a complete phrase that does not occur in S, then we return NULL with high probability; otherwise, we return Q's partial encoding. ◀

▶ **Corollary 4.** *If we allow $O(|Q|)$ query time with high probability, then we can modify M to always report NULL when Q contains a complete phrase not in S.*

Proof. We augment each Karp-Rabin (KR) hash stored in the hash table with the actual characters of its phrase such that we can check, character by character, whether a matched phrase of Q is indeed in D. In case of a collision we recompute the KR hashes of D and rebuild the hash table. That is possible since we are free to choose different Karp-Rabin fingerprints for the phrases in D. ◀

Continuing from our example above where the trigger strings are `AA`, `CG` and `TA`, suppose we have a query pattern Q,

$Q[0..34] =$ `CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT`

we can compute the parse of Q to obtain the following

`CAGAA`, `AAGAGTA`, `TATCTCCTCG`, `CGACATGTTGAA`, `AAGACATA`, `TAT`.

Next, we use $M(h(\cdot))$ to map the complete phrases of this parse of Q to their phrase IDs – which is their rank in D. If any complete phrase maps to NULL then we know Q does not occur in T. Using our example, we have the partial encoding

`CAGAA`, 2, 4, 3, 1, `TAT`.

Next, we consider all possible cases. First, we consider the case that the last substring β in our parse of Q ends with a trigger string, which implies that it is a complete phrase. Here, we can immediately start backward searching for the parse of Q in the FM-index for P. Next, if β is not a complete phrase then we backward search for β in the FM-index for S. If this backward search for β returns nothing then we know Q does not occur in S. If the backward search for β returns an interval in the BWT of P that is not contained in the BWT interval for a trigger string then β does not start with a trigger string so $Q = \beta$ and we are done backward searching for Q.

Finally, we consider the case when β is a proper prefix of a phrase and the backward search for β returns a BWT_S interval contained in the BWT_S interval for a trigger string. In our example, $\beta =$ `TAT` and our backward search for β in the FM-index for S returns the interval $\mathsf{BWT}_S[31..32]$, which is the interval for the trigger string `TA`. Next, we use B to map the interval for β in the BWT_S to the interval in the BWT_P that corresponds to the cyclic shifts of S starting with β.

▶ **Lemma 5.** *We can store in space (in words) proportional to the number of distinct trigger strings in S a data structure B with which,*

- *given the lexicographic range of suffixes of S starting with a string β such that β starts with a trigger string and contains no other trigger string, in $O(\log\log n)$ time we can find the lexicographic range of suffixes of P starting with phrases that start with β;*
- *given a lexicographic range of suffixes of P such that the corresponding suffixes of S all start with the same trigger string, in $O(\log\log n)$ time we can find the lexicographic range of those corresponding suffixes of S.*

Proof. Let $B[0..n-1]$ be a bitvector with 1s marking the lexicographic ranks of suffixes of S starting with trigger strings. There are at most as many runs of 1s in B as there are distinct trigger strings in S, so we can store it in space proportional to that number and support rank and select operations on it in $O(\log\log n)$ time.

If $\mathsf{BWT}_S[i..j]$ contains the characters immediately preceding, in S, occurrences of a string β that starts with a trigger string and contains no other trigger strings, then $\mathsf{BWT}_\mathsf{P}[B.\mathsf{rank}_1(i)..B.\mathsf{rank}_1(j)]$ contains the phrase IDs immediately preceding, in P, the IDs of phrases starting with β.

If $\mathsf{BWT}_\mathsf{P}[i..j]$ contains the phrase IDs immediately preceding, in P, suffixes of P such that the corresponding suffixes of S all start with the same trigger string, then $\mathsf{BWT}_S[B.\mathsf{select}_1(i+1)..B.\mathsf{select}_1(j+1)]$ contains the characters immediately preceding the corresponding suffixes of S.

The correctness follows from Lemma 1. ◀

Continuing with our example mapping $\mathsf{BWT}_S[31..32]$ yield the following interval:

$$\mathsf{BWT}_\mathsf{P}[B.\mathsf{rank}_1(31), B.\mathsf{rank}_1(32)] = \mathsf{BWT}_\mathsf{P}[4..5]$$

as shown in Figure 1. Starting from this interval in BWT_P, we now backward search in the FM-index for P for the sequence of complete phrase IDs in the parse of Q. In our example, we have the interval $\mathsf{BWT}_\mathsf{P}[4..5]$ which yields the following phrase IDs: 2 4 3 1.

If this backward search in the FM-index for P returns nothing, then we know Q does not occur in S. Otherwise, it returns the interval in BWT_P corresponding to cyclic shifts of S starting with the suffix of Q that starts with Q's first complete phrase. In our example, if we start with $\mathsf{BWT}_\mathsf{P}[4..5]$ and backward search for 2 4 3 1 then we obtain $\mathsf{BWT}_\mathsf{P}[2]$, which corresponds to the cyclic shift

`AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG`

of S that starts with the suffix

`AAGAGTATCTCCTCGACATGTTGAAGACATAT`

of Q that is parsed into $2, 4, 3, 1,$ `TAT`.

To finish our search for Q, we use B to map the interval in BWT_P to the corresponding interval in the BWT_S, which is the interval of rows in the BWT matrix for S which start with the suffix of Q we have sought so far. In our example, we have that $\mathsf{BWT}_\mathsf{P}[2]$ maps to

$$\mathsf{BWT}_S[B.\mathsf{select}_1(2+1)] = \mathsf{BWT}_S[2].$$

We note that our examples contain BWT intervals with only one entry because our example is so small, but in general they are longer. If the first substring α in our parse of Q is a complete phrase then we are done backward searching for Q. Otherwise, we start with this interval in BWT_S and backward search for α in the FM-index for S, except that we ignore the last w last characters of α (which we have already sought, as they are also contained in the next phrase in the parse of Q).

In our example, $\alpha =$ `CAGAA` so, starting with $\mathsf{BWT}_S[2]$ we backward search for `CAG`, which returns the interval $\mathsf{BWT}_S[14]$. As shown in Figure 2,

$$S[\mathsf{SA}[4]..n] = S[2..n] = \texttt{CAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT\$}$$

does indeed start with

$$Q = \texttt{CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT}\,.$$

This concludes our explanation of `count`.

To conclude, we give some intuition as to why we expect our two-level FM-index to be faster in practice than standard backward search. First, we note that standard backward search takes linear time in the length of Q and usually uses at least one random access per character in Q. Whereas, prefix-free parsing Q takes linear time but does not use random access; backward search in the FM-index of S is the same as standard backward search but we use it only for the first and last substrings in the parse of Q. Backward search in the FM-index for P is likely to use about $\lg |\mathsf{D}|$ random access for each complete phrase in the parse of Q: the BWT of P is over an effective alphabet whose size is the number of phrases in D. Therefore, a balanced wavelet tree to support rank on that BWT should have depth about $\lg |\mathsf{D}|$ and we should use at most about one random access for each level in the tree.

In summary, if we can find settings of the prefix-free parsing parameters w and p such that

- most query patterns will span several phrases,
- most phrases in those patterns are fairly long,
- $\lg |\mathsf{D}|$ is significantly smaller than those phrases' average length,

then the extra cost of parsing Q should be more than offset by using fewer random accesses.

4 Results

We implemented our algorithm and measured its performance against all known competing methods. We ran all experiments on a server with AMD EPYC 75F3 CPU with the Red Hat Enterprise Linux 7.7 (64bit, kernel 3.10.0). The compiler was g++ version 12.2.0. The running time and memory usage was recorded by SnakeMake benchmark facility [15]. We set a memory limitation of 128 GB of memory and a time limitation of 24 hours.

Datasets. We used the following datasets. First, we considered sets of SARS-CoV-2 genomes taken from the NCBI website. We used three collections of $25,000$, $50,000$, and $100,000$ SARS-CoV-2 genomes from EMBL-EBI's COVID-19 data portal. Each collection is a superset of the previous. We denote these as `SARS-25k`, and `SARS-50k`, `SARS-100k`. Next, we considered a single human reference genome, which we denote as `GRCh38`, downloaded from NCBI. We report the size of the datasets as the number of characters in each in Table 1. We denote n as the number of characters.

Implementation. We implemented our method in C++ 11 using the `sdsl-lite` library [8] and extended the prefix-free parsing method of Oliva, whose source code is publicly available here `https://github.com/marco-oliva/pfp`. The source code for `PFP-FM` is available at `https://github.com/marco-oliva/afm`.

Competing methods. We compared `PFP-FM` against the following methods the standard FM-index found in `sdsl-lite` library [8], `RLCSA` [19], `RLFM` [13, 12], and `FIGISS` [4]. We note that `RLCSA` and `FIGISS` have publicly-available source codes, while `RLFM` is provided only as an executable. We performed the comparison by selecting 1,000 strings from the genome file at random of the specified length, performing the `count` operation on each query pattern, and measuring the time usage for all the methods under consideration. It is worth noting that `FIGISS` and `RLCSA` only support `count` queries where the string is provided in an input text file. More specifically, the original `FIGISS` implementation supports counting with the entire content of a file treated as a single pattern. To overcome this limitation, we modified the source code to enable the processing of multiple query patterns within a single file. In addition to the time consideration for `count`, we measured the time and memory required to construct the data structure.

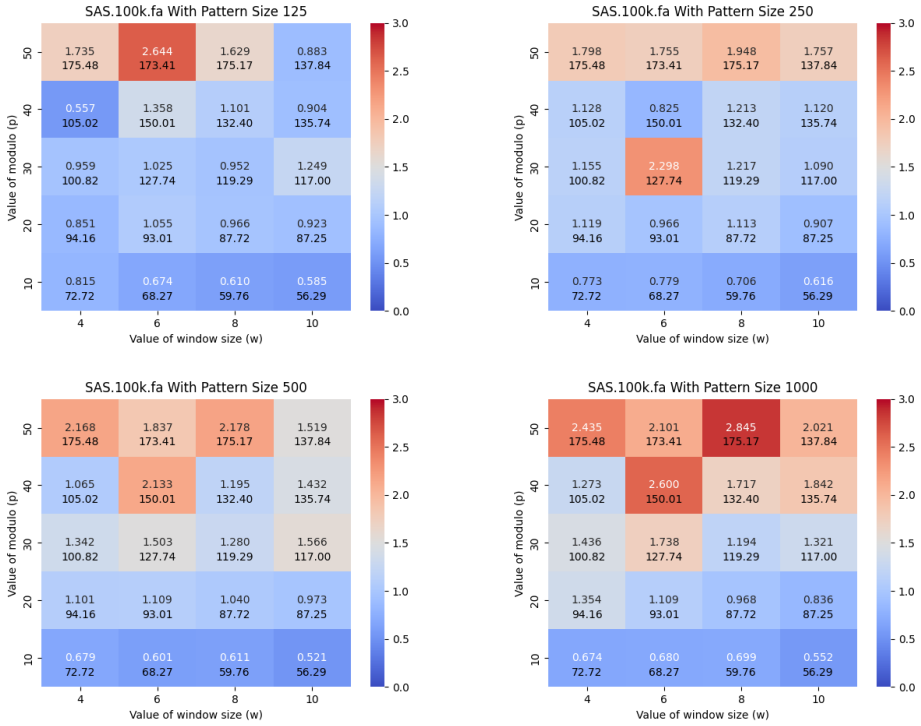

Figure 3 Illustration of the impact of w, p and the length of the query pattern on the acceleration of the FM-index. Here, we used `SARS-100K` dataset and varied the length of the query pattern to be equal to 125, 250, 500, and 1000. The y-axis corresponds to p and the x-axis corresponds to w. The heatmap illustrates the number of queries that can be performed in a CPU second with the acceleration versus the standard FM-index from `sdsl`, i.e., `PFP-FM` / `sdsl`. The second value in each block represents the average length of phrases.

4.1 Acceleration versus Baseline

In this subsection, we compare `PFP-FM` versus the standard FM-index in `sdsl` with varying values of window size (w) and modulo value (p), and varying the length of the query pattern. We calculated the number of `count` queries that were able to be performed in CPU second with `PFP-FM` versus the standard FM-index. We generated heatmaps that illustrate the number of `count` queries of `PFP-FM` verses `sdsl` for various lengths of query patterns, namely, 125, 250, 500, and 1,000. We performed this for each SARS-CoV-2 set of genomes. Figure 3 shows the resulting heatmaps for `SARS-100K`. As shown in this figure, `PFP-FM` was between 2.178 and 2.845 times faster than the standard FM-index with the optimal values of w and p. In particular, an optimal performance gain of 2.6, 2.3, 2.2, and 2.9 was witnessed for pattern lengths of 125, 250, 500, and 1,000, respectively. The (w, p) pairs that correspond to these results are $(6, 50)$, $(6, 30)$, $(8, 50)$, and $(8, 50)$.

4.2 Results on SARS-CoV-2 Genomes

We used the optimal parameters that were obtained from the previous experiment for this section. We constructed the index using these parameters for each SARS-CoV-2 dataset and assessed the time consumption for performing 1,000 `count` queries using all competing methods and `PFP-FM`. We illustrate the result of this experiment in Figure 4. It is clear

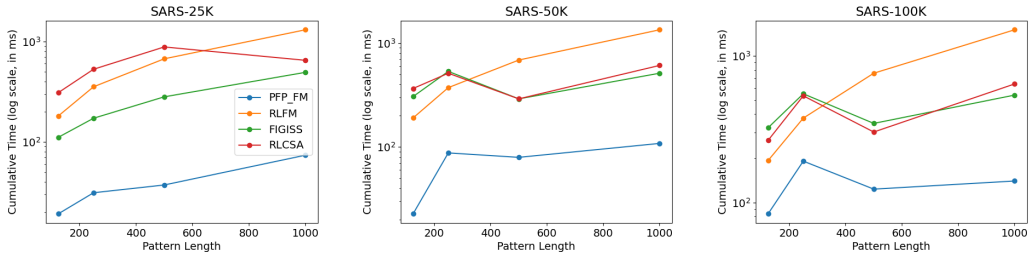

Figure 4 Illustration of the impact of the dataset size, and the length of the query pattern on the query time for answering `count`. We vary the length of the query pattern to be equal to 125, 250, 500, and 1000, and report the times for `SARS-25K`, `SARS-50K`, and `SARS-100K`. We illustrate the cumulative time required to perform 1,000 `count` queries. The y-axis is in log scale.

from this `PFP-FM` consistently exhibits the lowest time consumption and a gradual, stable trend. For the `SARS-25K` dataset, the time consumption of `FIGISS` was between 451% and 568% higher than our method. And the time consumption of `RLCSA` and `RLFM` was between 780% and 1598%, and 842% and 1705% more than `PFP-FM`, respectively. The performance of `FIGISS` surpasses that of `RLFM` and `RLCSA` when using the SARS-25k dataset; however for the larger datasets `FIGISS` and `RLCSA` converge in their performance .Neither method was substantially better than the other. In addition, on the larger datasets, when the query pattern length was 125 and 250, `RLFM` performed better than `RLCSA` and `FIGISS` but was slower for the other query lengths. Hence, it is very clear that `PFP-FM` accelerates the performance of `count` over all state-of-the-art methods.

The gap in performance between `PFP-FM` and the competing methods increased with the dataset size. For `SARS-50K`, `FIGISS`, `RLCSA` and `RLFM` were between 3.65 and 13.44, 3.65 and 16.08, and 4.25 and 12.39 times slower, respectively. For `SARS-100K`, `FIGISS`, `RLCSA` and `RLFM` were between 2.81 and 3.86, 2.45 and 4.59, and 1.96 and 10.75 times slower, respectively.

Next, we consider the time and memory required for construction; which is given in Table 1. Our experiments revealed that all methods used less than 60 GB of memory on all SARS-CoV-2 datasets; `PFP-FM` used the most memory with the peak being 54 GB on the `SARS-100K` dataset. Yet, `PFP-FM` exhibited the most efficient construction time across all datasets for generating the FM-index, and this gap in the time grew with the size of the dataset. More specifically, for the `SARS-100K` dataset, `PFP-FM` used 71.04%, 65.81%, and 73.41% less time compared to other methods. In summary, `PFP-FM` significantly accelerated the `count` time, and had the fastest construction time. All methods used less than 60 GB, which is available on most commodity servers.

4.3 Results on Human Reference Genome

After measuring the time and memory usage required to construct the data structure across all methods using the `GRCh38` dataset, we observed that `PFP-FM` exhibited the second most efficient construction time but used the most construction space (71 GB vs. 26 GB to 45 GB). More specifically, `PFP-FM` was able to construct the index between 1.25 and 1.6 times faster than the `FIGISS` and `RLFM`.

Next, we compare the performance of `PFP-FM` against other methods by performing 1,000 `count` queries on, and illustrate the results in Figure 5. Our findings demonstrate that `PFP-FM` consistently outperforms all other methods. Although `RLCSA` shows better performance than `RLFM` and `FIGISS` when the pattern length is over 125 but is still 3.9, 6.2, 3.4, and 7.1 times

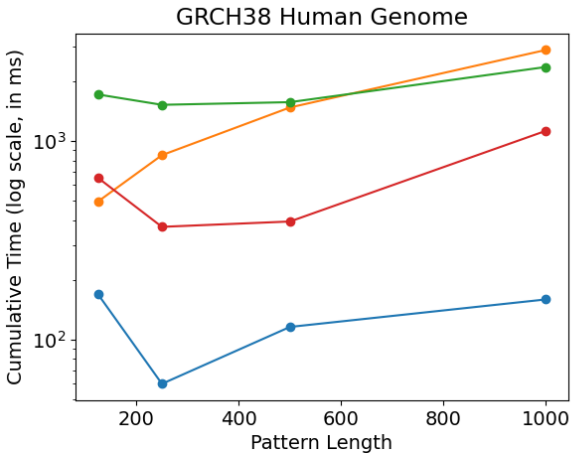

Figure 5 Comparison of query times for `count` between the described solutions when varying the length of the query pattern. For each pattern length equal to 125, 250, 500, and 1000, we report the times for the `GRCH38` dataset. We plot the cumulative time required to perform 1,000 `count` queries. The y-axis is in log scale. `PFP-FM` is shown in blue, `RLFM` is shown in orange, `RLFM` is shown in red, and `FIGISS` is shown in green.

slower than `PFP-FM`. Meanwhile, the `RLFM` method exhibits a steady increase in time usage, and it is 2.9, 14.2, 12.8, and 18.07 times slower than `PFP-FM`. It is worth noting that the `FIGISS` grammar is less efficient for non-repetitive datasets, as demonstrated in the research by Akagi et al. [1], which explains its (worse) performance on `GRCh38` versus the `SARS-100K` dataset. Hence, `FIGISS` is 10.1, 25.5, 13.6, and 14.8 times slower than `PFP-FM`. These results are inline with the performance of our previous results, and demonstrate that `PFP-FM` has both competitive construction memory and time, and achieves a significant acceleration.

5 Conclusion

In this work, we presented `PFP-FM` that shows significant acceleration over existing state-of-the-art methods. Hence, this work begins to resolve a relatively long-standing issue in data structures as to how we can parse input that has no natural word boundaries in a manner that enables acceleration of the FM-index. We note that it is possible to similarly augment `locate` queries since for that we need the suffix array samples only in the final step when matching α (or β in case that $Q = \beta$), which can be done by the usually suffix array samplings for the FM-index. If α is empty, then we can instead match the first block of the pattern with the FM-index on S and not on P. We leave this for future work. With respect to practical applications, as reads are getting longer and more accurate, we will soon see an opportunity to apply accelerations of finding patterns that have length between 125 and 1,000. Hence, a larger area that warrants future consideration is accelerating the backward search with approaches such as `PFP-FM` for aligning Nanopore reads to a database. Our last experiment shows significant acceleration with query patterns of length 1,000 to a full human reference genome, giving proof that the research community is in the position to begin such an endeavour.

Table 1 Comparison of the construction performance with the construction time and memory for all datasets. The number of characters in each dataset (denoted as n) is given in the second column. The time is reported in seconds (s), and the memory is reported in gigabytes (GB).

Dataset	n	Method	Construction Memory (GB)	Construction Time (s)
SARS-25k	751,526,774	RLCSA	9.90	322.85
		RLFM	3.47	363.74
		FIGISS	4.89	378.49
		PFP-FM	12.99	117.29
SARS-50k	1,503,252,577	RLCSA	19.88	679.89
		RLFM	6.94	701.36
		FIGISS	12.44	795.70
		PFP-FM	26.12	233.04
SARS-100k	3,004,588,730	RLCSA	39.47	1690.22
		RLFM	25.01	1432.16
		FIGISS	25.57	1840.80
		PFP-FM	53.90	489.45
GRCh38	3,189,750,467	RLCSA	45.45	924.60
		RLFM	26.31	1839.25
		FIGISS	34.65	1440.19
		PFP-FM	71.13	1154.12

References

1 Tooru Akagi, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar index by induced suffix sorting. In *Proceedings of the 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 85–99, 2021.

2 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Molecular Biology*, 14(1):13:1–13:15, 2019.

3 Christina Boucher, Travis Gagie, Alan Kuhnle, and Giovanni Manzini. Prefix-free parsing for building big BWTs. In *Proceedings of the Workshop of Algorithms in Biology (WABI)*, pages 2:1–2:16, 2018.

4 Jin-Jie Deng, Wing-Kai Hon, Dominik Köppl, and Kunihiko Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 63–72, 2022.

5 Paola Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52:552–581, 2005.

6 Paolo Ferragina and Johannes Fischer. Suffix arrays on words. In *Proceedings of the 18th Annual Symposium Combinatorial Pattern Matching (CPM)*, pages 328–339, 2007.

7 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):1–54, 2020.

8 S Gog, T Beller, A Moffat, and M Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the 13th Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

9 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019.

10 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(3):R25–R25, 2009.

11 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *CoRR*, 2013. `arXiv:1303.3997`.

12 Veli Mäkinen and Gonzalo Navarro. Run-length FM-index. In *Proceedings of the DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"*, pages 17–19, 2004.

13 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, pages 45–56, 2005.

14 Udi Manber and Gene W. Myers. Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

15 Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. Sustainable data analysis with Snakemake. *F1000Research*, 10, 2021.

16 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching Reads to Many Genomes with the r-Index. *Journal of Computational Biology*, 27(4):514–518, 2020. `doi:10.1089/cmb.2019.0316`.

17 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. `doi:10.1109/TC.2010.188`.

18 Daniel Saad Nogueira Nunes, Felipe Alves da Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. DCC*, pages 42–51, 2018. `doi:10.1109/DCC.2018.00012`.

19 Jouni Siren. Compressed suffix arrays for massive data. In *Proceedings of the 16th International Symposium String Processing and Information Retrieval (SPIRE)*, pages 63–74, 2009.