Lossy Compressor Preserving Variant Calling through Extended BWT

Veronica Guerrini[1][a], Felipe A. Louza[2][b] and Giovanna Rosone[1][c]
[1]*Department of Computer Science, University of Pisa, Italy*
[2]*Faculty of Electrical Engineering, Federal University of Uberlândia, Brazil*

Keywords: eBWT, LCP, Positional Clustering, FASTQ, Smoothing, Noise Reduction, Compression.

Abstract: A standard format used for storing the output of high-throughput sequencing experiments is the FASTQ format. It comprises three main components: (i) *headers*, (ii) *bases* (nucleotide sequences), and (iii) *quality scores*. FASTQ files are widely used for variant calling, where sequencing data are mapped into a reference genome to discover variants that may be used for further analysis. There are many specialized compressors that exploit redundancy in FASTQ data with the focus only on either the *bases* or the *quality scores* components. In this paper we consider the novel problem of lossy compressing, in a reference-free way, FASTQ data by modifying both components at the same time, while preserving the important information of the original FASTQ. We introduce a general strategy, based on the Extended Burrows-Wheeler Transform (EBWT) and positional clustering, and we present implementations in both internal memory and external memory. Experimental results show that the lossy compression performed by our tool is able to achieve good compression while preserving information relating to variant calling more than the competitors.
Availability: the software is freely available at https://github.com/veronicaguerrini/BFQzip.

1 INTRODUCTION

The recent improvements in high-throughput sequencing technologies have led a reduced cost of DNA sequencing and unprecedented amounts of genomic datasets, which has motivated the development of new strategies and tools for compressing these data that achieve better results than general-purpose compression tools – see (Numanagić et al., 2016; Hernaez et al., 2019) for good reviews.

FASTQ is the standard text-based format used to store raw sequencing data, each DNA fragment (*read*) is stored in a record composed by three main components: (i) read identifier with information related to the sequencing process (*header*), (ii) nucleotide sequence (*bases*), and (iii) quality sequence, with a per-base estimation of sequencing confidence (*quality scores*). The last two components are divided by a "separator" line, which is generally discarded by compressors as it contains only a "+" symbol optionally followed by the same header.

The majority of compressors for FASTQ files commonly split the data into those three main components (or *streams*), and compress them separately, which allows much better compression rates.

The *headers* can be efficiently compressed taking advantage of their structure and high redundancy. A common strategy used by FASTQ compressors, like SPRING (Chandak et al., 2018) and FaStore (Roguski et al., 2018), is to tokenize each header: the separators are non-alphanumerical symbols (Bonfield and Mahoney, 2013).

The *bases* and *quality scores* are commonly processed separately, although their information are correlated, and current specialized compressors only focus on one of these two components.

Most of FASTQ compressors limit their focus on the *bases*, compressing the *quality scores* independently with a third tool or standard straightforward techniques. The approaches that focus on compression the *bases* component are *lossless*, *i.e.*, they do not modify the bases, but they find a good strategy to represent the data by exploiting the redundancy of the DNA sequences. An interesting strategy is to reorder the sequences in the FASTQ file to gather reads originating from close regions of the genome (Cox et al., 2012; Roguski et al., 2018; Hach et al., 2012; Chandak et al., 2018).

[a] https://orcid.org/0000-0001-8888-9243
[b] https://orcid.org/0000-0003-2931-1470
[c] https://orcid.org/0000-0001-5075-1214

Guerrini, V., Louza, F. and Rosone, G.
Lossy Compressor Preserving Variant Calling through Extended BWT.
DOI: 10.5220/0010834100003123
In *Proceedings of the 15th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2022) - Volume 3: BIOINFORMATICS*, pages 38-48
ISBN: 978-989-758-552-4; ISSN: 2184-4305

On the other hand, approaches that focus on compression the *quality score* component are generally *lossy*, *i.e.*, they modify the data by smoothing the quality scores whenever possible. These approaches can be *reference-based* when they use external information (besides the FASTQ itself), such as a reference corpus of *k*-mers, *e.g.* QUARTZ (Yu et al., 2015), GeneCodeq (Greenfield et al., 2016) and YALFF (Shibuya and Comin, 2019). While, *reference-free* strategies evaluate only the *quality scores* information, such as the quantization of quality values using QVZ (Malysa et al., 2015), Illumina 8-level binning and binary thresholding, or evaluate the related biological information in the *bases* component (strategies known as *read-based*) – *e.g.* BEETL (Janin et al., 2014) and LEON (Benoit et al., 2015).

Our Contribution. In this work, we focus on a novel approach for the lossy compression of both the *bases* and the *quality scores* components taking into account both information at the same time. In fact, the two components are highly correlated being the second one a confidence estimation of each base call contained in the first one. To the best of our knowledge, none of the existing FASTQ compressor tools evaluates and modifies both components at the same time.

Note that we are not interested in compressing the *headers* component, for which one can use any state-of-the-art strategies or ignore them. Indeed, headers can be artificially structured in fields to store information only related to the sequencing process.

We focus on lossy reference-free and read-based FASTQ compression that makes clever modifications on the data, by reducing noise in the *bases* component that could be introduced by the sequencer, and by smoothing irrelevant values on the *quality scores* component, according to the correlated information in the *bases*, that would guarantee to preserve variant calling.

Hence, we propose a novel read-based, reference- and assembly-free compression approach for FASTQ files, BFQZIP, which combines both DNA bases and quality information for obtaining a lossy compressor.

Similarly to BEETL (Janin et al., 2014), our approach is based on the Extended Burrows-Wheeler Transform (EBWT) (Burrows and Wheeler, 1994; Mantaci et al., 2007) and its combinatorial properties, and it applies the idea that each base in a read can with high probability be predicted by the context of bases that are next to it. We also exploit the fact that such predicted bases add little information, and its quality score can be discarded or heavily compressed without distortion effects on downstream analysis.

In our strategy, the length of contexts is *variable-order* (*i.e.* not fixed *a priori*, unlike BEETL), and can be as large as the full read length for high-enough coverages and small-enough error rates.

We exploit the positional clustering framework introduced in (Prezza et al., 2019) to detect "relevant" blocks in the EBWT. These blocks allow us not only to smooth the quality scores, but also to apply a noise reduction on the corresponding bases, replacing those that are believed to be noise, while keeping variant calling performance comparable to that with the original data and with compression rates comparable to other tools.

2 PRELIMINARIES

Let S be a string (also called sequence or *reads* due to our target application) of length n on the alphabet Σ. We denote the i-th symbol of S by $S[i]$. A *substring* of any $S \in \mathcal{S}$ is denoted as $S[i,j] = S[i]\cdots S[j]$, with $S[1,j]$ being called a *prefix* and $S[i,n+1]$ a *suffix* of S.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings. We assume that each string $S_i \in \mathcal{S}$ has length n_i and is followed by a special end-marker symbol $S_i[n_i+1] = \$_i$, which is lexicographically smaller than any other symbol in $\mathcal{S}$, and does not appear in $\mathcal{S}$ elsewhere.

The Burrows-Wheeler Transform (BWT) (Burrows and Wheeler, 1994) of a text T (and the EBWT of a set of strings $\mathcal{S}$ (Mantaci et al., 2007; Bauer et al., 2013)) is a suitable permutation of the symbols of T (and $\mathcal{S}$) whose output shows a local similarity, *i.e.* symbols preceding similar contexts tend to occur in clusters. Both transformations have been intensively studied from a theoretical and combinatorial viewpoint and have important and successful applications in several areas, *e.g.* (Mantaci et al., 2008; Li and Durbin, 2010; Kimura and Koike, 2015; Shibuya and Comin, 2019; Gagie et al., 2020; Guerrini et al., 2020).

We assume that $N = \sum_{i=1}^{m}(n_i+1)$ denotes the sum of the lengths of all strings in $\mathcal{S}$. The output of the EBWT is a string $\mathsf{ebwt}(\mathcal{S})$ of length N such that $\mathsf{ebwt}(\mathcal{S})[i] = x$, with $1 \leq i \leq N$, if x circularly precedes the i-th suffix (context) $S_j[k, n_j+1]$ (for some $1 \leq j \leq m$ and $1 \leq k \leq n_j+1$), according to the lexicographic sorting of the contexts of all strings in $\mathcal{S}$. In this case, we say that the context $S_j[k, n_j+1]$ is associated with the position i in $\mathsf{ebwt}(\mathcal{S})$. See Table 1 for an example.

In practice, computing the EBWT via suffix sorting (Bauer et al., 2013; Bonomo et al., 2014) may be done considering the same end-markers for all strings.

Table 1: Extended Burrows-Wheeler Transform (EBWT), LCP array, and auxiliary data structures used for detecting positional clusters for the set $\mathcal{S} = \{GGCGTACCA\$_1, GGGGCGTAT\$_2, ACGANTACGAC\$_3\}$ and $k_m = 2$.

i	B_{min}	B_{thr}	lcp	ebwt	Sorted Suffixes
1	0	0	0	A	$\$_1$
2	0	0	0	T	$\$_2$
3	0	0	0	C	$\$_3$
4	0	0	0	C	A$\$_1$
5	0	0	1	G	AC$\$_3$
6	0	1	2	T	ACCA$\$_1$
7	0	1	2	T	ACGAC$\$_3$
8	0	1	4	$	ACGANTACGAC$\$_3$
9	1	0	1	G	ANTACGAC$\$_3$
10	0	0	1	T	AT$\$_2$
11	1	0	0	A	C$\$_3$
12	0	0	1	C	CA$\$_1$
13	0	0	1	A	CCA$\$_1$
14	0	0	1	A	CGAC$\$_3$
15	0	1	3	A	CGANTACGAC$\$_3$
16	1	1	2	G	CGTACCA$\$_1$
17	0	1	4	G	CGTAT$\$_2$
18	1	0	0	C	GAC$\$_3$
19	0	1	2	C	GANTACGAC$\$_3$
20	1	0	1	G	GCGTACCA$\$_1$
21	0	1	5	G	GCGTAT$\$_2$
22	1	0	1	$	GGCGTACCA$\$_1$
23	0	1	6	G	GGCGTAT$\$_2$
24	1	1	2	G	GGGCGTAT$\$_2$
25	0	1	3	$	GGGGCGTAT$\$_2$
26	1	0	1	C	GTACCA$\$_1$
27	0	1	3	C	GTAT$\$_2$
28	1	0	0	A	NTACGAC$\$_3$
29	0	0	0	A	T$\$_2$
30	0	0	1	G	TACCA$\$_1$
31	0	1	3	N	TACGAC$\$_3$
32	0	1	2	G	TAT$\$_2$

That is, we assume that $\$_i < \$_j$, if $i < j$, and use a unique symbol \$ as the end-marker for all strings.

The *longest common prefix* (LCP) array (Manber and Myers, 1990) of $\mathcal{S}$ is the array $\mathsf{lcp}(\mathcal{S})$ of length $N+1$, such that for $2 \leq i \leq N$, $\mathsf{lcp}(\mathcal{S})[i]$ is the length of the longest common prefix between the contexts associated with the positions i and $i-1$ in $\mathsf{ebwt}(\mathcal{S})$, while $\mathsf{lcp}(\mathcal{S})[1] = \mathsf{lcp}(\mathcal{S})[N+1] = 0$. The LCP-intervals (Abouelhoda et al., 2004) are maximal intervals $[i, j]$ that satisfy $\mathsf{lcp}(\mathcal{S})[r] \geq k$ for $i < r \leq j$ and whose associated contexts share at least the first k symbols.

An important property of the BWT and EBWT, is the so-called *LF mapping* (Ferragina and Manzini, 2000), which states that the i-th occurrence of symbol x on the BWT string and the first symbol of the i-th lexicographically-smallest suffix that starts with x correspond to the same position in the input string (or string collection). We will use the LF mapping to perform *backward searches* when creating a new (modified) FASTQ file. The backward search allows to find the range of suffixes prefixed by a given string – see (Ferragina and Manzini, 2000; Adjeroh et al., 2008) for more details.

3 METHOD

We structure our reference-free FASTQ compression method in four main steps: (a) data structures building, (b) positional cluster detecting, (c) noise reduction and quality score smoothing, and (d) FASTQ reconstruction.

(a) Data Structures Building. This phase consists in computing the EBWT and the LCP array for the collection of sequences $\mathcal{S}$ stored in the *bases* component of the input FASTQ file. We also compute $\mathsf{qs}(\mathcal{S})$, as the concatenation of the quality scores associated with each symbol in $\mathsf{ebwt}(\mathcal{S})$, *i.e.*, the string qs contains a permutation of the quality score symbols that follows the symbol permutation in $\mathsf{ebwt}(\mathcal{S})$. Note that the $\mathsf{lcp}(\mathcal{S})$ is only used in the next step, so we can either explicitly compute it in this phase or implicitly deduce it by $\mathsf{ebwt}(\mathcal{S})$ during the next step.

(B) Positional Cluster Detecting. A crucial property of the EBWT is that symbols preceding suffixes that begin with the same substring (context) w will result in a contiguous substring of ebwt, and thus of qs. Such a substring of the ebwt is generally called *cluster*. In literature, such clusters depending on the length k of the context w are associated with LCP-intervals (Abouelhoda et al., 2004).

The aim of the positional clustering framework (Prezza et al., 2019; Prezza et al., 2020) is to overcome the limitation of strategies based on LCP-intervals, which depend on the choice of k. Intuitively, meaningful clusters in the EBWT lie between local minima in the LCP array, and symbols of the same positional cluster usually cover the same genome location (Prezza et al., 2019). This recent strategy automatically detects, in a data-driven way, the length k of the common prefix shared by the suffixes of a cluster in the EBWT. Moreover, so as to exclude clusters corresponding to short random contexts, we set a minimum length for the context w, denoted by k_m.

Analogously to (Prezza et al., 2020), we define positional clusters by using two binary vectors: B_{thr} and B_{min}, where $B_{thr}[i] = 1$ if and only if $\mathsf{lcp}[i] \geq k_m$, and $B_{min} = 1$ if and only if $\mathsf{lcp}[i]$ is a local minimum *i.e.*, it holds $\mathsf{lcp}[i-1] > \mathsf{lcp}[i] \leq \mathsf{lcp}[i+1]$, for all $1 < i \leq N$, which depends on data only. A EBWT positional cluster is a maximal substring $\mathsf{ebwt}[i,j]$ such that $\mathrm{B}_{thr}[r] = 1$, for all $i < r \leq j$, and $\mathrm{B}_{min}[r] = 0$, for all $i < r \leq j$. See, for instance, Table 1.

(C.1) Noise Reduction. Given the base symbols appearing in any positional cluster of $\mathsf{ebwt}(\mathcal{S})$, we call as *frequent symbol* any symbol whose occurrence in the cluster is greater than a threshold percentage. The idea that lies behind changing bases is to reduce the number of symbols in a cluster that are different from the most frequent symbols, while preserving the variant calls. So, we take into account only clusters that have no more than two frequent symbols (for example, we set the threshold percentage to 40%).

The symbols in an EBWT positional cluster usually correspond to the same genome location (Prezza et al., 2019). Thus, given an EBWT positional cluster $\alpha = \mathsf{ebwt}[i,j]$, we say a symbol b is a *noisy base* if it is different from the most frequent symbols and all occurrences of b in α are associated with low quality values in $\mathsf{qs}[i,j]$ (*i.e.* there are no occurrences of b with a high quality score in α). Intuitively, a noisy base is more likely noise introduced during the sequencing process.

Then, for each analyzed cluster α, we replace noisy bases in α with a predicted base c as follows. We distinguish two cases. If the cluster α contains a unique most frequent symbol c, then we replace the noisy base b with c. Otherwise, if we have two different frequent symbols, for each occurrence of them and for the noisy base b, we compute the preceding context of length ℓ in their corresponding reads (*i.e.* left context of each considered base), by means of the backward search applied to $\mathsf{ebwt}(\mathcal{S})$ (for example, we set $\ell = 1$ in our experiments). If the left context preceding b coincides with the contexts preceding all the occurrences of d (one of the two most frequent symbols), then we replace the base b with d. We specify that no base changes are performed if the frequent symbols are preceded by the same contexts.

(C.2) Quality Score Smoothing. During step (c.1), we also modify quality scores by smoothing the symbols of qs that are associated with base symbols in clusters of ebwt.

In any cluster α, the value Q used for replacements can be computed with different strategies: (i) Q is a default value, or (ii) Q is the quality score associated with the mean probability error in α, or (iii) Q is the maximum quality score in α, or (iv) Q is the average of the quality scores in α. According to this smoothing process, apart from strategy (i), the value Q depends on the cluster analyzed. In the experiments we evaluated qs smoothing as follows. For each position r within $\alpha = \mathsf{ebwt}[i,j]$ we smooth the quality score $qs[r]$ with Q, either if $\mathsf{ebwt}[r]$ is one of the most frequent symbols (regardless of its quality), or if $qs[r]$ is greater than Q.

An additional feature to compress further quality scores is the possibility of reducing the number of the alphabet symbols appearing in $\mathsf{qs}(\mathcal{S})$. This smoothing approach is quite popular and standard in literature (Chandak et al., 2018). Then, in addition to one of the strategies described above, we can apply the Illumina 8-level binning reducing to 8 the number of different symbols in qs.

(D) FASTQ Reconstruction and Compression. Given the modified symbols in $\mathsf{ebwt}(\mathcal{S})$ and in $\mathsf{qs}(\mathcal{S})$ (according to the strategies described above), we use the LF mapping on the original ebwt string to retrieve the order of symbols and output a new (modified) FASTQ file.

The *headers* component with the read titles can be either omitted (inserting the symbol '@' as header) or kept as they are in the original FASTQ file.

At the end, the resulting FASTQ file is compressed by using any state-of-the-art compressor.

4 EXPERIMENTS

Implementations. We present two implementations of our tool BFQZIP: in internal memory and in external memory. Now we give a brief description of them.

Given a FASTQ file containing a collection $\mathcal{S}$, both implementations take as input the files containing $\mathsf{ebwt}(\mathcal{S})$ and $\mathsf{qs}(\mathcal{S})$. In the external memory, we also need the array $\mathsf{lcp}(\mathcal{S})$.

The construction of these data structures during the first step can be performed with any tools. *e.g.* (Bauer et al., 2013; Bonizzoni et al., 2019; Egidi et al., 2019; Louza et al., 2020; Boucher et al., 2021) according to the resources available (which is a good feature).

The two implementations largely differ in the detection of positional clusters. Indeed, alike (Prezza et al., 2020), the internal memory approach represents $\mathsf{ebwt}(\mathcal{S})$ via the compressed suffix tree described in (Prezza and Rosone, 2021) (see also (Belazzougui et al., 2020)), where it is shown that $\mathsf{lcp}(\mathcal{S})$ can be induced from the EBWT using succinct working space for any alphabet size. Whereas, in external memory, EBWT positional clusters are detected

Table 2: Paired-end datasets used in the experiments and their sizes in bytes. Each dataset is obtained from two files (_1 and _2), whose number of reads and read length are given in columns 2 and 3. We distinguish the size of the original FASTQ (raw data) from the size of the same FASTQ file with all headers removed (*i.e.*, replaced by '@'). In the last column we report the size of the bases component, that is equal to the quality scores component size.

Dataset	No. reads	Length	Raw (complete)	FASTQ	DNA/QS
ERR262997_1 - chr 20	13,796,697	101	3,420,752,544	2,869,712,976	1,407,263,094
ERR262997_2 - chr 20	13,796,697	101	3,420,752,544	2,869,712,976	1,407,263,094
ERR262997_1 - chr 14	18,596,541	101	4,611,888,574	3,868,080,528	1,896,847,182
ERR262997_2 - chr 14	18,596,541	101	4,611,888,574	3,868,080,528	1,896,847,182
ERR262997_1 - chr 1	49,658,795	101	12,211,743,094	10,329,029,360	5,065,197,090
ERR262997_2 - chr 1	49,658,795	101	12,211,743,094	10,329,029,360	5,065,197,090

by reading the $\mathsf{lcp}(\mathcal{S})$ stored in a file in a sequential way.

For the last two steps, the two implementations are similar, except that the data are kept in internal or external memory. In particular, during step (d), we use the LF-mapping either in internal memory – via the suffix-tree navigation as in (Prezza et al., 2020) – or in external memory – similarly to (Bauer et al., 2013).

Datasets. The easiest approach to evaluate the validity of our method could be to simulate reads and variants from a reference genome. However, it is not trivial to simulate variant artifacts for this purpose (Li, 2014), so we focus only on real data.

In this study, thus, we use the real human dataset ERR262997 corresponding to a 30x-coverage paired-end Whole Genome Sequencing (WGS) data for the CEPH 1463 family. Similarly to other studies (Ochoa et al., 2016), for evaluation purposes we extracted the chromosomes 20, 14 and 1 from ERR262997, obtaining datasets of different sizes (see Table 2).

Compression. We describe experiments that show that our strategy is able to compress FASTQ files in lossy way modifying both the bases and quality scores component, while keeping most of the information contained in the original file.

To the best of our knowledge, none of the existing FASTQ compressors evaluates and modifies both the *bases* and the *quality scores* components at the same time. Therefore, no comparison with existing tools is completely fair. We consider the tools BEETL[1] (Janin et al., 2014) and LEON[2] (Benoit et al., 2015) that are reference-free and read-based: they smooth the *quality score* component in a lossy way based on the biological information of the *bases*, without modifying the *bases* themselves.

We choose BEETL, because it is based on EBWT and takes as input the same data structures we compute during step (a). BEETL smooths to a constant value the quality scores corresponding to each run of the same symbol associated with the LCP-interval $[i, j]$. The quality scores associated to each run in $\mathsf{ebwt}[i, j]$ are smoothed if the length of the run is greater than a minimum stretch length s. So, it needs two parameters: the minimum stretch length s and the cut threshold c for the LCP-interval, failing to separate read suffixes differing after c positions (this is, indeed, a drawback of all c-mer-based strategies).

We choose LEON (Benoit et al., 2015) that, on the contrary, needs to build a reference from the input reads in the form of a bloom filter compressed *de Bruijn* graph, and then maps each nucleotide sequence as a path in the graph. Thus, LEON can be considered assembly-based, since it uses a *de Bruijn* graph as a de novo reference. If a base is covered by a sufficiently large number of k-mers (substrings of length k) stored in the bloom filter, its quality is set to a fixed high value. Thus, LEON depends on a fixed parameter k for the graph, as well.

Regarding the output of each tool, BFQzip produces a new FASTQ file with modified bases and smoothed qualities, whereas BEETL only produces a BWT-ordered smoothed qualities file, which is used to replace the quality scores component in the modified FASTQ file (with the original bases component). LEON produces a proprietary format compressed file that encodes the *de Bruijn* graph, thus, it was necessary to uncompress the output file to obtain the modified FASTQ. For a fairer comparison, these resulting FASTQ files were compressed with the same tools. In particular, we choose two well-known compressors for this task: PPMd (Cleary and Witten, 1984; Moffat, 1990) and BSC[3].

We run BFQzip and BEETL with similar parameters: in BEETL, we set the replacement quality score to '@' (as set by LEON) and the minimum LCP cut threshold to 30. For BFQzip, we used options `-T 30` to set the minimum context length $k_m = 30$ and `-Q`

[1] https://github.com/BEETL/BEETL/blob/RELEASE_1_1_0/scripts/lcp/applyLcpCutoff.pl

[2] http://gatb.inria.fr/software/leon/

[3] http://libbsc.com/

Table 3: Compression ratio for (original and three smoothing tools) FASTQ files (with headers replaced by '@') and their single components (qualities (QS) and bases (DNA)) obtained by both PPMd and BSC. The ratio is defined as compressed size/original size, where original file size is in Table 2. Since BEETL and LEON do not modify the bases component, their ratio for the DNA component is the same as the original.

	Tool	FASTQ	QS	DNA	FASTQ	QS	DNA
		ERR262997_1 chr 20			ERR262997_2 chr 20		
PPMd	Original	0.2473	0.2999	0.2037	0.2547	0.3142	0.2046
	LEON	0.1152	0.0317		0.1234	0.0473	
	BEETL	0.1900	0.1833		0.2002	0.2033	
	BFQZIP	0.1941	0.1918	0.2035	0.2043	0.2119	0.2043
BSC	Original	0.2005	0.2902	0.1154	0.2095	0.3034	0.1207
	LEON	0.0677	0.0241		0.0780	0.0394	
	BEETL	0.1413	0.1724		0.1534	0.1915	
	BFQZIP	0.1453	0.1830	0.1152	0.1572	0.2024	0.1200
		ERR262997_1 chr 14			ERR262997_2 chr 14		
PPMd	Original	0.2482	0.2956	0.2100	0.2544	0.3076	0.2106
	LEON	0.1175	0.0301		0.1249	0.0444	
	BEETL	0.1916	0.1805		0.2010	0.1989	
	BFQZIP	0.1957	0.1889	0.2098	0.2050	0.2074	0.2103
BSC	Original	0.1992	0.2862	0.1174	0.2071	0.2972	0.1224
	LEON	0.0674	0.0226		0.0770	0.0367	
	BEETL	0.1406	0.1698		0.1518	0.1874	
	BFQZIP	0.1445	0.1786	0.1164	0.1555	0.1962	0.1210
		ERR262997_1 chr 1			ERR262997_2 chr 1		
PPMd	Original	0.2461	0.2969	0.2046	0.2529	0.3097	0.2054
	LEON	0.1148	0.0299		0.1224	0.0445	
	BEETL	0.1876	0.1777		0.1973	0.1966	
	BFQZIP	0.1918	0.1864	0.2044	0.2015	0.2054	0.2051
BSC	Original	0.1984	0.2874	0.1146	0.2068	0.2991	0.1197
	LEON	0.0661	0.0224		0.0758	0.0367	
	BEETL	0.1379	0.1669		0.1494	0.1850	
	BFQZIP	0.1419	0.1759	0.1136	0.1533	0.1941	0.1183

Table 4: Evaluation of called variants by means of `rtg vcfeval`: comparison between called variants from a modified FASTQ and variants from the original FASTQ used as baseline.

	ERR262997 chr 20			ERR262997 chr 14			ERR262997 chr 1		
	PREC	SEN	F	PREC	SEN	F	PREC	SEN	F
LEON	0.9593	0.9356	0.9473	0.9626	0.9381	0.9502	0.9589	0.9348	0.9467
BEETL	0.9584	0.9529	0.9556	0.9626	0.9553	0.9590	0.9596	**0.9526**	0.9561
BFQZIP	**0.9613**	**0.9534**	**0.9574**	**0.9650**	**0.9555**	**0.9602**	**0.9628**	0.9523	**0.9575**

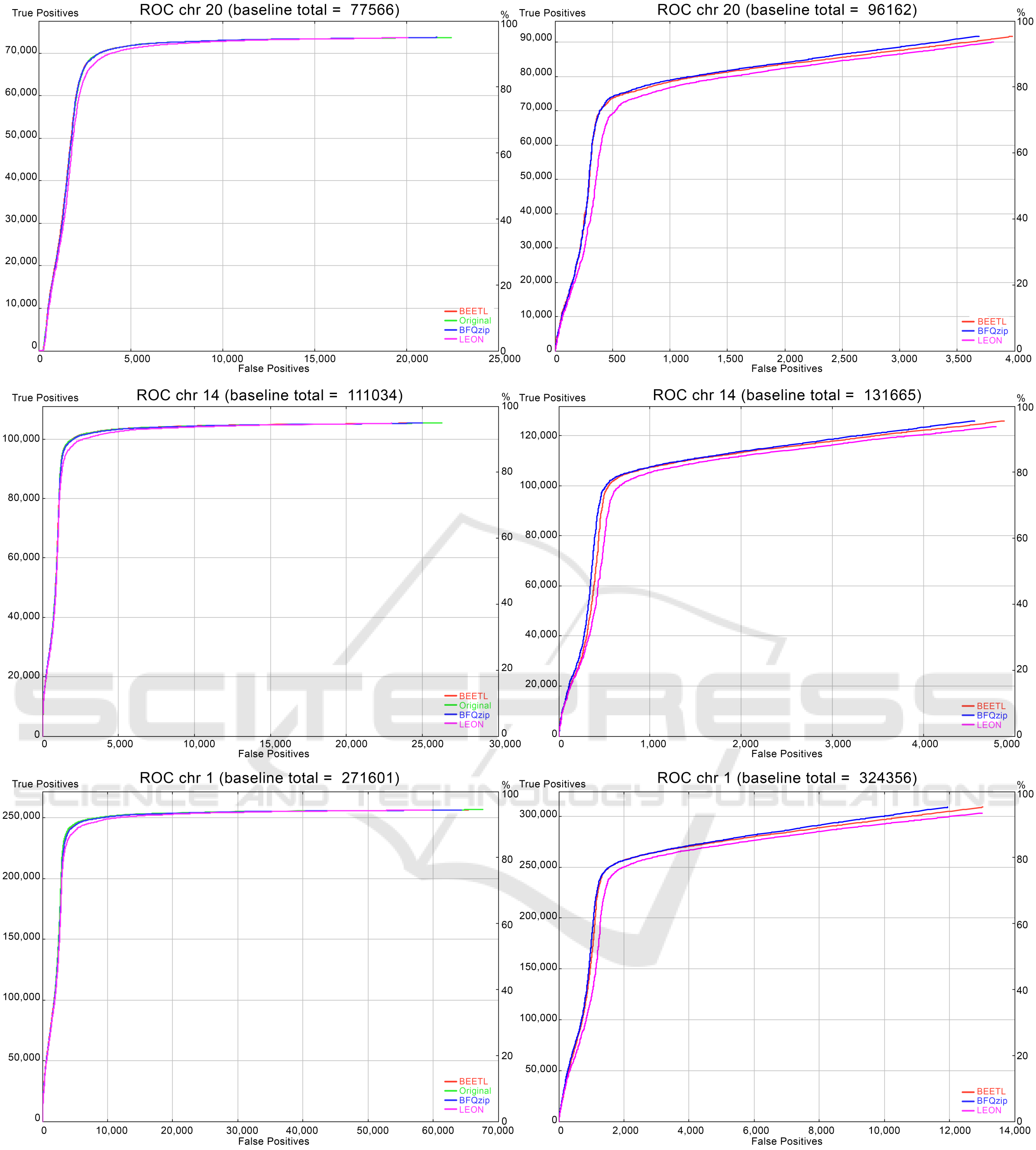
`@` to set the constant replacement value. LEON was executed with default parameters for *k*-mer size and minimal abundance threshold, as suggested by the authors. The exact commands for the tools are:

- `python3 BFQzip.py <input>.fastq -o <output>.fastq -T 30 -Q @`
- `applyLcpCutoff.pl -b <input>.ebwt -q <input>.ebwt.qs -l <input>.lcp -o <output>.ebwt.qs -c 30 -r 64 -s 5`
- `leon -file <input>.fastq -c` (for decompression `-d`) `-nb-cores 1`

In Table 3, we report the compression ratios achieved by PPMd and BSC given as input the FASTQ modified by any of the three tools and the original FASTQ file. Note that each of the two FASTQ files comprising any paired-end dataset is compressed separately.

Table 3 shows that all tools improve the compression of the data (compared with the original FASTQ). In particular, LEON is better in terms of compression ratios than other tools. This improvement is due to a greater ability to smooth the quality scores component. Recall that LEON truncates all quality scores above a given threshold (qualities higher than '@' are replaced by '@') and in these datasets the total frequency of symbols greater than '@' is about $80-90\%$. However, the results of BFQZIP and BEETL were similar in almost all cases.

Validation. In lossy FASTQ compression, it is important to take into account the impact of the modified data on downstream analysis, so we need to evaluate the genotyping accuracy.



ROC chr 20 (baseline total = 77566)

ROC chr 20 (baseline total = 96162)

ROC chr 14 (baseline total = 111034)

ROC chr 14 (baseline total = 131665)

ROC chr 1 (baseline total = 271601)

ROC chr 1 (baseline total = 324356)

Figure 1: ROC Curves obtained by `rtg rocplot`: true positive as a function of the false positive respect both to the Ground truth as baseline (left side) and to the original file as baseline (right side).

We compared the set of variants retrieved from a baseline with the set retrieved from the modified FASTQ. First, we considered as baseline the set of "ground truth" variants for NA12878 provided by Illumina[4] and then, we considered as baseline the set of variants obtained from the original FASTQ files.

The SNP calling pipeline is a bash script[5] (Li, 2013) to align sequences to the reference (in our case, the latest build of the human reference genome, GRCh38/hg38) and GATK-HaplotypeCaller (DePristo and et al., 2011) to call SNPs. The output

[4] https://github.com/Illumina/PlatinumGenomes

[5] https://github.com/veronicaguerrini/BFQzip/blob/main/variant_calling/pipeline_SNPsCall.sh

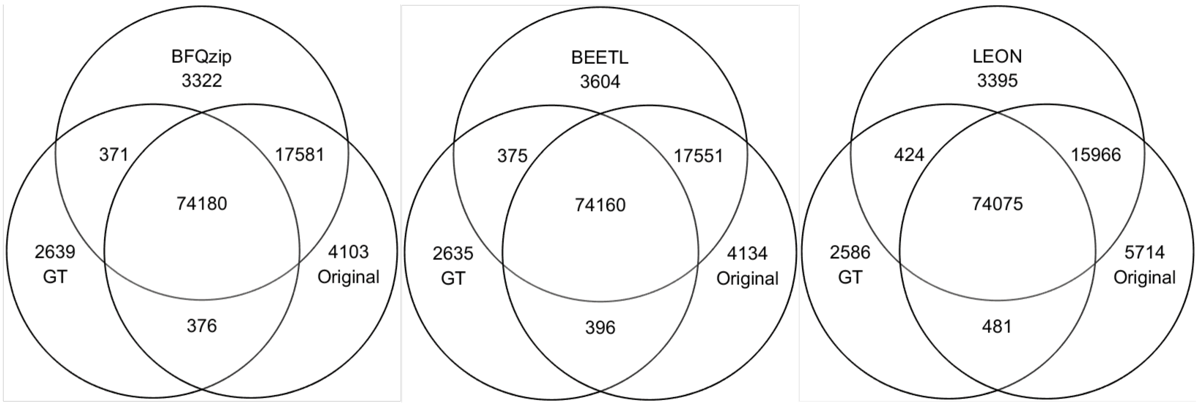


Figure 2: Venn diagrams for chr 20 variants: set comparison between the variants in the original FASTQ file (Original), those in the ground truth (GT) and those called from the FASTQ modified by BFQzip (left), or modified by BEETL (middle), or modified by LEON (right).

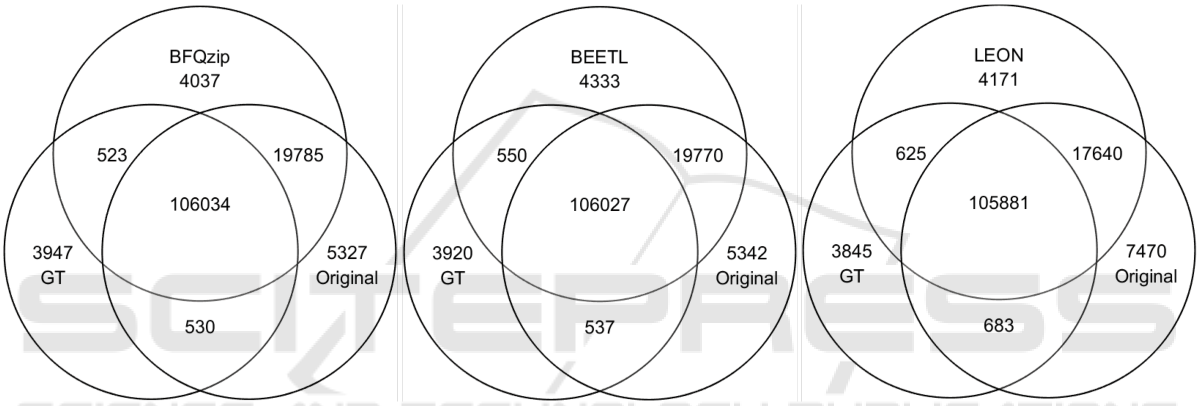


Figure 3: Venn diagrams for chr 14 variants: set comparison between the variants in the original FASTQ file (Original), those in the ground truth (GT) and those called from the FASTQ modified by BFQzip (left), or modified by BEETL (middle), or modified by LEON (right).

of the pipeline is a VCF file that contains the set of called SNPs, which are then compared to those contained in a baseline set by using RTG Tools[6]: `rtg vcfeval` evaluates agreement between called and baseline variants using the following performance metrics.

True positive (TP) are those variants that match in both baseline and query (the set of called variants); false positives (FP) are those variants that have mismatching, that are in the called set of variants but not in the baseline; false negatives (FN) are those variants that are present in the baseline, but missing in the query. These values are employed to compute precision (PREC) that measures the proportion of called variants that are true, and sensitivity (SEN), which measures the proportion of called variants included in the consensus set. A third metric is the harmonic mean between sensitivity and precision (known as F-measure):

$$\text{PREC} = \frac{TP}{TP+FP}, \qquad \text{SEN} = \frac{TP}{TP+FN},$$

$$\text{F} = \frac{2 \cdot SEN \cdot PREC}{SEN + PREC}.$$

The tool `rtg vcfeval` can also output a ROC file based on the `QUAL` field, which can be viewed by `rtg rocplot`.

Table 4 reports the evaluation results of the variants retrieved from a modified FASTQ (produced by any tool) compared to the set of variants from the original (unsmoothed) FASTQ, which is used as baseline. We observe that our method provides the highest precision and F measure. This means that, with respect to the original FASTQ, we have a higher number of common variants (TP) and the lowest number of newly-introduced variants (FP), thus we preserve information of the original FASTQ more than the others (see also Figure 1, right side). This is a desirable property useful in several applications.

[6] https://www.realtimegenomics.com/products/rtg-tools

In Figure 1 we provide the ROC curves associated with the variant comparison performed by `rtg vcfeval` (reporting the number of TP as a function of the FP), showing similar results.

Figure 1 (left side) shows that our tool preserves those variants that are in the ground truth: using the VCF file of the ground truth as baseline, all tools show an accuracy similar to the original. In particular, BFQZIP and BEETL curves in each graph overlap the original one, while LEON's curve is a bit lower.

Figure 1 (right side) also shows that our tool preserves the original variants (TP) introducing only a little number of new ones (FP) when the original FASTQ variants are used as baseline (as also shown by the percentages of sensitivity and precision in Table 4) .

With the idea of showing the preservation of variants from the original file and the possibility of losing variants due to the sequencer noise, we decided to manually check the variants obtained for any tool by intersecting the corresponding vcf file with that from the FASTQ original and with the ground truth vcf. Figures 2 and 3 show as our tool preserves the variants that are both in the original file and in the ground truth (*i.e.* GT) more than the others.

This analysis appears to confirm what we observed with the ROC curves. BFQZIP reports the smaller number of new variants introduced by the tool, that are those variants neither in the original file nor in the ground truth.

The majority of the variants present in both the original FASTQ and the ground truth is preserved by all tools. However, it appears that the heavy truncation of the quality scores carried out by LEON leads to a loss of variants present in the original file (see, for instance, the intersection between each tool and the original, or the intersection between the ground truth and the original). While BFQZIP (and in the similar way BEETL) preserves a high number of variants that are present in the original file (see intersection between each tool and the original).

We also made a detailed analysis of the modifications performed by our method, comparing the modified bases which correspond to parts aligned by BWA with the relative bases in the reference (taking into account how BWA aligned the reads). We have noticed that about 91-93% of the changed bases correspond to the reference. The remaining part includes positions we cannot evaluate, such as those in the reads skipped by the aligner.

5 DISCUSSION AND CONCLUSIONS

We propose the first lossy reference-free and assembly-free compression approach for FASTQ files, which combines both DNA bases and quality information in the reads to smooth the quality scores and to apply a noise reduction of the bases, while keeping variant calling performance comparable to that with original data. To the best of our knowledge, there are no tools that have been designed for this purpose, so we compared our results with two reference-free and read-based tools that only smooth out the quality scores component: BEETL (Janin et al., 2014) and LEON (Benoit et al., 2015).

The resulting FASTQ file with the modified bases and quality scores produced by our tool achieves better compression than the original data. In particular, by using our approach the *bases* component achieves better compression than the original (therefore also than competitors which do not make any changes to this component), whereas the compression ratio of the quality scores component is more competitive with BEETL than LEON, which also truncates all quality values greater than '@'. On the other hand, in terms of variant calling, our tool keeps the same accuracy as the original FASTQ data when the ground truth is used as baseline, and preserves the variant calls of the original FASTQ file better than BEETL and LEON.

From the viewpoint of the used resources, LEON has shown to be the fastest tool, although this comparison is not completely fair because our tool modifies different components of the FASTQ file and the outputs are different (not directly comparable). Moreover, the authors in (Benoit et al., 2015) state that for WGS datasets, the relative contribution of the Bloom filter is low for high coverage datasets, but prohibitive for low coverage datasets (*e.g.* 10x). We intend to improve our implementation also using, for instance, parallelization, and test our tool for low coverage datasets and longer reads.

Our implementations give as output the modified FASTQ file, so we have used PPMd and BSC for compression, but we could use any other compressor for this task and we could also combine existing lossless compression schemes to further reduce the size of the FASTQ file, for instance we could use SPRING (Chandak et al., 2018), FQSquezeer (Deorowicz, 2020), and others.

Moreover, our strategy could take advantage of the reordering of the reads based on their similarity, *e.g.* as in (Cox et al., 2012; Chandak et al., 2018; Deorowicz, 2020). Another feature we did not exploit in our compression scheme is the paired-end information coming from reads in pairs. (Indeed, we compress the FASTQ files in a paired-end dataset independently, as they were single-end.) Both the above aspects could be analyzed as future work.

We believe the results presented in this paper can motivate the development of new FASTQ compressors that modify the *bases* and *quality scores* components taking into account both information at the same time to achieve better compression while keeping most of the relevant information in the FASTQ data. As future work we intend to investigate the error correction problem that needs to take into account much more information (*e.g.* reverse-complement, or paired-end information).

ACKNOWLEDGEMENTS

The authors would like to thank N. Prezza for valuable comments and suggestions and for providing part of the code library, and E. Niccoli for preliminary experimental investigations on positional clustering and compression in his bachelor's thesis under the supervision of GR and VG.

Work partially supported by the project MIUR-SIR CMACBioSeq ("Combinatorial Methods for Analysis and Compression of Biological Sequences") grant n. RBSI146R5L and by the University of Pisa under the "PRA – Progetti di Ricerca di Ateneo" (Institutional Research Grants) - Project no. PRA_2020-2021_26 "Metodi Informatici Integrati per la Biomedica".

REFERENCES

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86.

Adjeroh, D., Bell, T., and Mukherjee, A. (2008). *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.

Bauer, M., Cox, A., and Rosone, G. (2013). Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134 – 148.

Belazzougui, D., Cunial, F., Kärkkäinen, J., and Mäkinen, V. (2020). Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2).

Benoit, G., Lemaitre, C., Lavenier, D., Drezen, E., Dayris, T., Uricaru, R., and Rizk, G. (2015). Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph. *BMC Bioinformatics*, 16.

Bonfield, J. K. and Mahoney, M. V. (2013). Compression of FASTQ and sam format sequencing data. *PLOS ONE*, 8(3).

Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M., and Rizzi, R. (2019). Multithread Multistring Burrows-Wheeler Transform and Longest Common Prefix Array. *Journal of computational biology*, 26(9):948—961.

Bonomo, S., Mantaci, S., Restivo, A., Rosone, G., and Sciortino, M. (2014). Sorting conjugates and suffixes of words in a multiset. *International Journal of Foundations of Computer Science*, 25(08):1161–1175.

Boucher, C., Cenzato, D., Lipták, Z., Rossi, M., and Sciortino, M. (2021). Computing the original ebwt faster, simpler, and with less memory. In *SPIRE*, pages 129–142. Springer International Publishing.

Burrows, M. and Wheeler, D. (1994). A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center.

Chandak, S., Tatwawadi, K., Ochoa, I., Hernaez, M., and Weissman, T. (2018). SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, 35(15):2674–2676.

Cleary, J. and Witten, I. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402.

Cox, A., Bauer, M., Jakobi, T., and Rosone, G. (2012). Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419.

Deorowicz, S. (2020). Fqsqueezer: k-mer-based compression of sequencing data. *Scientific reports*, 10(1):1–9.

DePristo, M. A. and et al. (2011). A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics*, 43(5):491–498.

Egidi, L., Louza, F. A., Manzini, G., and Telles, G. P. (2019). External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society.

Gagie, T., Navarro, G., and Prezza, N. (2020). Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM*, 67(1):2:1–2:54.

Greenfield, D. L., Stegle, O., and Rrustemi, A. (2016). GeneCodeq: quality score compression and improved genotyping using a Bayesian framework. *Bioinformatics*, 32(20):3124–3132.

Guerrini, V., Louza, F., and Rosone, G. (2020). Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinformatics*, 21.

Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. (2012). SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057.

Hernaez, M., Pavlichin, D., Weissman, T., and Ochoa, I. (2019). Genomic data compression. *Annual Review of Biomedical Data Science*, 2(1):19–37.

Janin, L., Rosone, G., and Cox, A. J. (2014). Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, 30(1):24–30.

Kimura, K. and Koike, A. (2015). Ultrafast SNP analysis using the Burrows-Wheeler transform of short-read data. *Bioinformatics*, 31(10):1577–1583.

Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv:1303.3997*.

Li, H. (2014). Toward better understanding of artifacts in variant calling from high-coverage samples. *Bioinformatics*, 30(20):2843–2851.

Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595.

Louza, F. A., Telles, G. P., Gog, S., Prezza, N., and Rosone, G. (2020). gsufsort: constructing suffix arrays, lcp arrays and bwts for string collections. *Algorithms for Molecular Biology*, 15.

Malysa, G., Hernaez, M., Ochoa, I., Rao, M., Ganesan, K., and Weissman, T. (2015). QVZ: lossy compression of quality values. *Bioinformatics*, 31(19):3122–3129.

Manber, U. and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *ACM-SIAM SODA*, pages 319–327.

Mantaci, S., Restivo, A., Rosone, G., and Sciortino, M. (2007). An extension of the Burrows-Wheeler Transform. *Theoret. Comput. Sci.*, 387(3):298–312.

Mantaci, S., Restivo, A., Rosone, G., and Sciortino, M. (2008). A new combinatorial approach to sequence comparison. *Theory Comput. Syst.*, 42(3):411–429.

Moffat, A. (1990). Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921.

Numanagić, I., Bonfield, J., Hach, F., Voges, J., and Sahinalp, C. (2016). Comparison of high-throughput sequencing data compression tools. *Nature Methods*, 13:1005–1008.

Ochoa, I., Hernaez, M., Goldfeder, R., Weissman, T., and Ashley, E. (2016). Effect of lossy compression of quality scores on variant calling. *Briefings in Bioinformatics*, 18(2):183–194.

Prezza, N., Pisanti, N., Sciortino, M., and Rosone, G. (2019). SNPs detection by eBWT positional clustering. *Algorithms for Molecular Biology*, 14(1):3.

Prezza, N., Pisanti, N., Sciortino, M., and Rosone, G. (2020). Variable-order reference-free variant discovery with the Burrows-Wheeler transform. *BMC Bioinformatics*, 21.

Prezza, N. and Rosone, G. (2021). Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138 – 156.

Roguski, L., Ochoa, I., Hernaez, M., and Deorowicz, S. (2018). FaStore: a space-saving solution for raw sequencing data. *Bioinformatics*, 34(16):2748–2756.

Shibuya, Y. and Comin, M. (2019). Better quality score compression through sequence-based quality smoothing. *BMC Bioinform.*, 20-S(9):302:1–302:11.

Yu, Y. W., Yorukoglu, D., Peng, J., and Berger, B. (2015). Quality score compression improves genotyping accuracy. *Nature biotechnology*, 33(3):240—243.