# BWT-RUNS COMPRESSED DATA STRUCTURES FOR PAN-GENOMIC TEXT INDEXING

by

Nathaniel K. Brown

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2023

*Dedicated to those who inspired me to do more,*
*when I could have settled for less*

# Contents

# List of Tables

# List of Figures

## Abstract

The FM-index is used in many bioinformatics applications, using the Burrows-Wheeler Tranform (BWT) to support pattern matching queries; however it cannot handle large text collections. This is of particular concern due to reference bias; DNA alignment is inaccurate when the references fail to capture the required genetic diversity. Pan-genomics attempts to avoid this by utilising the information of many genomes. A recently successful solution has been FM-indexes using the run-length compressed BWT (RLBWT). Gagie et al. showed how we can support queries efficiently with an RLBWT but relied on sparse bitvectors requiring worst case logarithmic time rank operations [20]. Nishimoto and Tabei improved this result to answer queries in optimal time given a polylog-sized alphabet by computing last-to-first (LF) steps in constant-time [46].

We show that this result can be made practical for LF, generalizing to run-length compressing permutations which contain $r$ runs which are permuted consecutively. This "table-lookup" approach is fast even without theoretical guarantees despite being worst case $\Omega(r)$-time; however, the simple formulation is large when implemented using uncompressed integers. We show how to compress columns of the table to support count queries, competitive in time/space with the best existing implementations. An algorithm for constant-time LF steps is evaluated, but does not impactfully improve the speed of count queries. LF steps can also be used to compute matching statistics, which describe exact matching substrings of a pattern, for pan-genomic datasets. The approach requires storing *thresholds* or computing LCE queries to re-orient in the BWT when a character mismatches. We show how storing additional LCE information alongside thresholds allows us to improve the speed of computing matching statistics with one pass of the pattern.

## Acknowledgements

I'm especially grateful to my supervisor Travis Gagie, who has involved me in many exciting topics while allowing me to mature as a researcher; I have him to thank for countless opportunities that have shaped my entire academic journey and my future. Following closely behind is Massimiliano Rossi, who has been more than just a co-author, mentoring me from my start in bioinformatics as an undergraduate. My research term supervisor Kimon Fountoulakis (U. Waterloo) for the advice and support in pursuing further study. Thank you to my readers at Dalhousie University, Meng He and Chris Whidden; not only for their insightful comments but for all the knowledge of algorithms and data structures I learned from them. Christina Boucher (U. Florida) and Ben Langmead (JHU) for all their collaboration and guidance in our shared research interests. Ben Langmead, Nicola Prezza (Università Ca' Foscari), Giovanni Manzini (University of Pisa), and Veli Mäkinen (University of Helsinki) for hosting me during my academic travels; and to all the professors and students who aided my work while there. The Faculty of Computer Science at Dalhousie University; particularly Norbert Zeh for helpful discussion of my thesis topics, and my fellow students Yansong Li, Sana Kashgouli, Nicola Cotumaccio, and Baorui Ja. To my other co-authours for work on the included topics: César Martínez-Guardiola, Fernando Silva-Coira, Dominik Köppl, and Susana Ladra. A heartfelt thank you to Tatum Arcon for her love and understanding support through thesis writing. Finally, my family and friends for everything; particularly my parents, Josh Brown and Sarah Renaud. This thesis was supported by the Natural Sciences and Engineering Research Council of Canada through a CGS-M scholarship.

# Chapter 1

# Introduction

A string is a datatype consisting of a sequence of symbols, with this thesis itself a convenient example. Given a sub-string, we can ask how many times it occurs within another string and where; in other terms, string matching of a pattern to a text. Relevant application areas include queries on natural language texts, DNA and protein sequences, website data, and software heritage. A text index is a data structure supporting pattern matching queries in sub-linear time with respect to the length of the text [23]. Compact text indexes take this further by occupying sub-linear space through compression of the input whilst supporting queries [17].

## 1.1 Compact Text Indexing

Pattern matching can be supported with efficient speed by using foundational data structures such as a suffix tree [27] and a suffix array (SA) [39]. By sorting suffixes of the text, any matching pattern is found as a prefix for some range of these suffixes. However, these approaches can occupy space larger than the text itself and still require access to the full text [23]. In contrast, the FM-Index of Ferragini and Manzini [17] encapsulates the input text while supporting queries over its compressed representation. Lossless compression is performed by applying the Burrows-Wheeler Transform (BWT) algorithm [12]: a reversible permutation of the original text. It supports queries by an underlying relation to the suffix array, but does so using $o(n)$-bits for compressible texts with a length of $n$ symbols [17].

The BWT is the last column of the Burrows-Wheeler Matrix (BWM) which is computed by sorting the rotations (cyclic shifts) of the text. This column is used to extract sub-strings or perform a full text reversal by performing backwards search, which maps the position of a character in the BWT to the position of the character immediately preceding it in the original text [12]. Ferragini and Manzini describe how to compute this step as a last-to-first (LF) mapping between characters in the

BWT (last column) and first column of the matrix in constant time [17]. LF-mapping is used not only for extraction, but as the foundation to support further sub-string queries.

LF steps and the FM-index is the basis for many key tools in computational genomics which operate over DNA sequences; FM-index based aligners such as Bowtie [33, 32] and BWA [35] see daily use in hospitals and clinics worldwide. By pre-processing a reference genome into a text index, sampled short reads of DNA are aligned using pattern matching and sequence alignment algorithms [32]. This is a success story for compact data structures as they directly address the need for compressed representations of large DNA sequences

## 1.2 Computational Pan-genomics

Although standard FM-aligners perform compression, they still grow linearly with the length of the input text. This is only sufficient for a reference of a few human genomes. The 1000 genomes project [57] is an influential example of advancements in the variety of sequenced genomes available. Since the original human genome project [29], the collection of references is growing exponentially and significantly outpacing Moore's law [56]; advancements in memory technology are not sufficient to make full use of these growing collections. Aligners which only use a few references fail to capture the diversity between sequences in the collection and are susceptible to errors [5].

Computational pan-genomics aims to utilize this information to avoid issues such as reference bias where reads are misaligned since they are not well represented in the reference. Surveys discuss this as a barrier to personalized medicine for some ethnic groups (such as African, Central/South Asian, Indigenous, Latin American, and Middle Eastern) whose genetic diversity is not sufficiently represented in the standard reference [51] or even in public databases of genomes [30]. Certain populations have aimed to construct their own references, such as China (using a consensus for Han individuals, China's majority ethnic group at 92%) [59] and Denmark [40], but it is not clear whether and how we can fairly represent multi-ethnic populations with a single reference genome. Reference bias also affects disease identification and the ability of alignment to adequately provide personalized medicine. Rare diseases fail to be diagnosed when mutation based variations cause reads to not align in the reference [5].

Reference bias exhibits the downside of a single reference and motivates bioinformaticians to develop tools which are able to capture the genetic diversity of humans or other species. Although some problems are side-stepped by introducing more computation, this solution limits who can access this technology, and the size of these datasets is still outpacing hardware improvements. Because of the underlying structure of these indexes, to improve on the approach necessitates using compact data structures that can support these queries in even smaller space . Pan-genome graphs represent commonalities and differences at the sequence level of genomic collections [3], leveraging the 99.9% similarity across human genes [14]. Indexing pan-genome graphs is the most publicized solution aiming to capture diversity by improving indexing and its data structures [54], but still has disadvantages in certain cases due to fixed query size and requiring careful selection of introduced variation to avoid prohibitively large indexes [50]. To provide an alternative with different functionality than graphs, we can attempt to scale FM-Indexes to handle a representative sample of a dozen genomes [13] or even thousands of references.

## 1.3 Run-Length FM-Indexes

To scale FM-indexes we can exploit that the fastest growing text collections are highly repetitive. Genomic data is our motivating example, where one sequence can be transformed into another with few modifications. Concatenating text collections and computing its resulting BWT groups together characters whose context are similar. For repetitive texts, this results in many characters in the BWT occurring in *runs*; adjacent characters are grouped since they occur with similar context in their respective string among a highly repetitive collection. This is exploited in BWT compression, but we can also support sub-string queries over the run-length compressed Burrows-Wheeler Transform (RLBWT) [12] to obtain space bounded by the number of runs $r$ rather than the text length $n$. Where concatenating genomes increases $n$ linearly, in practice $r$ grows sublinearly [20].

Text indexing based on the RLBWT for massive datasets was initiated by Mäkinen et al. [55, 38] showing how to quickly count the number of occurrences of a pattern in runs-bounded space. Policriti and Prezza [48] augmented the approach by storing sampled suffix array samples at the boundaries of runs in the BWT to locate *one* position where a pattern ocurred in the text. Gagie, Navarro and Prezza [20] showed how to locate *all* ocurrences of a pattern by using those sampled SA samples to support $\phi$: a function which steps forward and recovers other SA values given a current position. This $O(r)$-space index was referred to as the $r$-index in reference to its runs-bounded approach, and supports fast count and locate queries. Boucher et al.'s prefix-free-parsing (PFP) [8] allows us to efficiently construct such indexes in practice.

The $r$-index and other conventional RLBWT based indexes replicate the FM-index and rely on its LF-mapping method. Conventionally this requires rank queries on a bitvector; the rank at a position in a bitvector corresponds to the number of set bits prior that position. The $r$-index approach marks the starting positions of runs in the BWT in a length $n$ bitvector. To get a runs-bounded approach, a sparse bitvector [47] is used which achieves $O(r)$-space by compressing unset bits. However, rank queries on sparse bitvectors inherit lower bounds from predecessor queries [6] and cannot be computed in constant time like a traditional FM-Index. Many experts would have probably agreed that we cannot perform LF steps in constant time over an RLBWT because of this reliance on slow rank queries [11]. That is, until Nishimoto and Tabei showed how to replace them with a theoretically more efficient approach [46] which does not require rank queries over sparse bitvectors.

The $r$-index is $O(r)$-space and computes the LF-mapping in $O(\log\log r)$-time [20]. Nishimoto and Tabei's data structure is also runs-bounded, but improves LF-mapping to $O(1)$-time by adopting a table lookup approach [46]. Their *move structure* generalizes for any permutation $\pi$ which tends to be map in runs; i.e. where $\pi(i{+}1) = \pi(i){+}1$ occurs often. This property holds not only for LF but also for $\phi$ with these mappings correspond exactly to the $r$ runs of the BWT. The $r$-index computes locate optimally in $O(r)$-space, but Nishimoto and Tabei use the move data structure to also compute count and extract queries in optimal time for a polylog$(n)$ sized alphabets [46]. The table lookup approach itself is not constant-time; instead theoretical guarantees are met by bounding how many rows of the table are scanned when computing a single LF-step.

Although Nishimoto and Tabei's result focuses on exact pattern matching queries, LF-steps can be used to support approximate matching. Bannai, Gagie and I [4] designed a variant of the $r$-index which computes *maximal exact matches* (MEMs) used in approximate matching tools like BWA-MEM [34]. MEMs describe exact matching sub-strings of a pattern which cannot be extended by lengthening the sub-string to the left or right. Bannai et al.'s approach stores $r$ thresholds in the RLBWT which, given a position corresponding to a mismatch character when processing the pattern, tell us whether to "jump" forwards or backwards to reorient in the BWT [4]. Rossi et al. [52] showed how to find these thresholds using the *longest common prefix* (LCP) array and gave a practical implementation constructed using PFP, which they called MONI.

Rossi et al.'s MONI relies on access to a *straight line program* (SLP) to support random access to a text. Such an SLP can be constructed with significantly less space than the $r$-index itself [19, 18] and is practical for pan-genomics. Where MONI processes the pattern twice to compute *matching statistics* (MS), Boucher et al.'s PHONI [7] allows for online computation by using the SLP to instead support *longest common extension* (LCE) queries. Instead of computing MS, Ahmed et al.'s SPUMONI [2, 1] index for targeted nanopore sequencing computes an approximation called *pseudo matching lengths* (PMLs). SPUMONI relies on primarily LF-steps and thresholds without use of LCEs, using its PMLs to rapidly eject "nontarget" DNA molecules from the sequencer.

## 1.4 Contribution

The current state of FM-indexes over the RLBWT, specifically for pan-genomics, necessitates the use of LF-mapping in nearly all cases, including alongside thresholds when computing MS using MONI. Thresholds are a rather new approach and further improvements are likely; however, improvements to LF-mapping in runs bounded space is more surprising since rank queries on sparse bitvectors cannot improve past theoretical bounds. A practical approach to Nishimoto and Tabei's result provides an opportunity for speed improvements that otherwise would not have been expected. Their *move structure* applied to LF was first implemented[1] as part of undergraduate

[1]To our colleagues best knowledge by monitoring public avenues.

thesis work by myself and later presented at a workshop in 2021 [10], the same year Nishimoto and Tabei's original result was published [46]; although Nishimoto, Kanda, and Tabei provided an implementation in 2022 [45]. Our approach through this thesis is a static data structure, but this recent work by Nishimoto, Kanda, and Tabei demonstrate that a dynamic approach can be obtained.

This preliminary implementation[2] is the subject of Chapter 3, which includes the surprising result that the structure computes LF efficiently even without constant-time guarantees. Although we show LF to be worst case and average $\Omega(r)$-time using table-lookup, in practice the majority of steps do not require scanning any additional rows of the table (on genomic data). As presented, our table representation is an efficient introduction to both run-length indexing and a practical formulation of Nishimoto and Tabei's move structure. Its implementation is much faster than the $r$-index in full text decompression or sampling LF-mapping steps; however it is much larger due to storing uncompressed integers.

To further explore unbounded table lookup with LF, we can attempt to compress components of the table. The destinations of run mappings and the RLBWT run-head characters themselves can be efficiently compressed, which we explore in Chapter 4. Rows of the table are broken into blocks to mitigate locality concerns and compressed columnwise using various schemes. To experimentally verify the approach, count queries are supported with table-lookup LF alongside sequential scanning and queries on short uncompressed bitvectors. This index was published by myself[3], Gagie, and Rossi [11] and demonstrates that even without theoretical bounds, table-lookup can be made competitive in time/space with the best existing methods on pan-genomic data. However, we found that although the majority of scans were small, the worst case increased as more sequences were added to the text.

Nishimoto and Tabei's proof of a theoretical bound on scans can be reformulated and parameterized [11] and illustrates a practical implementation of the approach by splitting runs. However, it can double the size of the table in the worst case.

[2]This summarizes early results from Honours thesis work, but expands to include work presented subsequently during Masters for workshop and conference presentations

[3]I was primary author/presenter and implemented the approach at https://github.com/drnatebrown/r-index-f

Chapter 5 explores alternatives to splitting by capping run lengths or sampling table scans, but also implements the parameterized formulation[4]. It generalizes to any permutation in constant-time and $O(r)$-space for the number of permutable runs. Nisimoto, Kanda, and Tabei recently used a variation of the move structure to gradually build the RLBWT character by character [45]; my method is an alternative taking a pre-computed RLBWT and thus allowing construction using PFP or other tools first. Experiments show that in practice we can achieve theoretical guarantees without much space increase, but this does not have a large effect on count queries explored in Chapter 4.

Successful implementations of table-lookup for LF were partially motivated by potential improvements to MONI and its variants which rely on LF-mapping and thresholds. In Chapter 6 we focus instead on improvements to the thresholds. Martínez-Guardiola, myself, Silva-Coira, et al. showed how to augment thresholds with additional LCE information to avoid some LCE queries in practice when computing MS online [41]. By storing the lower-bound on potential LCE queries at each run, we avoid unnecessary computation in the mismatch case. This chapter focuses on my role in the paper, explaining the approach and how to generate and compress this additional information alongside the thresholds. Results show great speed gains compared to the traditional one-pass MONI of Boucher et al. [7] with respect to the additional space needed to store the augmented thresholds.

Demonstrating improvements in LF stepping and thresholds motivates new variants of tools which rely on them, such as MONI and SPUMONI. Chapter 7 explores possibilities of applying these results to pan-genomic indexing tools and other future work related to these improvements in run-length compressed data structures. To understand these improvements, our journey proceeds with Chapter 2, introducing background knowledge and notation required for the content of this thesis.

[4]Code available at https://github.com/drnatebrown/r-permute

# Chapter 2

# Preliminaries

This chapter focuses on introducing the relevant background needed to understand the RLBWT and LF-mapping in motivation of table-lookup and MS-finding. Notation for strings is introduced to discuss the BWT and its relationship to the suffix array through LF-mapping. We briefly remark on succinct data structures (bitvectors supporting rank/select, wavelet trees) and integer encoding (delta codes, Elias-Fano representations, and directly addressable codes) which are used in later sections. Background specific to computing MS is reserved for Chapter 6. These topics are covered in further detail in *Compact Data Structures: A practical approach* [44] by Gonzalo Navarro. Implementations of these data structures can be found in the C++ SDSL library [21].

## 2.1 Notation

Unless otherwise specified, log is assumed to be base 2 and often describes entropy in bits. The word RAM model of computation is used, in which we assume a computer word of $w = \Theta(\log n)$ for data of size $n$ in memory. In this model, we assume all basic arithmetic and logical bit operations can be performed in constant time. Addressing any data element is also assumed to take constant time. This model is used only when describing the space of indexes such as the $r$-index and Nishimoto and Tabei's move structure. For specific compression schemes, we will often use log and asympotic notation to describe the number of bits needed to represent the data.

We use 0-based indexing for arrays in most cases; i.e. an array $A[0..n-1]$ of $n$ elements whose size is also given by the notation $|A| = n$. A string is an array drawn from an *alphabet* of symbols. For our purposes, we assume the alphabet $\Sigma$ to be a totally ordered set of size $|\Sigma| = \sigma$; e.g. the lexicographic ordering of ASCII characters with $\sigma$=128. For a string $S[0..n-1]$, substrings are represented as $S[i..j]$ and correspond to $S[i], S[i+1], ..., S[j]$. Prefixes are substrings starting from the beginning of the string, such as $S[0..i]$, and suffixes are substrings to the end, such as $S[j..n-1]$. A subsequence is more general and does not have to be consecutive in the array/string we describe it over.

For indexing problems, we refer to a text $T[0..n-1]$ to be indexed and a substring pattern $P[0..m-1]$ which is used to query the index. In these cases it is sufficient to consider lexicographical order of ASCII characters; $T[i] < T[j]$ represents $T[i]$ being lexicographically less than $T[j]$. Ordering on substrings maintains convention of left to right character comparisons until they differ. The terminal character is a special symbol of least order represented by \$. A text $T[0..n-1]$ is assumed to end with this character \$; $T[i] \neq \$$ except when $i = n-1$. This simplifies suffix ordering since the $n$ suffixes of $T$ are now uniquely defined by the position of the terminal character. This is used to compute the *suffix array* SA$[0..n-1]$ for a text $T[0..n-1]$: $SA[i]$ stores the starting position of the lexicographically $i$th suffix in $T$. The *longest common prefix array* LCP$[1..n-1]$ is associated with the SA[1]; $LCP[i]$ stores the longest common prefix of $T[SA[i-1]..n-1]$ and $T[SA[i]..n-1]$. A *longest common extension* LCE$(i, j)$ returns the longest common prefix of the suffixes $T[i..n-1]$ and $T[j..n-1]$.

The rank of a character $T[i]$ is the number of prior occurrences of that same character in the range $T[0..i-1]$ of a text. In this sense, rank is 0-based, since the first character in a text is of rank 0. The function $rank_c(i)$ answers this query for a character $c$ where we do not assume $T[i] = c$ (it computes the occurrences of $c$ prior to $i$). The position of the $i$th occurrence of a character is denoted as $select_c(i) = j$ such that $T[j] = c$ and $rank_c(j) = i-1$. In this sense, select is 1-based. This notation is adopted from literature on the subject [44] and is useful since it allows rank queries to return 0 when there are no prior occurrences. Finding the character $c$ of rank $i$ is given by $select_c(i+1)$. Our notation of rank and select queries on symbols is heavily related to bitvectors and other succinct data structures.

## 2.2 Compact Data Structures

*Succinct data structures* refer to data structures using $\log N + o(\log N)$-bits where $N$ is the total number of distinct objects that can be encoded and $\log N$ represents the

[1]LCP[0] can be defined as 0, but we exclude it from being indexed.

worst case entropy [44]. For example, length $n$ strings over an alphabet give $N = \sigma^n$ objects and worst case entropy of $\log \sigma^n = n \log \sigma$ bits. Such data structures are differentiated from compressed encodings by also supporting queries in-place *without* requiring decompression [42]. *Compact data structures* refers to such compressed space and fast query data structures more broadly (not necessarily fulfilling the succint worst case entropy defintion). Bitvectors consist of a bit array $B[0..n-1]$ over the binary alphabet $\Sigma = \{0, 1\}$ which support access, rank, and select queries. This is consistent with our notation of rank/select over arbitrary alphabets, but we will implicitly assume these queries are with respect to 1 bits when over bitvectors. Conventional bitvectors initiated the study of succinct data structures [26], and they support access/rank/select queries in constant time using $n$ bits for the array and $o(n)$-bits to support the queries.

A *plain bitvector* refers to a bitvector whose representation is in the order of $n$-bits. A *sparse bitvector* instead optimizes space for the number of $m$ set bits, with the ones in this paper using $2m + m \log \frac{n}{m}$-bits [47, 21] which is advantageous when $m \ll n$. Sparse bitvectors support select in constant time; however, rank queries take $O(\log \frac{n}{m})$-time and cannot be computed in constant time since they inherit lower bounds from predecessor queries [6]. A *wavelet tree* supports access/rank/select over arbitrary alphabets by storing a tree structure of bitvectors (either plain or sparse). Each character is encoded to partition the alphabet; the tree has $\sigma$ leaves where each levels bitvector sets a 0 if the character is in the left subtree (and 1 for right subtree). This recursive procedure using plain bitvectors can support access/rank/select in $O(\log \sigma)$-time in $n \log \sigma + o(n \log \sigma)$ bits [43]. When $\sigma = O(polylog(n)) = \log^{O(1)n}$, meaning a polylog($n$) sized alphabet, we improve from $O(\log \sigma)$ to $O(1)$-time for queries. Figure 2.1 shows a wavelet tree for an example string.

## 2.3 Integer Encoding

Given a sorted integer sequence (non-decreasing) delta codes compress based on gaps between integers. For an array $A[0..m-1]$ with maximum element $n$, consider marking $A[i] + i$ in a bitvector of length at most $n + m$ supporting select queries; $select(i) - i$ computes the partial sum of 0s and returns the value (deltas are implicitly represented by the number of 0s between set bits). When strictly increasing, we can mark $A[i]$

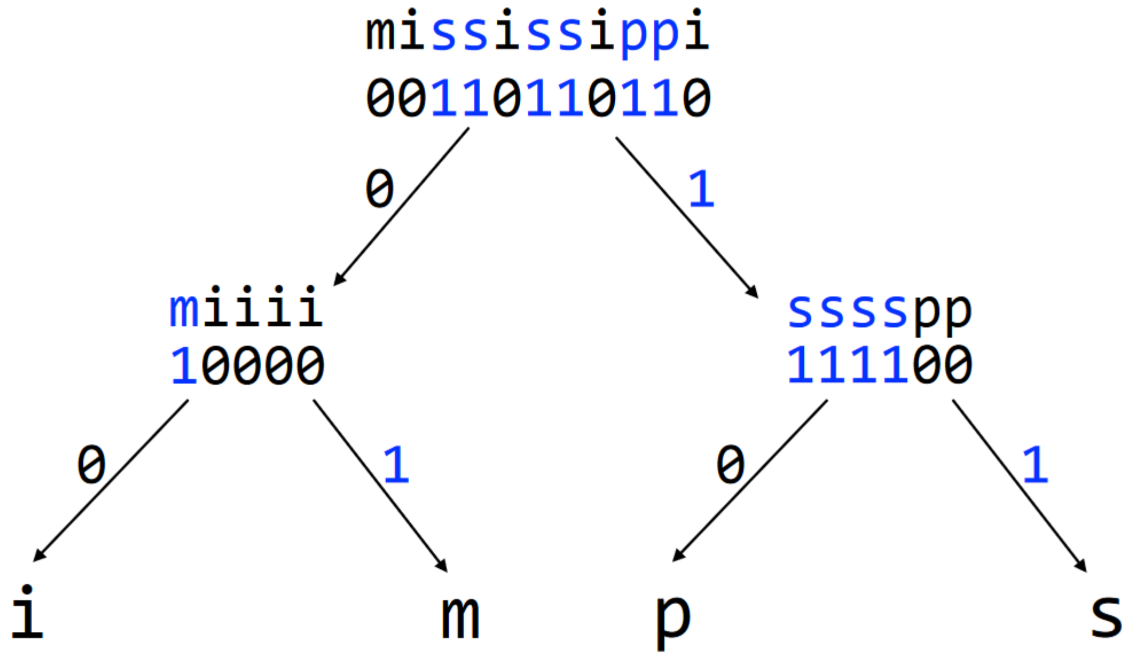


Figure 2.1: For a string $T = mississippi$, its corresponding wavelet tree is shown (copied from [31]).

in the bitvector reducing the length to solely $n$ bits. This can be used efficiently without change when gaps are small (e.g. Chapter 4), but it is often a component of other compact data structures [44]. Elias-Fano representations store explicitly the $\lceil \log \frac{n}{m} \rceil$ lower bits of each integer and store the remaining bits (also a non-decreasing sequence) using the simple approach using bitvectors [58]. The required select is constant time on plain bitvectors; for increasing sequences we require at most $2m$ bits for select on top of the explicit $m \log \frac{m}{n}$ lower bits. This bound looks familiar; select on sparse bitvectors is stored explicitly by using Elias-Fano representation on the set bits and these terms are used interchangeably during later discussion.

To store any integer sequence $A[0..n-1]$ with maximum element $U$, we can use a fixed length code using $n \lceil \log U \rceil$ bits. A *variable length code* attempts to compress further by storing each integer in varying length based on their smallest bit representation (e.g. Elias codes [16]) or statistics (e.g. Huffman codes [25]). Such codes do not always support efficient direct access of any $i$-th element, depending on sums of previous code lengths or sampling of these lengths [44]. Brisaboa et al.'s *directly addressable codes* (DACs) allow practical direct access for variable length codes without sampling [9]. It represents integers into chunks of size $b$; any chunk $C_1$

in the first sequence stores the least significant $b$ bits of each integer, $C_2$ stores the next $b$ bits for *only* integers that require it, and so on. A bitvector supporting rank is stored with each chunk level which is set if that integer has a corresponding chunk at the next level; we find the index needed at the next chunk level using a rank query. Figure 2.2 shows the structure of the code. DACs, optimized with implementation considerations, support direct access efficiently in practice for sequences with mostly small values which we see in Chapters 4, 5, and 6.

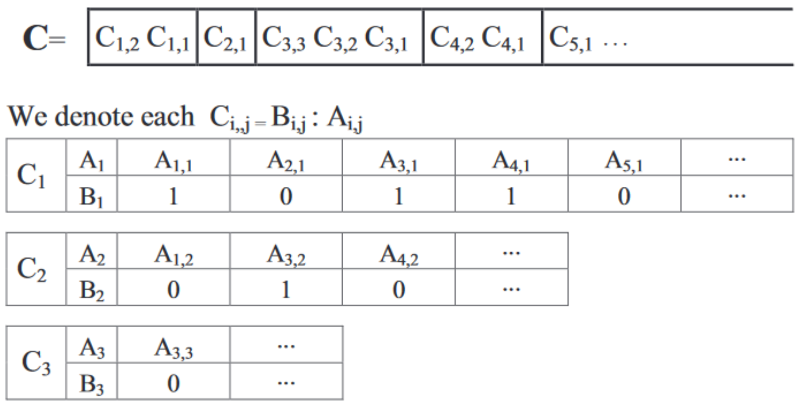

C= C1,2 C1,1 | C2,1 | C3,3 C3,2 C3,1 | C4,2 C4,1 | C5,1 ...
We denote each Ci,j = Bi,j : Ai,j
C1 A1 A1,1 A2,1 A3,1 A4,1 A5,1 ...
B1 1 0 1 1 0 ...
C2 A2 A1,2 A3,2 A4,2 ...
B2 0 1 0 ...
C3 A3 A3,3 ...
B3 0 ...

Figure 2.2: Visualizes a directly addressable code. For an integer sequence $C$, the first entry is represented using two chunks; we first load $A_{1,1}$ and then the set bit $B_{1,1}$ signals to also load $A_{1,2}$ from the next level. (copied from [9]).

## 2.4 Burrows-Wheeler Transform

Given a text $T[0..n-1]$, the *Burrows-Wheeler Matrix* (BWM) is computed by sorting the rotations (left cyclic shifts) of the text; the last column of this matrix is the *Burrows-Wheeler Transform* (BWT) of the text [12]. Figure 2.3 illustrates the process. Because of the terminal character, there are exactly $n$ unique rotations of the text. Any column of the BWM and specifically the BWT are permutations of the original text since each of the $n$ rotations place a character $T[i]$ in each position exactly once. Due to cyclic shifts, the preceding characters of a row in the BWT are sorted by their right context (i.e. the characters in preceding columns of the BWM). When context strongly predicts symbols in the text, they are grouped together in the BWT, which allows various simple compression schemes such as move-to-front or run-length encoding [12]. A run in the BWT is a sequence BWT$[i..j]$ where

BWT[$i$] = BWT[$i+1$] = ... = BWT[$j$]. We usually assume runs to be maximal such that neither BWT[$i-1$] nor BWT[$j+1$] equals the run character BWT[$i$]. The *run-length encoded Burrows-Wheeler Transform* (RLBWT) stores the $r$ runs of the BWT by their run-head characters and their run-lengths.

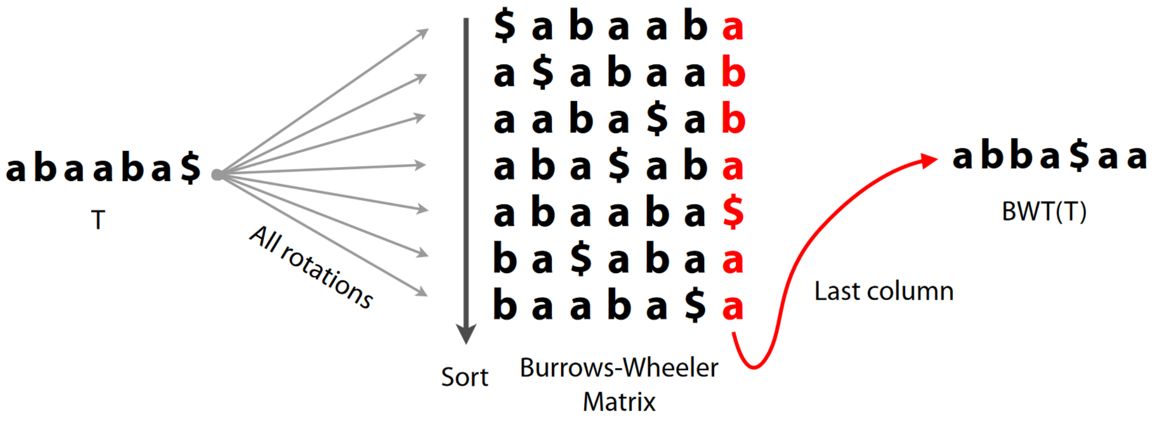


Figure 2.3: For a string $T = abaaba\$$, the BWM is constructed by sorting its cyclic rotations; the BWT is the last column (copied from [31]).

The BWM maintains the same order as the SA since we are essentially sorting suffixes due to the terminal character. This defines BWT[$i$] to be the character directly preceding SA[$i$] in the text[2] such that BWT[$i$] = $T$[SA[$i$] − 1]. If we find the position $j$ of SA[$i$] − 1 in the suffix array, we can find BWT[$j$] = $T[SA[i] - 2]$; this mapping allows us to reverse the BWT and recover the original text (partially or in full). The characters in the first column of the BWM, $F[0..n-1]$, are the first characters of the sorted suffixes and are representative of SA in that $F[\ell] = T[\text{SA}[\ell]]$; finding the position of $SA[i] - 1$ in SA is equivalent to finding the position of the character BWT[$i$] (with respect to ranks in $T$) in the $F$ column.

## 2.5 LF Mapping

Mapping the position of characters in the BWT (or last column $L$) to their corresponding position in the $F$ column is called the *last-to-first* (LF) mapping. By the BWM, $F$ corresponds to the sorted characters of $T$. The order of these characters with respect to their rank depends on the comparisons of the succeeding characters

[2]For simplicity we ignore the boundary case SA[$i$] = 0, but note we can work modulo $n$ such that BWT[$i$] = $T[SA[i] - 1] = T[-1] = T[n-1]$, which in all cases will equal the terminal $

in their BWM row (right context); hence, character occurrences in $F$ are sorted by right context. Since we established that $L$ is also sorted by right context, we have that character occurrences for each $c \in \Sigma$ preserve the same ranked order in both $L$ and $F$. This allows us establish ranks in ascending order looking down $L/F$, which simplifies $F$ into ordered blocks of ascending rank for each character. If BWT[$i$] is of rank $j$, then its position in $F$ is the $j$th character in its corresponding $F$ block. Let $C[0..\sigma - 1]$ be an array[3] where $C[c]$ is position of the first $c$ in $F$. As Figure 2.4 visualizes, any LF mapping can be computed as $\text{LF}(i) = C[\text{BWT}[i]] + rank_{\text{BWT}[i]}(i)$.

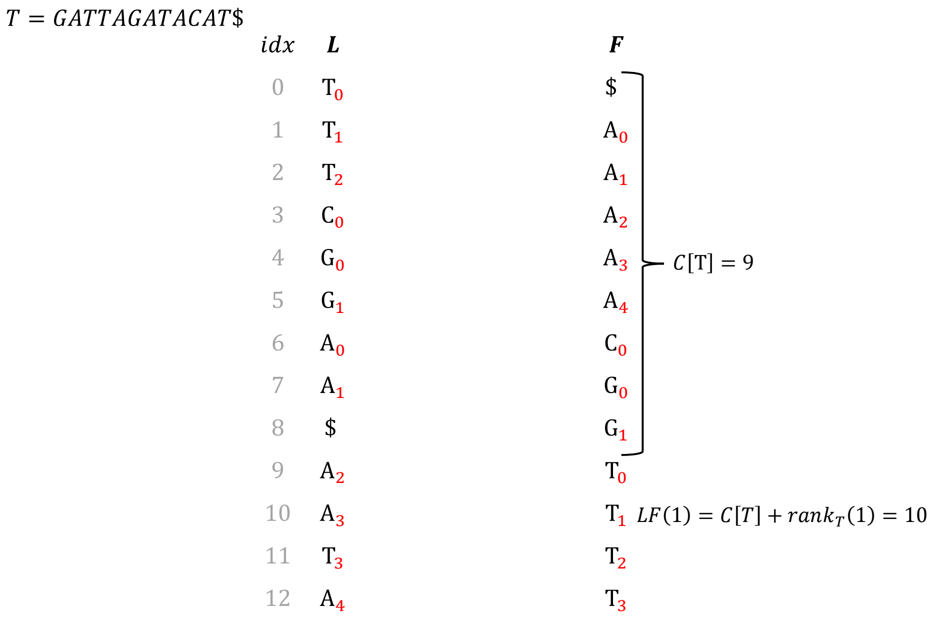


Figure 2.4: For a text $T$ = GATTAGATACAT, the corresponding $L$ = BWT and $F$ columns are shown in ascending rank order. An LF mapping is computed for a character $T$ of rank 1 in the $L$ column by finding the 1st character in the block of $T$s in $F$.

The FM-Index can store the ranks explicitly to achieve constant-time LF-mapping [17], and a wavelet tree achieves the same for polylog sized alphabets. The $r$-index, like other conventional indexes operating over the RLBWT in $O(r)$-space, stores run-heads in a wavelet tree of size $r$, compresses the run-lengths, and marks the starting positions of runs in a sparse bitvector [37]. This slightly more complicated LF-mapping requires a rank query on the run positions; a sparse bitvector operation which cannot achieve constant-time due to inheriting lower bounds from predecessor

[3]Any alphabet of size $\sigma$ can be represented equivalently as $\Sigma = \{0, ..., \sigma - 1\}$

queries [6]. The rank query itself in $O(\log\log_w(n/r))$-time also determines the total complexity of an LF step for conventional RLBWT indexes such as the $r$-index [20]. Although indexes using this sparse bitvector approach achieve $O(r)$-space compared to $O(n)$ for an FM-index, they lose the ability to backwards step using LF in constant time.

# Chapter 3

# Table Lookup for LF

Conventional RLBWT indexes cannot perform LF-mappings in constant-time. Nishimoto and Tabei's *move structure* shows how to do so by considering mapping as run-intervals [46]. In this chapter, we introduce the approach using the formulation of table-lookup. The notation and terminology in this section is altered from Nishimoto and Tabei's original result, but captures the same idea. By augmenting the RLBWT with explicit pointer information for mapping runs, an $O(r)$-space lookup table can be constructed. Brief remarks are made about generalizing this to any permutation and constant-time guarantees, but this discussion is largely left for Chapter 5. Due to sequential scanning in the unmodified table, LF is worst case $\Omega(r)$-time, and even average time $\Omega(r)$ on some strings. However, a simple implementation is is tested on a real genomic dataset and shows that steps are efficient in practice.

## 3.1 LF Mapping Runs

Recall that a run BWT$[i..j]$ is a contiguous same character sequence which we assume to be maximal such that BWT$[i-1]$ and BWT$[j+1]$ cannot extend the run. For any run, their ranks occur in ascending order; a property shared with these same characters in the $F$ column. As Figure 3.1 shows, any run in the BWT is found in the $F$ column as well. This observation gives an alternative to computing LF for runs; if BWT$[i]$ = BWT$[i+1] = c$ then LF$(i+1)$ = LF$(i) + 1$. This is proven directly by our rank definitions, since $rank_c(i+1) = rank_c(i)+1$ implies LF$(i+1) = C[c] + rank_c(i) + 1 =$ LF$(i) + 1$. Applying this property across an entire run allows us to LF map runs directly.

For a run BWT$[i..j]$ and a position $q$ within that run, its LF-mapping can be computed as LF$(i) + (i - q)$. As Figure 3.2 shows, this reformulation computes LF by first mapping the run-head and then computing successive positions as offsets of that initial mapping. Computing LF$(i)$ still requires a character rank query, but what if

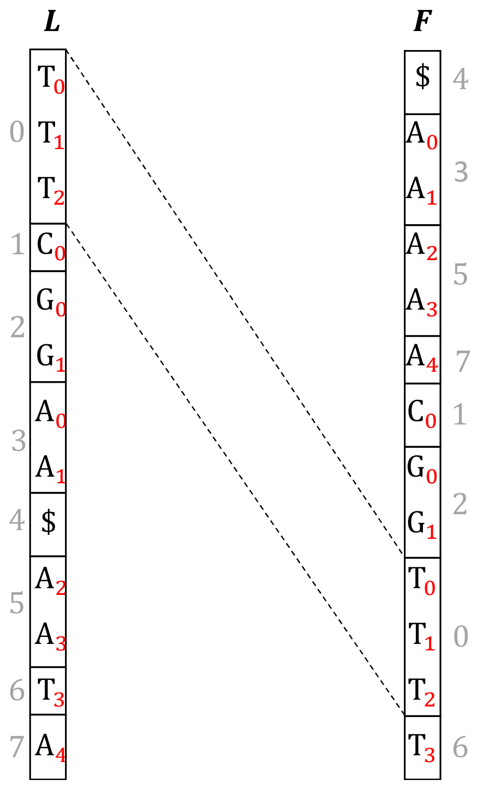


Figure 3.1: Each run in $L$ can be found permuted in $F$. The position indicates where the $k$th run of $L$ is found in the $F$ column.

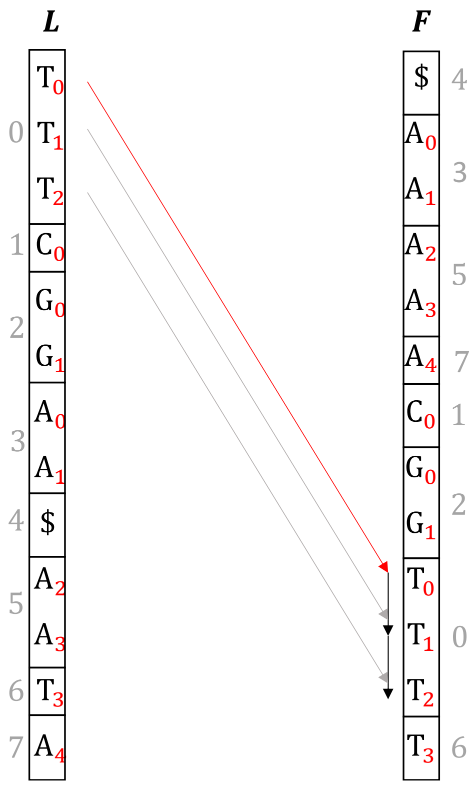


Figure 3.2: The LF mapping of a run can be reformulated as first mapping the representative run-head position and then computing the succeeding positions as offsets.

we stored $\mathrm{LF}(i)$ explicitly? For $q$ occurring in a run starting at $i$, we compute its offset $q-i$ and access our stored $LF(i)$, each of which is a constant-time operation. There are at most $r$ positions where $\mathrm{LF}(i+1) \neq \mathrm{LF}(i)+1$, which corresponds exactly to the runs of the RLBWT. Thus we store the LF-mapping of each run-head position of the RLBWT in $O(r)$-space.

To perform successive LF permutations, i.e. $\mathrm{LF}(j)$ where $j = \mathrm{LF}(i)$, we need to find the run containing $j$ to access the table. Doing so with a predecessor query is not constant-time in $O(r)$-space. Instead we store the $r$ run-head mapping destinations (run containing $\mathrm{LF}(i)$) alongside the run-head positions $i$ and their mappings $\mathrm{LF}(i)$. To maintain the functionality of the RLBWT, we store these columns alongside the run-head characters $c$ and the run-lengths $\ell$. For a value $q$ in run starting at $i$, we compute $\mathrm{LF}(q) = \mathrm{LF}(i) + (q-i)$, and then access the run containing $\mathrm{LF}(i)$ to find its run-head position $i'$ and length $\ell'$. If $LF(q) < i' + \ell'$, then this permutation *crosses* the boundary of $LF(i)$'s containing run $k'$. We proceed by checking $k' + 1$ and its length and so on, scanning until we find the run containing LF($q$). Each access is constant-time, since our scan is sequential. Since all other operations are constant time, the number of scans bounds the complexity of LF.

## 3.2 Permutations in $O(r)$-space

Nishimoto and Tabei's result, as we have introduced through LF, can be generalized to any permutation. The intuition, as Figure 3.3 visualizes, is when a permutation is "runny", such that it can be represented by relatively few contiguous sequences with elements which are also permuted consecutively, we can permute over these blocks instead of each position individually. Let $\pi$ be a permutation over $\{0, ..., n-1\}$ and $S$ be the $r$ values of $i$ such that $\pi(i-1) \neq \pi(i) - 1$ or $i = 0$. We store an $O(r)$-space table in which a row $j$ consists of a 4-tuple: the $j$th value $i$ in $S$; $\pi(i)$; the predecessor $j'$ of $\pi(i)$ in $S$; and the length of the permuted subsequence starting at $i$. This is analogous to our LF representation, and we equivalently evaluate for position $q$ with predecessor $i$ as $\pi(q) = \pi(i) + (q - i)$. To find the predecessor of $\pi(q)$, we scan sequentially in the table from $j'$ until we find the correct position.

Since we perform a linear scan over our $O(r)$-space table, we may have to access $\Omega(r)$ entries in the worst case. It is possible to find worst case examples in which the average time across all permutations is $\Omega(r)$. Figure 3.4 shows an example for LF in which BWT$[0..n-1] = (bc)^{n/10} \cdot a^{4n/5}$ with $r = n/5 + 1$ runs. For $\frac{3n}{5}$ of possible LF steps, we scan $r - 1$ rows. A string producing a similar result can be produced; start with a randomly generated binary string over $\Sigma = \{b, c\}$ and then interleave four consecutive $a$'s in between each original character. We have $\frac{4n}{5}$ $a$ characters, and the number of runs of the BWT cannot be much less than the number of runs in the original sequence; otherwise, we could improve run-length compression for any random sequence by interleaving characters. The expected runs of a random binary string is half its length, $\frac{n}{10}$, and thus mapping to $a$ characters requires crossing a factor of $r$ boundaries similar to Figure 3.4; in expectation we can produce a string requiring average time $\Omega(r)$ for LF steps.

Most queries of RLBWT based pan-genomics indexes can be supported using LF,

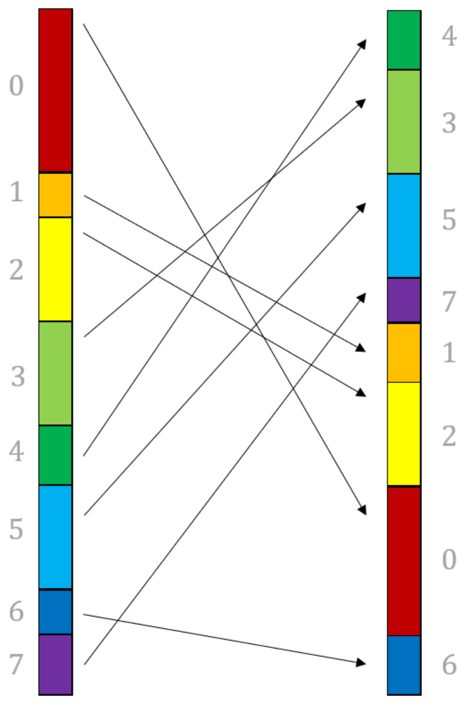


Figure 3.3: For any "runny" permutation, we can attempt to permute its blocks of runs instead of individual positions.

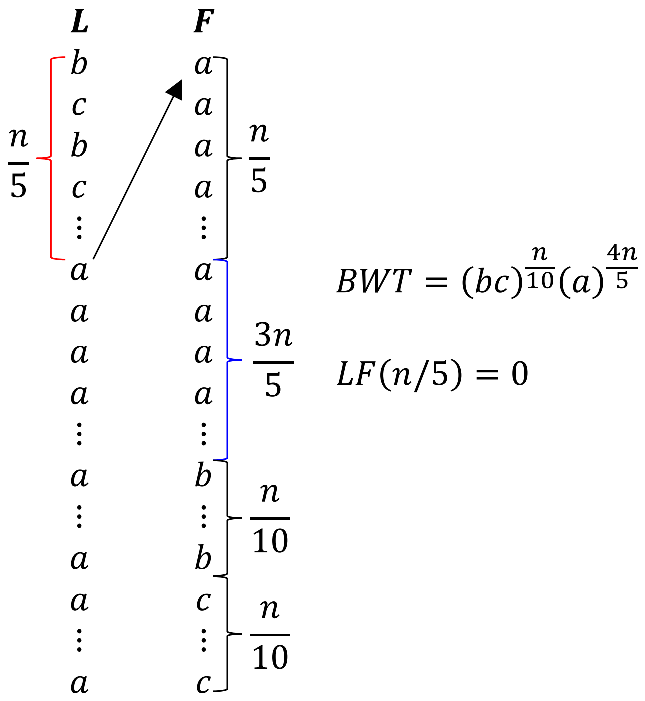


Figure 3.4: For a BWT$[0..n-1] = (bc)^{n/10} \cdot a^{4n/5}$, any LF step among $\frac{3n}{5}$ $a$ positions requires scanning $r-1$ rows.

rank/select over the run-heads with a wavelet tree, and the $\phi$ function and its inverse $\phi^{-1}$. For a position $i$ and $SA[i]$ value, $\phi$ gives the preceding value $\phi[i] = \mathrm{SA}[i-1]$ in the SA. We have shown LF has $r$ such permutable runs, and run-heads are easily stored in $O(r)$-space. Since BWT$[i]$ = BWT$[i+1]$ implies LF$(i+1) = LF(i)$ then SA[LF$(i)$] $= \phi(SA[LF(i+1)])$. As Figure 3.5 illustrates, this implies

$$\phi(\mathrm{SA}[i+1]) = \mathrm{SA}[i] = \mathrm{SA}[\mathrm{LF}(i)] + 1 = \phi(\mathrm{SA}[\mathrm{LF}(i+1)]) + 1 = \phi(\mathrm{SA}[i+1]-1) + 1$$

and choosing $i' = \mathrm{SA}[i+1] - 1$ gives $\phi(i'+1) = \phi(i) + 1$. Thus, there are also at most $r$ values for which $\phi(i'+1) \neq \phi(i') + 1$ and Nishimoto and Tabei's approach gives us LF, $\phi$ and $\phi^{-1}$ in $O(r)$-space. By inserting additional rows into the table, Nishimoto and Tabei show how to make them constant-time (proof in Chapter 5), but potentially double the size of the table [46].

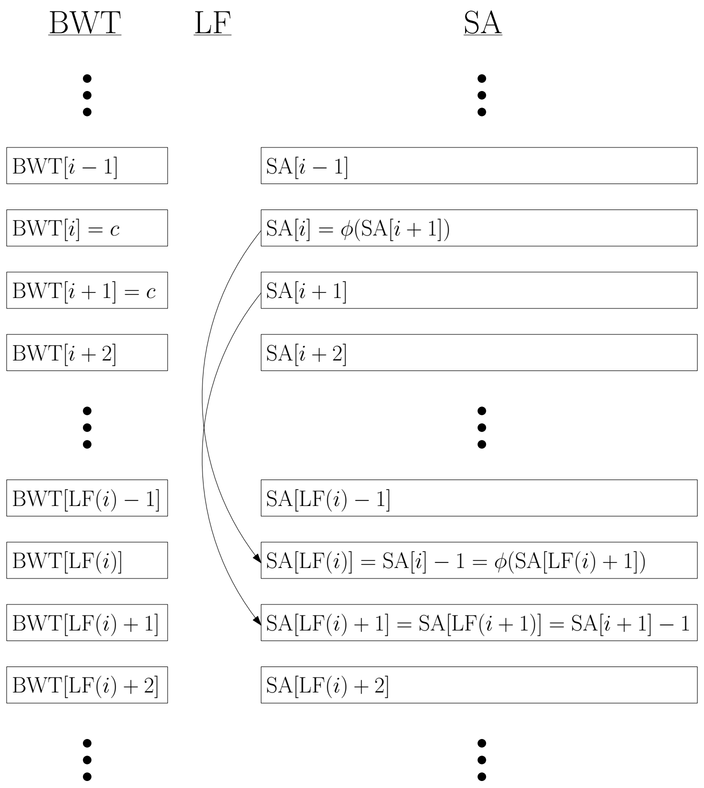
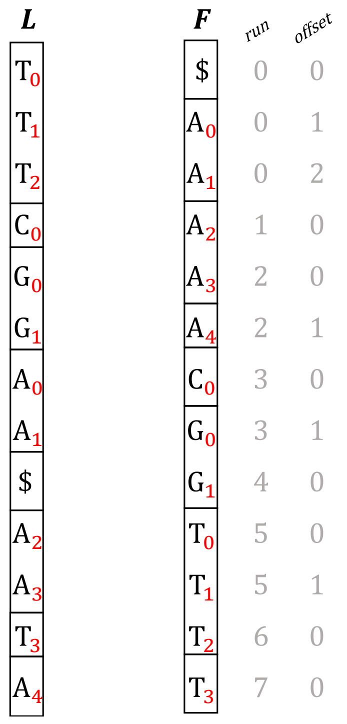


Figure 3.5: Illustrates that given $BWT[i] = BWT[i+1]$ it follows that $\phi(SA[i+1]) = \phi(SA[i+1]-1)+1$ (figure from [11]).

## 3.3 Table-Lookup with Runs/Offsets

To evaluate the practicality of table-lookup, we move towards an implementation *without* constant-time bounds; we predict in practice that the balancing of runs in $L$ and $F$ will lead to few scans on real datasets. Storing run-head positions $i$ and LF($i$) requires $r \log n$ bits. Instead, we can replace them by directly storing the offsets LF($i$) $- i'$ where $i'$ is the first position in the run containing LF($i$). This moves us towards a positional structure where a position $q$ is associated with pair $(k, d)$ such that the $k$th run contains $q$ and $d$ is the offset of $q$ in that run. Where $k'$ is the stored run containing the mapping of the $k$th run-head and $d'$ is the offset of the mapping within that run, then LF($q$) $=$ LF($k, d$) $= (k', d + d')$. To find the correct run containing $k'$ with offset $d + d'$, we sequentially scan while truncating the offset until we find a run whose length contains it; a reformulation identical to the prior approach.

For the $k$th run, we now store the run-head character $c$, the run-length $\ell$, the destination of its run-head mapping $k'$, and the offset of this mapping $d'$ within that run. Where explicit storing of run-head positions $i$ and LF($i$) required $\log n$ bits each, we can now store mappings $k'$ in $\log r$ bits each, lengths whose sum is at most $n$, and offsets in $\log n$ bits each (which should be smaller in practice). Further, our notion of LF is now over positions $k$ and $d$ where $k$ lets us access a table entry and $d$ describes which character in a run we'd like to map. Figure 3.6 shows a completed table for our altered approach. This formulation of LF given by our table-lookup is straightforward to implement.



| | character | length | destination | offset |
|---|---|---|---|---|
| 0 | T | 3 | 5 | 0 |
| 1 | C | 1 | 3 | 0 |
| 2 | G | 2 | 3 | 1 |
| 3 | A | 2 | 0 | 1 |
| 4 | $ | 1 | 0 | 0 |
| 5 | A | 2 | 1 | 0 |
| 6 | T | 1 | 7 | 0 |
| 7 | A | 1 | 2 | 1 |

Figure 3.6: A completed table-lookup approach to LF, where the run/offset positional approach with respect to the $L$ column is shown. For the first run of $T$s of length 3, we see its first character maps to run 5 with offset 0.

## 3.4 LF **Table in Practice**

Given a constructed RLBWT, we build our table using a two pass scan of Algorithm A.1 (in Appendix); first read the characters and lengths, then compute their containing runs and offsets from $\sigma$ stacks each holding the next position for each run-head character in $L$. The LF-mapping given a run $k$ and offset $d$ is computed using the sequential scan of Algorithm A.2 (in Appendix) which truncates offsets while checking rows of the table. The approach was implemented in C++[1] using a naive representation of one byte for characters and 64-bit integers for other fields; each stored as a row to preserve locality during steps.

To compare the speed of `table-lookup`, we experiment against conventional LF steps using the $r$-index method [20]; `rle-string` supporting rank/select on run-heads with sparse bitvectors marking run-head positions and lengths. A reference text containing 1370 E. Coli genomes (total file size 7GB) is described in Table 3.1. We measure the time to invert the text completely starting at the first position which precedes the terminal character and LF step $n$ times until wrap around to the terminal character. A randomly sampled 100,000,000 LF steps (using Mersenne-Twister pseudo-random generation with a seed of 23) are also measured. C++ code was compiled with flags `-O3 -DNDEBUG -funroll-loops -msse4.2`. We performed our experiments on a Linux server with an Intel® Xeon® Silver 4214 CPU running at 2.20GHz with 32 cores and 100 GB of memory.

| | |
|---|---|
| $n$ | 7,071,536,291 |
| $r$ | 80,621,220 |
| $n/r$ | 87.7131 |
| $\log(r)$ | 26.2647 |
| $\log(n/r)$ | 6.4547 |

Table 3.1: For a concatenated text of 1370 E. coli genomes, the length $n$ of their concatenation and runs $r$ in their BWT with related stats.

| **Metric** | `table-lookup` | `rle-string` |
|---|---|---|
| Construction (s) | 21.68 | 14.47 |
| Load (s) | 9.98 | 0.46 |
| Invert Time (s) | 1,943.98 | 12,372.30 |
| Invert Avg. (ns) | 274 | 1,749 |
| Sample Time (s) | 10.57 | 176.66 |
| Sample Avg. (ns) | 106 | 1,766 |
| Size (GB) | 2.56 | 0.18 |

Table 3.2: Results comparing times in seconds $s$ and nanoseconds $ns$ for constructing, loading, and performing LF steps for `table-lookup` and `rle-string`. Size is disk space in GB of the data structures.

Table 3.2 shows the results of experiments for `table-lookup` and `rle-string`: the average time per step for inversion/samples (using wall time), and the construction/load time for each method when an RLBWT is given (using Unix's `usr/bin/time`). The lookup approach is 6.38 times faster for inversion and 16.66 faster for sampled steps on average. Figure 3.7 shows that the number of scans across all LF steps is

[1]Code available at `https://github.com/drnatebrown/r-index-f/blob/master/include/r_index_f/LF_table.hpp`

small[2]; 76% require no scans and 98% less than 5, with long scans increasingly rare despite a maximum scan of 1620. However, `table-lookup` was 14.08 times larger than `rle-string` and slower in construction/loading. Despite being $O(r)$-space, storing integers is costly and motivates the compression schemes of the next chapter.

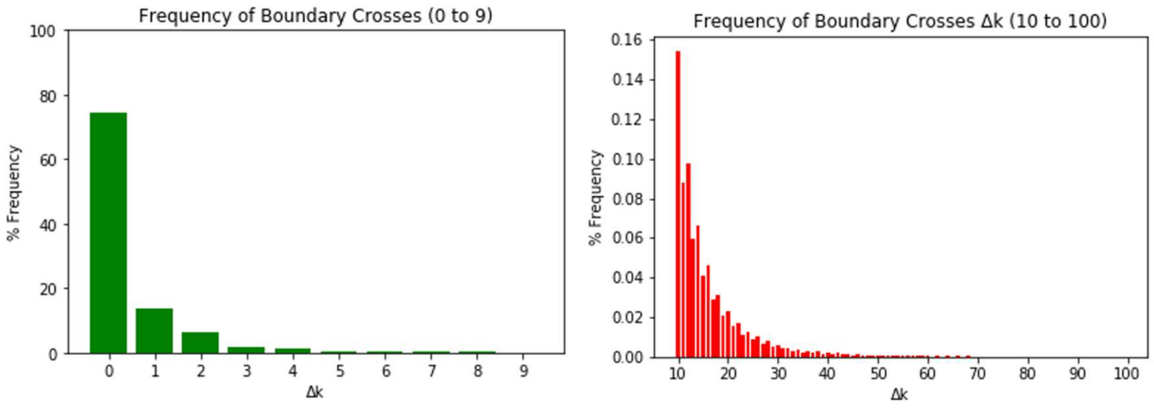


Figure 3.7: The number of scans $\Delta k$ required to perform LF steps for 1370 E. coli genomes. Left plot is restricted to range 0..10 and right from range 10..100. We note longer scans (with max 1620) similarly become decreasingly rare.

[2]Results from fellow student Sana Kashgouli's thesis shows that linear scans for $\phi$ are not as quick in practice.

# Chapter 4

# Compressed LF Table

The approach of Chapter 3 suggests that table lookup for LF can be efficient in practice on a genomic dataset even without constant-time guarantees. However, our simple formulation is large, requiring integer values to be stored. For pan-genomics datasets this size is not competitive with other approaches. In this chapter, we explore how to compress the table while supporting count queries (which rely on LF steps). Compressing the table column-wise is effective, but for locality concerns this is done while breaking rows of the table into blocks. Experiments show our approach can be made competitive in time/space with other methods. This Chapter's results correspond heavily to "RLBWT Tricks"[1] by myself, Travis Gagie, and Massimiliano Rossi [11].

## 4.1 Count Queries

Count queries are efficiently computed using the full SA$[0..n-1]$ by using binary search to find the range of suffixes which share the pattern as a prefix [39]. The FM-index finds this same range by backwards searching through the BWT for a pattern $P[0..m-1]$ by starting at its smallest suffix and increasing suffix lengths [17]. Starting at the range of $P[m-1]$, which is easily found using the representation of the $F$ column, we backwards step for the next character $P[m-2]$ for the first and last occurrence in that range of the BWT. It follows that the next range are all suffixes beginning with the pattern processed so far. Continuing this approach, right to left over the pattern, we return the length of the final range, or 0 if we fail to find any matching occurrences.

Since rank operations are directly computed during LF steps for an FM-index or conventional RLBWT based approaches, finding the start and end characters for a range is simple. For table-mapping, one option is a linear scan that is worst case

[1]This section does not contain self citations.

$O(r)$-time. Alternatively, we can suport rank/select over the run-head characters quicky using a wavelet tree. For a character $c$ and current range $[s, e]$, assume we know $s$ belongs to run $j_s$ and $e$ belongs to $j_e$ (we do not need to know $s$ and $e$). Then the next range starts at BWT[LF($s'$)] where $s'$ is the first character in $j_{s'} = select_c(rank_c(j_s + 1))$. Symmetrically, the next interval ends at LF($e'$) where $e'$ is the last character in the run $j_{e'} = select_c(rank_c(j_e))$. Since our table-lookup computes LF using run/offsets, we also sample run-head positions to recover $s$ and $e$ at the final stage and return the count.

## 4.2 Column-Wise Compression

A row of our LF table appears to have no simple properties to exploit for compression. Instead, we apply column-wise compression. The components of our table in columns becomes: characters $R[0..r-1]$ where $R[i]$ is the distinct character of the $i$th run, lengths $Len[0..r-1]$ of runs, destinations $K[0..r-1]$ where $K[i]$ is the run containing $i$'s run-head mapping, and $D[0..r-1]$ corresponding to the offset of that run-head mapping in the containing run. The sampled run-head positions (an increasing integer sequence) needed for calculating the final count are stored in a sparse bitvector supporting rank. However, we require a rank to change run/offset into index only after performing all LF steps.

Our formulation of rank will require rank/select on the run-heads $R[0..r-1]$. A wavelet tree achieves this whilst providing compression, and building it Huffman shaped [37] optimizes queries for high-frequency characters. However, for datasets of small alphabets, we could ignore a tree structure and store an $r$ length bitvector for each distinct character which marks where they occur. Genomic datasets in practice may support queries on only a few characters, i.e. the nucleobases $\{A, C, G, T\}$ for DNA alignment. In this case, using $4r$ bits is not much worse than the $2r$ bits needed for their smallest encoding when compared to the total size of the index (where other components such as length require $r \log n$ bits to encode). The operations rank/select/access are constant time on our full bitvectors, and should be faster than a wavelet tree in practice.

Both lengths $Lens[0..r-1]$ and offsets $D[0..r-1]$ have similar properties. Each offset cannot be longer than the length that contains it; the average of the offsets is always less than the average length. Further, the sum of lengths cannot exceed $n$. Despite both requiring worst case $\log n$ bits per element we expect the median to be comparatively small even on repetitive datasets based on prior experiments [52]. Our "naive" approach is to compress them both using DACs, allowing direct access which is fast and compact for integers whose bit representation is small. We use the implementation of SDSL [21] which adopts Brisboa et al.'s [9] approach using dynamic programming to choose the optimal chunk sizes at each level. Preliminary experiments found this implementation to be more efficient on average than using a fixed chunk size.

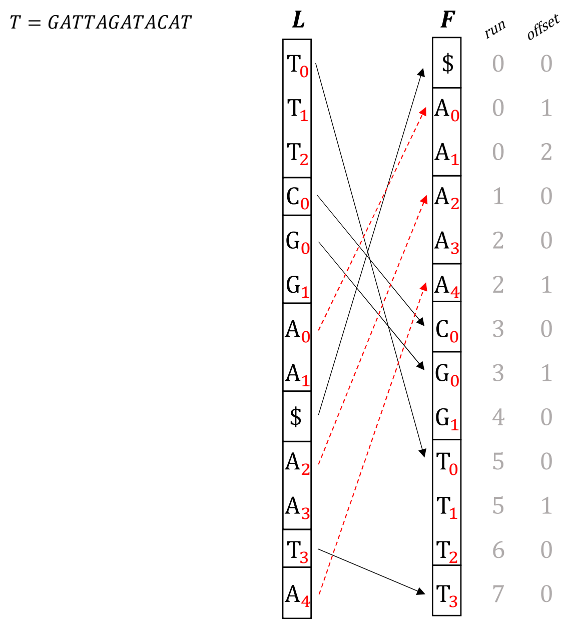
### 4.2.1 Run-Head Mappings

The run-head mapping destinations $K[0..r-1]$ do not form a non-decreasing sequence. However, as Figure 4.1 visualizes, the subsequences of $K$ corresponding to each of the $\sigma$ distinct characters form non-decreasing sequences. For a character $c$, we have that $R[i] = R[j]$ where $i < j$ with $rank_c(i) < rank_c(j)$ and thus their containing run-head destinations follow $K[i] \leq K[j]$ (they could map to the same run, in which case $D[i] < D[j]$). If we reduce the alphabet to chosen queryable characters (e.g. $\{A, C, G, T\}$) like we discussed on the run-heads we similarly require subsequences for only the chosen characters (all LF steps during count queries involve characters occurring in the pattern).

Various delta encoding schemes (see Chapter 2) apply to our decomposition of $K[0..r-1]$ into $\sigma$ non-decreasing subsequences $K_c[0..r_c-1]$ for distinct characters $c$ with $r_c$ run-head occurrences. Although Elias-Fano coding could be used, the simpler approach for prefix sum of deltas using bitvectors should be faster and still sufficiently small; we mark at most $r$ positions $K_c[i] + i$ in a bitvector of at most size $r + r$, achieving worst case $O(r)$ bits for each subsequence. We can also store the first value $K_c[0]$ explicitly as a base to our bitvectors. To illustrate, consider sequence $K_c[11, 16, 19, 21]$; we store $K_c[0] = 11$ and bitvector

$$10000010001001$$

and compute $select(k) - k$ to return the numbers of 0s before the $k$th set bit (the 0s



| | character | length | destination | offset |
|---|---|---|---|---|
| 0 | T | 3 | 5 | 0 |
| 1 | C | 1 | 3 | 0 |
| 2 | G | 2 | 3 | 1 |
| 3 | A | 2 | 0 | 1 |
| 4 | $ | 1 | 0 | 0 |
| 5 | A | 2 | 1 | 0 |
| 6 | T | 1 | 7 | 0 |
| 7 | A | 1 | 2 | 1 |

Figure 4.1: Highlighted LF-mappings of run-heads (destinations) for character $A$ form a non-decreasing subsequence; this is true for all $\sigma$ characters because of rank properties.

between each bit representing the delta values). For position $i = 2$, we have

$$K_c[0] + select(i+1) - (i+1) = 11 + 11 - 3 = 19 = K_c[i]$$

where $i$ corresponds to the $(i+1)$th bit due to 1-based select.

To compare the bitvector approach, we consider two alternatives. A relatively simple approach is to store deltas in a DAC (since values should be small on average) and sample values at a constant rate so retrieving the prefix sum is a bounded scan. Sampling and DACs can also be used in a linear interpolation approach. We sample values in $K_c$ at rate $s$, so each position $i$ with value $y$ has a prior sample $x$ and next sample $z$. Assuming a linear increase between samples, we define a weighted average

$$\epsilon = x + (z - x) \cdot (i - s \cdot (\lfloor i/s \rfloor)/s)$$

and store the difference $\Delta = y - \epsilon$ in a DAC. Given $i$ and $s$, we lookup $x$, $y$, and compute $\epsilon$; we recover $y$ by accessing $\Delta$ and adding the difference $\epsilon$. We expect $\Delta$ values to be small if growth between samples is linear, and can store a bitvector marking positive/negative sign to ensure recovery is still correct.

### 4.2.2 Table Blocking

Column-wise compression schemes for each element of the table save space, but they might affect locality. Where storing entries in a row should be stored contiguously in memory, each compressed column will likely not maintain locality per access. To attempt to mitigate these concerns, we consider partitioning the table rows into blocks of size $B$. Using fixed $B$, modular arithmetic is constant-time and allows easy reduction of positions into their corrct block and position. In order to maintain rank over the run-heads $R$, for each distinct character we store the position of the first run preceding the start block and following the end of the block. This scheme also aligns with compression of run-head destinations, storing the value of the first value in a block for each character since subsequent values are non-decreasing. Figure 4.2 shows experiments on block size using pan-genomic datasets explained in the subsequent section. The best block sizes are very large and do not fit in high-level cache, but we find $B = 2^{20}$ to achieve the best space and choose it for our experiments.

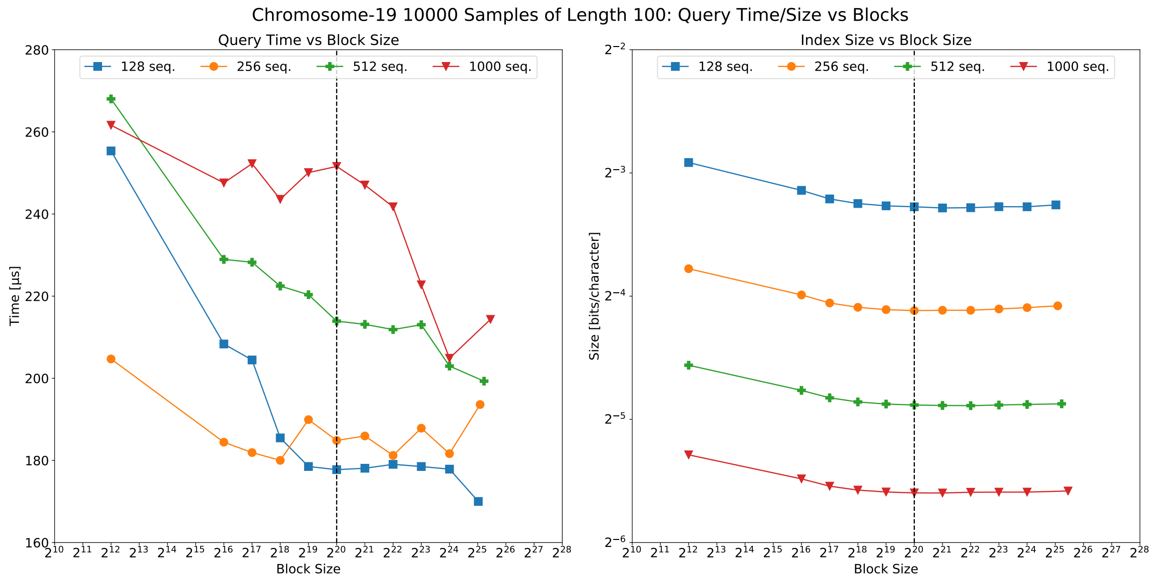


Figure 4.2: For pan-genomic data introduced subsequently, the time/space vs choice of block size is shown. Dashed line represents the chosen default of $2^{20}$. Rightmost value represents storing the entire table without blocking (one block).

The implementation hierarchy is shown in Figure 4.3; our proposed choices for compression of columns, queryable alphabet reduction, and table blocking are summarized. These choices were found during preliminary experimentation to achieve better results in query time for counting than the methods which were not chosen.[2]

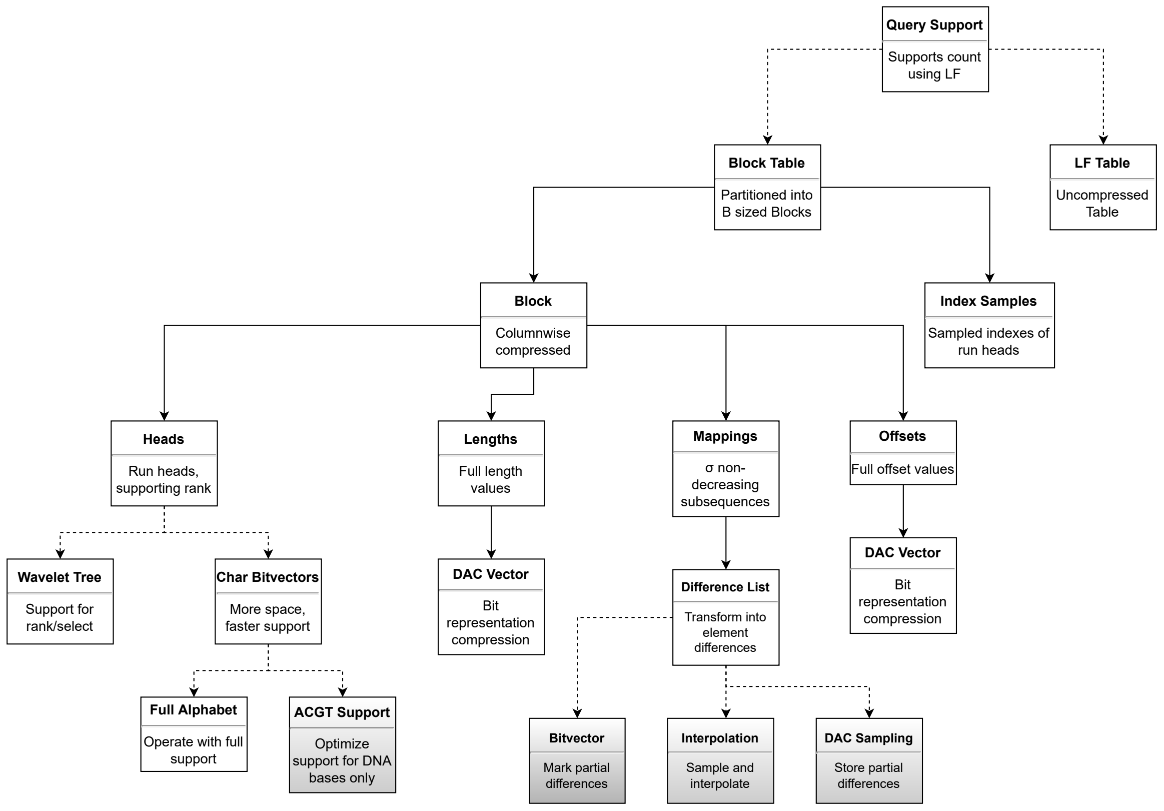


Figure 4.3: Shows hierarchy of implementation, outlining choices for constructing the table. Solid lines show required components; dotted lines denote that one can be chosen. Shaded nodes show paths that are presented in Section 4.3

## 4.3 Experiments

Our code was written in C++ and compiled with flags `-O3` `-DNDEBUG` `-funroll-loops` `-msse4.2` using data structures from `sdsl-lite` [21]. We performed our experiments on a server with an Intel® Xeon® Silver 4214 CPU running at 2.20GHz with 32 cores and 100 GB of memory. Our code is available at `https://github.com/drnatebrown/r-index-f.git`. Count query times were measured using Google Benchmark[3], and construction with the Unix /usr/bin/time. Our benchmarks only discuss the average time to compute the queries and do not involve statistical analysis.

[2]The source code still implements all the approaches shown in Figure 4.3.
[3]Available at `https://github.com/google/benchmark`.

### 4.3.1 Data Structures

For our table lookup implementations, we partition into blocks of size $B = 2^{20}$ and sample every 16th run-head position into a sparse bitvector. Since worst case scans and lengths/offsets are affected by the maximum run-length, we also explore variations capping run sizes by splitting runs into multiple rows. We compared the following data structures:

**lookup-bv** table-lookup with bitvector marking deltas with 0s, recovered with select.

**lookup-int** table-lookup with linear interpolation between sampled values.

**lookup-dac** table-lookup with DACs storing deltas with sample rate 5.

**lookup-split2** table-lookup using *lookup-bv* where runs larger than twice the average length $n/r$ are split.

**lookup-split5** table-lookup identical to *lookup-split2*, except runs larger than five times the average length $n/r$ are split.

**wt-fbb** fixed-block boosting wavelet tree of [22] using default parameters; implementation at `https://github.com/dominikkempa/faster-minuter`.

**rle-string** run-length encoded string of the $r$-index [20]; implementation based off `https://github.com/nicolaprezza/r-index`.

**RLCSA** the BWT component[4] of the run-length encoded compressed suffix array of [38] using default parameters; implementation at `https://github.com/adamnovak/rlcsa`.

### 4.3.2 Datasets

We tested our data structures for construction and query on 4 collections of 128, 256, 512 and 1000 haplotypes of chromosome 19 from the 1000 Genomes Project [57] (`chr19`) and 4 collections of 100k, 200k, 300k, 400k SARS-CoV2 genomes from the EBI's COVID-19 data portal [24][5] (`Sars-CoV2`). Each set is a superset of the previous one, with their BWT computed by concatenating the sequences in the collection. Our goal with these datasets is to verify the ability of the index to handle human DNA for alignment and viral genomes for classification. However, we have not attempted to use full human genomes and larger collections, or even other genomic data; we

[4]We build the data structure without suffix-array sampling.

[5]The complete list of accession numbers is reported in the repository.

aim to show only that our data structures are practical to motivate further work for full-scale applications. Table 4.1 describes the lengths $n$ and ratio $n/r$ of the datasets.

| Name | Description | Copies | $n/10^6$ | $n/r$ |
|---|---|---:|---:|---:|
| `chr19` | Human chromosome 19 | 128 | 7568.01 | 222.24 |
| `chr19` | Human chromosome 19 | 256 | 15136.04 | 424.93 |
| `chr19` | Human chromosome 19 | 512 | 30272.08 | 771.54 |
| `chr19` | Human chromosome 19 | 1,000 | 59125.12 | 1287.38 |
| `Sars-CoV2` | Sars-CoV2 genomes database | 100,000 | 2979.01 | 881.16 |
| `Sars-CoV2` | Sars-CoV2 genomes database | 200,000 | 5958.35 | 977.19 |
| `Sars-CoV2` | Sars-CoV2 genomes database | 300,000 | 8944.37 | 1178.00 |
| `Sars-CoV2` | Sars-CoV2 genomes database | 400,000 | 11931.17 | 1328.92 |

Table 4.1: Table of the different datasets. In column 1 and 2 we report the name and description of the datasets, in column 3 we report the number of sequences in the collection, in column 4 we report the length of the file, and in column 5 the ratio of the length to the number of runs in the BWT.

### 4.3.3 Construction

In Figure 4.4 we report the time and memory for construction of the data structures for the `chr19` and `Sars-CoV2` datasets. `RLCSA` is omitted, since it is the only data structure not built using prefix free parsing (PFP) [8], and its construction time far exceeded the other methods.

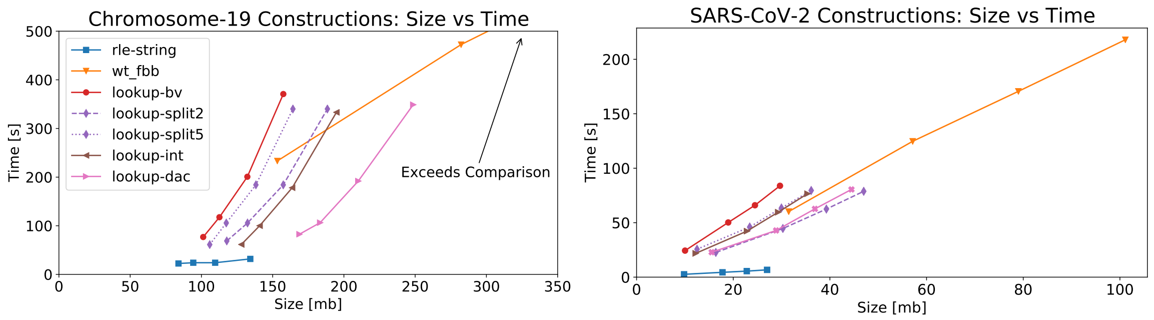


Figure 4.4: Construction for `chr19` of 128, 256, 512 and 1000 copies (left) and `Sars-CoV2` of 100k, 200k, 300k and 400k copies (right). Copies increase for an instance plotted left to right. For `chr19` we partially omit `wt-fbb` for being magnitudes larger than other values (approximately 4 times slower and larger than `lookup-bv` for 512 copies and similarly 5 times slower and 7 times larger for 1000).

### 4.3.4 Query

To query the data structures we performed counting queries for 10000 randomly chosen substrings each of length 10, 100, 1000 and 10000. In Figure 4.5 and 4.6 we report the time and memory for querying of the data structures for the `chr19` and `Sars-CoV2` datasets respectively.

## 4.4 Discussion

For table lookup implementations, `lookup-bv` and its variants (`lookup-split2` and `lookup-split5`) perform better than `lookup-int`, `lookup-dac` in time and space. For query lengths greater than 10 on `chr19`, `lookup-bv` variants are faster and larger than `rle-string`, while slower and smaller than `RLCSA`; we occupy a time/space trade-off between these implementations. The space of `wt-fbb` makes it an outlier despite best speeds for various queries.

On `Sars-CoV2`, table-lookup performs well on queries of length 10, with `lookup-split2` the fastest implementation and other approaches competitive in both time and space. For query lengths greater than 10, the non-splitting approaches (`lookup-bv`, `lookup-int`, `lookup-dac`) are slowest. With splitting approaches, we are comparable to `rle-string` in time but worse in space. Although again an outlier in space, `wt-fbb` performs fastest, with `RLCSA` occuping the least space with comparable speed to `wt-fbb`.

For construction, table-lookup approaches are slower than `rle-string` across all data. Across all table-lookup approaches `lookup-bv` and its variants are the definitive choice in both space and construction time. `RLCSA` is more space-efficient and faster on `Sars-CoV2` where and is a clear winner across all of the data structures in the experiment. This motivates table lookup to also speed up `RLCSA`; however, we note adding support for $\phi$ and $\phi^{-1}$ (thus, supporting locate) to `RLCSA` is still an open problem.

## 4.5 Worst Case Scans

Our run capping approaches are superior to `lookup-bv` for long query lengths and as $n/r$ rises. Figure 4.7 shows the number of scans required across LF steps during

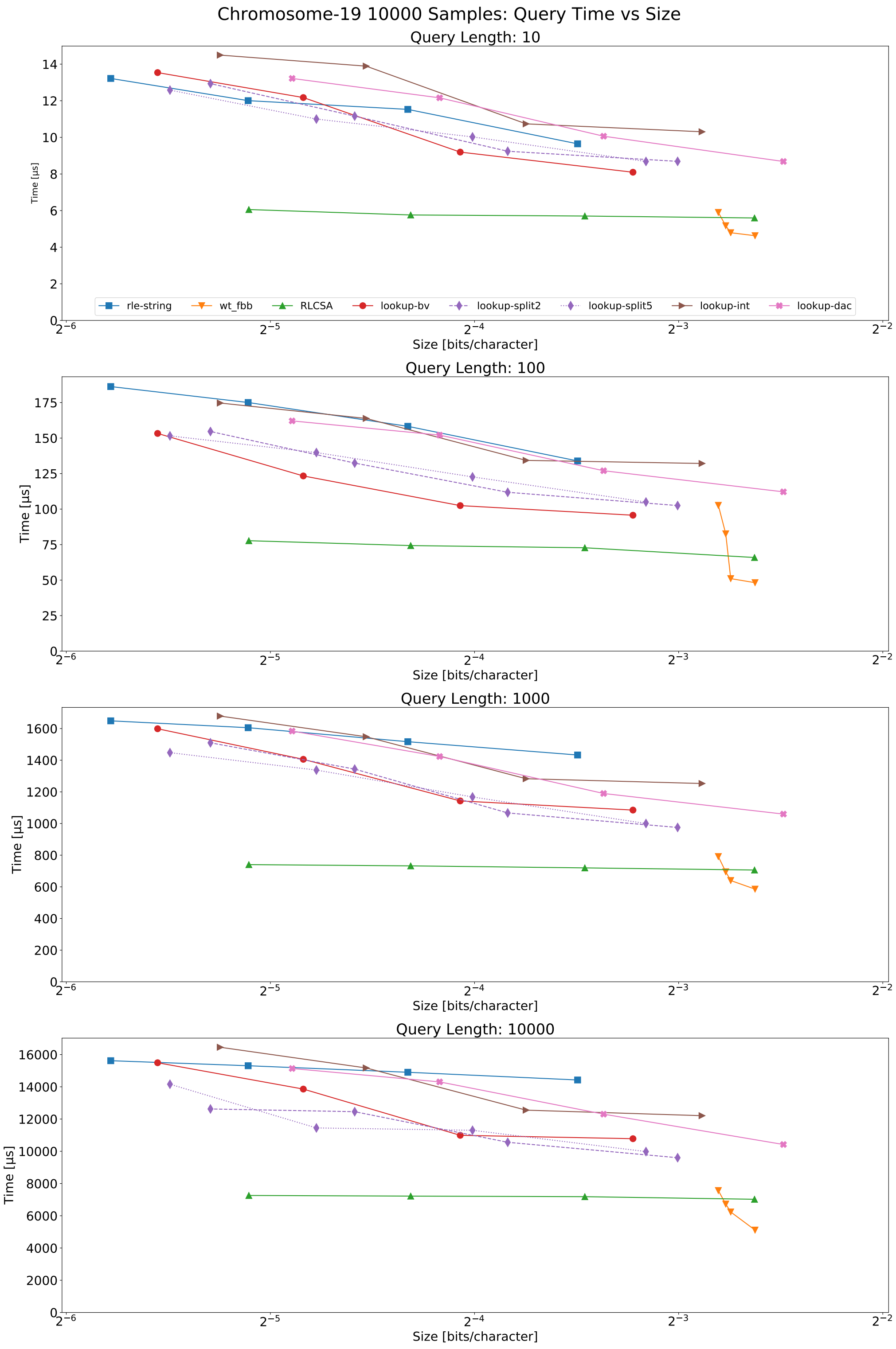


Figure 4.5: The time per query to count the occurrences of 128, 256, 512 and 1000 copies of `chr19` for 10000 randomly-chosen substrings of length 10, 100, 1000 and 10000 each. Copies for a single line are read from largest number of copies to smallest, left to right. The x axis is logarithmically scaled with base 2.

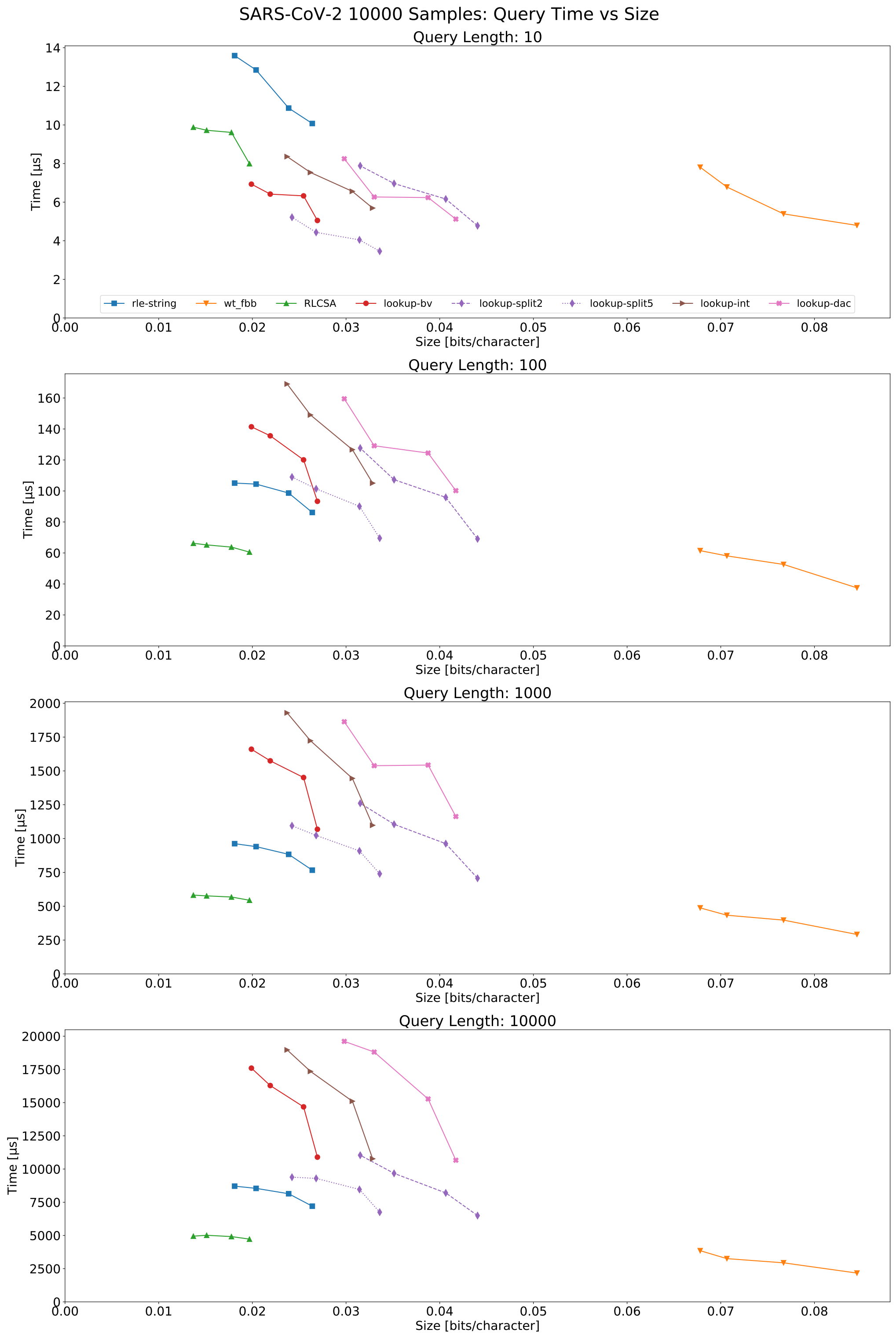


Figure 4.6: The time per query to count the occurrences of 100k, 200k, 300k and 400k `Sars-CoV2` copies for 10000 randomly-chosen substrings. Results are given for queries of length 10, 100, 1000 and 10000. Copies for a single line are read from largest number of copies to smallest, left to right.

count queries of length 100 for `chr19`. Although the distribution is similar across all copies near zero (majority requiring few rows to be scanned) worst cases become more prevalent and longer as the number of copies grows. This formulation of table-lookup is still competitive on pan-genomic datasets evaluated, but alternative formulations to support count could improve time/space. Due to success of run capping and evidence that increasingly repetitive text collections (growing $n/r$) cause longer and more likely scans, we are motivated to explore methods to bound them, such as Nishimoto and Tabei's constant-time theoretical guarantee [46].

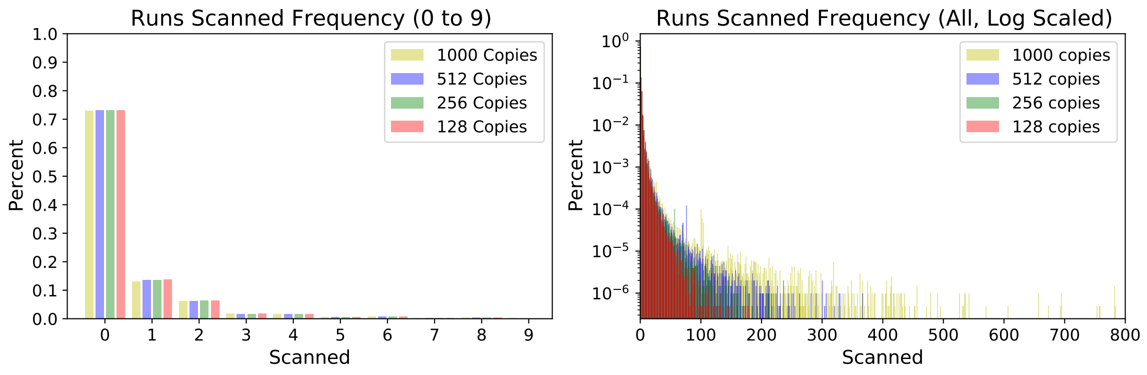


Figure 4.7: Frequencies in percentage of runs scanned for LF steps required across 10000 count queries of length 100 for 128, 256, 512 and 1000 copies of `chr19`. Left plot shows steps scanning 0 to 9 rows; right plot shows all scans (log scaled).

## 4.6 Profiling Components

Computing count queries requires each component of the table; the columns and our methods of compressing them. We time each component for a step of each count query to identify which components are slower for more repetitive datasets. Where a step corresponds to shrinking of the count range with LF, each component is measured when it is used: run-heads for access/rank/select, destinations for one access, offsets for one access, lengths for access during scans. Figure 4.8 shows the results for increasingly repetitive text collections for `chr-19` using `lookup-bv`. Although all components are slower as the average run increases, the time spent scanning and accessing lengths grows dis-proportionally larger. Longer runs both increase the longest scan, but also increase the access time using a DAC to retrieve their lengths.

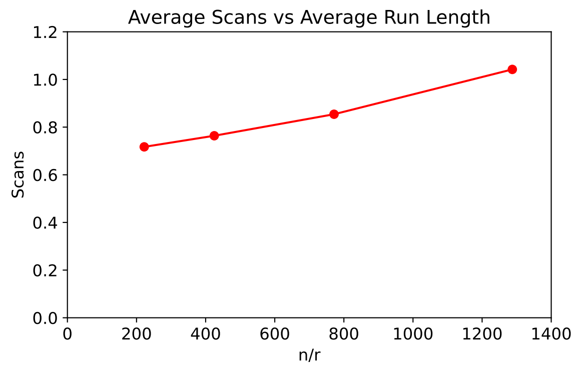


Figure 4.8: Left plot shows that the average scan as the average run length increases for `chr-19`; this relates to Figure 4.7 and long worst cases. The right plot shows the average time required for major components at each step of a count query for `chr-19` of 128, 256, 512, and 1000 copies.

# Chapter 5

# Runny Permutations

Using a table-lookup approach for LF is practically efficient within our experiments on genomic data, but is still subject to worst case $\Omega(r)$ scans. On these pan-genomic datasets the majority of steps require no scans, but increasingly repetitive texts cause the worst cases to grow. In this chapter, we discuss simple methods to bound scans given our framework which motivates a true row splitting approach. Nishimoto and Tabei's result is presented which splits runs to achieve constant-time LF steps in $O(r)$-space [46]. Our formulation hints at how we can actually perform this splitting procedure; for any "runny permutation" which has $r$ runs which are permuted in sequence we insert boundaries using its inverse permutation to achieve constant-time mappings. A practical implementation is given, and we explore its effects on our compressed table in computing count queries.

## 5.1 Bounding Scans

Results in Chapter 4 showed that naively splitting runs based on a maximum allowed length outperformed the default method in speed. The worst case scan cannot be worse than the longest run; a length $\ell$ run in the $F$ column cannot cross more than $\ell$ boundaries in the $L$ column. However, our naive run splitting also shortens the lengths of runs and thus the chunk level required during DAC access. Note that we can also support permutations in $O(r)$-space using our table by storing a sparse bitvector of runs and $O(\log \log r)$-time rank to find predecessors. As an alternative to motivate row splitting, consider directly bounding maximum scans by addressing what causes them; offsets into a run in $F$ which cross many run boundaries from the destination run of an LF-mapping.

To avoid introducing rows into the table, consider sampling offsets which intersect run boundaries. Figure 5.1 visualizes sampling every $x$th value corresponding to a run-head position. By supporting predecessor queries on these samples, we can skip some scans; if our offset has the $p$th value as predecessor we skip $p \cdot x$ table entries and scan at most $x$ additional rows. Across all runs, we sample at most $r/x$ entries since each sampled offset corresponds to a distinct run in $L$. Using a simple method like binary search we improve the worst-case scan to $O(\log r)$-time whilst maintaining $O(r)$-space. In practice, sampling plain bitvectors may be even faster while still small. Using the uncompressed formulation of Chapter 3 we invert a salmonella reference genome from the NCBI repository [15] with $n = 145594456$ and $r = 12823512$. Table 5.1 shows that despite theoretical improvements, average results are slower; it is likely that introducing such an approach slows down the average case which is already very fast.

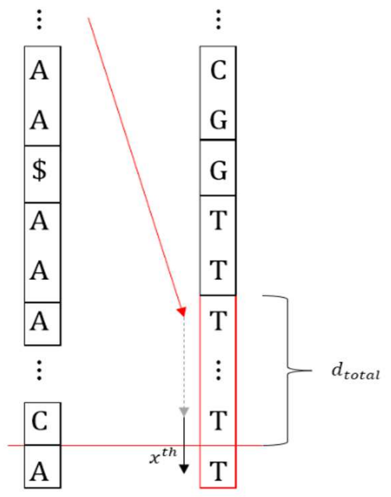


Figure 5.1: Sample every $x$th offset $d_{total}$ which crosses a boundary.

| | *Search* | *Bitvector* | *Unsampled* |
|---|---|---|---|
| $x = 2$ | | | |
| **Time (s)** | 28.9127 | 33.6014 | 20.5741 |
| **Space (GB)** | 0.3585 | 0.3750 | 0.3206 |
| $x = 4$ | | | |
| **Time (s)** | 25.4598 | 27.3216 | 20.5741 |
| **Space (GB)** | 0.3242 | 0.3261 | 0.3206 |
| $x = 8$ | | | |
| **Time (s)** | 23.4447 | 22.6571 | 20.5741 |
| **Space (GB)** | 0.3208 | 0.3210 | 0.3206 |
| $x = 16$ | | | |
| **Time (s)** | 21.0462 | 21.2288 | 20.5741 |
| **Space (GB)** | 0.3206 | 0.3206 | 0.3206 |

Table 5.1: Results of offset sampling with rate $x$ for inversion of a full salmonella genome using either binary search or a plain bitvector to find predecessors.

## 5.2 Row Splitting

Storing accessory data structures with runs to bound scans affects our efficient cases when scans are low. Instead, consider splitting rows at these positions where offsets cross boundaries. If we split rows when their corresponding run in $F$ crosses boundaries, we limit the length of that scan without changing the approach for unrelated runs. However, it is not clear if this solution works naively; Figure 5.2 shows that every row that is split also introduces another boundary for a run in $F$ to cross. The result of Nishitomo and Tabei proves that a splitting procedure will finish while inserting $O(r)$ rows into the table [46]. To see when this procedure converges, consider it as an interval weight problem between bitvectors $P$ marking run-head positions in $L$ and $Q$ marking the corresponding positions in $F$. Figure 5.3 shows the corresponding bitvectors given $L$ and $F$.

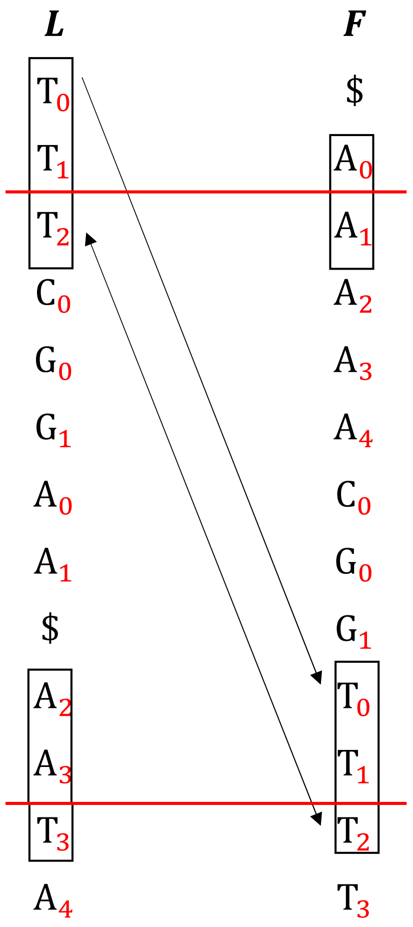

L
F
T0 T1 T2 C0 G0 G1 A0 A1 $ A2 A3 T3 A4
$ A0 A1 A2 A3 A4 C0 G0 G1 T0 T1 T2 T3

Figure 5.2: Inserting a boundary also splits a run found using the inverse permutation.

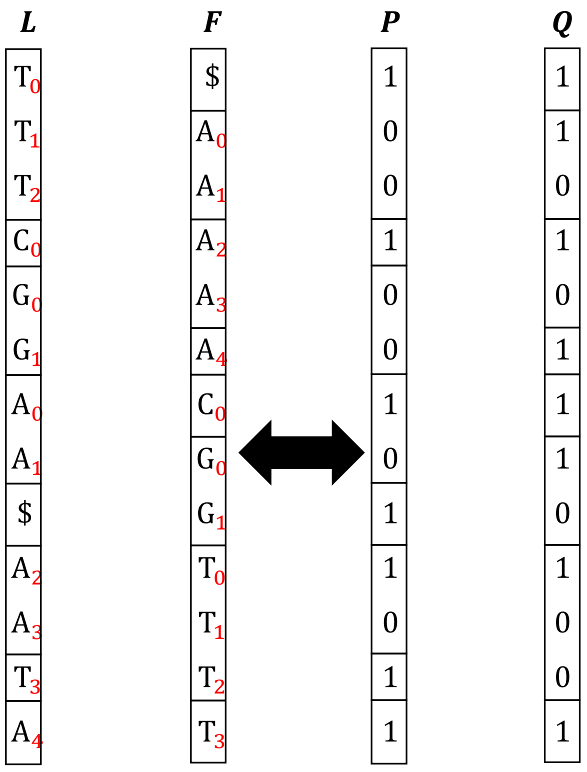

L
F
P
Q
T0 T1 T2 C0 G0 G1 A0 A1 $ A2 A3 T3 A4
$ A0 A1 A2 A3 A4 C0 G0 G1 T0 T1 T2 T3
1 0 0 1 0 0 1 0 1 1 0 1 1
1 1 0 1 0 1 1 1 0 1 0 0 1

Figure 5.3: Converts runs in $L$ and $Q$ into their corresponding bitvectors marking run-head positions.

Using bitvectors, bounding a scan becomes limiting the maximum number of bits in $P$ that overlap an interval in $Q$; let this number be the weight of an interval. Inserting a bit into $Q$ distributes the weight across the split intervals, but also introduces a bit in $P$ that affects the weight of another run. Since it introduces only one bit, each step increases the set bits of both $P$ and $Q$ by only one; let $P_i$ and $Q_i$ be these bitvectors after $i$ insertions. For a parameter $d$ and intervals with weight $w \geq d$

consider inserting at the $dth$ bit $P$ covering an interval in $Q$ such that the interval is split into one of weight $d$ and one of weight $w - d \geq d$. Let the set $E_i$ store intervals in $Q_i$ with weight $\geq d$ after $i$ insertions. This set must also have increased by at least $i$; with each step the first split interval of weight $d$ replaces the original interval, but we also must include[1] the second of weight $w - d \geq d$ since $w \geq 2d$.

Let $|P_i|$ denote the number of set bits in $P_i$ (or any bitvector). The intervals corresponding to the set $E_i$ contains at least $d|E_i|$ set bits and after inserting $i$ bits we have that $|P| \geq d|E_i|$. However, since $|E_i| \geq i$ then $|P_i| \geq d \cdot i$; it follows that we eventually reach an $i$ where splitting any interval would exceed the bits in $P_i$ and thus it must be of weight strictly less than $2d$. If we initially have $|P_0| = r$ set bits, then for any $i$ we have $|P_i| = r + i \leq d \cdot i$, so $r \geq i(d-1)$ and hence $i \leq \frac{r}{d-1}$. This ensures all interval weights are less than $2d$ after at most $\frac{r}{d-1}$ steps; for sufficiently small $d$ we achieve $O(1)$-time permutations in $O(r)$-space using Nishimoto and Tabei's result [46]. Where the initial proof uses a fixed $d = 2$ [46], our formulation is parameterized for time/space trade-off. Theorem 1 uses our discussion to give a formal proof for any permutation by adapting Nishimoto and Tabei's theorem [46] to our framework while introducing our parameter $d$.

**Theorem 1** (Permutation Scanning). *Let $\pi$ be a permutation on $\{0, \ldots, n-1\}$,*

$$P = \{0\} \cup \{i \ : \ 0 < i \leq n-1, \pi(i) \neq \pi(i-1) + 1\},$$

*and $Q = \{\pi(i) \ : \ i \in P\}$. For any integer $d \geq 2$, we can construct $P'$ with $P \subseteq P' \subseteq \{0, \ldots, n-1\}$ and $Q' = \{\pi(i) \ : \ i \in P'\}$ such that*

- *if $q, q' \in Q'$ and $q$ is the predecessor of $q'$ in $Q'$, then $|[q, q') \cap P'| < 2d$,*
- *$|P'| \leq \frac{d|P|}{d-1}$.*

*Proof.* We start by setting $P_0 = P$ and $Q_0 = Q$ and let $|P| = r$. Suppose at some point we have $P_i$ and $Q_i = \{\pi(i) \ : \ i \in P_i\}$. If there do not exist $q, q' \in Q_i$ such that $q$ is the predecessor of $q'$ in $Q_i$ and $|[q, q') \cap P_i| \geq 2d$, then we stop and return $P' = P_i$ and $Q' = Q_i$; otherwise, we choose some such $q$ and $q'$.

[1]We could increase by two because of the set bit in $P_i$, but this does not affect the proof.

We choose the $(d+1)$st largest element $p$ in $[q, q') \cap P_i$ and set $P_{i+1} = P_i \cup \{\pi^{-1}(p)\}$ and $Q_{i+1} = Q_i \cup \{p\} = \{\pi(i) \ : \ i \in P_{i+1}\}$. Since $q < p < q'$ we have $p \notin Q_i$ and so $\pi^{-1}(p) \notin P_i$. Therefore, $|P_{i+1}| = |P_i| + 1$ and so, by induction, $|P_{i+1}| = |P| + i + 1$.

Let $E_i$ be the set of intervals $[u, u')$ such that $u, u' \in Q_i$ and $u$ is the predecessor of $u'$ in $Q_i$ and $|[u, u') \cap P_i| \geq d$, and let $E_{i+1}$ be the set of intervals $[u, u')$ such that $u, u' \in Q_{i+1}$ and $u$ is the predecessor of $u'$ in $Q_{i+1}$ and $|[u, u') \cap P_{i+1}| \geq d$. When $\pi^{-1}(p)$ does not increase $|E_{i+1}|$, we have $E_{i+1} = (E_i \backslash \{[q, q')\}) \cup \{[q, p), [p, q')\}$, then $|E_{i+1}| = |E_i| + 1$ and, by induction, $|E_{i+1}| \geq i + 1$.

Since the intervals in $E_{i+1}$ are disjoint and each contain at least $d$ elements of $P_{i+1}$, we have $|P_{i+1}| \geq d|E_{i+1}| \geq d(i + 1)$. Since $|P_{i+1}| = r + i + 1$ and $|P_{i+1}| \geq d(i + 1)$, we have $r + i + 1 \geq d(i + 1)$ and thus $i + 1 \leq \frac{r}{d-1}$ and $|P_{i+1}| = |P| + i + 1 \leq \frac{d|P|}{d-1}$. It follows that we find $P'$ and $Q'$ after at most $\frac{r}{d-1}$ steps. $\square$

## 5.3 Practical Implementation

Formulating scanning as an interval weight problem using the proof of Theorem 1 illustrates how to perform row splitting in practice. Once we obtain $P'$, splitting maximal runs when constructing the table ensures a maximum scan given by our choice of $d$. To support $\pi^{-1}$ we can use a specific approach such as $\mathrm{LF}^{-1} = \mathrm{FL}$ using character ranks; however to support any permutation we can similarly create a table-lookup for $\pi^{-1}$ in $O(r)$-space. Note that using our boundary sampling, we can reduce scans to $O(\log\log r)$-time if we store samples in sparse bitvectors. Given $\pi$ we construct $P$ and $Q$ as sparse bitvectors and support rank/select, since performing $\pi^{-1}$ requires the predecessor and offset of a position in $Q$. To efficiently identify when an interval should be split we require a data structure which updates weights.

To identify intervals in $Q_i$ which have weight $w \geq 2d$ we maintain a maximum priority queue $H$ storing the starting positions of intervals ordered by weights. However, inserting a bit has many effects: reducing the weight to $d$ for the current interval, inserting a new interval of weight $w - d$, and incrementing the weight of another interval where we set a bit in $P_{i+1}$. Our approach is to use an indexable queue based on code by Sedgewick and Wayne [53]; we use an $O(r)$-space heap and an $O(r)$-space hash map to support indexing into the heap. Priority queue operations for insert and updating the maximum element are worst case $O(\log r)$-time, with expected $O(\log r)$-time for update due to using the hash function. Taking the maximum element gives us the position in $Q$ and its weight $w$.

To split for some interval position $q \in Q_i$ with weight $w \geq 2d$ we need to find the $(d{+}1)$st bit in $P_i$ starting at $q$; we first find the predecessor $p_{pred} = P_i$.rank$(q)$ of $q$ in $P_i$ and then the position to insert into $Q$ at $p = P_i$.select$(q_{pred}{+}d)$. We proceed by inserting into $P_i + 1$ using $\pi^{-1}$ and updating $H$; however rank/select on $P_i$ and $Q_i$ must be dynamic since we are inserting bits at each step. Prezza's DYNAMIC library [49] augments a B-tree to provide a dynamic sparse bitvector supporting insert/rank/select in $O(\log r)$-time and similar compression to our static sparse bitvector. Performing $O(r)$ total steps, this algorithm performs splitting in expected $O(r \log r)$-time[2] and $O(r)$-space. If we replace sparse bitvectors with plain ones it results in $O(n)$-space but also $O(n)$-time for splitting due to initialization stages; however, this option might be feasible depending on the dataset. Nishimoto, Kanda, and Tabei show how to split to achieve constant-time LF in worst case $O(n + r \log r)$-time and $O(r \log n)$ bits [45], but do so dynamically while building the RLBWT; our approach is a static alternative which used after other RLBWT construction methods. The splitting procedure is described in Appendix Algorithm A.3.

## 5.4 LF **Experiments**

To evaluate the result we perform row splitting with LF on the `chr19` dataset from Section 4.3, using 16, 32, 64, 128, 256, 512, and 1000 sequences, as increasing reptitiveness causes longer worst case scans. We experiment with splitting parameter $d$ in $\{2, 4, 8, 16, 32, 64, 128, 256, 512\}$ (since larger $d$ do not introduce any new rows) and measure the wall time and memory needed to perform the splitting. To evaluate in practice, we use `lookup-bv` both with and without splitting to compute count queries for `chr19` of 128, 256, 512, and 1000 copies and randomly sampled patterns of length 10, 100, 1000 and 10000 (akin to Section 4.3).

Our code was written in C++ and compiled with flags `-O3 -DNDEBUG -funroll-loops -msse4.2` using data structures from `sdsl-lite` [21] and `DYNAMIC` [49]. We

[2]Worst case $O(r^2 \log r)$-time due to hash map; we note that alternatives to the indexed heap such as a balanced binary search tree would improve to worst case $O(r \log r)$-time.

performed our experiments on a server with an Intel® Xeon® Gold 6248R CPU running at 3.00GHz with 48 cores and 1536 GiB of memory. Splitting code is available at `https://github.com/drnatebrown/r-permute`.

### 5.4.1 Splitting

To evaluate the affect on maximum scans, we perform splitting for various choice of $d$ on sequences of `chr19`. However, using sparse bitvectors for $P/Q$ and their dynamic variants caused slow runtimes which were did not finish in the allowed time for large datasets. Instead, we sacrifice space by using plain bitvectors to obtain splitting results quicker. Figure 5.4 shows the result of splitting for `chr19` of 16, 32, 64, 128, 256, 512, and 1000 copies. Table 5.2 shows the memory and time required for splitting parameter $d = 2$. We note that performing splitting for multiple $d$ is quicker, since the initialization of the data structures is slower than the splitting procedure itself.

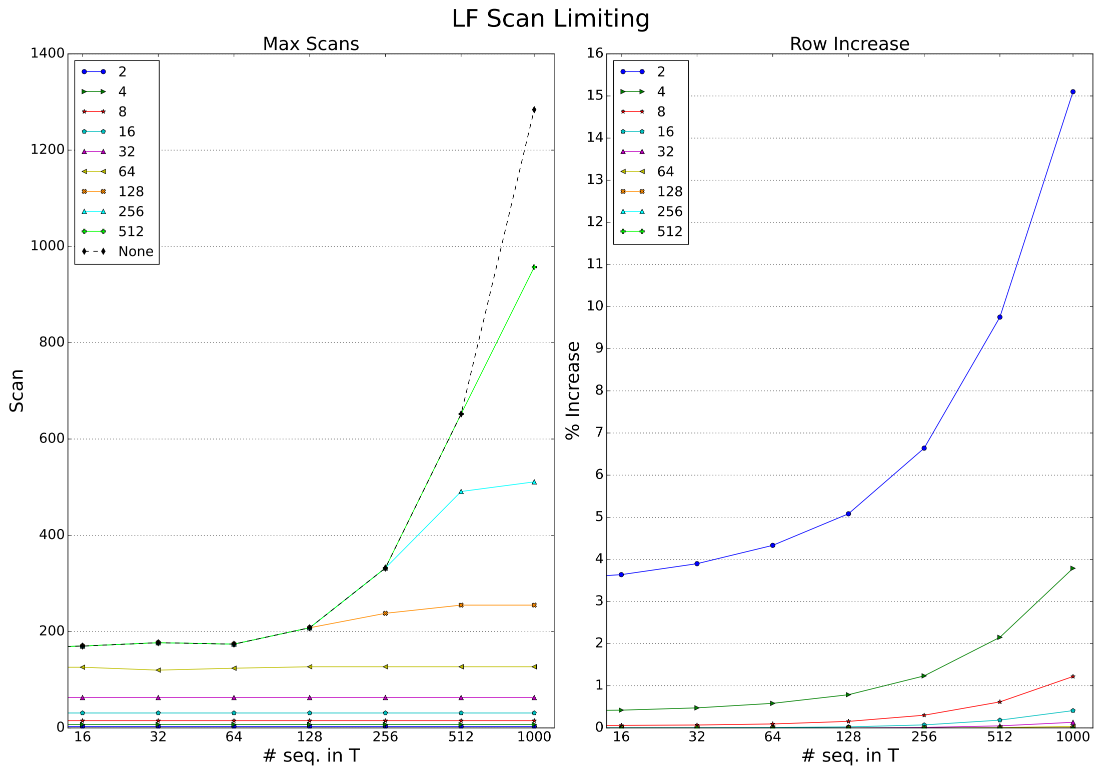


Figure 5.4: Shows the effect of performing splitting for different choice of parameter $d$ (shown in legend). Left plot shows the maximum scan before (dashed line) and after splitting across sequences of `chr19`. Right plot shows percentage increase of table rows resulting from splitting.

| **# seq. in T** | 16 | 32 | 64 | 128 | 256 | 512 | 1000 |
|---|---|---|---|---|---|---|---|
| $n/10^6$ | 946.01 | 1892.01 | 3784.01 | 7568.01 | 15136.04 | 30272.08 | 59125.12 |
| $r/10^4$ | 3240.02 | 3282.51 | 3334.06 | 3405.40 | 3561.98 | 3923.60 | 4592.68 |
| Inititialization (s) | 113.05 | 211.375 | 422.845 | 842.825 | 1705.44 | 3574.17 | 7396.98 |
| Splitting (s) | 46.2267 | 74.5394 | 125.788 | 231.538 | 465.058 | 949.25 | 1826.87 |
| Memory (GB) | 4.45 | 5.36 | 7.14 | 10.70 | 18.19 | 32.76 | 59.78 |

Table 5.2: Time/Memory needed to perform splitting on `chr19`. Initialization builds data structures and splitting perform the procedure.

### 5.4.2 Queries

To query the data structures we performed counting queries for 10000 randomly chosen substrings each of length 10, 100, 1000 and 10000. We build `lookup-bv` using splitting which is denoted by `lookup-bv-d` where $d$ is the splitting parameter. These approaches are compared against `lookup-bv` without splitting and `rle-string` from Section 4.3. Figure 5.5 reports the time and memory for querying of the data structures on `chr19`. We discuss by comparing the average query times directly without further statistical analysis.

## 5.5 Discussion

A splitting parameter of $d = 16$ bounds the worst scan from up to over 1200 on `chr19` to 31 while only increasing the rows of the table by a fraction of a percent, roughly 0.4%. Higher choices of $d$ eliminates the worst case to give constant time mapping with only a negligible practical increase to the table size. When computing count queries, $d = 2$ and $d = 4$ are faster on average than no splitting or other choice of $d$; on our data $d = 4$ is a small increase in space but shows a speed improvement. However, the overall speed improvement is not easily noticeable with splitting, and on the most repetitive datasets `rle-string` outperforms in both space and time. This result does support that, even without splitting, LF steps are practically similar to constant-time approaches on this data; however splitting does not solve performance issues on very repetitive datasets.

Other components of the index could be investigated to see if table-lookup can improve in this respect. Since our count queries require more than just LF, splitting should be investigated in an isolated context. Repetitiveness increases the access time in our DACs for lengths/offsets so we may consider removing them by storing $i$

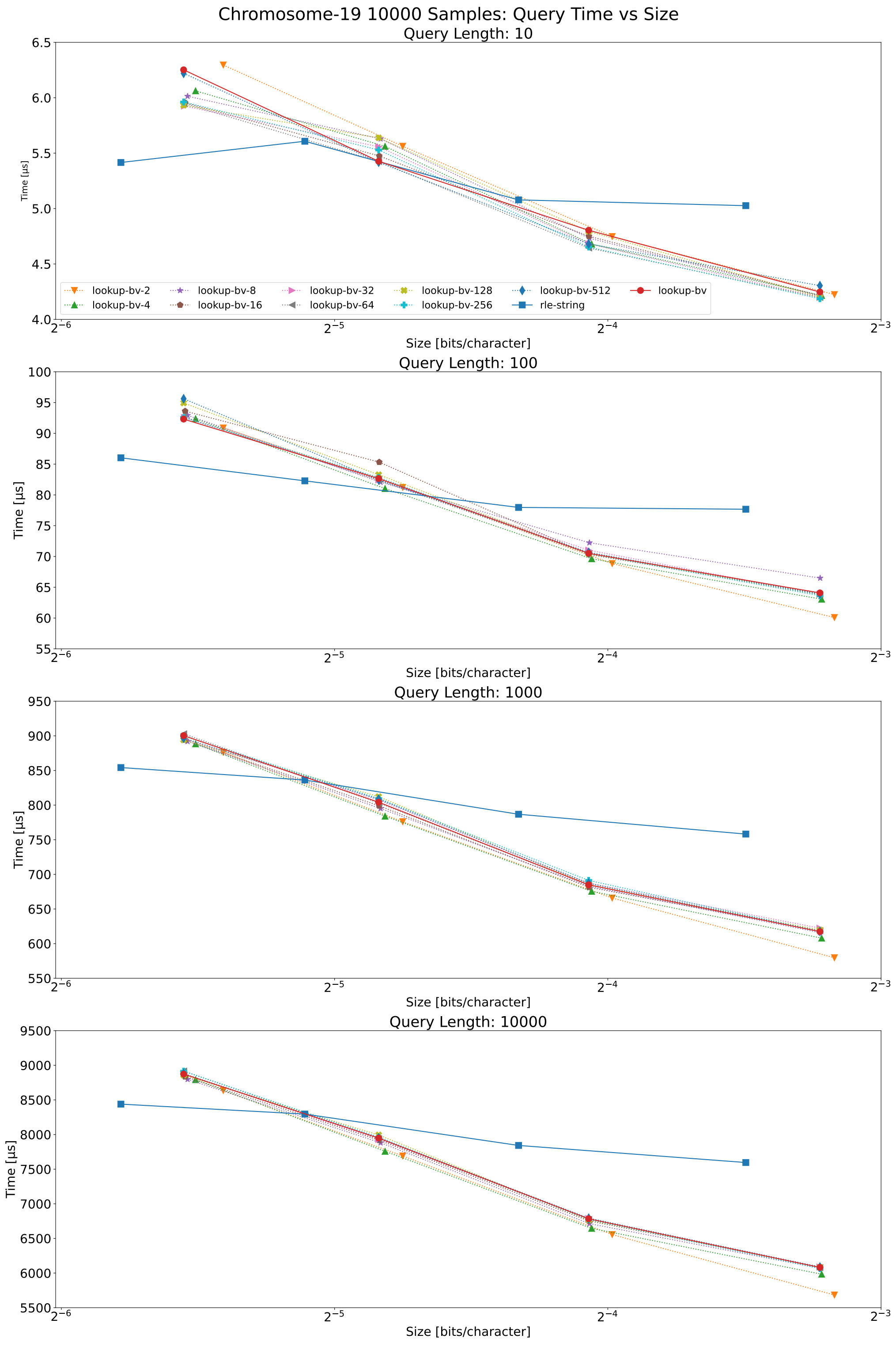


Figure 5.5: The time per query to count the occurrences of 128, 256, 512 and 1000 copies of `chr19` for 10000 randomly-chosen substrings of length 10, 100, 1000 and 10000 each. Copies for a single line are read from largest number of copies to smallest, left to right. The x axis is logarithmically scaled with base 2.

run-head positions and the explicit position of $\mathrm{LF}(i)$ instead of our run/offset pairs. Compressing the destinations of $\mathrm{LF}(i)$ can be copied from Chapter 4, and it follows that $\mathrm{LF}(i)$ itself also forms $\sigma$ non-decreasing subsequences. The run-head positions $i$ can be stored in a sparse bitvector and recovered in constant time. Note that we do not need lengths, since our scan is identical when over the run-head positions. Similarly, we find offsets easily since we now operate with a full position $j$ with predecessor at run-head position $i$. This approach is not as suitable for the table blocking of Chapter 4, but our block sizes did not prove to be useful on our test data.

Although our splitting procedure using dynamic bitvectors is $O(r)$-space and expected $O(r \log r)$-time in theory, the implementation of necessary data structures needs improvement since we cannot efficiently use a runs-bounded approach. In particular, the dynamic approaches used still require $O(n)$-time for initialization and using the sparse variants in $O(r)$-space proved too slow in our experiments. The proposed approach follows easily from our proof, but an algorithm which does not use such heavy compression as bitvectors may prove more efficient while still being $O(r)$-space.

For instance, consider storing the $L$ set bit positions in a balanced *binary search tree* (BST) and $F$ nodes in a balanced BST holding set bit positions, interval weights, and a pointer to the first set bit contained in the $F$ interval in $L$'s BST. By maintaining a queue of nodes in $F$ with weight $\geq 2d$ we can perform the necessary predecessor queries on the BST in $O(\log r)$-time and $O(r)$-space. To find the splitting position, we start at the pointer to the first covering node in $L$ for a node in $F$ and search $L$ tree nodes to find the $d$th in $O(d)$-time. Storing the inverse permutation of the set bit also allows us to compute any inverse by scanning nodes akin to table-lookup or using an $O(\log r)$-time predecessor query. This simple approach requires storing of integer values and pointers, but should be easy to implement in practice to provide a worst case $O(r)$-space and $O(r \log r)$-time algorithm for splitting with efficient run-time.

# Chapter 6

# Pan-genomic Matching Statistics

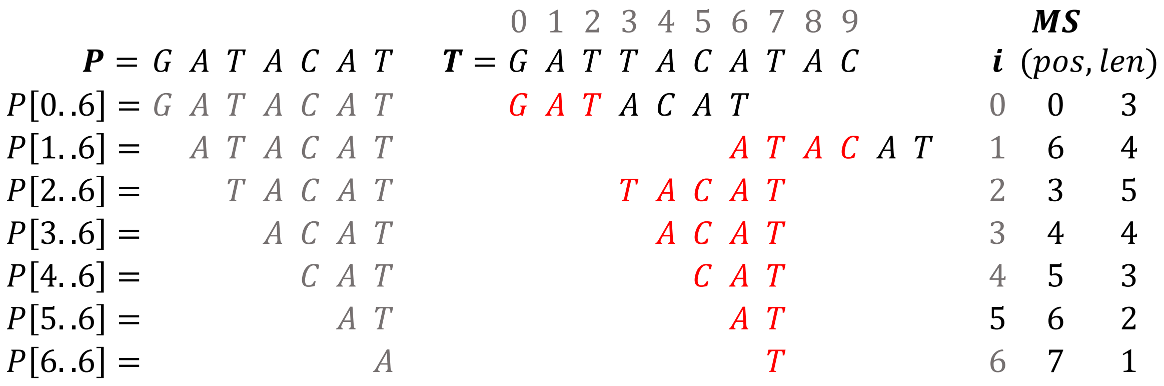
Backwards search using LF allows us to compute exact matching queries for a pattern, such as count and locate. However, approximate pattern matching is often more important in bioinformatics. When a pattern does not match exactly, we can still describe substrings which do using *matching statistics* (MS). This Chapter describes how to generate MS on pan-genomics datasets using runs-compressed data structures. Past runs-compressed indexes for MS depend on LF steps and SA samples, but also *thresholds* or LCE queries to reorient in the BWT when a character mismatches. We describe how to augment thresholds with additional LCE information to speed up computation of matching statistics by avoiding unnecessary LCE queries. This Chapter covers "Augmented Thresholds for MONI"[1] by Martínez-Guardiola, myself, Silva-Coira, et al. [41].

## 6.1 Matching Statistics

The *matching statistics* (MS) describe which substrings of a pattern match in a text. We define the matching statistics of a pattern $P[0..m-1]$ with respect to $T[0..n-1]$ as an array $\text{MS}[0..m-1]$ where $\text{MS}[i].pos$ is the starting position in $T$ of the longest prefix of $P[i..m-1]$ occurring in $T$ and $\text{MS}[i].len$ is the length of that prefix [7]. When there are multiple occurrences of a longest prefix $P[i..m-1]$ in $T$, $\text{MS}[i].pos$ can be any such position. Figure 6.1 shows possible matching statistics of an example pattern. Computing MS has many useful applications in bioinformatics [36]; particularly, *maximal exact matches* can be easily computed from MS and are used for DNA alignment in tools such as BWA-MEM [34]. For pan-genomic text collections, Bannai et al. [4] described how to compute MS using a variant of the $r$-index.

The key idea behind Bannai et al.'s index is to store $r$ *threshold* positions alongside the $r$-index; one between each pair of same character runs in the BWT. Given a

[1]This section does not contain self citations.



| | | | | *MS* | |
|---|---|---|---|---|---|
| ***P*** = *G A T A C A T* | | ***T*** = *G A T T A C A T A C* (0 1 2 3 4 5 6 7 8 9) | ***i*** | (*pos*, | *len*) |
| *P*[0..6] = | *G A T A C A T* | *G A T A C A T* | 0 | 0 | 3 |
| *P*[1..6] = | *A T A C A T* | *A T A C A T* | 1 | 6 | 4 |
| *P*[2..6] = | *T A C A T* | *T A C A T* | 2 | 3 | 5 |
| *P*[3..6] = | *A C A T* | *A C A T* | 3 | 4 | 4 |
| *P*[4..6] = | *C A T* | *C A T* | 4 | 5 | 3 |
| *P*[5..6] = | *A T* | *A T* | 5 | 6 | 2 |
| *P*[6..6] = | *A* | *T* | 6 | 7 | 1 |

Figure 6.1: For the given pattern $P$ and text $T$, visualizes the matching statistics MS; for $P[i..m-1]$, MS$[i].pos$ gives the positions of the longest matching prefix wrt. $T$ and MS$[i].len$ gives the length of that match.

position corresponding to a mismatch character when processing the pattern, the threshold tells us whether to "jump" forwards or backwards to reorient in the BWT. The approach of Bannai et al. is not a true runs-bounded index, since it requires fast random access to $T$ of which there is no known solution in worst case $O(r)$-space. However, Gagie et al. [19, 18] showed how we can use the same *prefix-free parsing* (PFP) used to efficiently build large indexes to also build a *straight-line program* (SLP); a context free-grammar that generates exactly one string. Their SLP supports fast random access while operating on a compressed text and thus takes much less space than the $r$-index itself in practice. Rossi et al. gave an implementation called MONI by providing an algorithm to find thresholds using the *longest common prefix* (LCP) array and PFP [52].

MONI makes two passes over the pattern; one right-to-left to generate the MS positions using LF steps and SA samples at run boundaries, and one left-to-right to find the lengths using random access with the SLP [52]. Boucher et al.'s PHONI [7] instead uses the SLP to support *longest common extension* (LCE) queries to generate both during the first pass. Since we can reorient based on which position gives the longer LCE, we can also omit thresholds. This online approach is advantageous in certain salutations since the first pass buffers a significant amount of data; using one pass reduces the workspace and allows us to run more queries in parallel. Using one-pass also suits applications which are inherently online, such as SPUMONI [2]

[2]SPUMONI does not generate MS, but an approximation, removing the SLP and SA samples to save space [2].

and its targeted sequencing of DNA strands as they are emitted from nanopore sequencers [2, 1]. LCE queries are only needed for mismatch cases; most characters in $P$ are processed using efficient LF steps and SA samples. However, the LCE queries it does compute dominate the query time [7].

## 6.2 Computing MS

A *threshold* between two consecutive same character runs $\mathrm{BWT}[s_1..e_1]$ and $\mathrm{BWT}[s_2..e_2]$ is defined to be a position $t$ with $e_1 < t \leq s_2$ such that

$$\mathrm{LCE}(\mathrm{SA}[e_1], \mathrm{SA}[j]) \geq \mathrm{LCE}(\mathrm{SA}[j], \mathrm{SA}[s_2])$$

if $j < t$, and

$$\mathrm{LCE}(\mathrm{SA}[e_1], \mathrm{SA}[j]) \leq \mathrm{LCE}(\mathrm{SA}[j], \mathrm{SA}[s_2])$$

if $j \geq t$. Bannai et al. did not describe how to find the thresholds [4], but Rossi et al. showed they could be computed as the minimum in $\mathrm{LCP}[e_1 + 1..s_2]$. Intuitively, this is since $\mathrm{LCE}(\mathrm{SA}[e_1], \mathrm{SA}[k])$ for $k$ from $s_2$ up to $e_1$ are non-decreasing, and similarly for $\mathrm{LCE}(\mathrm{SA}[k], \mathrm{SA}[s_2])$ with $k$ from $e_1$ down to $s_2$. For simplicity, let $\mathrm{LCE}_{e_1}(k) = \mathrm{LCE}(\mathrm{SA}[e_1], \mathrm{SA}[k])$ and $\mathrm{LCE}_{s_2}(k) = \mathrm{LCE}(\mathrm{SA}[k], \mathrm{SA}[s_2])$. Figure 6.2 shows the threshold position between runs for an example text.

Assume we have already computed $\mathrm{MS}[i+1].pos$ and the position $j$ of its preceding character $T[\mathrm{MS}[i+1].pos - 1]$ in the BWT. As Figure 6.1 hinted, if $\mathrm{BWT}[j] = P[i]$ then $\mathrm{MS}[i].pos = \mathrm{MS}[i+1].pos - 1$ since the character preceding our current matching position in the text equals the preceding character that we process to extend our suffix of $P$. We continue by computing the position of $T[\mathrm{MS}[i+1].pos - 1]$ in the BWT given by $\mathrm{LF}(j)$. If $\mathrm{BWT}[j] \neq P[i]$ then[3] $e_1 < j < s_2$ where $e_1$ is the last occurrence of $P[i]$ in the BWT and $s_2$ is the next, corresponding to runs $\mathrm{BWT}[s_1..e_1]$ and $\mathrm{BWT}[s_2..e_2]$. By definition of the BWT and thresholds, if $j < t$ for the corresponding threshold $e_1 < t \leq s_2$ then a prefix of $T[\mathrm{SA}[e_1]..n-1]$ is a longest match for $P[i..m-1]$; if $j \geq t$ then a prefix of $T[\mathrm{SA}[s_2]..n-1]$ is longest match for $P[i..m-1]$. Since $e_1$ and $s_2$ are run boundaries, we have $\mathrm{SA}[e_1]$ and $\mathrm{SA}[s_2]$ stored and can reorient in the BWT to find the position of a longest match by "jumping" based on the threshold and taking

[3]Assuming that $P[i]$ occurs again in the text in either direction for simplicity.

| $k$ | BWT$[k]$ | $T[\text{SA}[k]..n]$ | LCP$[k]$ | LCE$_{e_1}(k)$ | LCE$_{s_2}(k)$ |
|---|---|---|---|---|---|
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 1234 | A | GAGACATCA... | - | - | - |
| $e_1 = 1235$ | A | GATACATTA... | - | - | - |
| 1236 | C | GATAGATTA... | 4 | 4 | 3 |
| 1237 | G | GATATAGAA... | 4 | 4 | 3 |
| 1238 | G | GATCCAATA... | 3 | 3 | 3 |
| $t = 1239$ | G | GATTACATA... | 3 | 3 | 6 |
| 1240 | T | GATTACTTA... | 6 | 3 | 6 |
| 1241 | T | GATTAGATA... | 5 | 3 | 6 |
| $s_2 = 1242$ | A | GATTATCAT... | 5 | - | - |
| 1243 | A | GATTATGAA... | - | - | - |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Figure 6.2: For positions $e_1 < k < s_2$ between a pair of same character runs of A, shows the threshold position $t$ at a minimum in the LCP array. The LCEs between these run boundaries for $k$ visualizes why this threshold describes which run boundary gives a longer LCE.

the corresponding sampled SA value. That is, if $j < t$ then MS$[i]$.$pos$ = SA$[e_1]$ and we continue with the position of $T[\text{SA}[e_1] - 1]$ in the BWT given by LF$(e_1)$, and if $j \geq t$ then MS$[i]$.$pos$ = SA$[s_2]$ and we continue with the position of $T[\text{SA}[s_2] - 1]$ in the BWT given by LF$(s_2)$.

After finding the MS positions during a right-to-left pass, MONI finds the lengths by performing a left-to-right pass over the pattern to extend the matches using the SLP (comparable to an LCE). PHONI instead computes them alongside the positions using LCE queries during the first pass. Assume that we have computed MS$[i+1]$.$len$ alongside MS$[i+1]$.$pos$ and the position $j$ of $T[\text{MS}[i+1].pos - 1]$ in the BWT. If BWT$[j] = P[i]$ then we extend our match by exactly that character and set MS$[i]$.$len$ = MS$[i+1]$.$len + 1$. Instead, if $j < t$ then we set

$$\text{MS}[i].len = min(\text{LCE}(\text{MS}[i+1].pos, \text{SA}[e_1]), \text{MS}[i+1].len) + 1$$

since we can only take the LCE up to the length we have matched so far if it is less, increasing by 1 for the matching character BWT$[e_1]$ that precedes $T[\text{SA}[e_1]]$. Similarly, if $j \geq t$ then we set

$$\text{MS}[i].len = min(\text{LCE}(\text{MS}[i+1].pos, \text{SA}[s_2]), \text{MS}[i+1].len) + 1$$

and in both cases continue with an LF step. Further, if we compute both LCE(MS$[i+1].pos$, SA$[e_1]$) and LCE(MS$[i+1].pos$, SA$[s_2]$) then we know which position provides a longer match without needing to store the threshold. However, MONI stores thresholds to perform only one LCE during its second pass to recover the lengths; thresholds do not take much space compared to the RLBWT and the SA samples, whereas LCE queries are slow when compared to LF steps [7].

## 6.3 Augmented Thresholds

In practice, mismatch cases will occur in bunches on genomic datasets; if $P[i]$ is a sequencing error or part of a variation not present in $T$ then we are likely to find a short match when jumping and experience many mismatches for subsequent processed characters of $P$ until we have built enough context to reorient in the BWT. In this scenario, we will have short longest matches and perform LCEs in rapid succession (or threshold jumps) until the matches are long enough that we are likely to find characters of $P$ which do match. Since the length of the longest matches can only increment for each character processed in $P$ in this case, while performing LCEs in rapid succession we expect MS$[i+1].len$ to be smaller than the computed LCE query that we compare it to. Since we often end up setting MS$[i].len =$ MS$[i+1].len + 1$ for mismatch cases in this scenario, we are motivated to investigate if all these LCE queries are necessary.

Consider storing the threshold $t$ between two consecutive same character runs BWT$[s_1..e_1]$ and BWT$[s_2..e_2]$ while supporting LCE queries. For any $j < t$ we have $LCE_{e_1}(j) \leq \text{LCE}_{e_1}(t-1)$; this is easily seen when considering LCEs as *range minimum queries* (RMQs) over the LCP array with $\min \text{LCP}[e_1+1..j] \leq \min \text{LCP}[e_1+1..t-1]$ since $e_1 < j < t$. Similarly, if $j \geq t$ then $LCE_{s_2}(j) \leq \text{LCE}_{s_2}(t)$. These *threshold LCEs* are a lower bound for LCE queries we will compute on either side of the threshold. Suppose we have MS$[i+1].len$ when $P[i] = \text{BWT}[e_1] = \text{BWT}[s_2]$ and $j$ is the position of $T[\text{MS}[i+1].pos - 1]$ in the BWT with $e_1 < j < s_2$. If $j < t$ and

$$\text{MS}[i+1].len \leq \text{LCE}_{e_1}(t-1),$$

or $j \geq t$ and

$$\text{MS}[i+1].len \leq \text{LCE}_{s_2}(t)$$

| $k$ | SA$[k]$ | BWT$[k]$ | $T[\text{SA}[k]..n]$ |
|---|---|---|---|
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 1234 | 8765 | A | GAGACATCA... |
| $e_1 = 1235$ | 1519 | A | GATACATTA... |
| 1236 | 5450 | C | GATAGATTA... |
| $j = 1237$ | 1004 | G | GATATAGAA... |
| 1238 | 4242 | G | GATCCAATA... |
| $t = 1239$ | 3110 | G | GATTACATA... |
| 1240 | 1102 | T | GATTACTTA... |
| 1241 | 1978 | T | GATTAGATA... |
| $s_2 = 1242$ | 2505 | A | GATTATCAT... |
| 1243 | 2022 | A | GATTATGAA... |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Figure 6.3: If MS$[i+1]$.len $\le \text{LCE}_{e_1}(t-1)$ then, since $\text{LCE}_{e_1}(t-1) \le \text{LCE}_{e_1}(j)$, we have MS$[i+1]$.len $\le \text{LCE}_{e_1}(j)$ and we can safely set MS$[i]$.len $=$ MS$[i+1]$.len $+ 1$. For example, if $MS[i+1].len = 3$ for the above diagram, then since $\text{LCE}_{e_1}(t-1) = 3$ is a lower bound on $\text{LCE}_{e_1}(j)$ so it cannot be smaller than MS$[i+1].len$; we skip an LCE and set MS$[i].len = 3 + 1$.

then as Figure 6.3 explains we can set MS$[i+1].len =$ MS$[i].len+1$ without computing any LCE queries. By storing the values $\text{LCE}_{e_1}(t-1)$ and $\text{LCE}_{s_2}(t)$ (i.e. the threshold LCEs) explicitly alongside thresholds, we obtain *augmented thresholds* which allow us to filter out unnecessary LCE queries. Appendix Algorithm A.4 shows the complete algorithm.

We can compute threshold LCEs using LCE queries if we have the SA values for threshold positions. However, their relationship to thresholds allows us to compute both simultaneously using the procedure of MONI with PFP. Rossi et al. observed that we can set $t$ to be the position of $\min(\text{LCP}[e_1+1..s_2])$, which they support space-efficiently through PFP and a range-minimum data structure over the LCP array [52]. Since our LCE queries are also defined as RMQs over the LCP array, we can utilise the minimum information of thresholds to find $\min(\text{LCP}[e_1+1..t-1])$ and $\min(\text{LCP}[t+1..s_2])$. When a threshold position is updated to $t'$ by finding a new minimum, then the last minimum must be $\min(\text{LCP}[e_1+1..t'-1])$. Similarly, the second minimum after $t'$ must be $\min(\text{LCP}[t'+1..s_2])$. These minimums can be computed alongside the thresholds by updating until the final $t$ is found and performing RMQs when necessary. With this slight modification, the augmented thresholds can be efficiently computed using PFP in a similar manner to MONI.

## 6.4 Experiments

We compare the time and memory for querying using augmented thresholds against the one-pass approach of PHONI. To mitigate the size increase of augmented thresholds, we explore techniques for space-efficiency. Any single threshold LCE can be stored in $\lg n$-bits (since they inherit LCP bounds); however, many values tend to be smaller than others [28] and in practice our LCE values represent minimums over ranges of the LCP array. Further, some threshold LCEs can be ignored: if $t = s_2$ then for any position $j$ (with $e_1 < j < s_2$) we always have $j < t$ so we jump up to $e_1$ and the other threshold LCE is never used, and similarly for $t = e_1 + 1$ and always jumping down. Thus, we can safely ignore these values, choosing to "zero" them to not store any value at all. Thresholds form $\sigma$ increasing subsequences which we compress by storing them in $\sigma$ sparse bitvectors.

The variants of augmented thresholds differ in storing the threshold LCEs; we compare them against PHONI:

- `PHONI`: Standard version of one-pass MONI described as $\texttt{PHONI}_{std}$ in original paper [7].
- `Aug-Full`: One-pass MS using augmented thresholds, using $\lg n$-bits per threshold LCE stored
- `Aug-1`: Above, but restricting threshold LCEs to one byte. On overflow, we default to using an LCE query.
- `Aug-BV-Full`: Stores bitvector marking which threshold LCEs are used, storing these values in $\lg n$-bits and accessed using a rank query.
- `Aug-BV-1`: As above, but ignores storing values greater than one byte (default to LCE query).
- `Aug-DAC`: Stores threshold LCEs using a directly addressable code (DAC)
- `Aug-BV-DAC`: Same as `Aug-BV-Full`, but substituting in a DAC to store values.

The approach was implemented in C++ and compiled with flags `-O3 -DNDEBUG -funroll-loops -msse4.2`; the code is available at `https://github.com/drnatebr`

| # | $n/10^6$ | $r/10^4$ | $n/r$ | SLP$_{comp}$ [MB] | SLP$_{plain}$ [MB] |
|---|---|---|---|---|---|
| 16 | 946.01 | 3240.02 | 29.20 | 36.10 | 70.54 |
| 32 | 1892.01 | 3282.51 | 57.64 | 37.80 | 74.75 |
| 64 | 3784.01 | 3334.06 | 113.50 | 39.48 | 79.84 |
| 128 | 7568.01 | 3405.40 | 222.24 | 42.11 | 88.89 |
| 256 | 15136.04 | 3561.98 | 424.93 | 47.43 | 102.52 |
| 512 | 30272.08 | 3923.60 | 771.54 | 58.00 | 131.09 |
| 1,000 | 59125.12 | 4592.68 | 1287.38 | 80.63 | 186.98 |

Table 6.1: Table summarizing the datasets and sizes of SLPs built over them. The first column describes the number of concatenated sequences of `chr19`. SLP sizes are measured in megabytes [MB].

`own/aug_phoni` and is based on the original one-pass MONI code at `https://github.com/koeppl/phoni`. All experiments were executed single-threaded on a server with an Intel(R) Xeon(R) Bronze 3204 CPU and 512 GiB RAM.

We re-ran Boucher et al.'s query experiments from the original `PHONI` paper using the same datasets; chromosome 19 haplotypes (`chr19`) for concatenations of 16, 32, 64, 128, 256, 512, and 1000 sequences. We query the data structure with 10 distinct `chr19` sequences as patterns. To support random access and LCE queries efficiently we construct SLPs; SLP$_{comp}$ over the compressed text of the original PHONI experiments, SLP$_{plain}$ over the uncompressed text [18]. The datasets and SLP sizes are reported in Table 6.1. The average query times (computing MS for a single pattern) are shown in Figure 6.4 with results for both SLP types in distinct plots. Figure 6.5 shows the disk sizes for all variants using both SLP types. We use the average wall time for discussion.

## 6.5 Discussion

Variants using augmented thresholds are always faster on average than `PHONI` but always larger. Introducing the SLP$_{plain}$ speeds up LCE queries for all methods; although SLP$_{plain}$ can be over twice as large as SLP$_{comp}$ (Table 6.1) the difference is smaller when compared to the total index sizes shown in Figure 6.5. This LCE speedup reduces the gap between query times compared to `PHONI`, since it spends a larger percentage of execution on them; however, the LCE queries skipped by augmented thresholds still result in faster executions.

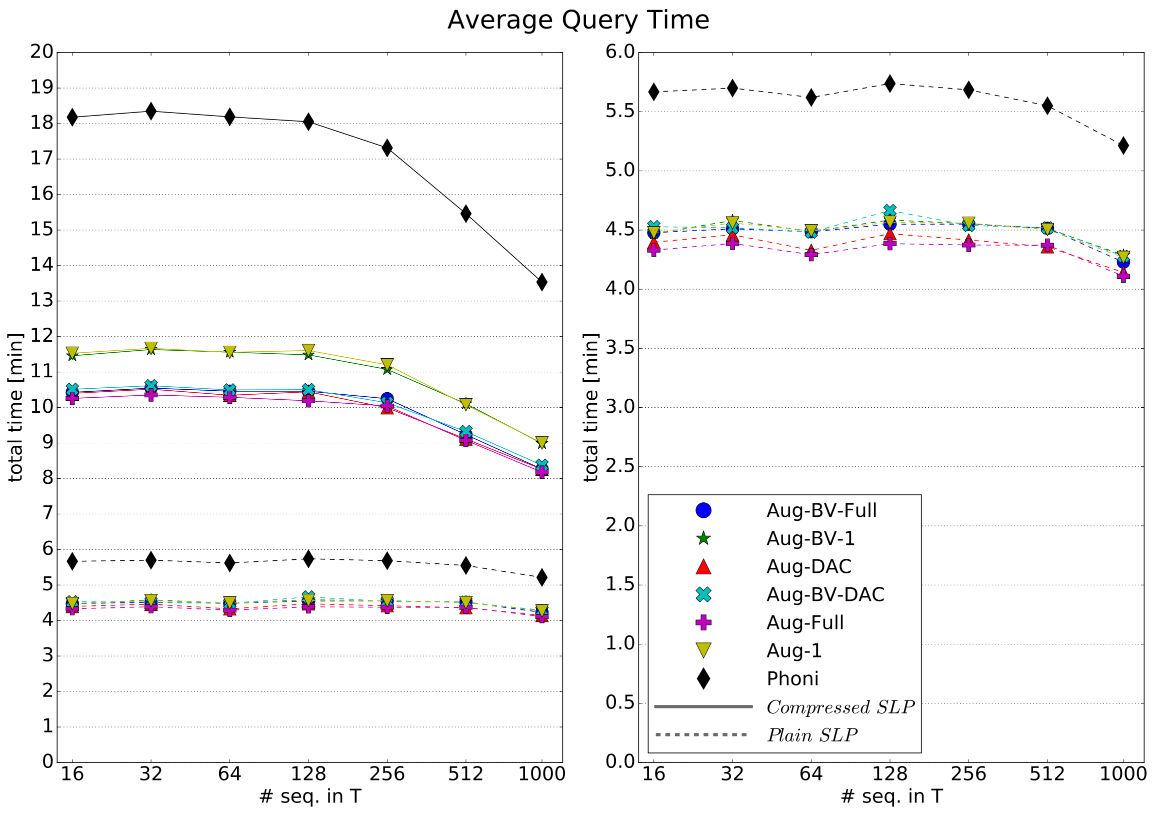


Figure 6.4: The average query time across data structures to compute MS using 10 distinct `chr19` sequences against 16, 32, 64, 128, 256, 512, and 1000 sequences of `chr19`. Solid lines use SLP$_{comp}$, dashed lines using SLP$_{plain}$ (focus of right plot).

We highlight some standout variants when compared to `PHONI` for the largest text size (1000 sequences of `chr19`). `Aug-DAC` is among the fastest approaches regardless SLP: 48.37% faster and 22.89% larger for SLP$_{comp}$, and 22.92% faster and 19.97% larger for SLP$_{plain}$. This is a significant improvement compared to the original `PHONI` method (SLP$_{comp}$) and a direct time/space tradeoff when using SLP$_{plain}$. The success of this variant aligns with expectation: the DAC is engineered for the LCP array which has small values on average. Threshold LCEs consist of small LCP values and zeroes which results in high compression and fast access on average. `Aug-1` is in the smallest class: 40.22% faster and only 14.60% larger for SLP$_{comp}$, while 19.95% faster and 12.66% larger for SLP$_{plain}$. Although `Aug-Full` is in the fastest class with `Aug-DAC`, it is much larger. Other variants fall between these approaches in both time and space.

When compared to the original one-pass MONI of Boucher et. al (`PHONI` with

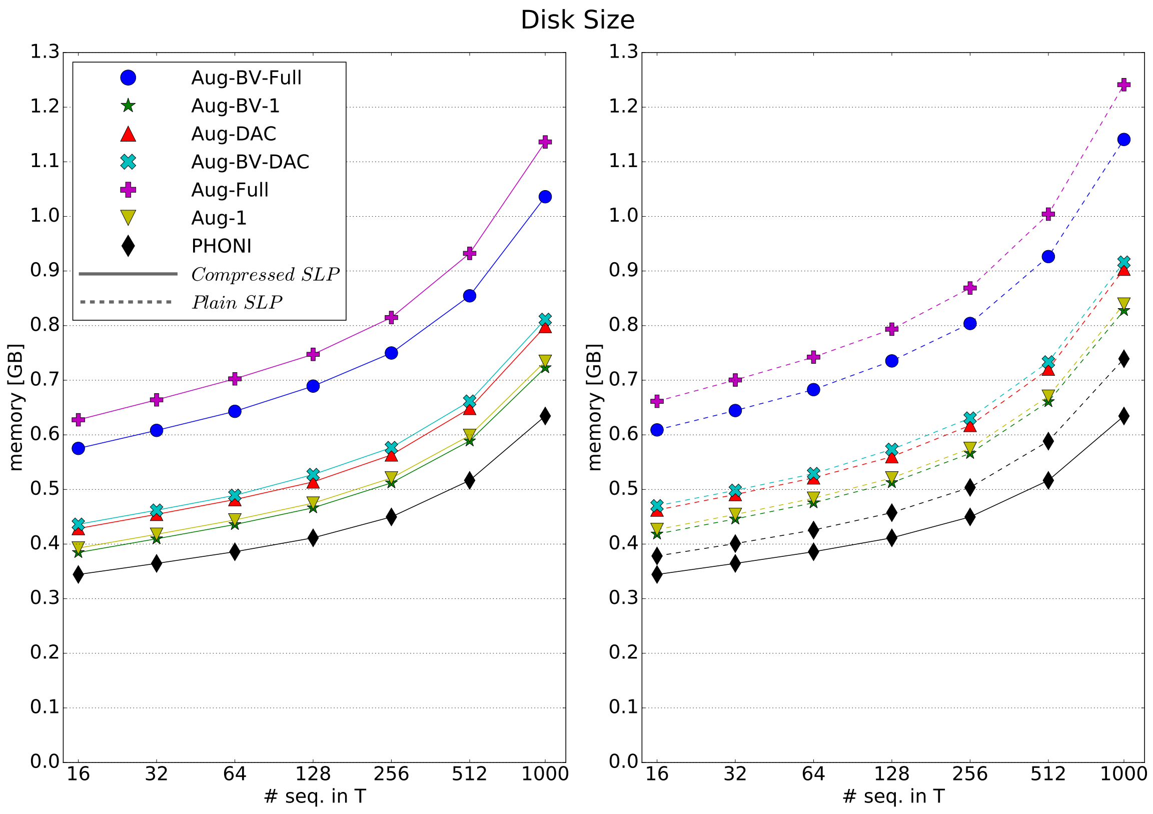


Figure 6.5: The disk size in GB for each data structure built on 16, 32, 64, 128, 256, 512, and 1000 sequences of `chr19`. Solid lines use SLP$_{comp}$ (focus of left plot), dashed lines using SLP$_{plain}$ (focus of right plot). `PHONI` using SLP$_{comp}$ is included on right to visualize the size difference of SLP choices.

SLP$_{comp}$), our best augmented threshold approaches showed over 40% speed improvements with under 20% space increase on the largest dataset with similar results across all data. Our applied compression schemes are space-efficient whilst still being faster on average than PHONI on the tested data. Introducing an uncompressed SLP (SLP$_{plain}$) was experimentally shown to benefit both LCE and total query speed while requiring a small size increase. Using this SLP$_{plain}$, augmented thresholds allow at worst a direct time/space tradeoff (increase speed/space by $\approx 20\%$) with some results gaining more in speed than the additional space required to store them.

# Chapter 7

# Conclusion

In this thesis we described recent approaches in run-length compressed data structures for pan-genomic text indexing related to the BWT. Nishimoto and Tabei's constant-time data structure [46] for permutations was introduced and formulated as a simple table-lookup approach which is $O(r)$-space for LF steps. We showed that without bounds the approach can be worst case $\Omega(r)$ on average for LF but that the approach is efficient in practice without introducing their theoretical bounds. The simple implementation as presented in Chapter 3 is much faster than conventional methods relying on sparse bitvectors to compute LF steps in runs-bounded space, but is much larger due to storing uncompressed integers.

The table can be compressed column-wise and, with slight modification, supports computing of count queries. This index, using our reduced alphabet and bitvector compression for run-head mapping destinations, is competitive in time/space with the best existing methods for pan-genomic data. However, long scans for computing LF were encountered during computation. We provided motivation for bounding the scans necessary during table lookup steps, and reformulated Nishimoto and Tabei's constant-time proof [46] using table row splitting to include a parameter for a time/space tradeoff. This proof illustrates how to perform splitting in practice; we experiment with our approach which succesfully bounds scans with minimal space increase, but it does not have a large effect on count query time.

Generating matching statistics for pan-genomic datasets using the RLBWT relies on LF steps and thresholds/LCE queries. By exploiting both two-pass [52] and one-pass [7] methods to compute the matching statistics we observed that threshold jumping and resulting LCE queries tend to occur in bunches on genomic data; by augmenting thresholds with additional LCE information we avoid unnecessary LCE queries in a one-pass approach. These threshold LCEs can be efficiently compressed alongside thresholds, and result in faster computation on average when compared to the prior one-pass method on pan-genomic datasets. For example, Using a directly addressable code to compress these threshold LCEs gave a result which on average achieves a larger percentage speed improvement when compared to additional space used. An auxiliary result showed that using a plain SLP greatly improves the speed of LCE queries when compared to the space required to store it.

## 7.1 Future Work

The discussion of Chapter 4 opens the question of implementing table-lookup alongside other data structures, such as RLCSA, or for other permutations, such as $\phi$. Alternative compression and table schemes may result in better results when blocking rows of the table for locality. This discussion might involve investigating the ability of table blocks to be distributed over multiple compute nodes; a result that seems obtainable based on the outlined structure. Chapter 5 shows that the row splitting procedure as described using bitvectors may not be practical given current implementations. The proposed approach supporting operations using balanced binary search trees should be implemented and experimentally evaluated, alongside other alternatives such as modifying splitting of a dynamic RLBWT [45] to the static case. Our results focus on count queries; splitting isolated to LF steps only or other queries could be useful. The easily compressible alternative without lengths/offsets suggested at the end of Chapter 5 may have use cases over our approach.

Pan-genomics matching statistics tools rely primarily on LF steps, thresholds and LCEs. Our first result show methods to improve the speed of LF steps, and our second result shows how to improve thresholds and speed up LCE queries. Combining these methods should result in an even faster approach; it remains to be shown if this can made space efficient. In the case of tools such as SPUMONI which rely only on LF steps and thresholds, our results could be applied to improve the speed and accuracy of its targeted sequencing approach. To evaluate feasibility, compressing thresholds alongside our table, with or without splitting, can be explored; specifically, testing which table formulation is best if we aim to use thresholds. In general, experiments included in this thesis suggest that our data structures will scale for pan-genomics using even hundreds of human genomes; however, future work should evaluate the feasibility in practice on such large datasets while exploring applications to alternatives in text indexing. This involves not just using these data structures for DNA alignment, but further applications within or even beyond genomics.

# Appendices

# Appendix A

# Pseudocode

## A.1 Construction [Chapter 3]

**Algorithm 1** LF-table Construction

```
    Input RLBWT[0..r − 1], RLBWT[i].c is character, RLBWT[i].ℓ is length
 1: $table[0..r-1] \leftarrow \{\}$
 2: $positions[1..|\Sigma|] \leftarrow \{\}$          ▷ Character indexed array of queues
 3: for $k \in 0..r-1$ do
 4:     $table[k].c = \mathrm{RLBWT}[k].c$
 5:     $table[k].\ell = \mathrm{RLBWT}[k].\ell$
 6:     $positions[\mathrm{RLBWT}[k].c].enqueue(i)$
 7: end for
 8: $k \leftarrow 0$
 9: $obv_L, obv_F \leftarrow 0$          ▷ Number of characters of $L/F$ column seen
10: for $i \in \{0..\sigma-1$ do
11:     while $|positions[i]| > 0$ do
12:         $pos \leftarrow positions[i].dequeue()$
13:         $table[pos].k \leftarrow k$
14:         $table[pos].d \leftarrow obv_F - obv_L$
15:         $obv_F \leftarrow obv_F + table[pos].\ell$
16:         while $obv_F \geq obv_L + table[k].\ell$ do
17:             $obv_L \leftarrow obv_L + table[k].\ell$
18:             $k \leftarrow k + 1$
19:         end while
20:     end while
21: end for
return $table[0..r-1]$
```

## A.2 Table-Lookup for LF-Mapping [Chapter 3]

**Algorithm 2** LF Mapping

```
   Input $(k, d)$
1: $k' \leftarrow table[k].k$
2: $d' \leftarrow table[k].d + d$
3: while $d' \geq table[k'].\ell$ do
4:     $d' \leftarrow d' - table[k'].\ell$
5:     $k' \leftarrow k' + 1$
6: end while
return $(k', d')$
```

## A.3 Row Splitting [Chapter 5]

**Algorithm 3** Interval Weight Bounding

```
    Input $P, Q, d, \pi^{-1}$
 1: $P' \leftarrow P$
 2: $Q' \leftarrow Q$
 3: $H.init(P, Q)$
 4: $(q, w) \leftarrow H.max()$
 5: while $w \geq 2d$ do
 6:     $p_{pred} \leftarrow P'.\text{pred}(q)$
 7:     $p \leftarrow P'.\text{select}(p_{pred} + d)$
 8:     $Q'[p] \leftarrow 1$
 9:     $x \leftarrow \pi(p)^{-1}$
10:     $P'[x] \leftarrow 1$
11:     $H.update(q, d)$                      ▷ Update interval at $q$ to have weight $d$
12:     $H.insert(p, w - d)$
13:     $x_Q \leftarrow Q'.\text{select}(Q'.\text{pred}(x))$
14:     $H.update(x_Q, H.weight(x_Q) + 1)$
15:     $(q, w) \leftarrow H.max()$
16: end while
return $P'$
```

## A.4 Aug-Moni [Chapter 6]

**Algorithm 4** Computes MS with augmented thresholds

```
1: $j \leftarrow \text{BWT.select}_{P[m]}(1)$
2: $\text{MS}[m] \leftarrow (\text{pos} : \text{SA}[j], \text{len} : 1)$
3: **for** $i = m - 1$ **down to** $1$ **do**
4:     **if** $\text{BWT}[j] = P[i]$ **then**
5:         $\text{MS}[i] \leftarrow (\text{pos} : \text{MS}[i+1].\text{pos} - 1, \text{len} : \text{MS}[i+1].\text{len} + 1)$
6:     **else**
7:         $c \leftarrow \text{BWT.rank}_{P[i]}(j)$
8:         $e_1 \leftarrow \text{BWT.select}_{P[i]}(c)$
9:         $s_2 \leftarrow \text{BWT.select}_{P[i]}(c+1)$
10:        $x \leftarrow \text{BWT.run\_of\_position}(s_2)$        ▷ Position $s_2$ belongs to the $x$th run
11:        $t \leftarrow \text{thresholds}[x]$
12:        **if** $j < t$ **then**        ▷ $\text{thr\_lce}_e$ stores $\text{LCE}(\text{SA}[e_1], \text{SA}[t-1])$
13:            **if** $\text{MS}[i+1].\text{len} \leq \text{thr\_lce}_e[x]$ **then**
14:                $\text{MS}[i].\text{len} \leftarrow \text{MS}[i+1].\text{len} + 1$
15:            **else**
16:                $\text{MS}[i].\text{len} \leftarrow min(\text{MS}[i+1].\text{len}, \text{LCE}(\text{SA}[e_1], \text{MS}[i+1].\text{pos})) + 1$
17:            **end if**
18:            $\text{MS}[i].\text{pos} \leftarrow \text{SA}[e_1]$
19:            $j \leftarrow \text{LF}(e_1)$
20:        **else**        ▷ $\text{thr\_lce}_s$ stores $\text{LCE}(\text{SA}[t], \text{SA}[s_2])$
21:            **if** $\text{MS}[i+1].\text{len} \leq \text{thr\_lce}_s[x]$ **then**
22:                $\text{MS}[i].\text{len} \leftarrow \text{MS}[i+1].\text{len} + 1$
23:            **else**
24:                $\text{MS}[i].\text{len} \leftarrow min(\text{MS}[i+1].\text{len}, \text{LCE}(\text{SA}[s_2], \text{MS}[i+1].\text{pos})) + 1$
25:            **end if**
26:            $\text{MS}[i].\text{pos} \leftarrow \text{SA}[s_2]$
27:            $j \leftarrow \text{LF}(s_2)$
28:        **end if**
29:    **end if**
30: **end for**
```

# Bibliography

[1] Omar Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. SPUMONI 2: Improved pangenome classification using a compressed index of minimizer digests. *bioRxiv*, 2022.

[2] Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C. Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *iScience*, 24(6):102696, 2021.

[3] Jasmijn Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, 21:1–28, 03 2022.

[4] Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the $r$-index. *Theor. Comput. Sci.*, 812:96–108, 2020.

[5] Sharon Begley. The reference genome is threatening dream of personalized medicine. https://www.statnews.com/2019/03/11/human-reference-genome-shortcomings, 2019.

[6] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4), apr 2015.

[7] Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. DCC*, pages 193–202. IEEE, 2021.

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

[9] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Inf. Process. Manag.*, 49:392–404, 2013.

[10] Nathaniel K Brown. Interval mapping of BWT-runs to efficiently compute LF in O(r) space. *16th Workshop on Compression, Text and Algorithms*, Oct 2021.

[11] Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[12] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC, 1994.

[13] Nae-Chyun Chen, Brad Solomon, Taher Mun, Sheila Iyer, and Ben Langmead. Reference flow: reducing reference bias using multiple population genomes. *Genome biology*, 22(1):1–17, 2021.

[14] James F. Crow. Unequal by nature: a geneticist's perspective on human differences. *Daedalus*, 131:81–88, 2002.

[15] Ron Edgar, Michael Domrachev, and Alex Lash. Edgar r, domrachev m, lash aegene expression omnibus: Ncbi gene expression and hybridization array data repository. nucl acids res 30: 207-210. *Nucleic acids research*, 30:207–10, 02 2002.

[16] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[17] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[18] Travis Gagie, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, and Yoshimasa Takabatake. Practical random access to SLP-compressed texts. In *Proc. SPIRE*, pages 221–231. Springer, 2020.

[19] Travis Gagie, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: rescaling RePair with rsync. In *Proc. SPIRE*, pages 35–44. Springer, 2019.

[20] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[21] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[22] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.

[23] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 397–406, New York, NY, USA, 2000. Association for Computing Machinery.

[24] Peter W Harrison et al. The COVID-19 Data Portal: accelerating SARS-CoV-2 and COVID-19 research through rapid open access data sharing. *Nucleic Acids Research*, 49(W1):W619–W623, 2021.

[25] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[26] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.

[27] Donald Ervin Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[28] Juha Kärkkäinen, Dominik Kempa, and Marcin Piatkowski. Tighter bounds for the sum of irreducible lcp values. *Theoretical Computer Science*, 656:265–278, 2016.

[29] Eric S. Lander, Lauren Linton, Bruce W. Birren, Chad Nusbaum, Michael C. Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William W. Fitzhugh, Roel Funke, Diane Gage, et al. Initial sequencing and analysis of the human genome. *Nature*, 409 6822:860–921, 2001.

[30] Latrice G Landry, Nadya Ali, David R Williams, Heidi L Rehm, and Vence L Bonham. Lack of diversity in genomic databases is a barrier to translating precision medicine research into practice. *Health Affairs*, 37(5):780–785, 2018.

[31] Ben Langmead. Burrows-wheeler transform and FM-index. https://langmead-lab.org/teaching-materials/, 2018.

[32] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[33] 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):1–10, 2009.

[34] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[35] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinform.*, 25(14):1754–1760, 2009.

[36] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. Genome-scale algorithm design: Genomics. In *Genome-Scale Algorithm Design: Genomics*, 2015.

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

[38] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

[39] Udi Manber and Eugene Wimberly Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22:935–948, 1993.

[40] Lasse Maretty, Jacob Malte Jensen, Bent Petersen, Jonas Andreas Sibbesen, Siyang Liu, Palle Villesen, Laurits Skov, Kirstine Belling, Christian Theil Have, Jose MG Izarzugaza, et al. Sequencing and de novo assembly of 150 genomes from Denmark as a population reference. *Nature*, 548(7665):87–91, 2017.

[41] Cesar Martinez-Guardiola, Nathaniel K. Brown, Fernando Silva-Coira, Dominik Koppl, Travis Gagie, and Susana Ladra. Augmented thresholds for MONI. In *Proc. DCC*. IEEE, 2023.

[42] J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

[43] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25, 01 2013.

[44] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[45] Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An Optimal-Time RLBWT Construction in BWT-Runs Bounded Space. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, pages 99:1–99:20, 2022.

[46] Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *48th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 101:1–101:15, 2021.

[47] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments*, page 60–70, USA, 2007. Society for Industrial and Applied Mathematics.

[48] Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

[49] Nicola Prezza. A framework of dynamic data structures for string processing. In *International Symposium on Experimental Algorithms*. Leibniz International Proceedings in Informatics (LIPIcs), 2017.

[50] Jacob Pritt, Nae-Chyun Chen, and Ben Langmead. FORGe: prioritizing variants for graph genomes. *Genome Biology*, 19, 12 2018.

[51] David Reich, Mike A. Nalls, Wen Kao, Ermeg L. Akylbekova, Arti Tandon, Nick J. Patterson, James C. Mullikin, Wen chi Hsueh, Ching-Yu Cheng, Josef Coresh, Eric Boerwinkle, Man Li, Alicja Waliszewska, Julie Neubauer, Rongling Li, Tennille S. Leak, Lynette Ekunwe, Joe C. Files, Cheryl L. Hardy, Joseph M. Zmuda, Herman A. Taylor, Elad Ziv, Tamara B. Harris, and James G. Wilson. Reduced neutrophil count in people of african descent is due to a regulatory variant in the duffy antigen receptor for chemokines gene. *PLoS Genetics*, 5, 2009.

[52] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.

[53] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[54] Jouni Sirén, Jean Monlong, Xian Chang, Adam M Novak, Jordan M Eizenga, Charles Markello, Jonas A Sibbesen, Glenn Hickey, Pi-Chuan Chang, Andrew Carroll, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

[55] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. SPIRE*, pages 164–175. Springer, 2008.

[56] Zachary Stephens, Skylar Lee, Faraz Faghri, Roy Campbell, Chengxiang Zhai, Miles Efron, Ravishankar Iyer, Michael Schatz, Saurabh Sinha, and Gene Robinson. Big data: Astronomical or genomical? *PLoS biology*, 13:e1002195, 07 2015.

[57] The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[58] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, page 83–92, New York, NY, USA, 2013. Association for Computing Machinery.

[59] Jun Wang, Wei Wang, Ruiqiang Li, Yingrui Li, Geng Tian, Laurie Goodman, Wei Fan, Junqing Zhang, Jun Li, Juanbin Zhang, et al. The diploid genome sequence of an Asian individual. *Nature*, 456(7218):60–65, 2008.