HIGH-THROUGHPUT SEQUENCING DATA COMPRESSION

Łukasz Roguski

TESI DOCTORAL UPF / 2017

DIRECTORS DE LA TESI

Dr. Paolo Ribeca
Dr. Sebastian Deorowicz

DEPARTAMENT DE CIÈNCIES EXPERIMENTALS I DE LA SALUT

upf. Universitat Pompeu Fabra Barcelona

The power for creating a better future is
contained in the present moment:
You create a good future
by creating a good present.
— Eckhart Tolle

ACKNOWLEDGEMENTS

First and foremost, I offer sincerely my gratitude to my supervisors Dr. Sebastian Deorowicz, from the Silesian University of Technology, and Dr. Paolo Ribeca, from the Pirbright Institute. Sebastian Deorowicz is an excellent researcher and professor who I met during undergraduate studies – his expertise has been a guidance during my adventure with developing methods for processing large volumes of data. His vast knowledge in the area, advice and encouragement have been essential during completing my master thesis and we have continued our fruitful collaboration on this work. I greatly appreciate his support and patience, always being able to count on him.

Paolo Ribeca is a talented researcher and a brilliant mind who I have been fortunate to work with, starting our collaboration as an intern and extending it to a long research stay at Centro Nacional de Análisis Genómico (CNAG) in Barcelona. His vast expertise in the area, advice, support, and thinking "outside the box", helped me to tackle the problems in bioinformatics field from different perspectives and helped me to understand the challenges of working with genomic data. I also enjoyed a lot crunching challenging problems together on the blackboard with discussions full of surreal sense of humor.

Furthermore, I am grateful to my tutor Roderic Guigó from Universitat Pompeu Fabra (UPF) and Centre for Genomic Regulation (CRG), who is an excellent researcher and professor. He has been always welcoming, helpful, motivating to look for answers for the large number of questions in the field of biology (and challenges in the field of bioinformatics), and encouraging me to pursue my goals in scientific career.

I also would like to acknowledge all the organizations, teams and people I have collaborated with. First of all, my research stay at CNAG has been of a great value. It allowed me to get insight into a vast number of biological concepts and processes, which we try to understand empowered with sequencing technologies and using computational methods from the bioinformatics field. This would not be possible without Ivo Gut, the director of CNAG, to who I am grateful for supporting my research, projects and for valuable advices during all these years.

Furthermore, I would like to thank Andrzej Polański and Joanna Polańska from the Silesian University of Technology, who encouraged and supported my initial short-term stay at CNAG under the project INTERKADRA. Collaboration with Mikel Hernaez and Idoia Ochoa from the University of Illinois has been also a very valuable experience within our new project. I have learned a lot working together and I greatly appreciate our collaboration. I would also like to thank David Man for his comments and help on correcting English grammar and style of this thesis (excluding this part, which was added at the same end).

While doing my research at CNAG, I have met and worked with fantastic people – I am grateful to all my close colleagues for these positive experiences. First of all, I would like to thank Leonor, one of the most skilled algorithmicians, a scientist, a dancer, and a volleyball player, who has been always supportive, encouraging and helpful; she is an exceptional person. I thank David, who has been a greats companion from the beginning, showing that bioinformatics and karate actually have a lot of things in common – together with Anna P., who I am also happy to have met, we spent great time working together and exploring Barcelona after work.

Moreover, I would like to thank Marcos for fruitful discussions, for his help and support both in solving various bioinformatic-related problems and in winning volleyball matches. My thanks also go to Enric, a passionate bioinformatician, with who we shared a lot of outstanding moments, not only crunching problems in the world of bioinformatics. I would also like to thank Emilio who I shared with a significant numbers of coffees – with his wife Giulia (a big 'thank you' to you too), they have been always supportive and helpful. I thank François, a true hacker in bioinformatics, for his support and fruitful, inspiring discussions. My thanks goes also to my PhD colleague Santiago, a very skilled scientist, from who I have

learned a lot. Thank you Miranda for always sharing a lot of positive energy, not allowing to stay down or bored (especially during Spanish classes).

Furthermore, I would like to thank my colleagues Fran, Marysia, Elisabetta, Thasso, Tyler, Jéssica, Jordi, Anna E., Nando, Sophia, Raúl, Pablo, Miguel, Justin, Davide, Irene, with who we shared a lot of worthwhile conversations, coffees, breakfasts, lunches, and/or drinks.

On a personal level, I would like to express my sincere gratitude to my close friends from my hometown, Gliwice, who have supported me during this PhD journey in Barcelona. They have been always present, despite the distance, both in the easy and difficult times, awaiting for the moment I finally close the thesis; they were also checking my progress in person, from time to time, while in Barcelona. Especially, I would like to thank Marcin, who has always been to me like "a brother, but from another mother and father". I thank Dominika, with who we could always understand each other without words. I thank Tomek for his enthusiasm and creativity, never being able to get bored together. Furthermore, I thank Ania & Wojtek, Tobek, and Marta, as I could always count on you, sharing valuable moments together independently on the place and the distance. I would also like to thank Agnieszka, Hiszpan, Monika, and Hania, for always remembering and being around.

Moreover, I am grateful to all my flatmates (and friends) who we shared together with unforgettable moments (apart from the flat), as they were also indirectly involved in this PhD journey almost every day. Firstly, I thank Peter for sharing good vibes while sharing the flat in Barceloneta. This journey, however, would not be so unique without Doron & Kamila, who gave me a sense of being my first family abroad – this was an amazing time in Gracia, thank you so much. I am also very grateful to Yannick for sharing a great time and part of life in Sants, both in our cozy self-assembled flat and in the bodega. Together with Yannick & Francesca, Leo & Marión, Ania & Stefan, Jeroen, Anna L., Fede, Davide, we have collected a lot of unforgettable memories, while living in Sants – thanks for being great people and thank you for making me feel in Barcelona "at home".

Furthermore, my sincere gratitude to my other close friends in Barcelona, sharing a lot both of positive and difficult moments together, who were always supporting me, despite me being (definitely too often) busy, work-

ing or “finishing” the thesis. First of all, I would like to especially thank Ewelina, who I met almost at the beginning of the journey and who has been was always close to me – our trips with Xavi, Pedro or Paola visiting Costa Brava will remain as great memories, among many others. Moreover, I would like to thank Damian for staying always creative, spontaneous, and not politically correct, guaranteeing having a lot of fun together. My big thanks goes to Doris & Remek, Olek & Mireia, for being always warm, welcoming people, and sharing a lot of positive energy (and some drinks). Many thanks to Oriol, for long and inspiring discussions; thanks a lot also for multiple hikes in the mountains and skiing together in the Pyrenees. Thank you Laia & Gerard for incredible hikes in the mountains together and sharing great moments outside Barcelona (and, of course, also in the city). Loli & Joan, thank you for always encouraging to staying creative and for experiencing together the Sónar. Thank you Sandie for your constant encouragements too, apart from the fantastic French crêpes you make. Many thanks go also to Ignasi, Solmaz, Edu and Pere, for sharing a lot of good moments and interesting discussions.

Finally, I am very grateful for my little family, for their indefinite support, presence, patience, and understanding – you have been always with me during this journey, despite me often being absent and fully focused on the thesis. I truly cannot express how grateful I am for my beloved Marina, who has always found positive energy to share, to smile, and to cheer up. We’ve been both going through the tough experiences together, but always finding the way to stay happy, and to enjoy even the little positive moments and successes on our way – and with more positive and great moments very soon to come! I am also very grateful to my parents, who supported me as much as they could both on the distance and while visiting me many times in Barcelona. My brother has been always an inspiration for me, his creativity, strength and stubbornness have been reminding me to keep going, despite the difficulties; thanks a lot “byczku”. Moreover, my sincere gratitude to Montserrat & Francesc for their positive energy and always cheering up. Finally, I would like to thank my uncle Kazik, Sumsi & Francesc for their kind support. Without all of you, I would not have been able to arrive up to here. Thank you.

ABSTRACT

Thanks to advances in sequencing technologies, biomedical research has experienced a revolution over recent years, resulting in an explosion in the amount of genomic data being generated worldwide. The typical space requirement for storing sequencing data produced by a medium-scale experiment lies in the range of tens to hundreds of gigabytes, with multiple files in different formats being produced by each experiment. The current *de facto* standard file formats used to represent genomic data are text-based. For practical reasons, these are stored in compressed form. In most cases, such storage methods rely on general-purpose text compressors, such as gzip. Unfortunately, however, these methods are unable to exploit the information models specific to sequencing data, and as a result they usually provide limited functionality and insufficient savings in storage space. This explains why relatively basic operations such as processing, storage, and transfer of genomic data have become a typical bottleneck of current analysis setups.

Therefore, this thesis focuses on methods to efficiently store and compress the data generated from sequencing experiments. First, we propose a novel general purpose FASTQ files compressor. Compared to gzip, it achieves a significant reduction in the size of the resulting archive, while also offering high data processing speed. Next, we present compression methods that exploit the high sequence redundancy present in sequencing data. These methods achieve the best compression ratio among current state-of-the-art FASTQ compressors, without using any external reference sequence. We also demonstrate different lossy compression approaches to store auxiliary sequencing data, which allow for further reductions in size. Finally, we propose a flexible framework and data format, which allows one to semi-automatically generate compression solutions which are not tied

to any specific genomic file format. To facilitate data management needed by complex pipelines, multiple genomic datasets having heterogeneous formats can be stored together in configurable containers, with an option to perform custom queries over the stored data. Moreover, we show that simple solutions based on our framework can achieve results comparable to those of state-of-the-art format-specific compressors.

Overall, the solutions developed and described in this thesis can easily be incorporated into current pipelines for the analysis of genomic data. Taken together, they provide grounds for the development of integrated approaches towards efficient storage and management of such data.

Resum

Gràcies als avenços en el camp de les tecnologies de seqüenciació, en els darrers anys la recerca biomèdica ha viscut una revolució, que ha tingut com un dels resultats l'explosió del volum de dades genòmiques generades arreu del món. La mida típica de les dades de seqüenciació generades en experiments d'escala mitjana acostuma a situar-se en un rang entre deu i cent gigabytes, que s'emmagatzemen en diversos arxius en diferents formats produïts en cada experiment. Els formats estàndards actuals *de facto* de representació de dades genòmiques són en format textual. Per raons pràctiques, les dades necessiten ser emmagatzemades en format comprimit. En la majoria dels casos, aquests mètodes de compressió es basen en compressors de text de caràcter general, com ara gzip. Amb tot, no permeten explotar els models d'informació específics de dades de seqüenciació. És per això que proporcionen funcionalitats limitades i estalvi insuficient d'espai d'emmagatzematge. Això explica per què operacions relativament bàsiques, com ara el processament, l'emmagatzematge i la transferència de dades genòmiques, s'han convertit en un dels principals obstacles de processos actuals d'anàlisi.

Per tot això, aquesta tesi se centra en mètodes d'emmagatzematge i compressió eficients de dades generades en experiments de sequenciació. En primer lloc, proposem un compressor innovador d'arxius FASTQ de propòsit general. A diferència de gzip, aquest compressor permet reduir de manera significativa la mida de l'arxiu resultant del procés de compressió. A més a més, aquesta eina permet processar les dades a una velocitat alta. A continuació, presentem mètodes de compressió que fan ús de l'alta redundància de seqüències present en les dades de seqüenciació. Aquests mètodes obtenen la millor ratio de compressió d'entre els compressors FASTQ del marc teòric actual, sense fer ús de cap referència externa. També

mostrem aproximacions de compressió amb pèrdua per emmagatzemar dades de seqüenciació auxiliars, que permeten reduir encara més la mida de les dades. En últim lloc, aportem un sistema flexible de compressió i un format de dades. Aquest sistema fa possible generar de manera semi-automàtica solucions de compressió que no estan lligades a cap mena de format específic d'arxius de dades genòmiques. Per tal de facilitar la gestió complexa de dades, diversos conjunts de dades amb formats heterogenis poden ser emmagatzemats en contenidors configurables amb l'opció de dur a terme consultes personalitzades sobre les dades emmagatzemades. A més a més, exposem que les solucions simples basades en el nostre sistema poden obtenir resultats comparables als compressors de format específic de l'estat de l'art.

En resum, les solucions desenvolupades i descrites en aquesta tesi poden ser incorporades amb facilitat en processos d'anàlisi de dades genòmiques. Si prenem aquestes solucions conjuntament, aporten una base sòlida per al desenvolupament d'aproximacions completes encaminades a l'emmagatzematge i gestió eficient de dades genòmiques.

PREFACE

Sequencing has become an essential technique extensively used in biological research. Access to the genetic code of different organisms has allowed us to improve our understanding of the organic world surrounding us. We can thus start deciphering the underlying biological processes, trying to better grasp the history of life on Earth. More recently, sequencing has also become used in the context of precision medicine, complementing clinical decision-making and improving a number of treatments for common and rare human disorders. However, with continuous advances in high-throughput sequencing technologies, we also observe an explosion in the amount of genomic data being generated. The data produced by a single state-of-the-art experiment can hardly be processed on a single PC workstation anymore. Handling such vast amounts of genomic data poses a lot of algorithmic challenges, which are progressively being addressed by the extensive research currently going on in the field of bioinformatics. The most obvious practical problems are related to data storage and transfer. They effectively hamper efficient processing, analysis and sharing of the data, and generate significant costs for an adequate IT infrastructure. Therefore, this work focuses on improving current approaches to processing, compressing and storing high-throughput sequencing data.

Sequencing a DNA molecule that represents a target genomic region typically yields a collection of reads, i.e., a set of substrings originating from the sequence of the region. Hence, one needs either to reconstruct the sequence of the DNA target molecule from these reads by performing *de novo* assembly, or to understand their original placement by mapping/aligning the reads to a known reference sequence. After that, depending on the goals of the study, the relevant sequence analysis is performed. For instance, this might involve assessing the potential differences between the

DNA molecule being studied and the corresponding molecule in the reference sequence. In practice, however, given the typical length produced by current short-read technologies and complex biological composition of some parts of the genomes, the input DNA molecule needs to be sequenced multiple times. This results in each experiment producing an output of millions of relatively short sequences each accompanied by auxiliary information. All this is usually stored in a text-based data format, and typically takes up from tens to hundreds of gigabytes of storage space.

Efficient processing and analysis of such large amounts of data clearly remains a challenging task. For practical reasons the data is usually stored in compressed form, most commonly using methods relying on general text-based compression algorithms such as gzip. Although they can reduce the data by up to 20–30% of its original size, these methods are unable to exploit the information models inherent in genomic and sequencing data, and hence the compression ratio they provide is insufficient. Aside from storage considerations, genomic data analysis workflows are also hampered by the current (and *de facto* standard) text-based file formats (such as FASTQ or SAM) in which information is stored. Such formats are cumbersome and do not allow for a rich representation of the data generated by either the ever improving sequencing technologies or novel bioinformatics tools. In fact, the magnitude of such problems tends to increase with the size and complexity of the experiment performed, placing the bottleneck of data analysis pipelines on data management, storage, and transfer.

This dissertation is composed of 4 chapters and is structured as follows.

In **Chapter 1** we provide a brief biological and technical background for this work. Firstly, we outline basic concepts in genomics and high-throughput sequencing technologies. Due to the fact that, at the moment, they are still used to generate the vast majority of genomic data, we primarily focus on second-generation short-read sequencing technologies. We then show the challenges facing the processing and storing of genomic data in a broader context. We describe a typical workflow for performing DNA re-sequencing data analysis, alongside the most commonly used data formats employed in the process. We put a particular emphasis on FASTQ and SAM formats, which are used to represent raw sequencing reads and reads aligned to a reference genome sequence, respectively. The

data stored in these formats usually take up most of the storage.

Going on, we outline a general data compression workflow and the steps it consists of. We briefly show the concepts used to compress text data, alongside the methods to model, encode and transform it.

In **Chapter 2** we describe the state-of-the-art methods, and the major challenges in compressing high-throughput sequencing data. As the data represented in FASTQ and SAM formats is composed of different kinds of information (such as DNA sequences, read identifiers, and base calling quality scores) we analyze different compression methods for each type of data. We first show approaches to compress data in FASTQ format followed by SAM format. Although the use-cases for these formats are different, the latter can be seen as a superset of the former, and the typical approaches to compressing data share a number of similarities and difficulties.

Then, we move on to outline alternative approaches to representing and storing genomic data. We show the limitations of the current formats and compression methods, and briefly show the challenges posed by the data generated by third-generation sequencing technologies.

Chapter 3 contains a compilation of the results of our research. As a starting point, we present DSRC2, a general purpose high-performance FASTQ files compressor. Compared to the most commonly used gzip, it achieves a significant reduction in the size of compressed data and offers high data processing speeds, which makes it a tool well suited for typical every-day usage.

Next, we focus on methods to compress short reads, exploiting the high sequence redundancy which is especially present in data coming from deep sequencing experiments. We illustrate our findings with ORCOM, a specialized DNA-only compressor.

Subsequently, we present FaStore, a complete compressor for the FASTQ format based on the methods of ORCOM and DSRC2. Compared to state-of-the-art methods it achieves the highest compression ratio so far. We also explore several lossy compression approaches to store base quality scores and read identifiers.

After that, and moving away from specialized format-specific compres-

sors, we present CARGO. CARGO is a general framework and data format which allows its users to semi-automatically generate compressors for any desired file format. All the user has to do is specify a record-based definition for the format, together with methods to parse records and to optionally perform data transformations on them. CARGO allows users to store multiple genomic datasets having heterogeneous formats in the same configurable container. As a proof-of-concept, we present a number of CARGO-based compression solutions. These allow the genomic data originally represented in FASTQ and SAM format to be stored in both a lossless and lossy way in CARGO containers. The results achieved are comparable to those obtained by the best methods so far available.

As a last step, we show a brief summary of lossless compression results, comparing our compression methods for FASTQ and SAM with the current state of the art.

Finally, in **Chapter 4** we provide a discussion of and outlook for the methods we have developed. First, we comment on the results of compressing high-throughput sequencing data both in a lossless and lossy way. We then elaborate on a strategy to integrate all our ideas into a single framework, showing other possible applications of our methods and future research directions they would open up.

The dissertation ends with an **Appendix**, which contains supplementary materials for Chapter 3.

CONTENTS

CHAPTER 1

INTRODUCTION

In this chapter we will show the main difficulties with representing, storing and managing the data generated using high-throughput sequencing technologies. Firstly, we will briefly explain the aim of DNA sequencing and give a short introduction to sequencing technologies from an historical point of view. We will explain important biological and technological concepts with their limitations. Then, we will show what the typical human re-sequencing data analysis workflow looks like. We will do this by briefly explaining the data processing steps, used data formats and the challenges facing the typical workflow from the data storage and management point of view. Then, we will give a brief introduction to data compression. We will explain the typical data compression workflow using concepts and methods commonly used in data compression. The information presented in this chapter will be used in Chapter 2 when describing the available approaches for storing and compressing high-throughput sequencing data.

1.1 High-throughput sequencing

1.1.1 The aim of DNA sequencing

DNA sequencing is the process of determining the sequence of the nucleotides that compose a DNA molecule. It is primarily used to discover the sequences of individual genes, larger genomic regions, full chromosomes or even entire genomes. Having access to the genetic code of a given organism allows us the possibility of better understanding the organism in question along with the underlying biological processes. Additionally, by using RNA sequencing, one can gain an insight into the gene expression

taking place inside the cell at any specific moment in time. The experiment can also be designed to focus on different scales – from sequencing an individual DNA molecule to a set of genes from one organism or even sequencing genomes of a whole population.

The whole genome sequencing of James Watson [207] and Craig Venter [107] assessed that humans share ~99.5% of the same genetic code. The differences present are known as *genetic variations* or just *variants*. The primary set of variants comes as a result of the recombination of genes during sexual reproduction involving germ cells. Another source of differences is genetic mutations. Mutations that occur during the sexual reproduction stage or during the normal replication of germ cells are transmitted to the offspring and are called *germline mutations*. On the other hand, the mutations that occur during a lifetime and which are not inheritable are called *somatic mutations*. Genetic variants which occur at frequency > 1% in the population are known as common variants. Yet, more than 95% of the rare ones are predicted to have a medical or biological consequence [197].

Thanks to sequencing, it is possible to discover a number of genetic variants in the human genome and to better understand their consequences. The presence of selected genetic variants can already be linked with phenotypes, e.g., how they influence the blood type [138], height [211], or scalp hair features [6]. In normal conditions, the presence of some of these may have no particular effect on the organism or even help when fighting diseases (e.g., a mutation present in one pair of genes can provide resistance to HIV virus [133]). However, some variants have already been associated with blood hypertension [106], type 2 diabetes [193], or Alzheimer disease [38]. For some some of them also a direct linkage has been found with different types of cancer [148, 203, 86].

Apart from providing essential contributions to medicine, sequencing has been applied to better understand not only humans but also their evolution and the history of different populations and migrations. [154, 200, 163, 140]. Sequencing has also been applied to improve genome engineering techniques in agriculture [202]. Moreover, discovering novel micro-organisms present in the environment through sequencing has also become possible, leading to the development of a new research field studying microbial communities – metagenomics [198]. Finally, sequencing

can offer more reliable individual identification methods to be applied in forensics sciences [214], as a result of the fact that no two humans have exactly the same genome, and some parts of it can be used as an efficient fingerprint.

1.1.2 DNA sequencing technology

Sanger sequencing

Since the discovery of the basic mechanisms of heredity and DNA structure, sequencing methods have always been of great interest. In the '70s, in parallel, Maxam with Gilbert [137] and Sanger [180], developed the first sequencing methods. However, as the former technique involved using hazardous chemicals it finally gave a win to the Sanger method, being the better and also a less complex process.

The Sanger sequencing method is based on the *DNA polymerase*, the enzyme used to synthesize DNA molecule from deoxynucleotides. At the beginning, the double-stranded DNA fragments need to be denaturated, i.e., heated up to split them into *template* and *complementary* strands. The obtained template strands are then divided into four separate reactions, each corresponding to the detection of a different nucleotide – either *A*, *C*, *G* or *T*. Next, DNA polymerase, primers, deoxynucleotides and dideoxynucleotides (performing as chain-terminating inhibitors of DNA polymerase) are added to each reaction in the following way. If the given reaction is to detect *A* nucleotides, it will contain *ddATP* dideoxynucleotides and the other remaining deoxunucleotites (i.e., *dCTP*, *dGTP* and *dTTP*). The primers or chain-terminating nucleotides are also previously radioactively tagged. In this way, the complementary strand will be synthesized by DNA polymerase by incorporating the present nucleotides to the template strand (stopping the reaction on encountering chain-terminating inhibitors). As a result, multiple complementary strands with different lengths will be synthesized per each reaction. Finally, the gel electrophoresis is applied to isolate and to analyze the synthesized DNA fragments by their length. The output is the X-ray of the gel (one example is presented in Fig. 1.1), which allows one to decode the DNA sequence. The initial Sanger sequencing method allowed for the first time for sequencing of a full genome, namely that of phi X 174 bacteriophage [179].

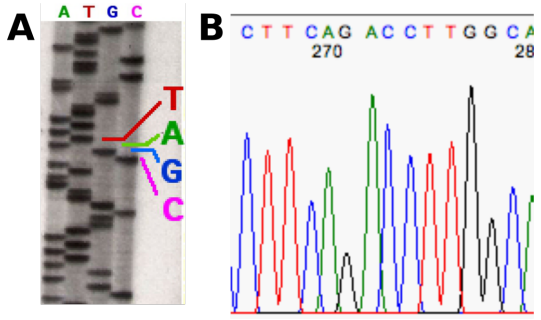

Figure 1.1: Decoding DNA sequence using as output: (A) autoradiograph from Sanger method (source: https://en.wikipedia.org/wiki/Sanger_sequencing), (B) chromatogram from dye-based automated Sanger method (source: https://seqcore.brcf.med.umich.edu/sites/default/files/html/interpret.html).

The initial Sanger method turned out to be very successful and made it possible to explore the genomes of microorganisms for the first time. It also went through a number of subsequent optimizations. The input DNA fragment could now be automatically cloned using bacteria to generate multiple copies of the same fragment. The average number of clones of each fragment is related to the *sequencing depth* or the *coverage*. Generally speaking, coverage refers to the number of times a single nucleobase at a specific position in the sample has been sequenced (or how many sequenced fragments it is covered by). However, one of the most important modifications to the original protocol was the introduction of nucleotide-specific dyes [188]. This allowed for the carrying out of a single reaction to detect DNA sequence instead of four, one per nucleotide. Such a modification also allowed for the detection of the nucleotides by using a fluorescent sensor (using a different wavelength per each nucleotide dye color) thereby replacing the X-ray step. The result of sequencing by this enhanced protocol is presented as a chromatogram, where each color peak represents a detected nucleotide – the process of assigning peaks to nucleotides is called *base calling*. Fig. 1.1 presents a comparison of the output image (X-ray autoradiograph) obtained by the classical Sanger method and the output (chromatogram) obtained by using the dye-based automated method. The first commercially available automated DNA sequencer was *ABI 370A DNA Sequencing System* introduced by Applied Biosystems (ABI) in 1986. The automation of the sequencing process allowed to carry out sequencing projects on a larger scale and lead to revolutionizing gene

discovery. The automated Sanger method is considered as the *first generation* sequencing technology and has been successfully applied in a variety of sequencing projects, most notably by providing a high quality human genome assembly [81, 201, 107].

The automated Sanger method, however, holds a number of limitations. First of all, it has a relatively low sequencing throughput due to a slow template preparation process and carrying on of the enzymatic reactions [144]. The chain-termination method also can be used only to sequence short DNA sequences, up to 1000 base pairs (bp). Longer sequences, e.g., chromosomes or genomes, need to be chopped into smaller fragments and sequenced independently. Such a sequencing process is called *shotgun sequencing* and the nucleotide sequences obtained from reading these small fragments are called *reads* (or *short reads*). Moreover, the resulting sequence reads have their average lengths r_{len} significantly shorter than the chopped input DNA molecule fragments of f_{len} length. Therefore, the initial molecule sequence needs to be reconstructed from these small reads using complex, computationally intensive methods in order to perform any meaningful analysis (as explained in more detail in Section 1.1.3). With these limitations in mind, the selection of a proper sequencing depth is one of the key considerations in biological analyses.

There exists a direct correlation between the required sequencing depth, the DNA molecule length, and the average read length offered by the sequencing instruments. As proposed by Eric Lander and Michael Waterman [98], the problem of the initial sequence reconstruction problem can be based on the mathematical covering problem, where the target is considered "sequenced" when an adequate coverage is achieved or when no gaps remain. Given randomly distributed (sequenced) reads of length r_{len} across a haploid genome of length G_{len} and the number N of reads generated (e.g., during a single sequencing run), the expected coverage C can be calculated as:

$$C = \frac{r_{\text{len}} N}{G_{\text{len}}} . \tag{1.1}$$

Moreover, the authors also state that a number of times n a nucleobase

has been sequenced follows a Poisson distribution and is defined as [79]:

$$P(x = n) = \frac{C^x e^{-C}}{x!} . \tag{1.2}$$

This sequencing model was used when designing The Human Genome Project [97], a large-scale international project to initially discover the DNA sequence of the human genome. Nonetheless, the model is quite simplified as it does not address some real problems, such as biases related to the composition of the genomes, its different ploidy levels, potential biases introduced in used chemistry, sequencing errors, or difficulties in computational sequence reconstruction from short reads in highly repetitive genomic regions (see Section 1.1.3). Therefore, the model underwent a number of improvements, which won't be covered here. A good overview of sequencing depth criteria in the context of different biological analyses can be found in [187].

Next-generation sequencing technologies

Since the introduction of the first commercially available DNA sequencer, a race began to increase sequencing throughput and decrease the sequencing time, while also reducing costs. Therefore, the initial automated Sanger sequencing method went through a number of changes and improvements.

The sample preparation steps have been simplified and further automated. The bacterial cloning step has been replaced with the generation of a sequencing *library*, a large collection of short genomic fragments from the input DNA molecule. Most importantly, during this step, platform-specific synthetic sequences (i.e., *adaptors*) are attached to the DNA fragments, which are later amplified by a polymerase chain reaction (PCR) creating many copies of them. The instruments perform a fully automated sequencing of multiple fragments in parallel in a repeatable manner, producing up to hundreds of millions of reads as a result. Hence, the techniques are called *high-throughput sequencing* (HTS), *second-generation sequencing* or *massively parallel sequencing* methods and were still formally seen a few years ago as the (future) *next-generation sequencing* (NGS) methods.

Due to the changes in the sequencing protocols and instruments themselves, the sequencing process as a whole (from an input DNA molecule

to a collection of sequenced reads in a digital form) is nowadays characterized by a set of different error probabilities, which have an impact on the final reported nucleobases. These include, for example, a probability of an error occurring during sample preparation or amplification steps (e.g., alteration of a nucleobases during PCR), the probability of a machine-specific error occurring during the sequencing a nucleobase (or set of), or the probability that during base-calling a nucleobase was called incorrectly. As a general feature, while sequencing throughput has been greatly improved, reads obtained with second-generation sequencing are much shorter than those produced by the Sanger method (see Table 1.1 for summary). As shown by Eqn. 1.1, when reducing the available read length more reads need now be generated if one wants to provide a coverage of the sequenced genomic region comparable to that offered by the automated Sanger method. Therefore, the great increase in sequencing throughput and the reduction in read size have shifted costs to the bioinformatics side, making analysis computationally more complex – as explained in detail in Section 1.1.3. Notably the automated Sanger sequencing method is still being used today, but primarily as a confirmatory protocol – typically to assess the results from shotgun-sequencing-based methods, but on selected relatively small DNA fragments up to a few kilobases (kb) in length.

To partially mitigate the problem of reads being shorter than those ones offered by Sanger sequencing methods, selected NGS platforms introduced the ability to read the DNA sequence from both ends of the library fragments. For a specially prepared library, the fragments can be sequenced either in *paired-end* or *mate-pair* mode (in addition to the standard *single-end* sequencing mode). The main difference between the modes (apart from the costs related to the sequencing protocol) is the expected length of the fragments, which is set up during the library preparation step. The length of the fragment, determines the distance between the sequenced reads from the fragment's at both ends, which is called *insert size*. With this in mind, paired-end libraries can typically sample a DNA region of ~300 – 500 bp. The mate-pair can sample a larger region of ~1.5 – 20 kilobases (kb) [129], which can give a better insight into possible complex structural information of the genome. The details of the chemistry used, the underlying reactions and the engineering methods of the DNA sequencing processes used by the instruments are specific to the platform. A summary of the main features of some selected past and current platforms

Table 1.1: Overview of selected sequencing platforms.

Sequencing platform	Read length (bp)	Sequencing throughput	Run time	Sequencing error rate
First-generation sequencing – Sanger sequencing				
ThermoFisher ABI 3730xl DNA Analyzer	400 – 900, up to 2k	1.9 – 84 kb	0.3 – 3 h	0.001%
Second-generation sequencing – massively parallel short reads sequencing				
ThermoFisher ABI SOLiD 5500xl	50 – 75 (SE) or 50 (PE)	160 Gb (SE) or 320 Gb (PE)	10 d	≤ 0.1%
BGISEQ-500 FCL	50 – 100 (SE/PE)	40 – 200 Gb	24 h	≤ 0.1%
Roche/454 GS FLX Titanium XL+	700 – 1000 (SE/PE)	700 Mb	23 h	1%
ThermoFisher Ion Torrent S5 530	200 – 400 (SE)	3 – 8 Gb	2.5 – 4 h	1%
Illumina MiSeq v3	75 – 300 (PE)	3.3 – 15 Gb	21 – 56 h	0.1%
Illumina HiSeq 2500 v4	36 (SE) or 50 – 125 (PE)	64 – 72 Gb (SE) or 180 – 550 Gb (PE)	29 h (SE) or 2.5 – 6 d (PE)	0.1%
Illumina HiSeq X	150 (PE)	800 – 900 Gb per flow cell	< 3 d	0.1%
Second-generation sequencing – massively parallel synthetic long reads sequencing				
Illumina Synthetic Long-Read	~100 k	Same as HiSeq 2500		
10X Genomics	Up to 100 k	Same as HiSeq 2500		
Third-generation sequencing – single molecule real-time long reads sequencing				
Pacific BioSciences Sequel	~10 – 15 k	5 – 10 Gb	4 h	13% or ≤ 1%*
Oxford NanoPore MK1 MinION	Up to 200 k	Up to 1.5 Gb	Up to 48 h	2 – 13%¶

Data based on own research, products' brochures and publications [62, 65]. The comparison excludes the prices of instruments and costs per sequenced Gb, as these are normally highly dependent on various factors including individual special discounts on machines, chemistry, bioinformatics infrastructure costs and staffing. *Depending on the sequencing mode: single pass or circular consensus read. ¶Using the newest R9 chemistry and depending on the sequencing mode: 1D or 2D.

is presented in Table 1.1. As the sequencing technology has been under intensive development in the last 25 years, here the focus is on the most important or influential ones available on the market – a more comprehensive technological review with some historic perspective and methods descriptions can be found in [128, 141, 62, 144, 130, 65].

Sequencing platforms can be divided into two main categories depending on the underlying sequencing method used, which is either *sequencing by ligation* or *sequencing by synthesis* – a detailed description of the differences between these sequencing methods and platforms can be found in [65, 141]. Applied Biosystems SOLiD (Sequencing by Oligonucleotide Ligation and Detection) or Complete Genomics (now BGISEQ) are the platforms based on the former method. These platforms provide very high sequencing throughput, however obtained reads are relatively very short, thus limiting their usefulness in further data analysis steps.

The Roche/454 FLX Pyrosequencer [131] introduced in 2004 was the first commercially available NGS platform based on a sequencing by synthesis method. Together with ThermoFischer IonTorrent platform they offer relatively longer reads, yet with also relatively small sequencing output. Solexa was another company to develop a technology based on sequencing by synthesis method, which was later bought by Illumina. Nowadays, Illumina offers a variety of different sequencing platforms depending on the resources and the needs. Their platforms account for generating the majority of the overall sequencing data today. Notably, in 2014 Illumina introduced the HiSeq X sequencing system[1], a set of either 10 (HiSeq X Ten) or 5 (HiSeq X Five) connected sequencers. It is based on previously successful HiSeq technology and is aimed at large-scale human whole genome sequencing experiments. HiSeq X Ten is able to provide > 1800 genomes per year when running full-time. The platform is also the first to break the milestone of 1000$ in amortized[2] cost per whole genome re-sequencing (i.e., sequencing an individual when the reference genome of th specie is available). Although Illumina platforms also produce reads of relatively short length, the robustness of the platform, low sequencing error rate and resulting wide adaptation made it somehow to be seen as a “standard”

[1] https://www.illumina.com/systems/hiseq-x-sequencing-system.html

[2] Amortized costs include: work and management costs, utilities and reagents, cost of sequencing instruments (amortized over three years), IT protocols cost and some indirect costs.

in terms of NGS technology. It is important to note, however, that the reported sequencing error rate is related to the sequencing platform, the chemistry used, and may vary depending on different experiment setup.

Long-reads sequencing technologies

While in many aspects second-generation technologies have already been broadly applied, other technologies, labeled as *third generation sequencing* (TGS) methods are rising and gaining a lot of positive attention. Most of these methods offer real-time sequencing, a feature not available in second-generation platforms. Their main advantage, however, lies in the ability to sequence a single DNA molecule at a time and without the need for the prior DNA fragmentation and clonal amplification. As previously noted, the most limiting factor of second-generation technologies is that they offer very short read lengths compared to the size of the whole input DNA molecule. Being able to produce very long reads, TGS technologies give rise to the possibility of performing an analysis of very complex and highly repetitive genomic regions, which are common and whose roles are important from a biological perspective. For example, in the human genome repeated DNA fragments make up to ~50% of the total genome [96]. Moreover, depending on the experiment, in order to achieve a reliable genomic coverage of the sequenced region, NGS technologies need to generate a large number of short reads (as denoted by Eqn. 1.1). On the other hand, the longer the reads, the possibly a smaller number of them needs to be generated in order to provide reliable results.

Pacific Biosciences was the first one to commercialize this new type of sequencing by introducing *single-molecule real time* (SMRT) sequencing technology [50]. SMRT was initially implemented in PacBio RS platform which was released in 2010. The technique uses a specialized flow cell with transparent bottom and which contains thousands of individual wells (called zero-mode waveguides (ZMWs)) with DNA polymerase attached to them [105]. Before sequencing, the DNA molecule needs to be specially prepared, by ligating hairpin adaptors to both ends of the input DNA molecule, creating as a result a circular *template* sequence. When sequencing, the DNA molecule passes through the stationary polymerase, emitting fluorescent light when nucleotides are incorporated into the second strand. In this way, the light sensor can focus on decoding the single

passing molecule. As the SMRT method uses a special circular template that allows each sequence to be read multiple times, the sequencing error rate is greatly reduced. By further reducing the error rate, operating costs and while simultaneously improving the sequencing throughput, PacBio technology is becoming a strong competitor in the current market dominated by the previous generation methods. *De novo* sequence reconstruction (sequence assembly, see Section 1.1.3 for description of the assembly process) is one of the most successful applications of PacBio sequencing. It has already been applied to a variety of projects, thereby improving existing genomes [17] obtained previously by NGS or Sanger technologies. A notable contribution is a significant update of human genome assembly [17, 26], closing more than 50 existing gaps and discovering new variants, especially the long structural ones. A more comprehensive review of successful PacBio application can be found in [169].

Another TGS technology is *nanopore sequencing* [31], developed and commercialized by Oxford Nanopore Technologies (ONT) with its first platform – MinION. Unlike the SMRT technique, when reading the input DNA molecule the device can directly detect the DNA nucleotide sequence from a native molecule without needing any special chemistry, light-detection methods or secondary signals. While performing sequencing, the template sequence passes through a very small protein pore, which results in detectable voltage changes. Those continuously sampled differences in voltage are then interpreted as possibly different short nucleotide subsequences. As the detection process can be quite error-prone (especially when reading long homopolymers [65], i.e., sequences of identical nucleobases) before sequencing a hairpin is ligated into both ends of input DNA molecule allowing it to be read from both strands (similarly to what happens in SMRT). In this way, a sequence can be read both from the forward and reverse strand. If the sequence has been read only from one strand it is called a *1D* sequence. If the sequence has been read from the second strand too, it can be later aligned to the one read from the first one, creating a more precise *2D* sequence. Similar to PacBio, ONT technology can be especially useful in *de novo* assembly [120] or metagenomics [67] experiments. Notably, thanks to the high portability of the MinION sequencer (being of a the size of a USB pendrive) and non-complicated sequencing process, it has been successfully tested live in analyzing the recent Ebola outbreak in Africa [165]. The results of the analysis were

available in less than 24h after receiving the Ebola-positive sample and with sequencing process taking from 15 min to 1h. ONT technology is still very young, yet it seems very promising.

There also exists another approach for generating long reads while is not formally considered as part of third-generation sequencing. This is a hybrid approach, based on NGS technology and categorized as *synthetic long reads* technology. It gives a possibility to create relatively long reads by using additional steps during library preparation, namely *barcoding*. While creating a library, the input DNA molecule is partitioned into relatively long chunks (~10 kb). Next, these fragments are put into separate wells, each corresponding to a different barcode. The fragments are then further fragmented into shorter ones, as when using standard NGS protocols. Finally, the well's barcode is added to the fragmented short sequences. Such obtained reads can be later sequenced using the existing NGS platforms. The barcoding information is then used during local assembly steps to link the reads coming from the same, larger fragment. The synthetic long reads technology have been already implemented in Illumina TruSeq library preparation kits[3] and in 10X Genomics solutions[4]. Although being a relatively young technology, already some sample metagenomics experiments discovering human microbiome diversity [94] have proven the usefulness of this approach. There are some downsides, however, as such techniques can have troubles when sequencing highly-repetitive genomic regions. Therefore, they can be considered as an intermediate technology between the second- and the third-generation sequencing technologies.

The perespective

The Human Genome Project [97] was a large-scale project that initially discovered the DNA sequence of the human genome. The project started in 1990, took 13 years, and consumed around 2.7 billion USD, being also the world's largest collaborative biological project. With the introduction of second-generation sequencing platforms in 2004, the cost of resequencing of human genome was reduced to an estimated 20 million $ USD. Since then, it began to fall dramatically. Today, we have practically reached the symbolic barrier of 1000$ USD in amortized costs for

[3] http://www.illumina.com/products/truseq-synthetic-long-read-kit.html
[4] http://www.10xgenomics.com/technology/

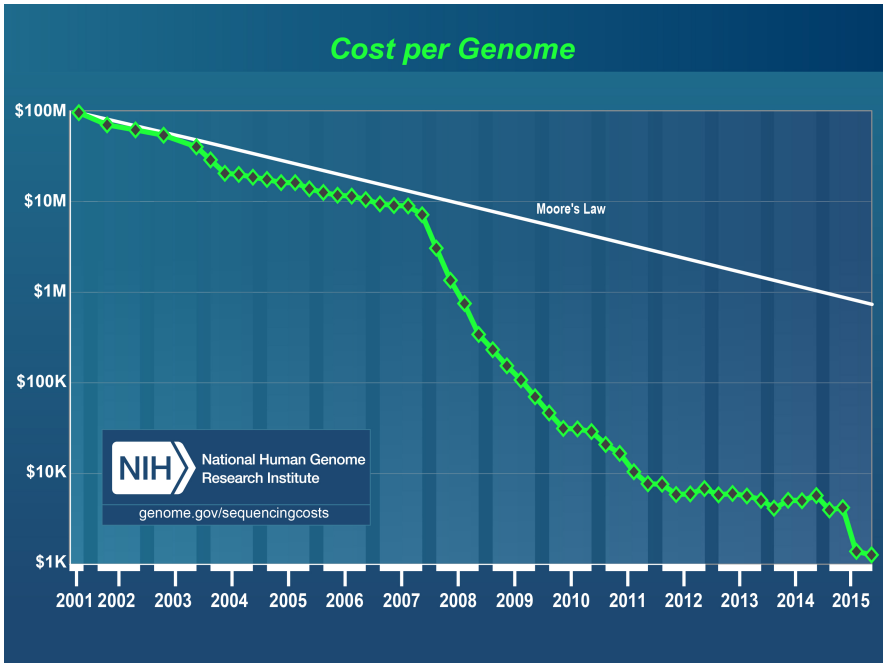

Figure 1.2: Costs reduction in time of human genome sequencing. Source: https://www.genome.gov/27565109/the-cost-of-sequencing-a-human-genome/

re-sequencing the genome of an individual. The reduction of sequencing cost is visualized in Fig. 1.2.

Thanks to advancements in sequencing technology, a variety of important large-scale studies, comprising thousands of individuals, have been accomplished. Of the highest importance are human population studies trying to discover genetic diversity and links to possible genetic disorders. The most famous one – the 1000 Genome Project [1, 2] was a multi-phase project, launched in 2008 with the aim of discovering the most common human genetic variants (with frequency > 1%). In its final phase [3, 192], the project covered sequencing of 2504 individuals from 26 populations identifying 88 million variants. The project provides a solid database for verification of variants obtained in DNA sequencing analysis projects, covering > 99% of single-nucleotide polymorphisms variants with frequency > 1%. Another notable large-scale sequencing project was led by the Exome Aggregation Consortium (ExAC) [104] performing exome analysis of 60,706 human individuals of diverse ancestries. The compiled catalogue of genetic diversity contains an average of one variant every eight bases of

exome, forming a comprehensive database for mutation verifications. On the other hand, the Pan-Cancer Analysis [205] project, a large-scale international project being led by The Cancer Genome Atlas (TCGA) Research Network and International Cancer Genome Consortium (ICGC), focuses on the extensive sequencing and analysis of human tumors with the aim of understanding the underlying genetics of cancer.

Apart from international and world-scale projects some nation-wide projects also turned out to be important. For example, whole-exome sequencing of 2000 Danish individuals has helped to better understand the role of rare coding variants in type 2 diabetes [119]. Similarly, whole-genome sequencing of 250 trios from a Dutch population performed by the Genome of the Netherlands (GoNL) Project [61] gave better insights into Dutch population genetic diversity, population structure and provided a look into the history of migration. On the other hand, the recent UK10k Project [199] performed whole-exome and genome sequencing and analysis of 10,000 UK individuals with a focus on rare and low-frequency variants linked with diseases.

Such large-scale studies are critical for our understanding of the genetic diversity of human populations and to our understanding of various diseases. Since DNA sequencing is one of the fundamental bases of precision medicine [10], re-sequencing experiments will, hopefully, become a commodity one day. Moreover, some scientists predict that by 2019 the genetic mapping of babies at birth will become a common approach [73]. Still, however, there are multiple challenges to solve, especially in the efficient analysis and storage of large amounts of sequencing data from various experiments.

1.1.3 DNA re-sequencing data processing

Goals and difficulties

Since two individuals of the same species share nearly identical genome sequence (in the case of human ~99.5% identity [107]), performing a comparison of one's DNA sequence (or its fragments) with an established and known universal sequence template for the species can give a better insight into one's underlying biological characteristics and functions. Hence, the goal of human DNA re-sequencing experiments is to explore the genetic

differences found in individuals, families or populations, particularly with respect to human genetic diseases [187]. This is usually done by comparing an individual's DNA sequence (or parts of it) with a reference sequence to find the possible differences. This follows further comprehensive validation steps and, in case of a study comprising multiple individuals, combined analyses. The procedure should ideally reveal single-nucleotide variants (SNVs or SNPs[5]), small insertions or deletions (INDELs), larger structural variants (SVs) and copy-number variants (CNVs), finally leading to meaningful clinical results. In the context of precision medicine, understanding the patient's underlying genetic variants is crucial to providing optimal treatment for many complex diseases [10].

As the design of a study depends on the biological hypothesis in question, different sequencing strategies and protocols can be used. There are three most common scenarios for human genetic analyses [155] – identification of inheritable, Mendelian disorders (germline mutations), identification of genetic mutations appearing in cancer cells (somatic mutations), and identification of candidate genes in complex diseases for further studies. Other, less frequent studies involving re-sequencing may include genome-wide association studies aiming to explore genome-wide sets of variants across different individuals and recreational or genealogical studies of families. On the other hand, two of the most popular sequencing strategies are whole exome sequencing (WEX) and whole genome sequencing (WGS). The difference between these two is that the former concentrates only on capturing and sequencing the genomic coding regions, being only a small fraction (~1% [155]) of the whole genome. Therefore, depending on the study, WEX experiments obtain a higher sequencing depth, while being up to an order of magnitude less expensive than WGS. The recommended sequencing coverage depends on the study, with the $30-40\times$ coverage for WGS being considered as a "standard", whereas for WEX it is $100-200\times$, with $200\times$ being the recommended one for clinical applications [187, 35]. Moreover, for studies analyzing somatic mutations the required coverage for both WGS and WEX needs to be even higher in order to be able to assess the low-frequency tumor-only variants [68].

Having multiple sequencing platforms available on the market with differ-

[5]Technically, SNP is a SNV, when it is considered as a common variant present inside a population. SNV is used as a general term to denote a single-nucleotide variant.

ent characteristics, selecting one that is suitable for the needs of the experiment is already a difficult task. Despite the choice, modern sequencers generally output massive amounts of short DNA sequences, coming from the fragmented input DNA molecule. This molecule, in the end, needs to be reconstructed in order to perform a meaningful analysis. The length of the generated reads is relatively small (tens to thousands of base pairs) in comparison to the genomic features being studied (which can span tens of thousands to billions of base pairs [145]). This is one of the biggest technological limitations, which greatly complicates the variant discovery process. With the variety of sequencing platforms, each having different sequencing process characteristics, different sequencing strategies and sequencing protocols, a variety of bioinformatic data processing tools have been developed to aid genomic analysis. Therefore, the current bottleneck of sequencing experiments lies in the sophisticated computational data analysis and data management [182].

From raw sequenced reads to genomic variants

In order to obtain meaningful clinical information from the sequenced biological sample, a series of data cleanup, processing and evaluation stages are necessary, forming together a pipeline. Figure 1.3 presents a generalized DNA re-sequencing data processing and analysis pipeline with each stage shown with corresponding utilized data formats. The most challenging part of the pipeline, both from the perspective of computational complexity and data management, is the data processing stage. Our focus, hence, is on the data processing stage and the utilized data formats (the formats are described in detail at the end of this section).

After completing the sample preparation stage and sequencing in the laboratory, massive amounts of digitalized raw biological data are produced in the process (which, due to current technological limitations, is not a completely lossless one). Large collections of short DNA reads are finally stored in FASTQ [34] file format. Single or multiple FASTQ files can be generated per one experiment. For a sample human re-sequencing experiment the size of generated FASTQ files can be about 20 GB (WEX, ~100× coverage) or around 200 GB (WGS, ~30× coverage). As the initial raw reads generated by the instrument can contain errors, quality assessment of the produced data needs to be performed.

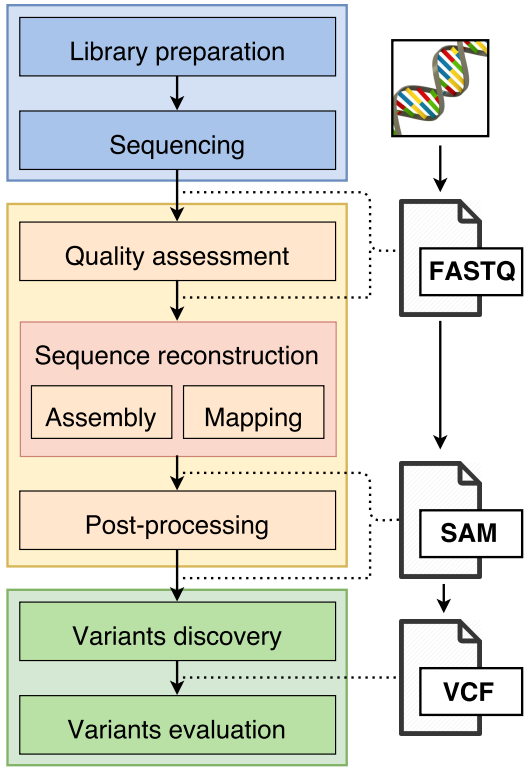

Figure 1.3: General concept of DNA re-sequencing pipeline.

At the next stage, the input DNA molecule sequence is reconstructed from the short reads. This reconstruction, so far, is the most computationally expensive data processing step. In the re-sequencing pipelines, the most common way to achieve this is to map the short reads to the studied organisms universal reference sequence, which is used as a template for reconstruction. The result of the mapping process – a collection of alignments typically stored in human-readable text Sequence Alignment / Map (SAM) [111] format. The generated files occupy more space than the initial or pre-processed FASTQ files. The example SAM files can occupy ~30 GB (WEX) or ~300 GB (WGS), and their size can still grow during the alignments post-processing stage.

The initial alignments may still require optional quality assessment, cleaning up and post-processing. After that, one can proceed to the variant discovery and evaluation stage. Depending on the biological hypothesis in question and experiment design, a different variant discovery strategy may be used. The summary of variant discovery results is stored in Variant Call

Format [39] (VCF) files. In this final stage, the resulting files can occupy ~30 MB (WEX) or ~6.5 GB (WGS), being only a small fraction of the size of the input raw FASTQ reads files or (temporary) SAM alignments files.

Since the introduction of the first massively parallel sequencing platforms it took a significant amount of research effort until the first reliable genomic data processing pipelines could be established, and became able to provide high-quality and reproducible results. However, the exact design of processing stages in the pipeline depends highly on the variant caller and the study. Currently, the most commonly used pipelines are the ones based on application suites like The Genome Analysis Toolkit (GATK [44, 139]) and SAMtools [111]. The recommended pipelines guidelines to follow are described in *GATK best practices*[6] [11] and Samtools workflows[7]. SAMtools (with SAM and VCF formats) and GATK have been successfully applied to the analysis of the 1000 Genomes Project [1, 2] and since then the tools have been seen as a *de facto* standard, which is being constantly improved. Notable to mention, a lot of research has been recently put into the development of fully integrated pipelines, which are trying to overcome the problems and limitations of the existing ones, and provide high-performance data processing protocols and scalability [166, 122, 40].

Quality assessment

Quality assessment is the first and crucial stage, as raw data generated by sequencing platforms can contain different artifacts, which are difficult to trace or can cause problems at further pipeline stages [155]. These could have been introduced before or during library preparation and amplification step (e.g., adapter contaminant sequences) or during the sequencing run itself. Moreover, some nucleic acid sequences are known to raise error rates for most of the sequencing platforms (which otherwise are characterized by additional specific sequencing error profiles). These include, e.g., regions with very high GC-content (an overall fraction of 'G' and 'C' nucleobases present in a given context) or long homopolymer sequences. One also usually observes the decay of base signal along the read [95]. It has a direct impact on base calling performed by the machine, where values specifying a low certainty of the called base are more probable to appear

[6] http://www.broadinstitute.org/gatk/guide/best-practices
[7] http://www.htslib.org/workflow/

Table 1.2: Sample values of Phred quality scores and their corresponding error probabilities.

Phred Q-score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

at the ends of the sequenced reads. These reported values are known as *quality scores*.

Given the probability P that the base was called incorrectly, the quality score Q of a called base is logarithmically related to the P reported in Phred scale [54]:

$$Q = -10\log_{10} P\,. \tag{1.3}$$

Table 1.2 presents sample quality scores on Phred scale with their corresponding base calling probability error rate.

The common strategy applied during the quality assessment stage is filtering and trimming of the raw reads. The quality score of each nucleobase present in reads is evaluated and either the reads not meeting required standards are removed, or the sequence (with its base quality scores) is trimmed accordingly at its end. To automate the quality assessment process, a variety of standalone tools have been developed for different sequencing platforms. Some, in addition to implementing the basic reads preprocessing just mentioned, add experiment report generation with rich statistics visualization features. This step can also give a preliminary insight into the quality of the sequencing experiment and can reveal more information about the potential problems that occurred during the sequencing itself (e.g., sample contamination or sequencing errors deduced from meta-data). Moreover, the information obtained can be useful to better adjust further analysis stages in the data processing pipeline. The most commonly used tools for raw sequencing reads quality assessment are FastQC[8], FASTX Toolkit[9], PRINSEQ [183], and NGS QC Toolkit [157]. A

[8] http://www.bioinformatics.babraham.ac.uk/projects/fastqc/
[9] http://hannonlab.cshl.edu/fastx_toolkit/

more comprehensive overview of the quality assessment stage with the solutions used can be found in [155].

Apart from the standard reads filtering and trimming strategies, there also exist more complex approaches. These focus on correcting possible sequencing errors found in the reads in order to improve accuracy of the further sequence reconstruction and/or variant discovery stage. The selection of an error correction method is highly linked with the sequencing platform, the error profile and the laboratory protocols used, hence a large variety of tools have been developed. A good overview of algorithms and tools used for error correction within the context of supported sequencing platforms can be found in [95] with their comparisons available in [213]. However, in a DNA re-sequencing pipeline they are not commonly used, due to lack of systematic and comprehensive benchmarks, and for practical performance reasons. In the case of *de novo* assembly most users also rely on built-in error handling / correction approaches already implemented in the assemblers [95].

Sequence reconstruction

Since most platforms are unable to provide the full sequences of the supplied input molecules, a computational reconstruction is required. In general, two main classes of sequence reconstruction methods are available – *de novo* assembly and sequence mapping. The former is used to assembly the reads into (possibly) full-length sequence(s) from scratch. The latter is used to link the reads by aligning them to an existing reference template sequence. Therefore, broadly speaking, *de novo* assembly can be used to discover the unknown genome sequence of a given organism. Then, this sequence can be used as a reference when performing sequence mapping in further re-sequencing experiments.

The goal of *de novo* assembly is to build the longest sequence(s) from the supplied short reads, which will hopefully represent the analyzed genomic fragment. In general, assembly approaches rely on a simple assumption that DNA fragments sharing high similarity originate from the same genomic region [145]. By this principle, similar short reads are grouped together creating larger contiguous sequences – *contigs*. During this step, to achieve a sufficient overlap between the reads, a significant depth of sequencing is desirable – it also depends on the available reads length.

The longer the reads, the possibly better overlappings can be found between the fragments, thus lowering the required sequencing coverage. After the initial assembly step and using some auxiliary information (e.g., paired-end, mate-pair reads, or very long reads from the third generation sequencing technologies), contigs are joined together to create even longer sequences – *scaffolds*. Due to problems with assembling difficult genomic regions (especially the highly repetitive ones), not enough coverage or sequencing errors, there may appear gaps inside such created scaffolds. Finally, the scaffolds are put together into linkage groups or – when assembling whole chromosomes or genomes – they are placed on the chromosomes. The process of assembling the genomes can be further aided with providing auxiliary information for improving the linkage between contigs. This information may include, e.g., DNA sequences from already pre-assembled genomes of related species [51] or genome-wide maps of DNA sequence fingerprints (known also as *optical mapping* [194]). The final result of the assembly process is stored in (or converted to) multiple FASTA files, representing scaffolds or chromosomes. As a side note, the official currently available human genome reference sequence[10], version 38, contains multiple sequences, not only limited to chromosome sequences or mitochondrial DNA. It also contains a number of auxiliary unplaced contig sequences, since the human genome contains a lot of repetitive and difficult to assess regions, especially around centromeres. A comprehensive review of *de novo* sequence assembly challenges, main algorithmic approaches and tools can be found in [51, 145] with comparisons and benchmarks in [178, 21, 146].

In NGS re-sequencing experiments, however, the other type of initial sequence reconstruction is preferred, namely sequence mapping. As two individuals of the same specie share a nearly identical genome sequence, a previously assembled genome sequence can be used to aid this re-assembly process instead of reconstructing the input molecule sequence directly from scratch. Hence, in the read mapping procedure, the short reads are aligned to the provided reference sequence, with the goal to track their locations. Since generated reads contain sequencing errors and differences with respect to the reference (the part we want to assess), the mapping task boils down to solving an approximate string matching problem. This process is computationally far less expensive than *de novo*

[10] https://www.ncbi.nlm.nih.gov/grc/human

assembly. Hence, our focus is on re-sequencing experiments, as these are the most common.

To efficiently deal with the processing of the massive amounts of inaccurate input data, two main algorithmic ideas are used in modern mappers – *indexing* and *filtering* [168]. Indexing is primarily used to build a binary representation of the reference sequence in order to perform fast string matching, searching for the given string's origin(s). Optionally, an index can be built on the read itself to allow for faster access to its substrings. There are basically two categories of indexing methods [112] – *hash-table*-based and *suffix-tree*-based. The idea behind hash-table based methods is to keep the position(s) of each k-mer (i.e., any possible subsequence of length k) of the reference in a hash table with the k-mer being the key. Although naive implementations of hash tables only allowed for exact matches, for more sophisticated ones it is also possible to find inexact matches. Some modern mappers from this category are MAQ [113], mrFast [9] / mrsFast [69], and Novoalign[11].

Another class of indexing methods is based on a suffix tree, a data structure which is used to store all suffixes of a given string with their corresponding locations. Once constructed, the suffix tree allows for fast searches for exact matches of substrings, these being starting points for finding inexact ones. However, constructing and storing it in memory requires a significant amount of resources, hence optimized derivatives were developed, namely enhanced suffix array [5] and FM-index [55]. Indexing methods based on a FM-index gained the greatest popularity, mainly due to small memory footprint they have, while still providing satisfactory search capabilities (compared to methods based on suffix arrays, which are typically faster). In the end, inexact matching algorithms based on FM-index provide a good tradeoff between search time and memory usage and have been implemented by modern, commonly used mappers, namely: BWA-MEM [110, 108] (recommended by GATK Best Practices), Bowtie2 [101, 100], SOAP2 [115, 116], or GEM [127].

As for filtering approaches, they can quickly exclude large regions of the reference sequence where no satisfying matches can be encountered. The filters are typically based on one of two lemmas [168] – q-gram (or k-mer) lemma and pigeonhole lemma. Both lemmas define necessary conditions

[11] http://www.novocraft.com/

to find a candidate match for any given genomic position within a specified error level. In general, when using the k-mer lemma, for a given length of k, all (or some selected) overlapping k-mers of the read are extracted. Next, the positions in the reference matching these k-mers are located by using the preferred indexing method. The locations which contain the biggest number of shared k-mers matching the reference and which are within the specified maximum error level are considered as candidate matches.

The pigeonhole lemma describes a simpler approach. Given a possible maximum number of e mismatches, the lemma states that if the read is divided into $e + 1$ disjoint substrings (seeds), then at least one substring will match without errors [12]. The read is hence divided into (possibly) equal-sized non-overlapping seeds and for each the matches are searched in the reference sequence. Modern mappers, however, use the mixture of both lemmas, which further increases speed and/or precision of the matching [108, 127, 100, 9].

Having obtained the initial set of candidate matches, it is next validated using dynamic programming approaches. Validation is done by using an edit-weighted distance algorithm [212] such as Smith-Waterman [189] or Needleman-Wunsh [147] to assess the similarity between the read and the reference at each potential mapping position. The locations that pass the verification step (e.g., those having less than the specified maximum number of possible substitutions, insertions or deletions) indicate the possible set of final mapping locations. In many mappers, this verification step is the main bottleneck [168] hence further optimizations are greatly desirable. Due to the highly repetitive nature of DNA fragments in the genome, auxiliary data from paired-end or mate-pair (if available) is used to help with the validation procedure. In this way, the read composed of repetitive sequence(s) can be placed more precisely if its mate read was placed unambiguously. However, it may often happen that when mapping reads into complex genomic regions, multiple possible alignments will be reported.

Finally, reads with their mapping information (usually called alignments) are stored record-wise in (or converted to) SAM format [111]. At this processing stage, there are no requirements for the alignments to be sorted by position inside the output SAM file. As the sequence re-assembly stage is highly parallelizable, multiple FASTQ files coming from a single experi-

ment (e.g., produced by different sequencing lanes or an initial large file split into a number of smaller chunks) can be processed simultaneously, producing as an output the corresponding number of SAM files.

Alignment post-processing

The first step in the alignment post-processing stage is to merge all the mapping results into one (in the case of alignments spread into multiple SAM files) and to sort the alignments according to their calculated genomic position. In this way, the alignments can be accessed by their mapping position, which is essential for further data processing and analysis. For example, with the alignments sorted by their genomic position, alignment quality assessment can be performed and the obtained depth of coverage can be calculated from the data in order to compare the results with the experiment setup. The resulting file can be optionally split by chromosome, and post-processing with further variant discovery can be done independently in parallel [164]. Finally, the SAM files can be compressed into BAM format (which is a compressed SAM representation, explained in Section 2.2.2) and indexed, as the further post-processing and variant discovery tools usually support working with alignments stored in such form. As a side note, although the mapping process is highly parallelizable and the alignments can be processed independently chromosome-wise (split into a number of separate SAM files), merging and sorting all the initially generated alignment files is a global operation on all the data, and remains one of the data-intensive bottlenecks in the pipeline.

If PCR amplification was used during the library preparation process, a fraction of DNA molecules could have been duplicated and sequenced more times than expected, introducing some information bias into further analyses. There are also other possible cases in which reads could have been read and sequenced multiple times (e.g., optical duplicates coming from the same cluster location in the same flowcell) due to technical problems with the instrument. As these reads usually share the same genomic location and (possibly) the sequence, the goal is to find and mark (or remove) them. Hence, this step is called *marking duplicates* [11]. A number of auxiliary information, such as paired-end data, alignment direction, or sequencing depth, are used to determine potential duplicates.

Quite often, during the sequence mapping stage, ambiguous alignments

can be reported (e.g., reads mapping "well" to multiple locations – the so-called problem of multiple alignments), especially when reads map to complex genomic regions with a high degree of sequence repetitions. In such cases, reads with identical sequences mapped to the same genomic region may have different alignments reported with respect to the reference sequence, especially around potential INDELs. Hence, the *indel realignment* [11] procedure identifies the most suitable placement of the reads, analyzing all the reads within the same genomic region context and normalizing their alignments. In the case of standard workflows studying germline mutations, this step may have only a little effect when using high-coverage data [109]. Therefore, sometimes it is omitted, especially as it is computationally expensive. As a side note, when somatic mutation are studied this step usually entails a more expensive analysis, as both tumor and normal sample data must be jointly analyzed and realigned [181].

As mentioned for the quality assessment step, the quality scores reported by sequencing machines (during the base calling process) are prone to contain systematic (non-random) technical errors. Therefore, they may introduce some bias during the variant discovery stage, in which they are used as one of the sources of information. To enable integration of the data generated from different platforms, the quality scores need to be fine-tuned [44]. Hence, *base quality score recalibration* (BQSR) [11] is a process adjusting the quality scores using machine learning to model the errors empirically. As the performed method is computationally expensive and may have little impact on high quality high-coverage data [109], this step is not always applied. An interesting benchmark comparing the results of germline variant discovery using selected tools with and without applied prior indel realignment with BQSR step is presented in [161].

It is important to mention that in standard alignment post-processing workflows, some additional data may be attached to each original record after each step is performed. This data, including for instance, the original base quality score values before recalibration step is usually stored in SAM format optional fields[12] including. Therefore, the size of the final file is usually lager than the initial one generated directly from the alignment stage. In basic implementations of post-processing routines (e.g., following the GATK Best Practices), after each step a new, modified SAM (or BAM) file is

[12] http://samtools.github.io/hts-specs/

generated. Hence, alignment post-processing entails intensive data I/O usage and creates a significant number of temporary files.

Variant discovery and evaluation

Having had alignments cleaned-up and prepared as an analysis ready SAM (or BAM) file, the crucial task left is to identify the sites where the data displays variation relative to the reference genome [11]. That is also known as *variant calling*. The choice of applied strategies for genotype calling, somatic mutation identification or structural variants exploration is, however, highly related to the study design and there exists plenty of tools available to choose from. Tools for variant identification can be basically grouped into four categories [155] : germline callers, somatic callers, CNV callers, and SV callers. Nevertheless, as CNVs refers to intermediate-scale SVs [220] and SVs detection strategies can also be applied to detect CNVs, for simplicity we will consider them together.

The detection of germline mutations is central to finding the causes of common and rare diseases [155] and it is also the most frequently performed study. Variant discovery procedure focuses on assessing single nucleotide variants and very short insertions/deletions with respect to the reference sequence. The underlying variant detection methods are usually based on Bayesian inference, whereas to aid in the detection of INDELs local read re-assembly methods can be used [44, 170]. The most commonly used variant callers are GATK [44, 139] HaplotypeCaller (recommended by GATK Best Practices), FreeBayes [60], SAMtools [111] mpileup (alongside with BCFtools[13]), and Platypus [170], to name a few. A good overview of variant callers with their benchmarks can be found in [77, 109].

Cancer studies focus on somatic mutations calling by performing variants analysis of tumor/normal samples pairs from one subject [155]. In contrast to the germline mutations discovery, somatic mutations are much more difficult to detect. Depending on the type of tumor and its stage, it may not have uniform genome as a whole – the tumor may be composed of multiple different small tumors, with each clone containing different mutations. Therefore, the frequency of the variants present in the tumor will be low in the total population. Moreover, the alterations may also be present

[13] https://samtools.github.io/bcftools/

only in a small fraction of the DNA material originating from the specific genomic region [29]. Therefore, it is necessary to perform an analysis of germline variants in order to discover somatic ones. Hence, additional read analysis, filtering and processing approaches may be required during the pipeline post-processing stage. Some of the notable somatic variant callers are MuTect [29] (recommended by GATK Best Practices), Strelka [181], FreeBayes [60], VarScan 2 [90, 91], and Platypus [170]. A good overview of somatic variant callers with benchmarks is available in [53, 7].

The last types of study focuses on discovering large structural alterations inside the genome, as they impact phenotypic diversity and may play an important role in diseases [195]. These alterations, including large deletions, translocations, invertions, tandem duplications, copy-number variants, or novel insertions, can range in length from kilobases to large chromosomal-level alterations [8]. Due to limitations of NGS technologies (they can only produce reads which are short when compared to the genomic features being studied) these variants are the most difficult to detect. Reliable detection hence poses big algorithmic and computational challenges. In addition to the general alignment post processing stage, some variant callers may implement their own more sophisticated data processing steps or even perform alternative processing – e.g., full *de novo* assembly from short reads [117]. Notable tools for detecting SVs include: Pindel [215], Delly [167], BreakDancer [27], Lumpy [102], or Manta [28]. A good overview of SV detection challenges can be found in [8] and a comparison of tools with benchmarks can be found in [160, 220].

The raw results of variant calling are typically stored in VCF format. Since the obtained raw results will most probably contain some bias, they may need to be post-processed. This may include refinement of variants probability to fine-tune the balance between specificity and sensitivity [11]. This is proposed in the GATK Best Practices for germline variants discovery as a *variant quality score recalibration* (VQSR) stage. Furthermore, variants may now need to be filtered to narrow the range of interest or in order to eliminate false positives (by assessing the results using external resources, e.g., databases). In general, the final variant annotation and verification strategies are dependent on the experiment and study design.

```
>hg19_dna range=chr1:3548729-3548935
ACAGGACACAGTAAAGGGTGAGACAGCACCTGCGTCAGCACAACTGACCG
TTCCTTGTCGCCAGGAAGTAGCTGTCAGGACTAAATGCCAGCATTCCTAT
GCCGATTTTCGGGTTTGCTCTGTCGGTAACAGGTTTCAGTGTCTGTAAGG
AGACTGGGACAGAGGCGATCTCATCTAGAACACCAACAGGAAGAACACGC
CATTGTC
```

Figure 1.4: A sample FASTA format record. The sequence represents a genomic region on chromosome 1 starting from position 3548729 to 3548935 of the human genome reference sequence version *hg19*. The specially marked nucleobase **G** in line 4 resides on position $chr1:3548832$

Common data formats

The most common genomic data representation file formats used in DNA re-sequencing pipelines are FASTA, FASTQ, SAM, and VCF. All of them are represented in ASCII text-based, human-readable format. For practical reasons, the data physically are usually stored in their compressed form (explained in detail in Section 2.2.2). Generally speaking, the formats are defined by an optional header section (containing meta-information) followed by a records section. The biological information is represented in a record-wise manner, where single or multiple records can be stored in one file. The formats description, the relation between them, and their applications are as follows.

FASTA format FASTA format is used to store biological sequences. Fig. 1.4 shows a sample FASTA record. The record begins with an optional single-line sequence description (starting with '>' symbol; the content after the first space character is considered as an optional *comment*), followed by line(s) of sequence data – nucleotides or amino acids. The sequence is encoded in characters following the standardized IUPAC notation [36], including single letter codes with possible lowercase letters. Since the length of the sequence can vary from single bases to kilo- or mega-bases it can be broken up to span multiple lines. Usually, in DNA re-sequencing pipeline, the reference sequences of the analyzed organisms are stored in FASTA format. For example, when working with human data each reference chromosome can be stored in a separate FASTA file.

FASTQ format FASTQ format [34] is used to store biological sequences with their corresponding (base-calling) quality scores. Each record in FASTQ format is usually simply called as *read*. Fig. 1.5 shows a sample record.

```
@HWI-D00119:97514
AATGCCAGCATTCCTATGCCCATTTTCGGGTTTGCT
+
FFFFFFEEDEEDDEDDDDDDDDDDEEEDDDDDDDDD
```

Figure 1.5: A sample FASTQ format record. The specially marked nucleobase **C** in the line 2 resides on the 21-st position from the beginning of the read – has offset of 20 nucleobases. Snipper based on the exome sequencing data published by Genome in a Bottle (GIAB) [222] (source: ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/NA12878/Garvan_NA12878_HG001_HiSeq_Exome/)

The format can be seen as an extension of FASTA, but with some minor modifications. A FASTQ record is represented in file as 4 consecutive lines containing in the following order: read identifier, sequence, control line, and quality scores. Read identifier (also known as *title*; starting with '@' symbol) contains meta-information about the read, sequencing platform and/or other related to the used protocol. Since it is a free format field with no length limit, it allows arbitrary information or comments to be included [34]. Next is the sequence line, which is encoded as in FASTA format, but, in special contexts, can also include special gap characters [34]. The control line (starting with '+' symbol) can be left only as one character, reducing the file size, since historically it could contain repetition of the read identifier line. Finally – the quality line, contains sequence base-called quality scores in Phred scale (see Eqn. 1.3). The numeric values are mapped onto human-readable ASCII characters by adding a constant offset of 33[14]. In this way, a sample Phred value of 40 (see Table 1.2) can be represented as ASCII character of value 40 + 33, which is 'I'. FASTQ format allows to represent up to 93 distinct values of quality scores. Finally, the sequence and quality strings are of the same length and, in contrast to FASTA, are represented in single lines[15].

Important to mention, if the FASTQ reads originate from a library sequenced in paired-end (or mate-pair) configuration, then the paired reads originating from the same DNA fragment are usually stored in a set of 2 files. The pairing information between the reads is normally preserved in their read identifiers, i.e., the two reads share a significant common content of the identifier, differing only with some pair indicator (usually, a number '1' or '2' denoting the read number in the pair). Moreover, the

[14] Historically, old Solexa/Illumina sequencers were using 64 as an offset.
[15] Historically they could span multiple lines.

pairing between reads is also expressed in the way the file(s) are structured – the reads originating from the same DNA fragment reside in the same lines in two separate FASTQ files or – when the reads are stored in one file – they are stored one after another. Sometimes, however, some reads may have been sequenced unpaired (or filtered, removing one of the reads from the pair), and these are typically stored in an additional, separate file.

SAM format Sequence Alignment / Map (SAM) format [111] is used to store the results from the sequence reconstruction stage, particularly – sequence mapping. It is a tab-delimited text format consisting of an optional header section and records section. Fig. 1.6 presents a sample part of SAM file, which consists of 3 lines of header and 4 SAM records.

The header section (with each line starting with '@' symbol) contains meta-information about the alignments, including, e.g., information about the used references sequences, experiment read-groups, used mapper, or post-processing tools. In contrast to FASTA and FASTQ formats each record is stored in a single line for a better consistency in data representation.

Table. 1.3 presents an overview of the SAM record fields. Each record consists of 11 mandatory fields containing essential alignment information and variable number of optional fields for flexible or aligner-specific information. SAM format can represent both mapped (which are usually referred to as alignments) and unmapped reads, where the latter will have the mapping information missing and assigned a special indicator. In a very simplified way, a SAM record can be seen as an extension of FASTQ record with the mapping information, where fields *QNAME*, *SEQ*, and *QUAL* specify accordingly the FASTQ read identifier, the sequence and the base quality scores. Such a description is true for unmapped reads, but is more complicated for the aligned reads.

In SAM naming terms, the short DNA fragment (originating from the fragmented input DNA molecule), which is sequenced and later mapped to the reference sequence is called a *template*. If the library has been created in paired-end mode, two paired DNA sequences can originate from one DNA fragment and stored initially as two separate FASTQ records (one per each fragment's end). Each DNA sequence originating from one template is referred to as a *segment*.

```
@HD     VN:1.4  SO:coordinate
@SQ     SN:chr1 LN:249250621
@RG     ID:1    PL:Illumina
HWI-D00119:97514  99   chr1  3548812  60  36M  =  3548755  104  AATGCCAGCATTCCTATGCCCATTTTCGGGTTTGCT
  FFFFFFEEDEEDDEDDDDDDDDDEEEDDDDDDDDD  MD:Z:20G15  RG:Z:1  XG:i:0
HWI-D00119:88620  99   chr1  3548812  60  36M  =  3548779  129  AATGCCAGCATTCCTATGCCCATTTTCGGGTTTGCT
  FEFDEDCECCECDDCCCDCDAACDEDDBDBA@?B@C  MD:Z:20G15  RG:Z:1  XG:i:0
HWI-D00119:16648  99   chr1  3548812  60  36M  =  3548758  109  AATGCCAGCATTCCTATGCCGATTTTCGGGTTTGCT
  CFEFFFEEDDEDDDDDDCCCBBDBDDEDDDDDBBDC  MD:Z:36      RG:Z:1  XG:i:0
HWI-D00119:23358  163  chr1  3548814  60  34M  =  3548809  158  TGCCAGCATTCCTATGCCCATTTTCGGGTTTGCT
  HHHFFFFFDEEEEEEDDDDDDDDDEEEDDDDDDD     MD:Z:18G15  RG:Z:1  XG:i:0
```

Figure 1.6: A sample SAM file with selected header fragment and 4 records. The alignments are mapped to reference chromosome 1 (*RNAME = chr1*). Some of the alignments have been mapped with mismatches with respect to the reference sequence. The first of the alignments (line 4) represents the result of mapping the FASTQ record previously shown in Fig. 1.5 with one mismatch. The position on which the mismatch has been found in this read has an offset of 20 bases from the beginning of the mapping position *chr1:3548812*, it is *chr1:3548832*. The used reference sequence is referenced in the file header (line 2). Based on the exome sequencing data published by GIAB [222].

Table 1.3: A brief description of SAM record fields. Source: the official SAM format specification from: https://samtools.github.io/hts-specs/.

Col	Field	Type	Brief description
1	QNAME	String	Query template name
2	FLAG	Int	Bitwise flag
3	RNAME	String	Reference sequence name
4	POS	Int	1-based leftmost mapping position
5	MAPQ	Int	Mapping quality
6	CIGAR	String	CIGAR string
7	RNEXT	String	Ref. name of the mate/next read
8	PNEXT	Int	Position of the mate/next read
9	TLEN	Int	Observed template length
10	SEQ	String	Segment sequence
11	QUAL	String	ASCII of Phred-scaled base quality+33
12	OPT	String	A collection of tab-separated optional fields

What is important to note is the fact, that during the mapping process, in some special cases, the segments can be broken into multiple smaller ones each mapping to a different loci (e.g., chimeric reads). Therefore, one fragment can be composed of multiple smaller segments. Moreover, for some reads, which map to complex and difficult genomic regions, the mapper can report multiple possible mapping positions – these reads will be represented as *alternative alignments*. In such cases, the content of the input FASTQ record is duplicated and stored in multiple SAM records. A read, which is reported to fully map to only one position in the reference (with possible mismatches, trimming, etc.) is called a *linear alignment*. Therefore, the number of SAM records (including alignments and unmapped reads) after the mapping step can be larger than the number of the input FASTQ records.

Alignment information in SAM format is represented and stored in multiple fields. The *FLAG* field always contains information related to read properties and the mapping results, represented as a bitwise sum of different flags. If the read has been successfully mapped, *RNAME* specifies the reference sequence name, *POS* is the position, and *MAPQ* is the quality of the mapping. The *CIGAR* (Compact Idiosyncratic Gapped Alignment Report) field defines a relation between the resulting alignment sequence and the reference, by specifying a sequence of operations that transforms the former to the latter, but only with respect to the inserted, deleted or

clipped sequences. The fields *RNEXT* and *PNEXT* are related to the name and position of paired sequences (paired-end or mate-pair if present) and *TLEN* field corresponds to the observed fragment length. Some reads can also be unmapped (or having the paired read unmapped) and the corresponding fields are left empty (having value either *** or *0*). The optional fields follow the *TAG:TYPE:VALUE* format where *TAG* is a two-character string naming the field and *TYPE* is a predefined enumerator defining the format of the underlying *VALUE* data. There exists a number of custom tags used by a variety of data alignment and post-processing tools described in the Sequence Alignment / Map Optional Fields Specification[16]. Nonetheless, some fields can be also defined by the user and those are usually the most commonly used ones.

As mentioned previously, SAM is usually stored in a compressed form in Binary Alignment / Map format (BAM) format and, once sorted by position, it can be indexed for fast retrieval of alignments from a range of positions on the reference genome and which is an important feature used when performing variant calling. A more detailed description of the SAM format can be found in the initial publication [111] or in the official specification.

VCF format Variant Call Format (VCF) [39] is a format for storing DNA alteration information such as SNPs, INDELs, and SVs, together with their rich annotations. It is used to store genotyping information from preliminary variant calling results or summary results of variant analysis from a single re-sequencing experiment analysis or multiple joint results from population studies. Fig. 1.7 presents a sample part of VCF file, which consists of 5 lines of header and 4 SAM records.

A VCF file consists of header section and data (variants) section. The header contains an arbitrary number of meta-information lines (specific to the given dataset), each starting with characters "##", and a tab-delimited data field definition line, starting with a single '#' character. Each VCF record consists of a set of 8 mandatory fields containing information about the observed variant (or invariant) site in the analyzed population. If the variant information is unavailable, the field is left with '.' symbol. Optionally, a VCF file can contain information from multiple samples describing their genotypes.

[16] https://samtools.github.io/hts-specs/

```
##fileformat=VCFv4.1
##INFO=<ID=AC,Number=A,Type=Integer,Description="Allele count in genotypes">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##contig=<ID=chr1,length=249250621,assembly=hg19>
#CHROM  POS       ID          REF   ALT   QUAL      FILTER   INFO           FORMAT   NIST7035   NIST7086
chr1    3546935   rs2821007   G     A     43.17     .        AC=2;AF=1.00   GT:GQ    1/1:6      ./.
chr1    3548136   rs2760321   T     C     5136.20   .        AC=4;AF=1.00   GT:GQ    1/1:99     1/1:99
chr1    3548832   rs2760320   G     C     3820.44   .        AC=2;AF=0.50   GT:GQ    0/1:99     0/1:99
chr1    3552780   rs2821061   A     G     946.44    .        AC=2;AF=0.50   GT:GQ    0/1:99     0/1:99
```

Figure 1.7: A sample VCF file with selected header fragment and 4 records. The discovered variants on the chromosome 1 are shown, starting from line 6. The genotyping information (columns 9–11) is provided for two samples: *NIST7035 NIST7086*. For us, the most interesting variant was discovered on *chr1:3548832* site (line 8) which is a single nucleotide variant *G/C*. As seen in Fig. 1.6 this variant could have been suggested by the alignments, yet at this level the information was incomplete. Based on the exome sequencing data published by GIAB [222].

Table 1.4: A brief description of VCF mandatory record fields. Based on *Variant Call Format Specification*.

Col	Field	Type	Brief description
1	CHROM	String	An identifier from a reference genome or pointer to contig
2	POS	Int	The reference position
3	ID	String	Variant identifier
4	REF	String	Reference base(s)
5	ALT	String	Alternate base(s)
6	QUAL	Float	Phred-scaled quality score for the call
7	FILTER	String	Indicator about filtering status and result
8	INFO	String	Additional information, variant annotations
9	GFORMAT	String	Genotype information format, order and data description
10+	GINFO	String	Genotype information per sample

In Table 1.4 the record fields description is presented. Since our focus is primarily on files stored in FASTQ and SAM formats (due to their large storage requirements as compared to files stored in VCF format), the detailed format description is not covered – a complete description can be found in the official Variant Call Format Specification[17].

Similar to SAM, VCF files are usually stored in a compressed way as Binary VCF (BCF). The files are sorted by position and indexed for fast retrieval of variant information. A detailed description of the VCF format can be found in the initial publication [39] or in the most recent official Variant Call Format Specification.

1.2 Text data compression

1.2.1 The aim of data compression

One of the most natural ways of representing information for humans is human-readable text – a sequence of characters. Computers, on the other hand, store and process the information in binary representation. Therefore, any type of information needs to be *encoded*, where a code can be perceived of as a set of rules to convert a given type of information to its digital form and back and forth.

[17] https://samtools.github.io/hts-specs/

When encoding text, characters can be either encoded using ASCII, Unicode or other encoding standard and stored digitally using one or more bytes per character, depending on the selected standard. As has been shown in the previous section, the most commonly used data formats in genomics still use ASCII characters to represent the information, e.g., DNA sequences, sequence alignments or meta-information. The size of generated sequencing data is in orders of gigabytes and a possible reduction of its size is highly desirable.

Data compression is a set of techniques allowing one to reduce the data size, specifically – the number of bits required to store and/or transmit it. Some of the key benefits of compression include significant reduction of storage costs and more feasible sharing between individuals or research institutes. Compression can be lossless and lossy. In the former case, the decompressed data matches the original one. In the latter, the controlled loss of information can be applied, e.g., by lowering the data resolution (like downsampling the quality scores in genomic datasets) or discarding some auxiliary data, unnecessary from the point of view of further data analysis (like removing selected unnecessary optional fields from generated SAM alignments). When storing sequencing data we would be mainly interested in performing the lossless compression to be able to precisely assess important clinical information. In some cases, however, we might allow for a controlled degree of auxiliary data loss to further improve the compression factor.

As will be shown in the following subsections, there exists no "best" method to compress the data, neither using lossless nor lossy methods. Moreover, a method, which may perfectly fit to compress one type of data (e.g., sequencing-platform-specific raw image or signal data used in basecalling) may not provide satisfactory results when applied to an other type (e.g., sequencing data generated by basecaller). The selection of the compression method may also depend on the requirements of data processing and analysis pipeline – whether the priority should be on fast random access (with possible larger data size) or on maximum space savings (with possible slow access). The data compression can hence be perceived of as both an art and an optimization problem.

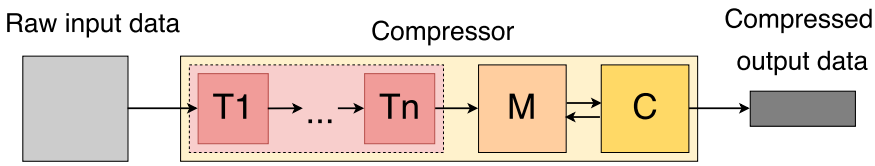

Figure 1.8: General concept of data compression workflow. $T1...Tn$ denotes optional data transformation steps, M denotes modeling stage and C – coding stage.

1.2.2 General data compression workflow

Any data compressor consists of at least a *model* and a *coder*. In order to efficiently encode the input message (a sequence of symbols) we need to build an appropriate statistical model of the source of information, which generates the messages. However, quite often the characteristics of the source are unknown and, therefore, the model is an approximate representation of the source. Such a model is later used by the encoder to determine the number of bits each symbol contributes to the compressed representation of the message. The more adequately the model describes the source, the better, as the encoder can encode the message more efficiently, i.e., using possibly the least number of bits. For example, the most frequent symbols in the message can be represented in some binary code using smaller number of bits than the less frequent symbols.

Moreover, optional data *transformation(s)* can be performed on the input message as a data preprocessing stage. The goal of data transformation (or multiple transformations chained together) is to cast the message into an alternative form, which can be encoded more efficiently using a different model. For example, if there are multiple repetitions of some sequence in the message, such a sequence can be encoded only once and its repetition can be specially encoded by pointing to its previous or initial occurrence. The general concept of compression workflow is illustrated in Fig. 1.8. The decompression can be perceived of as the reverse of the process.

As a side note – although, in a broader context, the data preprocessing stage can be perceived of as part of a modeling the source (we try to model some unknown characteristics of the information source casting the input message into a different form), we decided to explicitly distinguish between these two stages. After applying transformation(s) on the input message, the resulting message will have different characteristics, which

will be equivalent to being generated by a different source of information. Nonetheless, the boundaries between these two stages are not well defined.

1.2.3 Modeling

Source of information

A message (or multiple of), which we try to compress, was generated by some source of information, whose characteristics are usually unknown. The source can be seen as a stochastic process, characterized by a set of random variables. A stochastic process can be *stationary* or *non-stationary*. In the former case, its joint probability distribution does not change in time, in the latter – its current characteristics change in time. Typically, when compressing the data we assume that the message was generated by a stationary source and, if possible, we model a non-stationary source using a special case of a stationary one.

In order to encode the message efficiently, firstly we need to build a statistical model of the unknown source. Therefore, a model is an approximate representation of the source that generates the data. Its main purpose is to approximate a probability distribution of the (successive) symbol(s), which will be used by the coder, while encoding (or decoding). Multiple classes of sources of information exist, however, in this dissertation we will briefly focus on the most commonly used ones.

Memoryless source

The simplest class of source is the *memoryless source*. Let $\Sigma = \{a_1, a_2, ..., a_\sigma\}$ be the set of distinct symbols, called *alphabet* with σ specifying its size. Let $\Theta = \{\theta_1, \theta_2, ..., \theta_\sigma\}$ be a set of associated probabilities of occurrence of the symbols. A memoryless source generates as an output a sequence of randomly chosen symbols according to their probabilities. There are no dependencies between symbols and their probabilities, the only regularity in the generated message is that the frequency of occurrence of each symbol is close to Θ which defines the source characteristics [41].

Another one is the *piecewise stationary memoryless source*[18] (PSMS). When

[18] Formally, this type of source is a non-stationary source, as when generating messages

generating the i-th symbol of the message of length l ($1 \leq i \leq l$), the actual set of probabilities of occurrence of symbols depends on the position i in the generated message (the number of symbols generated so far). Given the sequence of probabilities of occurrence of symbols $\langle \Theta_1, \Theta_2, ..., \Theta_l \rangle$ and related sequence of positions $\langle t_1, t_2, ..., t_l \rangle$ in the message, the current state of the source is characterized by the set Θ_i.

For example, as mentioned in Section 1.1.3, the base quality scores reported by, e.g., Illumina basecaller may degrade with the length of the read being currently sequenced. The more DNA nucleobases that have been generated closer to the end of the read, the more probable low quality values will occur. Therefore, although simplified, in such cases a PSMS can be used to model the probability distribution of the quality scores depending on their position in the reads.

Finite-state machine source

Another very common class of sources is *finite-state machine sources* (FSM), with a *Markov source* being a generalized FSM. A Markov source is characterized by a set of states $S = \{s_1, s_2, ..., s_m\}$ and a number of conditional transitions between them with probabilities $\{p(s_i \rightarrow s_j)\}$. There can be at most σ transitions from one state to another. Each transition has some probability of being selected and it corresponds to generating a different symbol by the source. The next state is determined by the previous state and the current symbol.

Finite-order FSM sources are a subset of Markov sources in which every state is associated with a set of sequences of length no greater than k. Therefore, the current state is specified by the last k symbols, whereas the next state is specified by the current state and the current symbol. k is called the *order* of the source. The maximum number of possible states is σ^k.

As a side note, the previously mentioned memoryless source can be perceived of as a 0-order FSM model having a single state with corresponding σ transitions and their probabilities Θ.

---

the current source characteristics vary in time.

Context modeling

As already mentioned, usually, when compressing, the source characteristics are not known in advance and its model is being built while processing the generated message(s). The probabilities of occurrence of symbols are being estimated by analyzing the context in which they occur. The context of the analyzed symbol is usually the preceding k symbols, where k is the order of the model (and the length of the context). For a finite-order FSM source, a straightforward way to estimate some symbol s occurrence probability is to assign a number of times f to the symbol which occurred in the context c and to calculate its conditional relative frequency [74]. For a memoryless source (basically, order-0 FSM), there is no need to analyze the context in which the symbols occur.

There are two different approaches used to estimate the probabilities of occurrence of symbols while building the model. In the first one, the input message (or a part of it) is analyzed and, depending on the used model, the occurrences of symbols (in a context) are calculated. Based on them, an approximate representation of the probability distribution of the source's symbols is calculated, which will be used by the encoder. Such model is called *static*. Although simple, static models have some drawbacks. First of all, the message needs to processed twice – to gather the statistics for building the model and to encode the message (the model needs to be available before starting the encoding). Therefore, the model needs to be stored alongside the encoded message, where the amount of information representing the estimates of model's symbols' probabilities grows with the order of the model.

Alternatively, a model can "train" itself while processing the successive symbols of the message and can update the statistics. Such a model is called *adaptive* (or *dynamic*). The message can be encoded (or decoded) directly when data is being read and there is no need to store the model with the encoded message. However, the message may not be encoded as efficiently as when using a static model – when starting encoding, the model's probability estimates are empty or set to some initial arbitrary value. Therefore, some initial part of the message will possibly be encoded with overhead, before the dynamic model converges to the underlying characteristics of the source. However, by not requiring to store the model alongside the encoded message, overall the adaptive model will usually

perform better than the static one [132]. Hence, the most popular compressors use adaptive modeling.

A model representing FSM source can be either a fixed-order or variable-order model, with the former being the simplest one. When encoding a message using an adaptive fixed-order model, exactly k previous symbols are used as a context to predict the current symbol. However, the encoding efficiency degrades with the length of the used context, as many longer contexts appear for the first time and no prediction can be made [124]. One of the solutions is to collect statistics for different orders simultaneously and use the longest matching context to make a prediction of the symbol. A model performing symbols predictions using contexts of different lengths is called a *blended* model and is usually composed of multiple sub-models [74], for example, consisting up to k sub-models each with a different order $\leq k$. However, the challenge remains in encoding the symbols (and contexts) that occur in the input for the first time and how to indicate switching between the contexts.

The most popular family of compression methods using finite-order variable-length context modeling are *prediction by partial matching* (PPM) [32, 143] methods. When encoding the input message symbol by symbol, PPM firstly tries to find the longest match of k symbols. If no prediction can be performed, it continuously switches to a shorter context using $k-i$ previous symbols until a prediction can be made or until no matching symbols remain in context. Seeing the symbol (and the context) for the first time, PPM methods encode a special *escape* symbol, normally not present in the message's alphabet. This indicates the switching of the context to a shorter one. Next, it updates the models with the newly seen context. The way PPM methods handle the escape symbol leads to the development of a variety of PPM variants, where one of the most popular variants – PPMd [186] is currently widely used in several commonly used data archiving tools.

1.2.4 Entropy coding

Shannon entropy

Information entropy, a concept introduced by Shannon [185], provides a measure of the average amount of information generated by some source

of information, a stochastic process. It allows us to estimate the lower bound of the lossless compressibility of message(s), which are emitted by the source. Let X be a random symbol coming from alphabet $\chi = \{X_1, X_2, ..., X_n\}$, representing a message. The entropy $H(X)$ of the message can be estimated as:

$$H(X) = -\sum_{i=1}^{n} p(X_i) \log_b p(X_i) \,, \tag{1.4}$$

where $p(X_i)$ is the probability of X_i symbol occurrence and b is the logarithm unit (usually, a binary logarithm base $b = 2$ is used). If the symbol is emitted by a stationary memoryless source, characterized by generating symbols from alphabet $\Sigma = \{a_1, a_2, ..., a_\sigma\}$ with probabilities $\Theta = \{\theta_1, \theta_2, ..., \theta_\sigma\}$, then the entropy of the message corresponds also the entropy of the source. Analogously, $\chi \equiv \Sigma$ and $p(X_i) \equiv p(a_i) \equiv \theta_i$.

Let us assume a message $X = \langle x_1, x_2, ..., x_l \rangle$ representing a random sequence of characters and which is of length l. The x_j symbols are drawn from alphabet Σ. The probability of the a_i symbol occurrence at position j is $p(a_i, j)$. If the message is generated by a stationary memoryless source then $p(a_i, j) = \theta_i$. Therefore, the per-letter average entropy of such a message (and entropy of the source) can be calculated using Eqn. 1.4. Now, in order to estimate the minimum lossless representation of the message, the calculated entropy needs to be multiplied by its length.

As an example, let us consider a random sequence of characters (a string) $X = GCACTTTG$ generated by stationary memoryless source. Its length is $l = 8$ and its symbols are from alphabet $\Sigma = \{A, C, G, T\}$ of size $\sigma = 4$. The calculated relative frequencies of the symbols are: $p(A) = 0.125$, $p(C) = 0.25$, $p(G) = 0.25$, and $p(T) = 0.375$. The information entropy expressed in bits per symbol is $H(X) = -(0.125 \log_2 0.125 + 0.125 \log_2 0.125 + 0.25 \log_2 0.25 + 0.375 \log_2 0.375) \approx 1.781$. Hence, $lH \approx 14.248$ is the minimum number of bits in which the sequence can be represented (or $lH \approx 15$, rounded up to the nearest integer).

However, if the message is generated by a piecewise stationary memoryless source, the probabilities of occurrence of symbols depend on the position j in the message $p(a_i, j) = \theta_{i,j}$. Moreover, in case of stationary Markov sources, one needs to take into account conditional probabilities of occurrence of symbols, which depend on the occurrence of previous

symbols. For such sources, estimating the entropy is a bit more complex and it will not be covered here – a detailed description can be found in [41, 184, 84].

Basic prefix codes

A *prefix code* is a variable-size code which satisfies the *prefix property*, i.e., that there exist no sequence being a prefix of any other sequence (of bits or bytes) in the given code system. The sequences used in a code system are called *code words*. This property allows us to unambiguously store and retrieve code words from a binary information stream. One of the most simple and commonly used methods of assigning prefix codes are: the family of *Elias* codes (like *Elias gamma* code [52]), unary[19], and *Golomb-Rice* [64] codes.

In an unary coding scheme a positive integer x is represented as a series of $x-1$ values of one followed by a single zero value. Alternatively and less commonly used, the integer can be represented by a series of 0's followed by a single 1. The length of such a constructed code word for a given integer x is x bits.

An Elias gamma coding scheme represents the positive integer value x ($2^N \leq x < 2^{N+1}$) as a concatenation of two binary sequences. The value of N power is encoded using unary encoding. Next, the remaining $x-2^N$ difference is represented in binary encoding.

In contrast to the previous schemes, Golomb-Rice coding provides a flexible encoding method, which allows for creating parametrized prefix codes according to the source characteristics. For a given nonnegative integer x and a tunable parameter M, the encoding parameters are: $q = \left\lfloor \frac{x}{M} \right\rfloor$, $r = x - qM$ and $c = \lceil \log_2 M \rceil$, where q and r are the quotient and the remainder parts respectively. Having calculated the parameters, the code is constructed in two parts. The quotient value is represented using unary code. The remainder is encoded using *truncated binary encoding*, i.e., if $r < 2^c - M$ then binary encode value r using $c-1$ bits, otherwise binary encode value $2^c - M + r$ using c bits. In Table 1.5 sample Golomb-Rice code words are shown alongside binary, unary, and Elias gamma code words.

[19] Formally, unary codes are also known as *Elias alpha* codes, whereas binary codes of fixed length as *Elias beta* codes

Table 1.5: Sample of 10 code words for binary, unary, Elias gamma and Golomb-Rice coding. In Golomb-Rice coding $M = 10$ parameter was used for code word generation. The spaces between bits were added only to improve readability.

Number	Binary	Unary	Elias gamma	Golomb-Rice
1	1	0	1	0 001
2	10	1 0	0 1 0	0 010
3	11	11 0	0 1 1	0 011
4	100	111 0	00 1 00	0 100
5	101	1111 0	00 1 01	0 101
6	110	11111 0	00 1 10	0 1100
7	111	111111 0	00 1 11	0 1101
8	1000	1111111 0	000 1 000	0 1110
9	1001	11111111 0	000 1 001	0 1111
10	1011	111111111 0	000 1 010	10 000

As an example, to encode the message $X = GCACTTTG$ using unary encoding we could map its alphabet $\Sigma = \{A, C, G, T\}$ onto integer numbers with their corresponding code words as follows: $A \rightarrow 1$ (0), $C \rightarrow 2$ (10), $G \rightarrow 3$ (110) and $T \rightarrow 4$ (1110). As a result, the encoded message will be 110 10 0 10 1110 1110 1110 110 and of 23 bits in length. Compared to the theoretically possible information representation of the message, which is approximately 15 bits, such an encoded message contains a significant information overhead of ≈ 8 bits. To reduce the overhead, one of the solutions may be to perform a different mapping of symbols to code words. Knowing the relative frequency distributions of symbols inside the message we can assign the shorter codes to the symbols with the higher relative frequencies first, i.e., $T \rightarrow 1$ (0), $G \rightarrow 1$ (10), $C \rightarrow 2$ (110), $A \rightarrow 3$ (1110). As a result, the message will be encoded now as 10 110 1110 110 0 0 0 10 with length of 17 bits (≈ 2 bits of overhead compared with the theoretical limit).

Huffman coding

Huffman coding [76] is one of the most popular and widely used methods for data compression. For a given alphabet Σ with relative frequencies F of occurrence of symbols the algorithm can generate *optimal prefix codes* used to encode the symbols – the code is optimal if no other code with a lower mean codeword length exists.

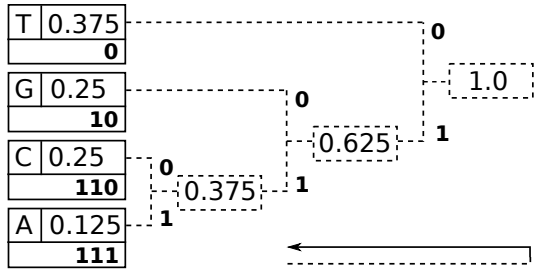

Figure 1.9: Illustration of generating Huffman code words for symbols to encode the sequence *GCACTTTG*.

It starts by creating a list of symbols sorted in descending order of their relative frequencies. Next, it constructs a binary tree with leaves representing each symbol going from traversing from the bottom to the top. Each time, while traversing the tree to the top, the two leaves (or nodes) with the lowest relative frequencies are merged together into a new parent node. Its relative frequency equals the sum of its children. This process continues until the last parent node is created – the root node, with the relative frequency of 1.0. The final code words are then assigned by traversing the tree from the root node to the leaves, appending 0 whenever a left child is selected and 1 – when right.

Fig. 1.9 illustrates how a Huffman tree is constructed for a sample message *GCACTTTG* to assign its code words. Each leaf represents the symbol, its relative frequency and the final deduced code word. The assigned code words for each symbol are: $A \rightarrow 111$, $C \rightarrow 110$, $G \rightarrow 10$ and $T \rightarrow 0$. As a result, the Huffman encoded message is 10 110 111 110 0 0 0 10 and is of 16 bits in length. In this case, Huffman encoding is more efficient than unary encoding (i.e., it uses less bits to represent the given message), yet, the encoded message still has an overhead of ≈ 1.75 bits compared to its theoretical entropy.

The presented algorithm describes the static Huffman coding algorithm, which uses precalculated relative frequencies of occurrence of symbols prior to encoding. However, to encode any message "on-the-fly" using adaptive modeling, there exists an adaptive version of Huffman coding algorithm [89]. The advantages and disadvantages between static and dynamic models have been previously discussed in Section 1.2.3. Although Huffman encoding is computationally a relatively non-complex and efficient algorithm, it has a strong limitation. As with other prefix-codes

generation methods, it generates a code with an integral number of bits assigned to each symbol in the given alphabet. For example, a symbol with probability 0.99 carries only $H(X) = -\log_2 0.99 \approx 0.0145$ bits of information, but using Huffman code it cannot be encoded in less than 1 bit. Therefore, the only case where it produces ideal variable-size codes (with average code size being close to the entropy) is for alphabets with probabilities of occurrence of symbols having a natural power of $\frac{1}{2}$. The maximum difference between the expected code length and the theoretical entropy denoted by Eqn. 1.4 is bounded by $p_m + \log \frac{2\log e}{e} \approx p_m + 0.086$ bits, where p_m is the probability of the occurrence of the most frequent symbol [59].

Arithmetic coding

Arithmetic coding [171, 99, 210], as with Huffman coding, encodes the message according to the relative frequencies of occurrence of symbols. However, in contrast to Huffman coding, it encodes the entire message as a single floating point number x, where $0.0 \leq x < 1.0$. In this way, the output symbols can have non-integral length codes possibly leading to a higher compression.

To encode a message, the method starts with setting an initial interval $[0.0, 1.0)$ (also called as *range*). Next, it divides the interval according to the relative frequency distribution of the symbols. Hence, each subinterval resembles the appearance of each symbol. After encoding the first symbol, the selected subinterval is then further divided according to the relative frequency distribution. In this way, the input message is encoded symbol by symbol, by narrowing the obtained subintervals. As a result, an interval encoding the whole message is obtained – any value within the final interval thus represents the input message in compressed form. The encoding is best shown visually – encoding of *GCACTTTG* sequence is presented in Fig. 1.10.

Unfortunately, available arithmetic systems in computers offer limited precision in representing numerical values. Encoding a floating point value representing the final interval with possibly unlimited precision is practically impossible. Therefore, while encoding the message, frequent renormalizations (expansions) and proportional reductions of the intervals are required to properly encode the numerical values [177]. A more

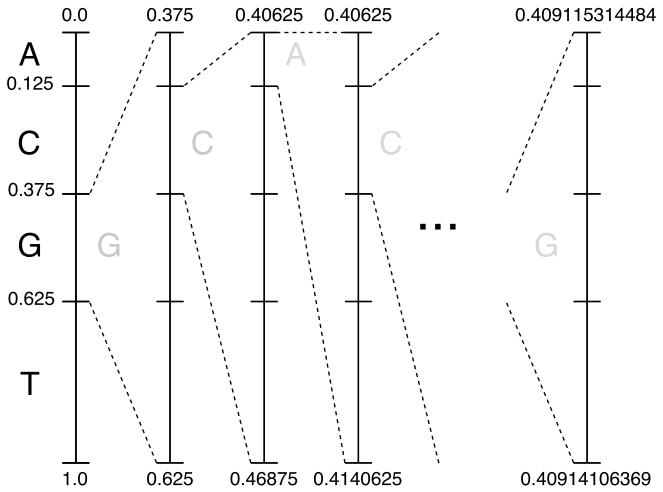

Figure 1.10: Illustration of arithmetic coding of *GCACTTTG* sequence. The result is an interval [0.409115314484, 0.40914106369).

comprehensive algorithm description with supporting examples showing both the encoding steps in detail and showing the problem of limited arithmetic precision can be found in [177].

The presented algorithm describes the arithmetic coding using a static model. Similarly, as in case of Huffman coding, an algorithm using an adaptive model is available. The strong point of arithmetic coding is that it can achieve optimal encoding performance. That is, with the length of the encoded message increasing to infinity, the number of bits required to represent the message converges to its theoretical entropy limit. This is why, the most successful compression methods are based on adaptive modeling with arithmetic coding. Some examples include family of PPM algorithms and PAQ[20] compressor series, where the former scores top positions in benchmarks and has won multiple awards, including the Hutter Prize[21] by achieving a maximum compression ratio among other available solutions.

Although arithmetic coding can achieve a higher compression ratio than Huffman coding, it unfortunately stems from higher computational complexity. The initial algorithm was computationally very expensive and

[20] http://mattmahoney.net/dc/zpaq.html
[21] http://prize.hutter1.net/

the algorithm was heavily patent-encumbered, so an alternative version was designed to partially overcome these issues. A *range encoder* [134] (although also formally an arithmetic encoder) reduces the number of necessary range renormalizations to encode the message, which significantly decreases the computational complexity. Implementations of range encoding algorithm can achieve speedup factor up to 2 compared to the implementations of standard arithmetic coding algorithm [177] (still with the compression performance speed inferior compared to Huffman encoding). Recently, however, a new family of entropy coders has been developed, namely *asymmetric numerical systems* (ANS) [47]. They provide encoding efficiency close to the one offered by arithmetic encoding, yet, with a compression performance speed comparable to (or better than [48]) Huffman encoding.

1.2.5 Data transformations

Basic data transformations

The goal behind the data transformation step is to cast the data into an alternative form, which can be encoded more efficiently, by using a better, but different model. Moreover, if applicable, some transformations can be chained together in order to improve the compression ratio even further. A proper choice of what data transformations are to be performed depends on the input data characteristics. However, it my not be a trivial task, especially if the characteristics of the source of information are unknown.

One of the simplest transformations is *differential encoding* (or *delta encoding*). Let $S = \langle x_1, x_2, ..., x_n \rangle$ be a message of length n, which consists of numerical values. When delta encoding S the algorithm emits only numerical differences between consecutive number $\Delta_i = x_i - x_{i-1}$. Only the first value needs to be encoded as original. Therefore, as a result it gives a tuple $(x_1, \langle \Delta_2, \Delta_3, ..., \Delta_n \rangle)$. This method noticeably reduces the entropy of the whole message, especially when the differences between consecutive numbers are small or when the numbers are represented in non-decreasing order.

As an example, let us encode a sequence of integers $\langle 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \rangle$ which, e.g., could be the read identifier number fields extracted from consecutive read identifiers in FASTQ file. Since in this example the

numbers are unique, each having the same probability of occurrence, the theoretical entropy (Eqn. 1.4) in bits per symbol is high and equal to 3.322. The theoretical minimum number of bits to represent the sequence is $3.322 \times 10 = 33.22 \approx 34$ bits. However, applying delta transformation would yield as an output a tuple $(11, \langle 1,1,1,1,1,1,1,1,1\rangle)$. Encoding all the values altogether, the theoretical entropy in average bits per symbol of the whole sequence is 0.469. Therefore, the sequence could be encoded in no less than $0.469 \times 10 = 4.69 \approx 5$ bits.

Another method, *run-length encoding* (RLE) [63] method encodes the consecutive symbol repetitions. While processing the input message, the number n of consecutive symbol s occurrences is encoded as a tuple (s,n). This method is useful especially in image compression (e.g., in JPEG format), where long runs of the exact pixel values can be represented in a more compact way. On the other hand, *move-to-front* (MTF; or *symbol ranking*) [16] method uses a control stack of recently used symbols when transforming the input message. In this way, each symbol is replaced by its index in the control stack. The main rationale behind applying the method is that the most recently seen symbol has the highest probability to occur. Both methods are best shown applied to an example, as discussed below.

Let us encode a DNA sequence $S = ACTACTACTACTACTACTTT$ (a short tandem repeat) of length 20 using alphabet to set of integers mapping $\{A,C,G,T\} \rightarrow \{0,1,2,3\}$. The theoretical information entropy of S in bits per symbol is 1.571 and the minimum number of bits to represent the sequence is $1.571 \times 20 = 31.419 \approx 32$ bits. By applying MTF with initial control stack $\langle A,C,G,T\rangle$, the sequence would be transformed to $S_{\text{bwt}} = \langle 0,1,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0\rangle$. The encoding process, step by step shows Figure 1.11. The entropy of the transformed sequence in bits per symbol is now 1.154 and the minimum number of bits to represent it is $1.154 \times 20 = 23.080 \approx 24$. Since the sequence contains runs of repeated symbols 2 and 0 it can be further transformed using RLE. The result will be a collection of tuples $\langle (0,1),(1,1),(3,1),(2,15),(0,2)\rangle$. If we split the result into two separate arrays containing indices R_{idx} and repetitions R_{num}, i.e., $R_{\text{idx}} = \langle 0,1,3,2,0\rangle$ and $R_{\text{num}} = \langle 1,1,1,15,2\rangle$, we now need to encode two short sequences of length 5. The R_{idx} sequence has an entropy of 1.922 in bits per symbol and can be represented using minimum $1.922 \times 5 = 9.61 \approx 10$ bits. Analogously, the R_{num} has an entropy of 1.371 bits per symbol and can be represented using minimum $1.371 \times 5 = 6.855 \approx 7$ bits. Combined,

A	ACGT	0
C	ACGT	1
T	CAGT	3
A	TCAG	2
C	ATCG	2
T	CATG	2
A	TCAG	2
	...	
C	ATCG	2
T	CATG	2
T	TCAG	0
T	TCAG	0

Figure 1.11: Encoding a sequence *ACTACTACTACTACTACTTT* using MTF. Columns denote respectively the current symbol to be encoded, the control stack buffer and the index of the current symbol in the stack.

the output sequence could be encoded using minimum 17 bits.

Lempel-Ziv coding

In the '70s Abraham Lempel and Jacob Ziv designed two variants of the most-widely used dictionary-based compression algorithms, namely LZ77 [221] and LZ78 [221]. When encoding, the algorithms try to find repeated subsequences in the input message and encode the repetitions with respect to the previously seen one, residing in the dictionary. The main difference between the methods, however, relies in defining the dictionary and the representation of the matches found.

The main idea behind LZ77-family methods is the use of a *sliding window* technique, which can be seen as "sliding" a buffer through the input message from its beginning to its end and encoding the repetitions of subsequences inside this buffer. This buffer keeps track of the previously-seen symbols (and sequences) and serves as a dictionary. Practically, this buffer is divided into two parts: *search buffer* (i.e., the dictionary) of size b_1 and *look-ahead buffer* of size b_2, containing the future part of the message to be encoded. The size of the buffer (and the length of the window) is $b = b_1 + b_2$. The encoding algorithm works as follows.

At the beginning, when the search buffer is empty, it is initialized with b_1

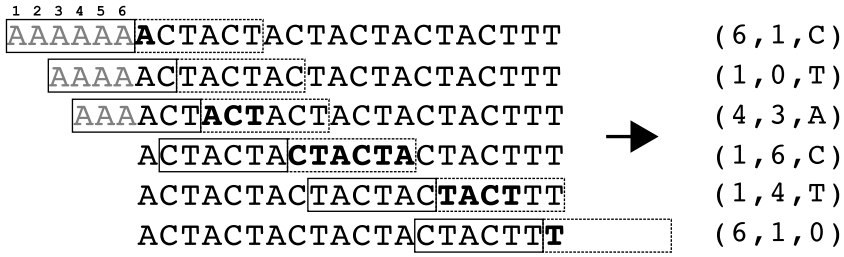

Figure 1.12: Illustration of encoding of a sequence *ACTACTACTACTACTACTTT* using LZ77 sliding window. Values on the right represent emitted triplets on each encoding step.

repetitions of the first symbol of the alphabet and b_2 repetitions of first symbol of the input message. Then, when processing the input message, it tries firstly to encode the longest prefix shared between the next part of the message in the look-ahead buffer (starting at $b_1 + 1$ position in the buffer) and the sequence present in the search buffer (starting to search from its end, i.e., b_1 position). Then, if a match is found, the subsequence can be encoded with respect to some previously-seen shared prefix. The match is encoded as a triplet (p, m, c), where p is the match starting position inside search buffer, m is the length of the substring match and c is the following character (possibly mismatch). The buffer is shifted $m + 1$ position to the left, filling with proceeding successive characters (the window is "slided"). Therefore, the encoding result is represented as a collection of triplets.

As an example, Fig. 1.12 illustrates a simplified encoding (using a sliding window technique) of a sequence *ACTACTACTACTACTACTTT*. As a side note, for simplicity, in step 3 the match has been encoded as $(4,3,A)$. However, some alternative algorithms may allow for a more efficient match representation, e.g., $(4,6,A)$ (or even $(4,15,T)$ if it is allowed to search beyond the look-ahead buffer if a match was already found). In such case, the length m of the found match is greater than the actual length of the matching substring *ACT*. Such an operation can be interpreted so as to generate m symbols using the matching substring, repeatedly starting from its beginning, when its length is exceeded.

The LZ78-family of compression algorithms, on the other hand, uses as a dictionary a list of previously encountered prefixes. The algorithm starts with the empty dictionary and fills it up while processing the input message. To encode a sequence at some position, it searches for the longest

prefix in the dictionary (of sequences). The sequence match is encoded as a tuple (i, c), where i is the index of the longest prefix match from the dictionary (an index of the match entry in the dictionary) and c is the following character (mismatch) after the prefix. The concatenation of the prefix and the character c is added as a new entry to the dictionary. The encoding goes on until the end of the input sequence. Therefore, the result is a collection of tuples.

The output of LZ-family encoded sequence can be compressed using Huffman encoding or arithmetic coding. A variety of different modifications have been applied to the LZ-family of methods, greatly improving their encoding efficiency – LZSS [191], LZW [206], LZRW [209], or LZAP [190], to name a few. A comprehensive descriptions of algorithms with examples can be found in [177]. As LZ-family methods are pretty general in algorithm design and do not pose strict limitations on the size of the used dictionary buffers (e.g., the capacity of LZ77 sliding window or the size of LZ78 list), the compression performance both in terms of compression ratio and speed can be tuned to specific data processing requirements. Therefore, LZ-family methods are one of the most widely used algorithms in data compression. They are an essential part of DEFLATE or LZMA algorithms, which are implemented in the most popular data compression applications, such as in gzip[22], 7zip [23], Snappy[24] or image formats such as GIF or PNG.

Burrows-Wheeler transform

Burrows-Wheeler transform [23] (BWT), also known as *block-sorting lossless data compression algorithm*, is a sequence permutation algorithm developed by Michael Burrows and David Wheeler in 1994. It defines a reversible permutation of the original sequence, where subsequences sharing the same prefix, tend to appear close to each other. Therefore, the obtained runs of characters can be encoded more efficiently by using locally-adaptive models.

To obtain a BWT transform of a given sequence S of length n the algorithm proceeds as follows. Firstly, all cyclic rotations of the input sequence

[22] http://www.gzip.org
[23] http://www.7-zip.org
[24] https://google.github.io/snappy/

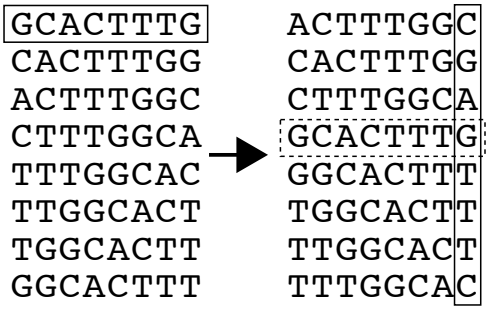

Figure 1.13: Illustration of BWT transformation of *GCACTTTG* sequence. Last column on the left indicates the result – *CGAGTTTC*.

are generated, forming an $n \times n$ matrix. The rotations are then sorted lexicographically. The last column of such a sorted matrix is the BTW transform of S. The backward transformation operation is straightforward and requires only knowing the index I, which denotes a row in the matrix containing the original sequence. The BWT algorithm, however, is best visualized by using a picture – Fig. 1.13 presents a BWT forward transformation of a sequence *GCACTTTG*, giving *CCGAGTTA* as a result.

The output of the BWT transform encoded using MTF, followed by RLE and Huffman coding is the basis of *bzip2*[25], one of the most commonly used textual data compression tools. Apart from being applied to data compression, the BTW transform has another very useful property. In 2000 Paolo Ferragina and Giovanni Manzini [55] designed *FM-index* data structure – a compressed full-text substring index based on the output of BWT transform. They showed that BWT transform of a given string S together with some auxiliary data structures can be used as a space-efficient index of S itself. Hence, this data structure is nowadays extensively used in bioinformatics, especially in sequence mapping algorithms to perform approximate string matching, as mentioned in Section 1.1.3.

1.3 Motivation

Data volumes generated by sequencing technologies, especially by 2nd generation sequencing technologies became a major issue for efficient storage and sharing. We have practically reached the symbolic barrier of 1000$ USD in amortized costs per whole genome re-sequencing of an

[25] http://www.bzip.org/

individual. However, there is growing concern about the explosion in growth of sequencing data, which emphasizes the importance on developing efficient high-throughput sequencing data storage, management and compression methods [33, 92, 72, 43, 19]. In 2012 the National Institutes of Health (NIH) organization in the United States launched a Big Data to Knowledge (BD2K) initiative to advance research in biomedical Big Data towards precision medicine; significant resources were allocated to advance research in genomic data compression methods[26]. With further advance in sequencing technologies, which will increase their accuracy, reliability, and throughput, it is very likely that soon the re-sequencing of individuals will become a commodity. Therefore, in order to prevent the potential explosion of sequencing data and to enable its efficient sharing and processing (for instance, in the cloud, which has become very popular), the development of new compression methods is of great importance.

[26] https://datascience.nih.gov/bd2k/funded-programs/software

CHAPTER 2

STORAGE OF HIGH-THROUGHPUT SEQUENCING DATA

In this chapter we will present the state-of-the-art in the area of high-throughput sequencing data compression and storage. We will show how the different data compression methods and genomic data processing algorithms presented in Chapter 1 are applied to reduce the size of HTS data. Since we are mostly interested in reducing the large storage space requirements of HTS data in DNA re-sequencing pipelines, we primarily focus on compressing the data stored in FASTQ and SAM formats. We will show the challenges and techniques to compress data in these formats by analyzing methods used to compress different parts of the files that they contain. Although the use-cases of FASTQ and SAM formats are different, the HTS data representation in SAM format can be seen as a superset of FASTQ format. Therefore, we will firstly analyze the techniques to compress the reads in FASTQ format and, then, the corresponding ones to compress alignments in SAM format. We will show that the techniques used to compress the data in both formats share a number of similar difficulties and follow analogous principles. Next, we will show alternative approaches to storing HTS data that are not strictly limited to compressed representations of data in FASTQ and SAM formats. Finally, we will briefly present the challenges and current solutions to storing the long-reads data from emerging third-generation sequencing platforms, which significantly differ from the most commonly used short-reads data. The results of compressing the HTS data represented either in FASTQ or SAM format and using different compression methods will be presented in Chapter 3.

2.1 Compression of raw reads in FASTQ format

2.1.1 Context

Raw sequencing reads generated by NGS machines are primarily stored in FASTQ format[1]. As mentioned in Section 1.1.3, each FASTQ record consists of three fields: read identifier, DNA nucleotide sequence and their corresponding base calling quality scores. Depending on the sequencing machine used and the sequencing experiment design, multiple FASTQ files can be generated in a single experiment and they can be either merged together or stored independently. The raw FASTQ files are often compressed using gzip and this already allows us to obtain around 3-fold reduction in size [43]. However, such a compression result is definitely not satisfactory, since gzip is a general purpose text compressor and as such is unable to exploit properties of genomic data represented in FASTQ format – fields containing data of different kinds are simply encoded together. Therefore, a variety of FASTQ format-specific compression methods have been developed, which share a number of common ideas.

A FASTQ file can be seen as a collections of records, each consisting of three different fields. The majority of modern[2] FASTQ compressors split each records' content into three separate *streams* of data, where each stream contains data of homogeneous type. The decomposition of the content of the records into different streams, each of which can be compressed by using different method, is a fundamental approach in order to achieve a reasonable compression of HTS data. For example, it has been demonstrated in [151] that instead of compressing a whole FASTQ file, as it is, one can achieve > 10% improvement in the compression ratio just by splitting the FASTQ files into three streams and compressing each separately using the same compressor (in this case: gzip).

Methods for compression of DNA sequences can be divided into four categories: general, mapping-based, assembly-based and read-reordering-based methods. Compressors using methods from the first category encode the reads as they are and without any prior knowledge about the data

[1] Formally, sequencers usually output reads in their own custom format, from which are then converted to FASTQ.

[2] Historically, some solutions like G-SQZ [196] or SeqDB [75] used to store sequence and quality symbols together into one byte prior to compression – such an approach, however, did not result in significant savings in the data storage

– they apply simple transformations and statistical methods to exploit the local similarities between the consecutive reads in the file. Mapping-based compressors use information about the reference genome, possibly mapping the reads to it and encoding only the minimal information about the mapping necessary to reconstruct the original reads. Assembly-based compressors assemble the reads in order to find possible overlappings among them. Compressors using methods from the last category reorder the reads inside FASTQ file according to a specified criterion in order to exploit the similarity between the DNA sequences on the whole file scope, which allows to improve the compression ratio.

In practice, however, advanced compressors use a combination of methods from different categories, performing the compression of DNA sequences as a multi-step process. For example, in the first step they can assemble contigs from the input reads, which can be later used as reference sequences when performing mapping and encoding of the reads. Hence, the compressors can be categorized by the core method (or approach) they use to compress (or process) the DNA sequences. From a high-level point of view, advanced DNA compression methods can be perceived of as dictionary methods, where either the externally provided reference sequences, self-assembled contigs or previously processed reads can be used as referential sequence matches. As a side note, it is important to emphasize that while developing any DNA sequence compression technique it is essential to encode the DNA sequences losslessly.

Read-identifiers-encoding methods fall into two categories: encoding by tokenization or differential encoding. In the first case, the read identifier string is split into tokens and then each token is encoded separately. In the second case, the read identifier is encoded differentially with respect to some identifier, which appeared previously in the stream. Usually, read identifiers are compressed losslessly, however, some compressors may prefer to skip storing the comment (the content after the first occurrence of a white space character in the identifier line).

Quality-scores-compression methods can be either lossless or lossy. Since quality scores stream characterizes high entropy with a high level of noise [19], it is challenging to compress it efficiently in a lossless manner. Quality stream usually occupies the majority of the space in the compressed archive, hence, a number of lossless and lossy quality-scores-compression

strategies have been developed, with the research focus put on the latter in recent years.

The main differences between FASTQ format compressors, therefore, lie in the methods used to compress different kinds of data encoded into (possibly, separate) streams. Hence, in this section, we provide a breakdown analysis of the FASTQ compressors focusing on compression methods applied per each kind of data.

2.1.2 Compression of read identifiers

Fields tokenization

FASTQ read identifiers can contain information such as unique instrument name, read number, flowcell lane, read length, etc. The format of the field is unfortunately arbitrary and sequencing-machine specific [19]. Nevertheless, read identifiers can be perceived of as a collection of different fields (tokens) separated by delimiters – as presented in Fig. 1.5. These tokens can be either of numeric, character or string type, having either a constant (fixed) or variable value among all the reads identifies in the whole FASTQ data set – these characteristics define the token class, which later determines the compression method. Having this in mind, some compressors perform tokenization of the read identifiers and compress each token using a dedicated encoding tailored to its characteristics. When encoding, usually, the previous read identifier is used as a context. Since read identifiers can share a significant amount of information among them in the dataset (e.g., reads coming from the one experiment may share the same experiment identifier), the idea is to store the fixed fields only once and to encode efficiently the variable ones.

A good example of a tool utilizing tokenization approach is a block-based (i.e., reading and compressing the file on a block-by-block manner) DSRC [42] compressor. DSRC inspects each field separately gathering all the fields statistics at the end of processing each block. Then, knowing the data characteristics, numeric fields can be either differentially encoded, binary encoded using a possibly minimal bit representation or encoded using order-0 Huffman coding. Non-numeric fields are encoded columnar-wise, i.e., per each field a vector is constructed storing characters occurrences at the field's positions observed in all the reads within the block. Lastly,

non-fixed fields are encoded using order-0 Huffman coder.

Similarily, Quip [85], Fqzcomp [19] and LEON [15] tokenize the read identifiers into n separate fields. They use different statistical models per each field. Depending on its value type(s) they may use multiple models per one field in order to achieve the best compression ratio. In this way, only a minimal representation of the identifier is stored, encoding only the differences in values (if those occur) with respect to the previous read identifier. Therefore, a numerical difference (delta) or a length of the longest common prefix with the mismatching characters is encoded. The final entropy coding is performed using arithmetic coders.

Differential encoding

In contrast to the tokenization, some compressors operate directly on whole read identifiers, storing the differences between the consecutive ones. For example, Fastqz [19] encodes read identifier lines differentially and column-wise. It stores the differences between the consecutive identifier lines and encodes them as a collection of triplets: a numeric field increment in the range $0-255$ selecting the starting column (position), length of the string match and the trailing differences (on a similar basis as Lempel-Ziv coding, but using only the last read as LZ buffer). In the next step, such encoded differences are passed through a mix of different context models and compressed using ZPAQ[3] [125].

Similarily, LW-FQzip [219] encodes read identifiers differentially, but with respect to the number of previous ones. When compressing read identifiers, it applies incremental encoding, storing as a result only common prefixes or suffixes and their lengths. FQC [49], on the other hand, performs a one-time indexing of the invariant parts found across a number of read identifiers – such information about the variability in the identifiers is further stored in a delta-encoded format. The encoded output in both LW-FQzip and FQZ is compressed using a LZMA-based compressor.

[3] http://mattmahoney.net/dc/zpaq.html

2.1.3 Compression of DNA sequences

General methods

Compression methods falling into this category are used to compress the DNA sequence data as it is, i.e., they only use data transformations and statistical methods to compress the reads in the local context. Usually, also the read order is kept intact. DSRC is a good example of a compressor implementing this category of methods and which introduces some interesting concepts. In order to lower the entropy of the DNA stream (limiting the alphabet possibly to only 4 DNA symbols), the rare N symbol occurrences and other IUPAC symbols are encoded in the quality stream. Since the numerical range for the quality values is at most $0-94$ [34], 161 additional values can be stored in a single byte by mixing together the base symbol and quality score. When the DNA alphabet consists of only a maximum of 4 values, DSRC encodes the sequence using two bits per symbol. Otherwise, it encodes the sequence data using order-0 Huffman coder. Additionally, to improve the compression ratio even further, DSRC provides an optionally LZ77-based dictionary compression method. When compressing, it looks for sequence matches in a buffer (of configurable capacity) consisting of previously encoded sequences. A similar approach of reducing DNA stream alphabet size has been applied in the recently published FQC [49] compressor where the final compression stage is performed by using LZMA-based compressor.

On the other hand, Fqzcomp [19] uses a configurable order-k Markov model when encoding sequences, using k previous bases as a context. This model is considered the main one with the possible maximum order of 14. Moreover, Fqzcomp provides an additional fixed size order-7 model to improve the accuracy of the predictions of the main model. Trying to improve the compression efficiency even further, the main model can be updated using the reverse compliment of the encoded sequence. The entropy encoding is performed using an arithmetic coder. As an important constraint, Fqzcomp limits the available DNA alphabet to only $\Sigma = \{A, C, G, T\}$ – the information about occurrence of N symbol is stored in the quality stream (similarly to DSRC), encoding the value of the corresponding base quality score as 0.

Quip [85] and Fastqz [19] also implement general methods for DNA se-

quence compression in one of their modes. Quip, in its default mode, uses a 12-th order Markov model with an arithmetic coder to compress the sequence data. Fastqz, by contrast, packs the bases symbols together into triplets or quartets by assigning $A = 1$, $T = 2$, $C = 3$ and $G = 4$ (the N base symbol is assumed to be uniquely identified by its corresponding quality score value of 0) and compresses them using ZPAQ. The focus of these solutions is put, however, on a mapping-based compression of the reads, which will be explained in the following subsection.

Mapping-based methods

The underlying idea of compression methods falling into this category is to map FASTQ reads to the provided reference sequence and to encode the reads within the mapping context. Such an approach gives us the possibility of eliminating high sequence data redundancy by efficiently storing only the sequence match position, size of the match and the potential differences from the reference. The non-mapped sequences, however, can be stored using the general methods, as presented in the previous subsection. The drawback of such an approach is that the same reference sequence needs to be available both during compression and decompression.

Fastqz [19] (in the reference-based mode) and recently published LW-FQZip [219] follows this approach. Both implement their own lightweight mapping algorithms with indexing based on hash tables to perform efficient substring searches. While processing reads, Fastqz will try to match the sequences to the reference and encode the best match as a 32-bit pointer, a direction bit and a list of up to four mismatches. The result is finally encoded using different context mixing strategies implemented in ZPAQ. By contrast, LW-FQZip after finding the best match, tries to perform a local realignment of the read around the mapping position. For each read, the position and edit operations (match, insertion, deletion, substitution) with their lengths are reported. The final output is compressed using LZMA-based compressor.

Assembly-based methods

Compression methods falling into this category apply sequence assembly methods to compress the reads in an efficient way. For example, Quip

[85] uses the first 2.5 million (by default; a configurable parameter) reads in a file to perform *de-novo* assembly of reads creating possibly large contigs. The assembly method is based on probabilistic de Brujin graphs, which utilize a Bloom Filter [20] data structure for efficient k-mer counting and filtering. The constructed contigs are in the encoding stage used as reference sequences, to perform reference-based compression of the remaining reads. The resulting alignments are encoded using different models with an arithmetic coder.

A similar approach was demonstrated in the recently published LEON [15] compressor. Prior to compression, LEON performs *de novo* assembly of the sequences constructing a probabilistic de Brujin graph based on Bloom filters, which are used for efficient k-mer filtering. Nevertheless, in contrast to Quip, LEON constructs the graph using all the reads inside the FASTQ file. After building the graph (which will be stored in the output archive at the end of the compression procedure) it proceeds to encode the reads within the graph context. Each read is encoded as its *anchor k-mer* (k-mer serves as an anchor node in the graph) with the branching information and the sequence length. The output is encoded using 0-order arithmetic coder.

Reads reordering methods

During the library preparation process, DNA molecules are cut randomly into many small fragments. These fragments are later sequenced by an instrument, which outputs the resulting short reads into a number of FASTQ files. It is a known property that the order of the DNA sequences in the resulting FASTQ file is completely arbitrary [56]. Hence, the read order can be altered, but remembering to preserve the information about the reads pairing if the library was sequenced in a paired-end configuration. Therefore, compression methods falling into this category reorder the reads according to some criterion in order to improve the compression ratio. Since the generated sequencing data is characterized by a high redundancy (especially present in the data generated from deep-sequencing experiments), by reordering the reads compressors try to exploit large sequence similarities between groups of reads.

The first compressor following this approach was SCALCE [70]. Its reads reordering technique is based on a Locally Consistent Parsing (LCP [13])

algorithm, which is used to search for sufficiently long patterns shared between the reads (called *core substrings*). For each read, LCP looks for the longest core substring and places the read in the corresponding bucket. As a result, all the reads inside one bucket share the same, possibly long core substring. Then, in each bucket the reads are ordered lexicographically with respect to the position of the core string. Finally, the reads are compressed using gzip.

Another interesting method was proposed by BEETL [14, 37, 83]. Its primary use is efficient k-mer retrieval from large collections of sequencing reads. However, as a proof-of-concept, it has been also applied for compression of raw DNA sequences and FASTQ reads, achieving very good compression results. The core approach is based on applying the Burrows-Wheeler transformation over the collection of input DNA sequences, which allows to exploit high redundancy present in sequencing data. Moreover, BWT transformation is also used to build an FM-index (see Section 1.2.5) for the data, which allows for fast substring lookups. The generated BWT output is run-length encoded and indexed. By compressing the output with PPM-based or LZMA-based compressor a high compression ratio can be achieved.

kPath [87] is a recent DNA sequence-only compressor, which achieves a high compression ratio. Its compression algorithm is as follows. Given the reference sequence and k-mer length, it builds a de Brujin graph based on k-mers from the provided sequence. Then, it trains the probability distribution model based on the calculated k-mer frequencies. When processing the reads, they are firstly reverse-complemented, where heuristics are used to determine which orientation matches better the provided reference sequence. In the next step, the reads are sorted lexicographically according to their initial k symbols – the goal is to place the reads with the same possibly longest substrings close to each other. The remainder of each read is encoded as a path within the built graph. The final output is encoded using arithmetic coding with the probability distribution model trained using the supplied reference sequence.

Mince [158] is another recently published DNA-only compressor. It clusters the reads into different bins which correspond to different minimzers, i.e., lexicographically smallest k-mers [172]. For each read, Mince searches for its k-mers and assigns a bin if the bin's minimizer is present in the read

and if the read is sharing possibly the maximum number of k-mers with the reads already stored in the bin. After clustering, reads inside the bins are sorted lexicographically starting from the position of the minimizer. Finally, the reads are compressed using a LZMA-based compressor Lzip[4].

2.1.4 Compression of quality scores

Lossless methods

It has been widely noted that in the data produced by most sequencing technologies there is a direct correlation between position and quality score values and that the read quality typically degrades along the length of the sequence [95, 93, 19]. In addition, a significant number of quality sequences generated by Illumina machines end in a run of score 2 (ASCII symbol '#'), which may be related either with a low quality of the sequence or with a common issue of Illumina base calling process [142]. Therefore, a majority of FASTQ compressors use the above observations trying to compress quality scores efficiently. They try to model the stream of quality scores either as a piecewise stationary memoryless source or as a Markov model (see Section 1.2.3). Hence, they implement different context-based models, where a context can be built of the preceding quality values and the base position inside the read.

For example, the authors of DSRC [42] noticed three types of quality streams characterized by different statistics and proposed different compression methods accordingly. The first type consists of quasi-random values with a mild dependency of its position onto the value's symbol distribution. DSRC handles this type of quality stream by gathering statistics and encoding the values columnar-wise, i.e., the statistics of quality scores occurrences per each position are gathered and are later encoded using Huffman coding. The second type is a special case of the first one, where quality sequences end in runs of low score 2 – this information is encoded simply as a bit flag. Lastly, in some cases, the authors observed a strong local correlation of quality scores values clustered within individual records. This type of quality data is compressed using RLE followed by an order-1 Huffman entropy coder. By contrast, authors of FQC [49] simplify the above approach and just use RLE following LZMA-based compressor

[4] http://www.nongnu.org/lzip/

to store the quality scores.

Moreover, Fqzcomp [19] authors observed that sequences tend to be overall either good or bad. This observation has a direct application for encoding the quality scores. When processing reads, if a read already contains a significant number of low quality values it is more likely that even more low quality values will appear later when encoding its quality scores. Therefore, to support multiple quality data models (including also the general ones mentioned at the beginning of this subsection), Fqzcomp utilizes a context mixing technique, using 5 different contexts to predict the quality values and to compress them using arithmetic coders – the exact definition of how the contexts are built can be found in the Fqzcomp publication [19].

Fastqz [19] encodes quality scores using special byte codes, which are as follows. Quality scores of 38 (the highest observed value) are encoded as runs up to the length of 55 using RLE. Quality scores in range $35-38$ are packed as triplets or, if not possible, are either packed into pairs (applies to scores in range $31-38$) or stored as single bytes (applies to all other quality scores). If a read ends with run of low quality value of 2 – it is indicated by just a single symbol. The resulting byte code is compressed by passing through a mix of context models provided in ZPAQ.

Another approach was proposed by Quip [85], which uses an order-3 arithmetic coder. As a context, three preceding quality scores are used (but binning coarsely the last two). In addition, Quip conditions the context on 2 variables: the encoding base quality position within the read and a number of "large" differences between the previous quality scores between adjacent positions. Other compressors, like the LEON [15] or the SCALCE [70], just use order-3 arithmetic coder to store the quality data in lossless mode – their primary focus is on the lossy compression.

Lossy methods

Quality scores are the most difficult type of data to compress – they occupy more than a half of the gzip-compressed file and contain a high degree of noise [19]. Moreover, NGS data analysis applications rely on quality scores in a heuristic manner [153], which puts in doubt the usefulness of the currently used high resolution of values. Therefore, Illumina has recently proposed a reduced quality scores resolution [80], by lowering the

Table 2.1: Illumina 8-level quality values bining scheme. The quality values are in Phred scale. Source: [80]

Quality values range	**Mapped bin value**
'N' (no call)	'N' (no call)
2 – 9	6
10 – 19	15
20 – 24	22
25 – 29	27
30 – 34	33
35 – 39	37
≥ 40	40

available quality scores range to only 8 values (or bins[5]). Such a quantization already allows us to reduce the size of the gzip-compressed FASTQ file by about 30% compared to one without quality bining. Moreover, Illumina showed that such a quantization does not degrade the results variant calling and can be considered as an upcoming standard. The new Illumina instruments, such as Illumina HiSeq X sequencers, already output scores in the binned form. Therefore, in order to store the FASTQ files in an efficient way, the lossy compression of quality scores is becoming a must and a number lossy compression methods have been investigated.

In [204] the authors state that quality scores are a result of an irreversible quantization of some original (floating point) sequencing error probabilities with $|\Sigma| = 94$ distinct values. Therefore, the quantization can be viewed as the process of partitioning the probability $[0, 1]$ into Σ sub-intervals or bins. Quality scores falling into the same bin share the same quantized quality score. Therefore, under the above assumptions, the authors propose a number of different bining strategies and show how they affect the sequence mapping accuracy. The most important scheme is the 8-level bining, which is presented in Table 2.1. The above mentioned Illumina's reduced quality scores resolution is based on this scheme.

In SCALCE [70] the authors observed that when compressing quality scores, for any basepair the quality values of its surrounding basepairs would be either the same or within a small range of the current value. Therefore, given an error threshold, they apply a lossy transformation of the values

[5]The scheme is sometimes also called as *Illumina bining*

and reduce the quality scores alphabet size. The goal is to reduce variability among the quality scores in the proximity of local maxima up to the threshold value, which helps to reduce the size of the output stream.

Fqzcomp [19] authors observed that quality scores of similar values tend to be clustered in consecutive blocks. Therefore, as a lossy transformation, they propose to smooth the qualities within blocks to ensure that, for each value, the difference from the original one is no more than the given threshold. On the other hand, FQC [49] proposes 3 levels of lossy transformations, with a predefined quality cutoff threshold and group size. Each quality score below the cutoff is replaced by 0 value, whereas quality scores above it are smoothed in groups.

To lossy compress quality scores, LEON [15] uses k-mer statistics coming from the input dataset calculated during data preprocessing step. It assumes that if the nucleotide bases are covered by a sufficient number of solid k-mers (i.e., k-mers that occur more than a minimal number of times in the dataset), they can be safely considered error-free and they are assigned an arbitrary high quality value. A similar approach was also explored by Janin *et al.* [82] where authors performed smoothing the values of the scores by computing the longest common prefix for the reads using Burrows-Wheeler transformation. The authors of both projects also show that performing a conservative lossy transformation of quality scores has a minimal impact on the results of downstream analyses and it greatly reduces the compressed file size.

QualComp [152] is a quality-only compressor. It utilizes techniques from rate distortion theory and assumes that quality scores are generated by multivariate Gaussian source. Given the user-provided maximum distortion rate, QualComp tries to use an optimal number of bits to represent scores. To do so, it tries to minimize the distortion of the values by minimizing the mean square error rate between original and transformed values. It uses k-means algorithm [123] to cluster the qualities into groups for which the mean square error is minimal. Finally, it applies a lossy transformation to the quality values inside the clusters and encodes the values using an arithmetic coder.

QVZ [126] is another quality-only compressor based on rate-distortion theory. It assumes a positional correlation between the neighboring values

and models a sequence of consecutive quality scores as a Markov chain of order 1. Firstly, for all the quality sequences in the input file (or inside a block of a given size), it calculates the Markov model state transition probabilities between the adjacent values. These transitions are in the following step used to compute a list of quantizers (called a *codebook*) , indexed by the base position in the read. With the codebook ready, the scores are transformed using a different quantizer per position. Finally, the transformed scores are encoded using an arithmetic coder.

Quartz [217] is a recent quality-only compressor, which, similar to LEON, utilizes pre-calculated k-mer statistics to correct the quality scores. However, in contrast to LEON, Quartz utilizes an external resource of k-mers statistics. It counts k-mers ($k = 32bp$ by default) in a given external set of files, which will be later used as a dictionary while compressing a new file. Such a dictionary can be generated once and used multiple times. When compressing quality scores, Quartz firstly tries to identify all k-mers present in the reads and within a small Hamming distance from the k-mers present in the provided dictionary. Any quality value at a given position that is consistent with at least one of the supporting k-mers is set as the default (maximum) value, whereas the quality score that is divergent from all supporting k-mers at a given position is kept intact. The authors show that their approach only slightly affects the accuracy of downstream analysis and, in some cases, it may even improve the genotyping accuracy of some selected pipelines.

2.2 Compression of mapped reads in SAM format

2.2.1 Context

Following its successful application in the 1000 Genomes Project [1], SAM format became a *de facto* standard for the representation of information about the sequence alignment (the data is stored usually in a compressed BAM format as explained below). In general terms, information stored in SAM format can be seen as a superset of the equivalent stored in FASTQ format, where FASTQ reads are extended by information of their alignment and with some optional information. Although the use-cases of FASTQ and SAM formats are different, they share a number of similar methods to compress the HTS data they represent.

Analogously, as in the case of compressing raw FASTQ reads, to efficiently compress SAM alignments, the content of the records fields can be split into a number of separate streams. The streams can be then compressed using possibly different compression methods. However, when compressing SAM alignments, it is more practical to analyze the available compression solutions in terms of proposed methods covering different categories of data. In general, four different categories of data can be distinguished: query template names, sequence alignment data, quality scores and optional fields. Therefore, different SAM format compressors use a set of possibly different methods to compress these kinds of data.

However, in contrast to FASTQ files, the read order of the reads (alignments) stored in SAM files is important for practical reasons. The alignments are typically sorted according to their mapping position (and are indexed), which allows for fast random access to alignments by specifying the chromosome and the position within the query range. Hence, such reordering of the reads may influence the efficiency of the methods used to compress different categories of data, as compared to ones used in FASTQ files compression.

Methods used to compress the alignment data can be divided into two main categories, namely: non-reference-based and reference-based methods. The difference between them is whether or not the reference sequence is used during compression and/or decompression stages. Similarly, as in the case of analyzing compressors of FASTQ format, some advanced SAM compressors can also utilize a mix of different compression techniques to compress the sequence alignment data. Since the data represented in SAM format can be seen as a superset of data represented in FASTQ format, SAM compressors can also utilize previously described methods to compress DNA sequences (as presented in Section 2.1.3). For example, a compressor can assemble the sequences into contigs and then encode the alignments with respect either to the provided reference sequences or self-built contigs, depending on which option can provide a better compression result. Nonetheless, SAM format compressors are usually analyzed in context whether they use the reference sequence or not.

Considering methods to compress the query template names, the classification of the methods with the techniques used to compress them in FASTQ format also apply here, hence, we won't cover them in detail. The

description of the methods with their potential difficulties can be found in Section 2.1.2. However, what is important, some of the methods used to compress the read identifiers can be also applied to compress the optional fields and we will cover these techniques in brief.

Similarly, as in the case of FASTQ files, in SAM files the quality scores tend to be the most challenging kind of data to compress, occupying in the lossless form the majority of space in the compressed archive [25, 71]. Most of the methods used to compress quality scores in FASTQ format can be also used to compress them in SAM format, both in lossless and lossy modes. The solutions have been described in detail in Section 2.1.4, hence, we will only focus on the methods, which are specific to SAM format, i.e., the ones utilizing the alignment information to aid the compression.

As a side note, when compressing FASTQ files it is critical to preserve all the DNA sequence data, agreeing only to apply a controlled degree of information loss to either quality or read identifiers streams. However, some SAM format compressors may not fully preserve all the data, not only by lossily compressing quality scores or removing query template names, but also, e.g., by discarding the optional fields of the alignments or by completely discarding the unaligned reads.

2.2.2 BAM and CRAM formats

SAM content is normally represented in raw text format and, for practical reasons, the data is typically stored in its compressed form – in Binary Alignment/Map format (BAM). Storing alignment data in BAM format is also currently considered as a *de facto* standard. The alignments inside BAM file are packed into independent blocks which are compressed using Blocked GNU Zip Format (BGZF), a gzip-based compression method. Some strong advantages of BGZF are random access and indexing, which allow for selective decompression of blocks. Unfortunately, the alignment data is stored in blocks as it is, i.e., the data from the fields is lumped altogether and not decoupled. As a consequence, the compressed files exhibit similar problems as gzip-compressed FASTQ files, where the compression ratio is not impressive.

Recently, however, CRAM[6] format has started being adapted as a prominent replacement of BAM format. A notable improvement is that the alignment data can be stored using a reference-based encoding scheme, greatly reducing the sequence data redundancy. Moreover, in CRAM format, in contrast to BAM, each field can be encoded separately as a data stream. It is also possible to select a different codec per each data stream, either from the built-in codecs or some external ones. The built-in codecs include: binary encoding, Golomb encoding, Gamma coding, Huffman coding. The external codecs may include: rANS [47], gzip or bzip2 and possibly more. Therefore, a wide number of features supported by CRAM format allows compressors using it to achieve a very good compression ratio.

Support for CRAM format has been implemented as a part of HTSlib and HTSjdk[7] libraries, which are used for reading and writing high-throughput sequencing data formats. These libraries are also used by most popular tools in genomic data analysis such as SAMtools, GATK or Picard and compression tools such as SCRAMBLE [18] and CRAMTools[8]. Therefore, apart from significant improvements in compression ratio and with offering data processing speed comparable to BAM, it provides a smooth transition from the BAM format in the current genomic data analysis ecosystem. As a side note, the initial implementation of CRAM format in CRAMTools was not fully supporting lossless compression, i.e., the compressor was not storing query template names and was dowsampling the quality scores. Hence, its potential adoption in genomic data processing pipelines as a replacement for BAM format has been very slow and cautious.

2.2.3 Compression of query template names

A query template name of SAM alignment is similar to the read identifier field in FASTQ read[9]. Hence, the methods used to compress query template names (with the present difficulties) are similar to the ones used to compress FASTQ read identifiers. Analogously, they fall into two categories – based on tokenization and differential encoding. However, the main dif-

[6] https://samtools.github.io/hts-specs/

[7] http://www.htslib.org

[8] http://www.ebi.ac.uk/ena/software/cram-toolkit

[9] The query template name field does not contain the initial '@' symbol present at the beginning of the FASTQ read identifier field nor it can contain any comments.

ference between compressing this kind of data in FASTQ and SAM formats lies in the fact that the reads are reordered according to their mapping position, which differs from the one in original FASTQ file and which may influence the compression. This is due to fact, that in original FASTQ files, some fields present in the identifiers of the consecutive reads tend to exhibit some sort of order (leading to a better compression), which is not kept, when the reads are stored sorted by their sequence mapping position.

Nonetheless, Quip [85], which is both FASTQ and SAM compressor, utilizes the same tokenization method to compress this kind of data in both formats. Similarly, Samcomp [19] utilizes read identifiers compression routines from Fqzcomp [19] to compress query template names. Other tokenization-based compressors include SCRAMBLE [18], DeeZ [71], whereas compressors utilizing differential encoding are SAMZIP [176], NGC [162] and HUGO [114]. The principles behind tokenization-based and differential-encoding-based methods have been described in Section 2.1.2.

2.2.4 Compression of alignment data

Non-reference-based approach

Methods falling into this category, do not use externally provided reference sequences in order to compress the alignment data. Some solutions, however, may apply specific data remodeling techniques by transforming the initial data representation into a different form prior to compression in order to improve the compression ratio. Some may also skip encoding parts of alignment data and to re-generate it while decompressing by analyzing and combining data present in other fields.

SAMZIP [176] was one of the early solutions to compressing segment and alignment data. It performs decoupling of the data from SAM fields and encodes each field separately. To encode *SEQ* field, SAMZIP considers only 4 DNA symbols and encodes them using 2 bits per base. For any rare occurrences of the *N* symbols it stores their positions separately. The alignment fields are encoded as follows. The *FLAG*, *RNAME*, *MAPQ*, *PNEXT* and *CIGAR* fields are encoded using run-length encoding. As *POS* values can span into large values and appear in an increasing order (in sorted SAM

file) the values are delta encoded, and followed by run-length encoding. To encode *RNEXT* it uses a mix of two methods – binary encoding (to encode the frequent values of '*' and '=' symbols) and run-length encoding. SAMZIP does not encode value of *TLEN*[10]. The produced output files with possible unaligned reads (stored as they are) are compressed using WinRAR. Such an approach already shows up as a significant improvement in reducing the storage footprint – alignments compressed by SAMZIP can occupy up to 40% less space than stored in BAM format.

To encode sequence data, Samcomp [19] utilizes *FLAG*, *POS* and *CIGAR* fields to anchor each base to a virtual reference coordinate and then encodes the bases according to a calculated per-coordinate model. Then, as more and more segments align to the virtual reference coordinates, the statistical model improves its accuracy and the data can be compressed more efficiently. Optionally, Samcomp can use an external reference sequence to seed the model's initial probabilities. However, given the usually deep coverage of the generated sequencing data, the model adapts itself well to the DNA stream characteristics. This, on the other hand, manages to achieve the encoding efficiency as close as when supplying an external reference sequence. The remaining alignment fields are encoded using arithmetic coders and using a different model per each field's type. A similar approach is used by Quip [85], which is, however, strictly a reference-based compressor, requiring the reference sequence to be present both when compressing and decompressing the SAM files.

CSAM [25] proposes a method to store sequence data based on sequence assembly principles. For a group of reads, it assembles from segments a consensus sequence called a *Presumed Reference Sequence* (PRS). To reduce the DNA alphabet size, it uses only standard 4 base symbols. Then, while encoding the segments, the sequence is stored as differences relative to PRS (as an alternative to the reference sequence, which is not present) – either as a set of copy (match) or replace (followed by the replacement symbol(s)) operations. The position is encoded differentially with respect to the previous alignment's position. Such transformed data with the remaining fields and unmapped segments are stored in separate streams, which are finally compressed using gzip.

NGC [162] introduces a more advanced alignments encoding. To store

[10] In the initial version of SAM format, this field was not yet present.

SEQ data it applies *vertical differential run length encoding* (VDRLE). The process can be sketched as follows. Firstly, on a similar basis as CSAM, NGC analyzes all the bases covering each genomic position and builds the common consensus sequence. In the next step, it processes reads in groups, but instead of encoding each segment separately (known also as a *horizontal approach*), NGC encodes bases per each genomic position (a *vertical approach*) as differences with respect to the built consensus sequence. As in the smajority of cases, there will be a high degree of matches, the differences can be efficiently encoded using RLE. The remaining *FLAG*, *RNAME* and *MAPQ* fields are encoded using RLE. *POS* and *PNEXT* values are delta encoded with the resulting *POS* values further encoded by Golomb/Rice coder. *CIGAR* values are not directly stored and are re-generated while decompressing the data (based on the VDRLE-encoded segment data with the consensus metadata). Such transformed data with possible unaligned reads is compressed using a general purpose compressor, namely, either gzip, bzip2 or LZMA-based one.

Reference-based approach

Having reads already mapped to the reference sequence, it can be more practical to encode the alignments with respect to the reference in order to further reduce the sequence data redundancy. Moreover, mappers usually store the information about sequence matching results in the mandatory SAM *CIGAR* and optional *MD* field, which allows downstream applications to perform the variant analysis without looking at the reference sequence. Having access to the reference, storing the variation information from these fields can be either re-modeled into a more compact form or, possibly, omitted and re-generated during decompression.

MZip [58] was the first[11] proof-of-concept solution exploring reference-based compression of SAM files. It encodes alignment data as follows. It encodes alignment position with respect to the reference and relative to the previous alignment using Golomb coding. Any information about sequence differences with respect to the reference is stored as an offset relative to the read's anchor position along with the corresponding variants.

[11] Around the same time as MZip, SlimGene [93] was published, however, compressing reads in (today deprecated) Illumina Export Format, which substantially differs from the SAM format.

These include: the variant type, the altered bases (in case of substitution or insertion) or the length (deletion). Variants information is encoded using special binary codes and Gamma code (for lengths). MZip also encodes in a compact way read's mate-pair information (if present), reducing further data redundancy. In case there are unmapped segments in the SAM file, MZip performs a de novo assembly on them. It tries to build possibly large contigs, which will be later used as an alternative reference to encode the unmapped segments. The compression of the remaining fields was not supported by authors, since MZip was more a proof-of-concept solution designed to showing potential storage savings by applying reference-based compression. The ideas behind MZip were successfully implemented and improved in CRAMTools, which also provided the initial implementation of CRAM format. The format and the compression methods has gone through a number of improvements since its initial release. For example, SCRAMBLE [18], which implements CRAM format, makes it possible to embed the reference sequence in the compressed archive, eliminating the need to be explicitly provided during the decompression.

HUGO [114] is another reference-based solution, but offering remapping of the reads and using multiple reference sequences. It introduces three classes of mapped reads: the unmapped reads (UMR), the inexact mapped with more than 4 mismatches (IMR) and the exact matched (EMR). For the IMRs and UMRs, it splits the reads into two shorter ones, each time splitting them in half of their length. Next, it tries to remap the reads to different reference sequences using SOAP 3 [118] mapper. The resulting EMRs are output for compression, whereas the remaining IMRs and UMRs are further remapped. The remapping is performed iteratively until a specified number of iterations has been reached or until the fraction of the unmapped reads is below a given threshold. The resulting record alignment fields are compressed using a mix of different codecs. *FLAG*, *MAPQ*, *TLEN* are compressed using Huffman coding. *POS* and *PNEXT* are differentially encoded with respect to the previous alignment followed by Huffman coder. *RNAME* and *RNEXT* are compressed using RLE, and *CIGAR* is compressed using LZW [206].

DeeZ [71] is a recently published SAM compressor achieving very high compression ratios. Similarly, as the non-referential based solution CSAM, it compresses alignments in blocks and constructs from the segments an updated consensus sequence. In this way, variants present in alignments

are encoded only once in the consensus. Such an approach reduces the required amount of space to store variant data in a significant way. Moreover, it simplifies the representation of *CIGAR* and *MD* fields or even eliminates the need for storing them, as with respect to the updated consensus reference there may be no variant present in the alignments. The differences between the built consensus sequence and the provided reference are stored only once – it will be used to decompress the records. Such a transformed alignment information with the other remaining fields is stored in separate data streams, where each is compressed using gzip. Only the alignment position is differentially encoded (with respect to the previous alignment's position) followed by an arithmetic coder.

2.2.5 Compression of quality scores

As mentioned previously, the majority of the methods used to compress quality scores in FASTQ files, both in lossless and lossy modes, can be also applied in case of SAM files. For example, considering the lossless methods, Quip [85] compresses quality scores using the same method both for SAM of FASTQ files. Other tools, such as Samcomp [19], DeeZ [71], and SCRAMBLE [18] utilize quality compression methods derived from Fqzcomp [19]. Considering lossy compression of quality scores, DeeZ [71], for example, utilizes the lossy scheme derived from SCALCE [70], whereas CSAM [25] proposes a strategy similar to the one offered by Fqzcomp [19]. Moreover, RQS [216], although being a quality-only compressor, is based on the same method as proposed by Quartz [217]. Finally, some solutions, such as QualComp [152] and QVZ [126] are format-independent and can be applied both to compress quality scores present in FASTQ and SAM files. The principles behind the above mentioned methods have been described in detail in Section 2.1.4.

Regarding the compression methods specific to SAM format, an interesting approach to compressing quality scores was presented in MZip [58], where the authors proposed to support only partial storage of the quality values. Instead of storing quality scores individually per each read (known also as a *horizontal* mode), they encode sequences of quality scores per each genomic position, encoding them in a *vertical* mode. Primarily, quality scores are stored at the positions where the corresponding bases show difference with respect to the reference sequence. When the bases match

the reference sequence, only a user-defined small percentage of quality scores per matching position is kept. Such transformed qualities are encoded using Huffman coding, encoding them in runs column by column (as per position in the genome). This technique was later implemented in CRAMTools and extended to allow selective compression of quality scores, e.g., by applying different compression schemes depending on the variation type or the calculated base coverage. Moreover, in the current version of CRAMTools (and SCRAMBLE [18]) the 8-level quality scores bining scheme proposed by Illumina [80] has been added as one of the lossy transformation methods.

NGC [162] proposes a similar approach as MZip and introduces four categories for compressing quality scores. The category selection depends on whether at a given position (in genomic coordinates) all the bases (or only a part of them) in segments match the reference sequence and if not – whether the variant at that position suggests a multiallelic region. NGC then applies different quantization of the quality value depending on the selected category. After quantization, the values are encoded either independently per alignment (the horizontal mode) or per-position (the vertical mode). In the second case, NGC uses vertical run-length encoding (VRLE), which provides a significantly better compression ratio than when encoding quality values per alignment. The final output is compressed using one of the general purpose compressors, such as gzip, bzip2 or a LZMA-based one.

Compression of optional fields

Regarding the compression of the optional fields, as mentioned in Section 1.1.3, in SAM format they are represented as a tab-separated collection of fields, where each is represented as a triplet in form *TAG:TYPE:VALUE*. Therefore, the methods to compress the query template names can be also applied to compress the optional fields.

A good example here is Quip, which utilizes the same compression methods both to compress query template names and optional fields. It applies tokenization on optional fields and encodes the values using a different statistical model per each type of field. Then, it encodes the values using an arithmetic coder with the field tag identifier serving as an additional context. In a similar way, Samcomp [19] utilizes read identifiers compression

routines from Fqzcomp [19] to compress the optional fields.

SAMZIP [176] performs tokenization of the fields and encodes only the field's tag and type information using Huffman coding, storing its value as it is. Then, it compresses the encoded fields using a general purpose compressor WinRAR. On the other hand, NGC [162] performs run-length encoding over a number of optional fields, compressing the encoded data using gzip, bzip2 or a LZMA-based compressor.

DeeZ [71] proposes a more sophisticated way to compress the optional fields. It tokenizes the fields and stores together the values denoted by the same tag identifier and the same type in different streams. In this way, each stream stores the values of homogeneous type with the values sharing possibly the same characteristics. All the resulting streams are then compressed using gzip or bzip2. On a similar basis, SCRAMBLE [18], tokenizes all the fields and stores the values in separate streams, but offers more advanced encoding methods. For example, while encoding fields containing read group names or fields containing auxiliary quality scores (denoted by different tags), SCRAMBLE can apply a different encoding scheme best suited for the field's data characteristics. The final streams compression stage is performed either by applying a general purpose compressor like gzip or bzip2 or by using an rANS entropy coder with some custom model.

2.3 Alternative HTS data storage solutions

2.3.1 MonetDB – a column-oriented database

A database is an organized collection of data, which allows for easy access to, management of, updates of and queries on the data. In computing, a database is a collection of *schemas*, *tables*, *queries*, *reports*, *views* and other objects. The schema defines the structure of the database in a formal language. The schema is supported by a database management system (DBMS), which includes definition of tables, fields, types, relationships between the objects, etc. The table is a collection of related data and is represented in a structured format consisting of columns and rows. The row is the single information item in the database. The information stored in the database is usually organized in a way to model aspects of reality or

a specific use-case.

Databases can have a row-oriented (horizontal) or column-oriented (vertical) architecture. The most general difference between these two (from the data storage point of view) is as follows. The column-oriented database vertically partitions the database into a collection of individual columns which are stored separately. The row-oriented database stores the data compacted as tables. This internal architectural property has a direct impact on performance. The column-based systems enable queries to read just the attributes they need rather than having to read the entire rows from the disk and discard unneeded data once they have been loaded into memory [4]. Analogously, column-oriented databases have many opportunities to reduce storage space by applying compression on the data stored in columns, since the data type of the values in the same column is usually uniform.

Compared to the genomic data representation and storage – popular flat file formats keep the data stored record-wise, where each record consists of a number of separated fields. Therefore, the data can be easily decomposed into a number of separate streams or columns, which store the data of homogeneous types. This property of genomic data is frequently exploited by format-specific compressors and can be a great fit for column-oriented databases. Moreover, since the data stored inside a database could be modeled in a more intuitive and descriptive way, storing the genomic data in a column-oriented database may allow us to perform advanced queries on the data, which is impossible to achieve by using format-specific tools.

As an example, in [30] the authors used column-oriented DBMS MonetDB [78] as a proof of-concept to store and analyze the sequenced genome of the Ebola virus. By using MonetDB to store alignment data, the authors were able to perform advanced queries on the data mimicking a selected set of functionalities provided in SAMTools. Moreover, they could import into a single database multiple SAM/BAM files, each represented as a separate set of tables. Such a method allowed them to perform a joint analysis of the datasets.

Since the data stored inside a database could be modeled in a more descriptive way, the authors proposed two different schemas to store and

represent SAM record. The first schema, called *Sequential storage schema*, is a straightforward mapping of the alignments fields in SAM format to their respective columns in the database table. The alignments are stored in two tables. The first one stores the content from alignment mandatory fields, whereas the second one – the optional fields. Such representation is, however, cumbersome for sequence analysis using paired-end sequencing data (which is the most common case), as the alignment pairs reconstruction needs to be performed per each analysis, which is a computationally expensive task. Therefore, the authors also introduced the second schema – *Pairwise storage schema*. With this schema, primary and secondary alignment pairs are explicitly stored in separate tables, which allows for a more efficient query execution. In addition, unpaired alignments are stored in a separate table. The optional fields are stored as in the case of sequential storage schema.

Since MonetDB focuses on providing a fast performance speed for data analysis, it physically stores the data in the same way both in-memory and on disk. Therefore, it implements only minimal compression on the data. It optimizes the columnar representation of the data and tries to represent it as a dense array. It also encoding the string data using a dictionary encoding. This approach has significant advantages versus row-oriented databases, in which empty attributes usually need to be explicitly stored. However, in the case of storing large volumes of genomic data, such an approach may not be efficient. Storing BAM files in a MonetDB database has unfortunately a significant overhead [46] versus storing it in BAM format – it requires at least 3-4 times the storage space of the BAM file itself.

2.3.2 cSRA format

Sequence Read Archive (SRA) is a large bioinformatics database, which provides an open public repository for DNA sequencing data. It is a part of the International Nucleotide Sequence Database Collaboration (INSDC), and is run as a collaboration between the NCBI, the European Bioinformatics Institute (EBI), and the DNA Data Bank of Japan (DDBJ). Its focus is on efficient archival and distribution of sequencing data, especially on short reads data generated by high-throughput sequencing platforms. The majority of the data stored in the repository comes from human or human-

related sequencing projects, such as the 1000 Genomes Project [1] and the Human Microbiome Project [159]. In order to support submission of data in multiple formats (some of which can be machine-specific) and efficient distribution of the archived data a special internal data format was developed.

The SRA format [103] (further replaced by cSRA) is a special column-oriented data format designed with column-oriented databases in mind. It allows for efficient storage of genomic data (as submitted in FASTQ or SAM format) by decoupling the content in the record fields and encoding the data column-wise. Therefore, it allows for accessing a reduced set of columns and removal of individual columns in the file. Additionally, in cSRA the genomic data format is defined by a special and flexible schema.

SRA Toolkit[12] is a set of utilities supporting cSRA format and providing API to store and access genomic data of different types. The primary use of a SRA Toolkit is submission of the data to an SRA archive. However, local storage of the data is also possible. Storing genomic file of a specified format into a cSRA archive involves a number of data preprocessing steps [13]. Firstly, the input file is parsed by a format-specific parser and the records' content is split into a number of separate (input) columns. Next, the values in the input columns follow a number of data transformations to possibly simplify the initial representation and/or to reduce the number of output columns required to represent and store the transformed data. The columnar data are encoded and serialized to a disk. The content is distributed into a number of different directories (representing the hierarchical data structure) and files (representing the columns) with additional meta-information and indexing files. Such data structure can be further compressed into a single cSRA archive file to enable efficient sharing of the content. To decompress the content, selected columns in the archive are deserialized, followed by decoding and data transformation steps. The decoded output will match the format of the input data as specified in the schema. In addition, while creating the archive, SRA Toolkit allows one to include a set of useful meta-information to accompany the data. For example, tracking information about the used reference sequences, which can be downloaded during the decompression or which can be accessed

[12] https://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?view=toolkit_doc
[13] Described in https://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?view=doc

directly while residing in the local filesystem.

When storing DNA sequencing data, it is recommended to store them in the aligned form. An SRA Toolkit provides an efficient way to store SAM files, by exploiting reference-based compression. Moreover, it can store only the sequence variants with respect to the used references, greatly reducing the sequence data redundancy. SRA Toolkit also provides an option to compress the input data with a controlled degree of information loss. Some lossy approaches include removing the data from read identifiers, performing lossy compression of quality scores or eliminating the unaligned reads. As a good use-case example, SRA archives hold the compressed data from the 1000 Genomes Project. The initial size of the aligned files in BAM format required 250 TB of disk space to store. However, by applying semi-lossless compression the required storage size was reduced to 85 TB. The operations included stripping off selected information from query template names (and saving them as unique integers) and performing 40-level read qualities binning using recalibrated quality scores. Furthermore, by applying a 8-level binning (as proposed by Illumina) and uniform binning of recalibrated quality scores, the overall size has been reduced to 30 TB, being 12% of the original BAM. By contrast, the variant calling analysis results stored in compressed VCF format occupy 0.1 TB.

2.3.3 HDF5 format

HDF5[14] [57] is a technology framework consisting of a data model, a library and a file format for storing and managing data. It supports a variety of different data formats and data types and is designed for efficient I/O operations for high-volume and complex data. The (possibly complex) data format can be defined by a user by using a set of provided basic HDF5 data types. Moreover, HDF5 allows one to define additional operations (called *filters*) to be performed on the data during I/O operations. For example, such as applying compression filters, which include ZLIB[15] (upon which gzip is based) or SZIP[16]. Finally, HDF5 is a mature technology with multiple wrappers and bindings available in different programming languages (such as C, C++, Java, Python, R) and is supported by large scientific

[14] http://www.hdfgroup.org/HDF5
[15] http://www.zlib.net/
[16] http://www.compressconsult.com/szip/

applications (such as MATLAB).

Data stored in HDF5 format is treated as *HDF5 information set* and is represented as a single HDF5 file. Each information set is a container holding annotated associations of array variables, groups and types and with a designated root inside. Each item contained in HDF5 format is an *HDF5 information item*. Inside the information set, the annotated associations define *HDF5 datasets*, *HDF5 groups*, *HDF5 links* and custom *HDF5 data objects* with optional annotations – *HDF5 attributes*. HDF5 datasets are array variables of homogeneous data type. They are logically shaped in a multidimensional array, which is defined by its rank (number of dimensions), along with the current and maximum extent of their respective dimensions. The array elements can be either of HDF5 predefined data type or custom HDF5 object type. HDF5 object type is a user-defined complex data type, composed from HDF5 data types. The available types include: integer, floating-point, string, bit-field, opaque (data type whose concrete data structure is not defined), compound, reference, enumeration, variable-length sequence and array. HDF5 groups represent a similar concept as directories present in file systems and define an association between zero or more HDF5 information items. Analogously, HDF5 links allow destination referencing between HDF5 information items residing inside HDF5 file. Finally, HDF5 attributes allow annotation of individual HDF5 datasets, groups and datatype objects with extra meta-data.

Typical applications of HDF format are found in research areas generating and operating on large volumes of multi-dimensional data. As an example, NASA adopted HDF5 format to support the NASA Earth Observing System (EOS) [17] [88] science data. Similarly, in bioinformatics, to efficiently deal with large volumes of genomic data a BioHDF5 [135] format was developed as an alternative to SAM/BAM format. It proposes storing multiple files inside one container, such as SAM/BAM files, optionally with their corresponding reference sequence files. The data inside a container are divided into three groups. All the DNA sequence data are stored in *sequences group* as a table with bases represented as columns and reads represented as rows. Reference sequences – stored in *references group* – are represented as very large concatenated character strings with an index to access individual sequences. Alignment descriptions – stored in *alignments group* – are

[17] https://earthdata.nasa.gov/standards/hdf-eos5

stored decomposed into a number of separate subgroups, corresponding to the fields names. Such decomposition allows for selective access to the contained data. The representation of the data is further compressed using standard HDF5 library lightweight compression filters such as ZLIB or SZIP.

Unfortunately, BioHDF5, although providing a very flexible approach for storing and representing the data of any format, introduces a significant overhead for representing the alignment data. The compressed size and compression performance is inferior compared to BAM format. The stored alignments occupy more space than compressed BAM files with longer compression (import to HDF5 container) and decompression (export from HDF5 to SAM file) times.

Another solution based on HDF5 format is SeqDB [75] – it uses HDF5 to store and compress FASTQ files. SeqDB packs both sequence and quality data together into one byte and compresses them using an external filter BloscLZ[18]. Read identifiers are compressed also using BlockLZ. In this way, SeqDB provides very high compression and decompression throughput, but at the cost of low compression ratio compared to the commonly used gzip.

2.3.4 GOBY – a data management framework and data format

Goby [24] is a genomic data management framework designed to facilitate the implementations of efficient genomic data analysis pipelines. It provides an efficient storage of FASTQ and SAM files (with the focus on the latter) using its own format, where the input genomic file format is defined in special schema. Such an approach already allowed for the integration of the Goby framework with such third-party tools as the sequence aligner BWA [110], the interactive data visualization tool IGV [173], the data analysis and management platform GobyWeb[45] and more.

Genomic data formats (FASTQ and SAM) in Goby are represented using *Protocol Buffers*[19]. Protocol Buffers is a data interchange format developed at Google, which is a language-neutral, platform-neutral, flexible mechanism for serializing structured data (stored as messages) and which

[18] http://www.blosc.org/
[19] https://developers.google.com/protocol-buffers/

automatizes reading and writing of the data. The record type (called as *message*) is defined using a domain-specific *Protocol Buffers Interface Description Language*. By using Protocol Buffers and by providing an explicit definition of the message type, Goby eliminates the need for implementing manual parsing and serialization mechanisms of the message data type. Moreover, in order to support storing large datasets (as present in genomics), the authors extended the Protocol Buffers message storage back-end by introducing *Goby Large Collection Storage Protocol* (GLCSP). GLCSP allows one to store collections of serialized messages by Protocol Buffers compacted in blocks of specified size. Additionally, GLCSP provides functionality for random access, which is performed on a per-block basis.

Goby provides a set of 4 different compression techniques to compress the data packed inside blocks. In the *separate field encoding* technique the message's content is decoupled and the fields of the same type are compressed together. The *field modeling* technique allows for expression of the value of one field as a function of other fields and constants, reducing the data overhead. The *template compression* technique detects whether a subset of the message (the template) repeats in the currently processed block. Then, it stores the template with the number of its repetition – a similar technique to run-length-encoding. Finally, the *domain modeling* technique allows for advanced representation of the message in compressed blocks by, e.g., replacing the message's selected content with links pointing to a different message sharing the same content. In this way, when decoding a message, previously decoded messages can be used to re-generate "on-the-fly" some of the current message's content.

Apart from compression techniques, a set of compression codecs have been implemented to compress the data. A general purpose gzip or bzip2 codec can be used to compress the data serialized directly by Protocol Buffers in blocks. The authors also implemented a specialized *Hybrid Codec* designed for compressing alignments, where each field is encoded separately which works as follows. Firstly, string values and floating point values are converted into lists of integer values. Then, the values are analyzed to create statistics per each field. Depending on the results, optional data transformations are applied on the fields, e.g., run-length encoding. The output is encoded using one of the implemented codecs, such as an arithmetic coder or minimal binary encoder (i.e., using minimum num-

ber of bits per symbol). Compressed streams are stored in a set of files, which are classified in *Tiers*. Following, files storing information from raw reads (FASTQ) can be classified as *Tier I*. Sorted and indexed alignment files (SAM) can be stored in another set of files classified as *Tier II*. Moreover, data stored in Tier II can be directly linked with the data stored in Tier I, which allows for the possible reduction of data redundancy when storing both SAM (aligned) and FASTQ (unaligned) files in a traditional file-oriented way. Unfortunately, due to complex architecture and different internal record representation of SAM alignment (and FASTQ read), the offered data processing speed is relatively slow (e.g., the resources are spent on format transcoding), hence, limiting its practical usage or as a possible replacement of BAM format.

2.3.5 ADAM – a cloud-oriented genomic data processing ecosystem

ADAM [136, 150] defines an alternative approach with respect to classical pipelines. It usually consists of a set of applications run sequentially and which uses different intermediate file formats for storing and exchanging the data. ADAM is an open-source programming framework, a set of application programming interfaces (APIs), and a set of data formats for genomic data processing in the cloud primarily focused on efficient data processing and scalability. It has been built on top of open-source Apache Spark [218] and Apache Hadoop [208] big data processing frameworks, which allow one to perform distributed computations on the data. Additionally, it provides a custom data format with the focus on data interoperability between different genomic data processing applications. Sample applications which have been developed on top of ADAM are variant caller Avocado [149] and Mango[20] – a scalable genome browser.

To achieve interopability between different applications ADAM utilizes a set of open-source solutions such as Apache Avro[21] and Apache Parquet[22]. Apache Avro is a data serialization system which is also used to define genomic data formats (similar to Google Protocol Buffers, used by Goby). Apache Parquet is a columnar data storage format, which shares underly-

[20] https://github.com/bigdatagenomics/mango
[21] https://avro.apache.org/
[22] https://parquet.apache.org/

ing ideas with column-oriented databases. It allows the data to be stored as a single file or as a distributed one, i.e., one large file partitioned into multiple smaller chunks, each stored on different nodes in the network distributed filesystem (such as Hadoop Distributed File System (HDFS)). Since the Apache Parquet format is mostly focused on providing rapid data access for performing efficient analyses over the data, it implements only some lightweight encoding schemes, such as run-length encoding or differential encoding. It also provides some lightweight compression codecs, such as gzip and Snappy[23]. Thanks to such decisions, different genomic record types can be easily defined in ADAM and stored in the filesystem without the need for developing specialized tools to handle different file formats.

As proof-of-concept, an implementation of SAM format in ADAM allowed to achieve resulting output files up to 25% smaller than their corresponding representation in BAM format. The reduction in data storage size seems promising, however, the main obstacle of ADAM is that it is heavily dependent on Hadoop ecosystem and its computational and data storage architecture. Therefore, its integration with current genomic data processing workflows can be very difficult.

2.4 Storage and compression of long-reads data

2.4.1 Context

The third generation single-molecule sequencing platforms can output reads in length of tens of kilobases (and more). The technology, potentially, allows for a lot of opportunities for discovering large structural variations present in genomes, which are very difficult or impossible to assess when using short reads. However, single-molecule sequencing technologies are still in development and the accuracy of generated long sequencing reads data is still inferior compared with the ones generated by second-generation short reads platforms (see Table 1.1 for the comparison between different platforms). With the continuous improvements of sequencing technology, base calling algorithms and chemistry, the format and the content of generated sequencing data is also susceptible to changes.

[23] http://google.github.io/snappy/

Currently, TGS reads are accompanied by auxiliary data (such as raw signal data) and meta-information (such as information related to a used instrument), which can be helpful to analyze and possibly correct the sequencing errors and to improve the accuracy of the analyses. It is clear that FASTQ format capabilities with respect to storing auxiliary data and meta-information are very limited. Therefore, a constantly improving technology requires a special, resilient format for storing the raw data coming from sequencers. In the case of Pacific Biosciences and Oxford Nanopore Technologies, which are the leading third generation sequencing platforms, the sequencing data are stored in HDF5 format. HDF5 format provides a high degree of flexibility and allows for easy incorporation of heterogeneous data with the DNA reads. Such a decision also requires adopting existing applications and data processing pipelines to be able to work with the new format or it requires developing converters to existing formats (such as FASTQ or SAM) which will always have a controlled degree of information loss.

Moreover, data compression techniques, which have been successfully applied to short-reads data, may not provide satisfactory compression results when applied to long-reads data generated by third generation sequencing platforms. First of all, the primary application of TGS data is in *de novo* sequence assembly experiments rather than re-sequencing. The data generated by TGS platforms is also characterized by a considerably lower coverage than the short-reads data, as the reads can span over longer genomic regions and fewer are required to be sequenced (see Eqn. 1.1 and 1.2 for the relation between the genome length, the length of the reads and required number of reads to be sequenced to adequately cover the genome). Hence, for example, the read-reordering techniques, used when compressing raw FASTQ files aiming to exploit the high redundancy present in short DNA sequences, may not be a good choice here. In addition, compared to the data generated by NGS platforms, the TGS data is characterized by a significantly higher degree of noise, primarily due to sequencing errors. As a consequence, the majority of the space is occupied by the auxiliary data, which can be used to possibly improve the quality of the obtained DNA sequences. Therefore, the commonly used short-reads sequence matching, mapping, assembly and compression methods would require significant modifications in order to be efficiently applied to long-reads data.

2.4.2 Pacific Biosciences platform

Data generated by Pacific Biosciences sequencers are stored in a set of HDF5 files – one main *bas.h*5 and three *bax.h*5 files [156]. Inside each of them, the data layout is structured in a hierarchical way, which allows for a comprehensive and consistent representation of the data produced by a sequencer. The main file is primarily used as an index file and contains mapping information necessary to access the data generated by each ZMW well (see Section 1.1.2), which are stored in *bax.h*5 files. These contain raw sequencing data such as signal pulses classified by a base caller with their timestamps (called *events*), probabilities of base calling errors, probabilities of INDELs occurrences, or whether the read passed filtering, etc. In addition to the events data, these files contain technical information about the sequencer such as ZMW wells parameters, change of their state in time, their associated metrics and others. As the format is constantly evolving, no specific compression methods have been defined – the data inside HDF5 containers are stored using default HDF5 format settings.

Since the initial release of the first *PacBio RS I* sequencer in 2010, the platform, available chemistry and protocols have undergone a significant number of improvements, reaching as of 2017 a high level of maturity. A large number of bioinformatics tools to perform data processing and analysis have been developed by Pacific Biosciences[24] and published as open-source. This step helped a wider adoption of the platform and allowed for development of novel methods for working with long reads data. Moreover, specific extensions for PacBio-generated data have been added to SAM/BAM format specification. As a result, long reads data (both mapped and unmapped) can be converted to SAM alignments representation, but at the cost of stripping a large amount of auxiliary data. Such a move allows long reads to be potentially used with pipelines and tools, which primarily work only with SAM format as an input.

2.4.3 Oxford Nanopore Technology platform

Similarly, as in the approach taken by Pacific Biosciences, data generated by Oxford Nanopore Technology sequencers are stored as a set of files

[24] http://pacbiodevnet.com/

in HDF5 format, namely – *FAST5* format and with corresponding *.fast5* file extension. However, the main difference is that one FAST5 file stores the data from only one sequenced DNA molecule. Therefore, a huge number of files will be produced by each sequencing run, where the total amount of files can vary from thousands to millions of files and depends on the number of sequenced DNA molecules. Such an approach allows for instant and continuous analysis of the data while sequencing, and continuously updating the analysis results once the new molecules have been sequenced. This also has a direct impact on how the sequencing data is represented internally.

In case of FAST5 format, the raw sequencing data stored inside the files are also structured in a hierarchical way, following the same schema, which allows for a comprehensive representation of the data coming from sequencing experiments. The raw sequencing data primarily consists of a time-series of nanopore translocations (events) with its corresponding detected signal strength and nucleotide probabilities. Depending on the experiment setup, the FAST5 files may also include the results from the base caller both for 1D and 2D configurations (see Section 1.1.2) with additional post-processing information and summary. Moreover, since the generated data can be directly processed in the cloud, in order to help with identification of the reads and coordination of the analysis, each FAST5 file contains unique tracking information of the read, the originating pore, the instrument and information about the used pipeline (with tools). Therefore, a large degree of meta-information is shared between reads coming from the same experiment.

The Oxford Nanopore Technology platform and available chemistry are still in their infancy, but are constantly being improved. This directly affects the data storage format, which is subject to change with the possibility of introducing improved chemistry or pore technology. Despite the difficulties around format, a number of bioinformatics tools have been developed to support the analysis of the produced data. Moreover, in order to overcome challenges with constantly evolving formats, a number of "wrapper" tools have been developed, which allow conversion from reads stored in FAST5 to FASTA or FASTQ, such as poretools [121]. This allows for further adoption of the long reads produced by Oxford Nanopore Technologies by a broader range of bioinformatic tools and pipelines.

OBJECTIVES

Sequencing technologies are continuously improving. Year after year the offered throughput greatly increases, while operational costs become progressively smaller. As has been shown in Chapter 1, efficient storage and processing of genomic information poses a lot of challenges. Data can be generated by multiple technologies, and its typical size ranges from tens to hundreds of gigabytes per single sequencing experiment. Moreover, downstream analysis generates multiple, large intermediate files that have different data formats. Therefore pipelines processing genomic data are very I/O intensive, which is currently one of the major computational bottlenecks of analyses. Apart from difficulties in processing such large volumes of data, efficiently sharing and querying data sets that are hundreds of gigabytes in size, or several terabytes, is even more challenging.

As a result, as described in Chapter 2, a number of compression methods to store data in both FASTQ and SAM format have been developed. Unfortunately the majority of compressors developed so far focus on maximizing the compression ratio and do not pay attention to the speed of data processing or reliability, which limits their practical usability within pipelines for genomic applications. In addition, the majority of raw or aligned data is still stored in gzip-compressed form (for files in FASTQ files) or using the gzip-based BZGF compression method (for files in BAM format), which do not provide satisfactory compression results.

Therefore, the research presented in this thesis focuses on methods for efficiently compressing and storing the data generated by high-throughput sequencing technologies. Not only would such methods be easier to incorporate into current genomic data analysis pipelines; they would also potentially eliminate the efficiency bottlenecks present in such pipelines, and enable a more effective sharing of HTS data.

In particular, our main objectives have been:

1. As a starting point, explore genomic data compression methods and develop a general, high-performance compressor for FASTQ files, which can serve as a good alternative to the most commonly used gzip.

2. Explore and develop methods allowing researchers to efficiently compress short-read data generated by sequencing experiments, especially those having high coverage. Such methods should exploit the significant sequence redundancy present in the data, providing a high compression ratio without using any external reference sequences.

3. Design and develop a flexible, configurable and format-independent solution which enables efficient storage and retrieval of genomic data no matter how the latter has been originally stored. It should be possible to use the solution as a replacement for any specific genomic file format and to perform simple queries on the compressed data, allowing the user to easily prototype new compression methods suitable for HTS information. As a proof of concept, the solution should be able to generate a range of different compressors for short-read data, be it originally represented either as raw reads in FASTQ format or alignments in SAM format.

4. Finally, briefly explore approaches allowing one to perform lossy compression of HTS data, in order to achieve additional savings in the amount of storage required. Information loss should happen in a controlled way, preserving the quality of the biological results that can be obtained from the data.

CHAPTER 3

RESULTS

In this chapter we will present results of the solutions for compressing and storing HTS data, which we developed during our research. They cover a broad range of use-cases for genomic data storage and were extensively tested, by comparing them with the state-of-the-art techniques, which were presented in Chapter 2.

Firstly, we will present DSRC2 [174], a general, high-performance FASTQ format compressor. It is based on the ideas of the previously published DSRC [42] compressor, implementing a number of improvements and novel compression methods.

Because of the limitations of the general methods used to compress the short-read datasets at high coverage; we will then shift our attention to a different approach. We will present ORCOM [66], a proof-of-concept DNA-only compressor for short-read data, which focuses on providing a maximum compression ratio of short DNA reads. It uses a novel read reordering method, which aims to exploit high DNA sequence redundancy present in the datasets coming from deep sequencing experiments.

In the next step, we will present FaStore, a full FASTQ format compressor based on the ideas presented in ORCOM and DSRC2. It extends the DNA compression method of ORCOM, by introducing additional reads reordering steps and by applying a sequence-assembly step in order to further improve the DNA compression ratio. Moreover, it also provides a set of different compression methods for storing read identifiers and quality scores in a lossy way.

Then, in contrast to specialized format-specific solutions, we will present CARGO [175], a framework and data format that allows one to semi-auto-

matically generate compressors that are not tied to any specific HTS data format. This allows for fast prototyping of HTS data storage solutions, where the record data type along with possible additional data parsing, transformation and querying operations are defined by the user. Moreover, all the datasets of heterogeneous types are stored in a compressed form in configurable CARGO containers.

Thorough our research we have been comparing our methods with those others of constantly improving state-of-the-art and using different input datasets. Therefore, in the next step, we will briefly compare compression results obtained by all our specialized (DSRC2, FASTORE) and semi-automatically generated (CARGO-based) compressors, while at the same time comparing them to the current state-of-the-art and using a concise input dataset.

Finally, all the developed solutions along with their compression results will be discussed in Chapter 4, which also provides a look at some novel approaches and future directions.

3.1 DSRC2 – Industry-oriented compression of FASTQ files

Roguski L, Deorowicz S. DSRC 2--Industry-oriented compression of FASTQ files. Bioinformatics. 2014 Aug 1;30(15):2213–5. DOI: 10.1093/bioinformatics/btu208

3.2 ORCOM – Disk-based compression of data from genome sequencing

Grabowski S, Deorowicz S, Roguski Ł. Disk-based compression of data from genome sequencing. Bioinformatics. 2015 May 1;31(9):1389–95. DOI: 10.1093/bioinformatics/btu844

3.3 FaStore – A space-saving solution for long-term storing of raw sequencing data

Roguski Ł, Ochoa I, Hernaez M, Deorowicz S. FaStore – A space-saving solution for long-term storing of raw sequencing data.
(Manuscript in preparation)

FaStore – A space-saving solution for long-term storing of raw sequencing data

Łukasz Roguski[1,2] Idoia Ochoa[3] Mikel Hernaez[4] Sebastian Deorowicz[5]

Abstract

The affordability of DNA sequencing has led to producing unprecedented volumes of raw sequencing data. These data must be stored, processed, and transmitted, which poses significant challenges. To facilitate this effort, we introduce FaStore, a specialized compressor for FASTQ files. The proposed algorithm does not use any reference sequences for compression, and permits the user to choose from several lossy modes to improve the overall compression ratio, depending on the needs. We demonstrate through extensive simulations that FaStore achieves a significant improvement in compression ratio than that of previously proposed algorithms for this task. In addition, we perform an analysis of the effect that the different lossy modes have on variant calling, the most widely used application for clinical decision making, especially important in precision medicine. We show that lossy compression can offer significant compression gains, while preserving the essential genomic information and without affecting the performance of variant calling.

[Supplementary material is available for this article.]

[1]CNAG-CRG, Centro Nacional de Análisis Genómico (CNAG) - Centre for Genomic Regulation (CRG), Barcelona Institute of Science and Technology (BIST), Barcelona, Spain

[2]Universitat Pompeu Fabra (UPF), Barcelona, Spain

[3]Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA.

[4]Carl R. Woese Institute for Genomic Biology, University of Illinois at Urbana-Champaign, IL, USA.

[5]Institute of Informatics, Faculty of Automatic Control, Electronics and Computer Science, Silesian University of Technology, Gliwice, Poland.

Correspondence should be addressed to sebastian.deorowicz@polsl.pl

Introduction

The growing interest in applications of genome sequencing, together with their dropping costs and continuous improvements in sequencing technologies, has led to the generation of unprecedented volumes of increasingly large and ubiquitous raw genomic data sets (Stephens et al. 2015). These data are characterized by highly-distributed acquisition, massive storage requirements, and large distribution bandwidth. For example, the 1000 Genomes Project (Clark et al. 2012) requires 260 Terabytes in storage size (both for raw sequences, alignments, and variant calls). Moreover, the 100,000 Genomes Project (2017) has already exceeded 21 Petabytes in size. Such flood of data hampers the efficiency of data analysis protocols, limits efficient data sharing, and generates vast costs for data storage and IT infrastructure (Schadt et al. 2010). This situation calls for state-of-the-art, efficient compressed representations of the raw genomic data, that can not only alleviate the storage requirements, but also facilitate the exchange and dissemination of these data.

The raw high-throughput sequencing data are primarily stored in FASTQ files (Cock et al. 2010), which are usually considered as the input for the genomic data processing and analysis pipelines. A FASTQ file can be perceived as a collection of "reads", each containing a sequence of nucleotides (generally referred to as the read), the quality score sequence that indicates the reliability of each base in a read, and the identifier, which usually contains the information about the sequencing instrument, flow cell coordinates, etc. Millions of such reads are produced in a single sequencing run. As a result, storing raw reads coming from a whole-genome sequencing of a single human individual can easily exceed 200 GB in an uncompressed form.

One of the most common protocols using DNA sequencing is the analysis of data from re-sequencing experiments, assessing the variants against a known reference genome. In order to assess these variants, the raw reads present in the FASTQ file are firstly mapped to a reference sequence of the analyzed specie. This process results in the generation of aligned reads in SAM format (Li et al. 2009), which contains the same information as the FASTQ file, together with the alignment information for each read, and with possible additional information provided by the mapper. Hence, the size of the intermediate data stored in SAM files is even larger than the input FASTQ files. Following a number of post-processing steps, the aligned reads are finally used to perform variant calling against the reference genome. The results of variant calling, following a number of additional data cleaning and validation steps, can be used as an entry point for further clinical analyses. However, one needs to be note that, the size of the resulting file is orders of magnitude smaller than the input raw reads stored in FASTQ format or the intermediate alignment data

Table 1: Datasets used in the experiments. WGS is for Whole Genome Sequencing. WES is for Whole Exome Sequencing.

Data set	Species	Experiment type	Sequencer	FASTQ size [GB]	Read len. [bp]	No. reads [M]	Coverage
HS2	*H.sapiens*	WGS	Illumina GA II	336.9	100	1339.7	43
GG	*G.gallus*	WGS	Illumina GA II	115.9	100	347.3	32
HSX	*H.sapiens*	WGS	Illumina HiSeq X	292.1	151	819.1	39
WGS-14	*H.sapiens*	WGS	Illumina HiSeq 2000	115.6	101	447.1	14
WGS-42	*H.sapiens*	WGS	Illumina HiSeq 2000	337.3	101	1304.5	42
CE	*C.elegans*	WGS	Illumina GA II	17.6	100	67.6	66
WEX	*H.sapiens*	WES	Illumina HiSeq 2500	44.7	126	150.4	∼220
WGS-235	*H.sapiens*	WGS	Illumina HiSeq 2000	1888.4	101	7363.7	235

stored in SAM format.

Although the data contained in the FASTQ file could potentially be recovered from the corresponding SAM file (or its compressed version), there may be cases in which this is impossible, depending on the used protocols for data analysis. For example, when performing post-processing of the alignments, some of the information can be discarded, such as the reads that failed to align to the reference genome or which were considered as duplicates. Moreover, the reference genome used for storing alignments in a compressed form may no longer be accessible during decompression. In case of performing other types of experiments, such as metagenomics, there may even not be a reference genome at all align the reads, as different organisms present in the sequenced sample are generally unknown prior to the analysis.

Therefore, we focus on the compression of the information stored in FASTQ format, i.e., the raw data containing only the DNA (nucleotide) sequences, read identifiers, and quality scores, since they represent a minimal subset of the data required for future reproduction of the analyses performed on the sequenced data. The existing specialized solutions for FASTQ files compression, extensively examined in (Numanagić et al. 2016), obtain significant compression gains over general compression tools such as gzip. However, in practice, gzip is still the *de facto* choice, mainly due to its popularity and stability. It seems that the community has not decided yet that the assets of specialized FASTQ compressors are worth some complications that may appear when moving to a different format of storage. Also, the fact that currently a number of good dedicated compressors are available does not make the right choice simple.

In this paper we propose FaStore, a new compressor for FASTQ files that, among others, can be used for long-term archival of raw genomic information and efficient sharing, especially with limited internet

bandwidth. FaStore inherits the assets of our previous attempts in the field, especially DSRC (Deorowicz & Grabowski 2011; Roguski & Deorowicz 2014), ORCOM (Grabowski et al. 2015), and QVZ (Malysa et al. 2015; Hernaez et al. 2016). The proposed compressor offers both lossless and lossy compression modes (the latter only for the quality scores and the identifiers), and does not use any external reference sequences.

We show that FaStore significantly outperforms the existing compressors in the lossless mode. We, however, advocate for the lossy option when suitable, which, as presented, gives much better shrinkage of the input files with negligible differences in variant calling. Although we emphasized the variant calling as the most common use-case for precision medicine, the analyses performed using FASTQ files are not only limited to variant calling, but also used for, e.g., gene expression analysis, assembly or metagenomics.

Results

Lossless and lossy compression of sequencing data

FaStore is a compressor for FASTQ files produced by next-generation sequencing platforms, which are characterized by generating massive amounts of short reads data and with a relatively low sequencing error rate. Moreover, the generated sequence data contains a high degree of sequence redundancy, especially coming from deep sequencing experiments. Hence, FaStore includes several compression modes to account for the different needs that the users may have.

In particular, parts of the data can be optionally discarded or quantized for additional file size reduction. The only strict requirement is that the DNA sequences are stored lossless. Due to the different nature of components of reads (DNA sequences, quality scores, identifiers) FaStore uses different specialized compression techniques for each of them (see **Supplementary Methods** for details). Moreover, as the sequencing data can be generated from a library in a single- or paired-end configuration, FaStore provides different techniques to handle both cases, guaranteeing that the pairing information between the reads is preserved (when available). In the following, when clear from the context, DNA sequences may be also referred to as reads.

Compression of the DNA sequences is done without the use of any external reference sequences. Relying on a reference sequence for compression requires the availability of the same reference at the time of decompression, which may be no longer accessible, thus making the compressed DNA sequences unrecoverable. Therefore, this design choice guarantees a perfect reconstruction of the sequences.

Due to the nature of the sequencing process as a whole, the reads are generated in a random order (Kavak et al. 2015) and thus the initial ordering of the DNA sequences in the output file carries no

significant information. With this in mind, FaStore exploits the existing high sequence similarity on the global scope, by reordering the reads. In particular, the reads are clustered in a manner such that reads coming from neighboring positions in the sequenced genome are likely to belong to the same cluster. When possible, within these clusters the reads are assembled into contigs, so that they can be stored relatively to the consensus sequences. Alternatively, a read can be also stored relatively to other reads belonging to the same cluster, or "as it is", depending on the degree of similarity with the other reads in the cluster.

As a trade-off between the computation time to cluster the reads and the attained compression ratio, FaStore offers two modes of operation, denoted by C0 (fast) and C1 (default). When compressing in C1 mode, more clustering steps between the reads are performed to obtain a higher compression ratio. However, the decompression speed is similar for both modes.

While the sequence redundancy present in the data can be efficiently reduced, the quality scores have proven more difficult to compress (Bonfield & Mahoney 2013). They are characterized by a high entropy with a significant level of noise. In addition, preserving precise quality scores is often unnecessary (i.e., some distortion is generally acceptable), in that no cost is incurred on the subsequent analyses performed on the data (Ochoa et al. 2016; Yu et al. 2015).Therefore, FaStore offers various types of lossy quality scores compression modes alongside the lossless. In particular, FaStore includes Illumina 8-level binning (Illumina 2014), a custom binary thresholding, and an adaptive scheme based on QVZ (Malysa et al. 2015; Hernaez et al. 2016).

Illumina binning maps the resolution of quality scores just to 8 distinct bins. The binary thresholding quantizes the quality scores according to the user-provided threshold, i.e. it sets the quality values below the threshold to $q_{\min}$, and those above to $q_{\max}$. QVZ quantizes the quality scores so as to minimize the rate allocation (number of bits per quality score) while satisfying a distortion constraint. To design the appropriate quantizers, QVZ relies on computing the statistics of the quality scores prior to compression. FaStore gathers these statistics while clustering the reads, and thus there is almost no added computational cost. The quantizers are generated at the end of reads clustering and one global codebook per dataset is used. For lossless compression, FaStore uses QVZ in lossless mode.

The read identifiers are initially tokenized to make use of the fact that some appearing tokens are constant, some are from a small dictionary, etc. Moreover, since the complete identifier string is usually unnecessary in practice, FaStore also offers a lossy mode for storing them , either by removing the comments (as do mappers by default) or by completely skipping them. One needs to note that in FASTQ format the representation of pairing information between the reads is not clearly defined, i.e., it can be carried by the read identifiers (i.e. a pair of reads share the same identifier) or at the file-level (i.e. the reads reside on the

same lines in two FASTQ files or are stored interleaved in a single file). Therefore, FaStore preserves this information with the sequences, allowing the identifiers to be removed and generating unique ones per pair of reads when decompressing.

Compression factors

For evaluation of the proposed compressor FaStore, we primarily used a subset of data sets benchmarked already in (Numanagić et al. 2016; Grabowski et al. 2015; Benoit et al. 2015) alongside new ones characterized by a high-sequence coverage. The details of the employed datasets are summarized in Table 1 (see **Supplementary Methods** for the download links). The collection consists of 7 large sets of FASTQ files (top 7 rows) and one vast dataset (bottom row), and it includes sequencing data from the *H. sapiens*, *G. gallus* and *C. elegans* species, generated in a paired-end configuration. We compared the performance of FaStore with that of gzip (the *de facto* current standard in storage of sequencing data) and the top FASTQ compressors according to (Numanagić et al. 2016), which are: DSRC 2 (Roguski & Deorowicz 2014), Fqzcomp (Bonfield & Mahoney 2013), Leon (Benoit et al. 2015), Quip (Jones et al. 2012), and Scalce (Hach et al. 2012). We also tested the top DNA-only compressors according to (Numanagić et al. 2016), which are: ORCOM (Grabowski et al. 2015), Mince (Patro & Kingsford 2015), and BEETL (Cox et al. 2012) Unfortunately, for several data sets, Mince run out of available memory (128 GB) and BEETL failed to process some in 48 hours time, so both of them are not included in our analysis.

Figures 1a–d show the average compression factor and compression/decompression speeds for the complete collection. The applications were run using 8 processing threads, when applicable. Due to space constraints and ease of exposition, we present results running applications in maximum compression mode. Moreover, we provide results for the main lossy settings (denoted by *reduced*, *lossy* and *max*), and refer the reader to the **Supplementary Worksheet W1** for an extensive evaluation of the whole range of lossy modes provided by FaStore alongside other tested applications and in different modes.

As one can notice, FaStore in the *lossless* mode (preserving all the input data) achieves significantly better compression factors than the competitors executed in maximal compression modes. In particular, the compression gains with respect to the results achieved by the best competitor (i.e., Fqzcomp for all datasets except for dataset HSX where Leon outperforms Fqzcomp), range from 7.6% to 20.3%. For example, for *H. sapiens* datasets HS2 and WGS-42, this corresponds to more than 10 GB of savings in both cases.

Although the *lossless* mode is used by default, we strongly recommend considering some of the provided lossy modes. By discarding parts of the read identifiers and reducing resolution of the quality scores one can achieve significant savings in storage space. For example, in the *reduced* mode, the compressed

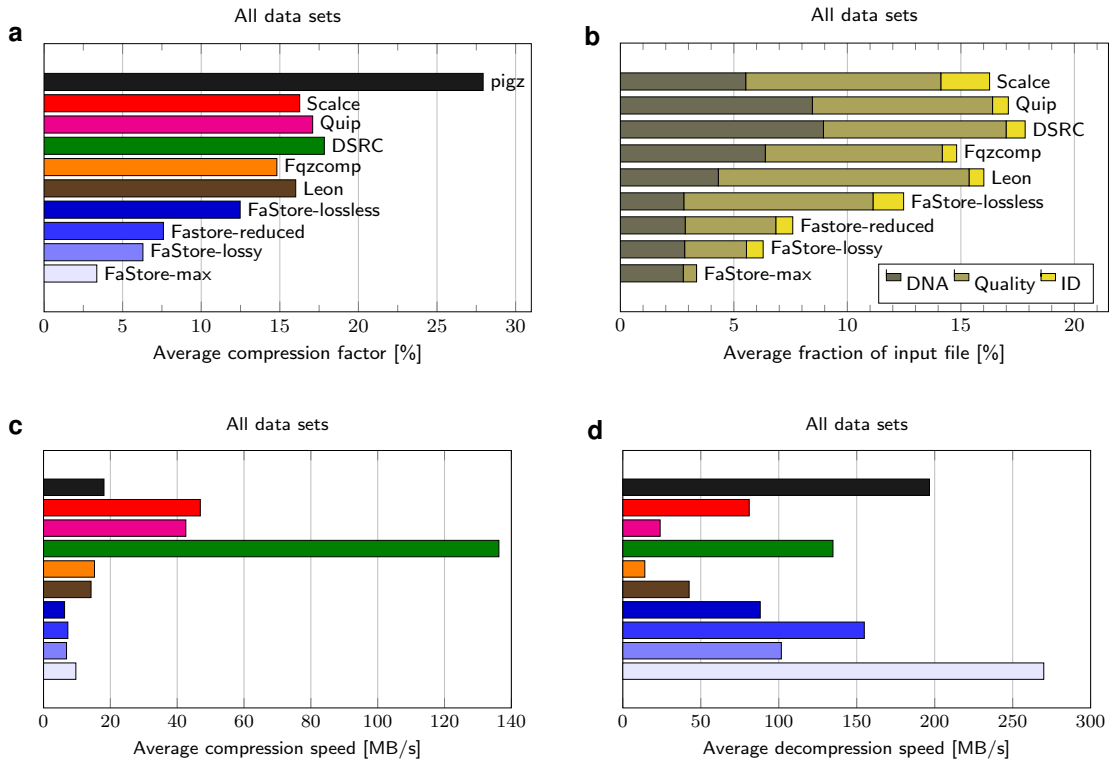

Figure 1: Compression results. (**a**) Average compression factors in % (compressed size divided by original size) for all examined datasets. (**b**) Average compression factors in % (compressed size divided by original size) for all examined datasets, divided by the different components: DNA bases, quality values, and IDs. (**c**) Average compression speeds for all examined datasets. (**d**) Average decompression speeds for all examined datasets. The sub-figures a–c share the same legend. pigz is multithreaded variant of gzip (same compression ratios, but faster processing).

size is about 60% of that of the *lossless* mode. The improvement is possible thanks to applying Illumina 8-level binning (Illumina 2014) and removing the comments from the identifiers (which, in some cases, leads to storing only a library name and a read number). As the identifiers are usually truncated in this way by mappers when producing SAM files and the 8-level binning becomes a default option in modern sequencers (although the actual mapping of the values can depend on the internal configuration of the instrument), this setting seems to be a reasonable choice.

Even better results (approx. half the size of the *lossless* mode) is possible when QVZ with distortion level 2 is applied (*lossy* mode). Nevertheless, the best compression factor (about a quarter of what was obtained in the *lossless* mode) is achievable when the identifiers are removed (only the pairing information

between the reads is preserved) and the binary thresholding for the quality values is applied (*max* mode).

Figure 1b shows fractions of archives consumed by various components: DNA bases, quality values, and identifiers. There is no result for gzip as, due to the design of this compressor, it is impossible to measure the exact fraction of each component (it compresses all the data "as it is"). The results show that FaStore uses much less space to store the DNA symbols than the competitors. Since there is no loss of the bases in the lossy modes, the amounts of space necessary for DNA symbols are almost identical. However, one needs to note, that in the *lossless* mode FaStore needs more space for storing identifiers than the other competitors (except for Scalce). The reason is that after reordering the reads it is much harder to compress their identifiers, as the neighboring ones differ more than in the original ordering. Nevertheless, the compression gains in of DNA stream overshadows the loss in the identifiers stream.

The most challenging to compress are, however, the quality scores. For most compressors, when the quality scores are stored in the lossless way they require more space than the DNA sequences and identifiers considered together. Thus, applying the lossy schemes for the quality values has a remarkable impact on the total compression factor. For example, to losslessly compress *H. sapiens* dataset HS2, FaStore requires 46.3 GB of space. From those, 32.7 GB correspond to the quality scores, which can be further reduced to 9.3 GB (Illumina binning, *reduced* mode), 8.4 GB (QVZ with distortion level 2, *lossy* mode) or even 1.1 GB (binary thresholding, *max* mode). In all the cases the total size is reduced by more than 50%, to as little as 14.7 GB (with binary thresholding). Moreover, one needs to note, that this reduction in total size is computed without considering lossy compression of the identifiers, which would provide even more storage savings. Next we demonstrate that such reductions in size are possible with little effect on variant calling.

Finally, the compression speed is some drawback of our solution, as it is somewhat smaller than 10 MB/s in the default C1 mode, as it applies multiple steps of reads preprocessing in order to achieve maximum compression of DNA sequences. Nevertheless, the decompression speed is comparable to the fastest algorithms, i.e., DSRC 2 and gzip. Figure 1c–d show the compression and decompression speeds for the different methods analyzed in this paper. However, for use cases where compression speed is of uttermost importance, FaStore provides a fast mode, namely, the C0 mode. This mode trades DNA compression ratio for compression speed, while still achieving better compression ratios than the competitors (see **Supplementary Worksheet W1**). As reported, mode C0 can, on average, reduce the compression time employed by mode C1 by a factor of 5 at a cost of increasing the size needed to store the DNA bases by a factor of 1.09. Note that switching between C0 and C1 has no significant effect on compression of quality scores and read identifiers.

Impact of lossy compression of quality scores on variant calling

Next, we assess the effects of using the different lossy quality compression modes provided by FaStore (i.e., Illumina binning, binary thresholding, and QVZ) have on variant calling. Since QVZ optimizes the quantization for an average distortion level that is specified as an input parameter, for the analysis, we considered distortion levels 1, 2, 4, 8, and 16.

For the evaluation, we selected the two datasets coming from whole-genome sequencing of *H.sapiens* individual, sequenced at coverage of 14x (WGS-14) and 42x (WGS-42) (see Table 1). These datasets pertain to the same individual, namely NA12878 and were sequenced as a part of Illumina Platinum Genomes (Eberle et al. 2017) . The reason for this choice is that the National Institute for Standards and Technology (NIST) has released a high-confidence set of variants for that individual as a Genome In a Bottle (GIAB) (Zook et al. 2014) initiative. This allows us to consider this set as a "ground truth" and use it to benchmark the different lossy modes supported by FaStore. We refer the reader to the **Supplementary Methods** for a detailed description of the pipeline used for the analysis.

In what follows we will report the results on Single Nucleotide Polymorphisms (SNPs), since SNPs are easier to detect and more curated in the high-confidence reference set. Nevertheless, for completeness, results for short insertions and deletions (INDELs) are provided in the **Supplementary Worksheet W2**. The GATK Best Practices proposes to apply VQSR for semi-automatic filtering of variants, however, the use of this machine-learning-based filter is still not widely adopted and should be used with caution for single-sample analyses. Hence, here we focus on the results obtained by applying hard filtering on the called set of SNPs, and refer the reader to the **Supplementary Worksheet W2** for the results achieved by applying both modes of filtering.

The results of the analysis are presented in Figures 2a–c. For completeness, Figure 2d shows the compression factors for the WGS-42x dataset. We focus on the recall vs precision results obtained using considered lossy modes for the WGS-14 and WGS-42 datasets respectively. Firstly, it needs to be noticed that the precision is similar for both datasets, whereas the recall is much higher for WGS-42. This already suggest that the higher ($\sim 3\times$) sequence coverage plays a key role and encouraging considering lossy compression scheme according to the available coverage.

The more interesting aspects are, however, the results achieved using various lossy modes. As expected, increasing the distortion level of QVZ, being an adaptive method, reduces both the recall and the precision. However, what is interesting is that the variant calling performance applying QVZ with distortion level 1 is comparable to that of Illumina binning, with slightly better results (in recall and in precision for WGS-42x)

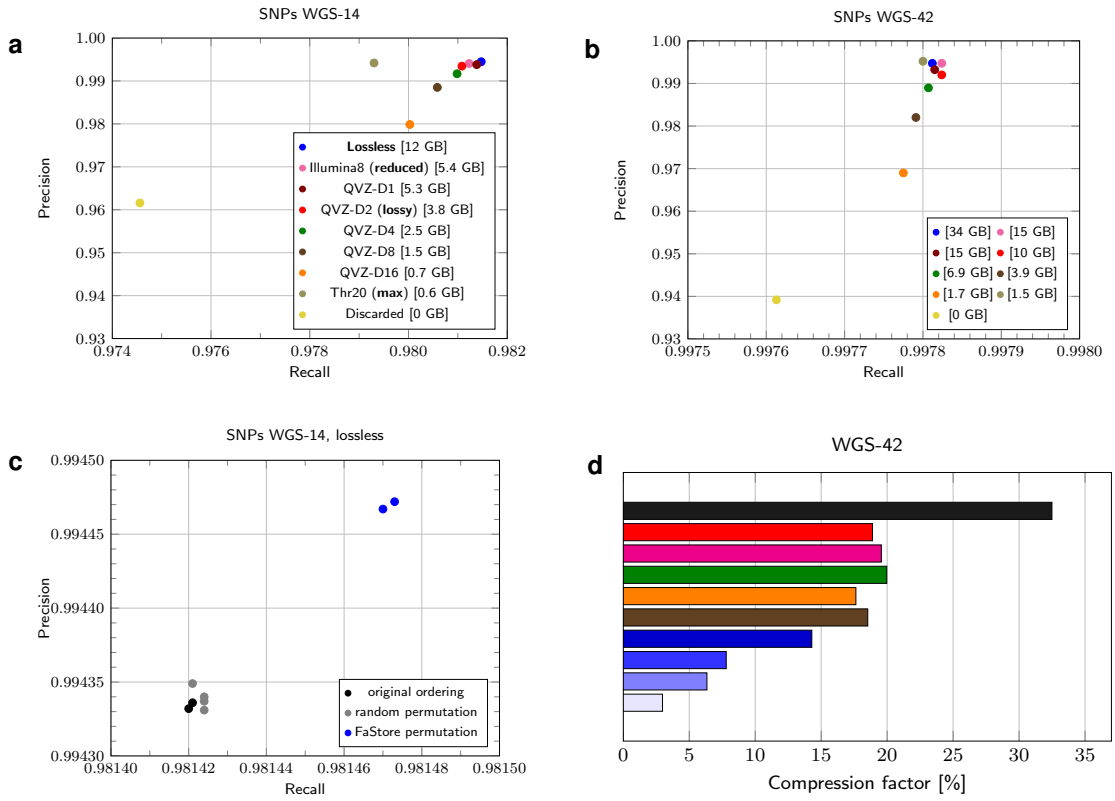

Figure 2: Compression results and variant calling analyses. (**a**) Results of variant calling for WGS-14 dataset. (**b**) Results of variant calling for WGS-42 dataset. (**c**) Influence of permutation of reads in the input collection for WGS-14 for variant calling. (**d**) Compression factors in % (compressed size divided by original size) for WGS-42 dataset.

in favor of Illumina binning. Moreover, both modes offer a similar compression factor, reducing the size of the quality scores by more than 44% compared to lossless (with slightly better result achievable by QVZ). Hence, the WGS-42 dataset could be compressed up to 7.8% of its original size (and almost a half of the space required to store it losslessy), by reducing the footprint of quality scores, removing the comments from the read identifiers, while preserving the variant calling accuracy. As a side note, in our test case, applying the lossy compression of read identifiers (as in *reduced* and *lossy* schemes), has no effect on variant calling, as the mappers usually do strip the present comments.

Nonetheless, in order to save more space, while also preserving variant calling accuracy, stronger lossy compression modes need to be considered. For that reason, QVZ applied with distortion level 2 seems to be a good trade-off between variant calling performance and compression factor. It offers comparable

performance to that obtained with the original data while reducing the size of the quality scores by more than 66%. Distortion levels above 2, although offer significant gains in compression, show a degradation on variant calling.

Quite surprisingly, for WGS-45x the results for the *max* mode are almost as good as for the lossless mode, with vast difference of the size of the compressed quality data (1.5 GB vs. 34 GB). For comparison, we also experimented with completely removing the quality data, but the results (series denoted as *discarded*) show that some information about the base quality is necessary for reliable variant calling results (at least in the examined range of coverages). Therefore, the obtained results suggest that for storing the datasets with a high coverage, keeping only the information on whether the called base is "good" or "bad" should be sufficient for achieving reliable results from variant calling.

Impact of read reordering on variant calling

As mentioned above, next-generation sequencing machines generate the DNA sequences in no particular order and the original order of the DNA sequences carries no significant information. Due to the large size nature of the produced data, several commonly used computational methods that operate on these files rely on heuristics to be able to run in a reasonable time in multi-threaded execution mode. For this reason, the reordering of reads, even if theoretically not relevant, may have some effect on variant calling. For example, authors in (Firtina et al. 2016) showed that, for some mappers, randomly shuffling the input FASTQ reads can lead to different alignment results, especially for reads originating from highly repetitive genomic regions.

Since FaStore permutes the input collection of reads, in this section we briefly examine the impact that various re-orderings have on variant calling (Fig. 2c). The goal is to analyze how FaStore shuffling may affect the performance of the variant calling, examining precision and recall for the obtained SNPs. In order to assess it we compare the variant calling results obtained using original files, randomly shuffled and reordered by FaStore. We shuffled the reads of the file four times creating four different pair-end FASTQ files. We also compressed losslessly the files using the C0 mode of FaStore, shuffling the reads as a result. Finally, we run the same experiments in two different machines using in each of them a different number of cores. As one can notice on Fig. 2c, the differences in precision and recall are negligible. Similar results can be observed for INDELs – for completeness, the results are provided in the **Supplementary Worksheet W2**.

Influence of coverage

Finally, we analyzed the compression ratio just for the DNA symbols (Fig. 3) using whole-genome sequencing data of *H.sapiens* (WGS-235 dataset) sampled at various coverages. As can be noted, for some algorithms (Scalce, Leon, Fqzcomp, FaStore) the increasing coverage leads to significant improvement in compression ratio. In case of FaStore, the advantage is more than 2-fold over the competitors. When testing read-reordering algorithms (FaStore and Scalce), we also added a series of values in which the reads were compressed as single-end (i.e., the pairing information was lost). For FaStore this leaded to further savings in storage space, giving about 2.6 times better compression ratios.

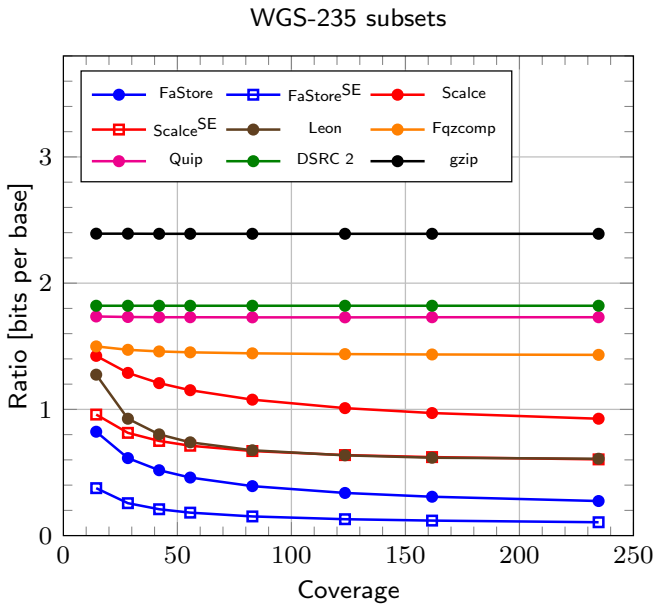

Figure 3: Compression ratio for storing only DNA symbols (bits used to encode a single base) for *H.sapiens* sampled at various coverages (WGS-235 subsets). Superscript SE stands for Single-End.

Discussion

The storage and transfer of huge files containing raw sequencing data has become a real challenge. Years ago the popular general-purpose compressor gzip was applied to reduce file sizes by about 3 times, with significant gains in cost of storage and speed of transfer. Unfortunately, in modern times much more is necessary. Our proposed compressor, FaStore, is designed to achieve excellent compression factors, i.e., about 3 times better than gzip and significantly better than the existing specialized FASTQ compressors. In

addition, FaStore offers several compression modes for the quality scores and the read identifiers, which result in significant compression gains.

We strongly suggest the community to consider resignation from storage of all the raw sequenced data. As we presented, together with the increasing sequencing throughput and the dropping costs of sequencing reflected in higher coverages for the smaller prices, the high resolution of quality values seems to be unnecessary. An important stage in this direction was made by Illumina, which has already reduced the resolution of the available quality scores to only 8 values in some of their newest sequencers. We show that similar variant calling results could be obtained when even more reduction of the quality stream is applied. For sufficiently large coverage it seems to be enough to provide just a binary information about each base telling whether it is "good" or "bad".

To imagine the possible gains in reduction of cost thanks to the lossy approaches let us say that the FASTQ files for *H. sapiens* sequenced at 42-fold coverage in the paired-end mode could consume as little as 10 Gigabytes (in *max* mode), which can be compared to 110 Gigabyte of gzipped FASTQ files. Finally, for both datasets the quality of variant calling is almost the same.

Methods

Compression workflow

In FaStore, the compression workflow has been designed as a multi-step process, trying to exploit the high sequence redundancy present in the sequencing data. It consists of: (1) reads clustering, (2) optional reads re-clustering and (3) reads compression stages, where each stage is divided into multiple smaller steps. In this section only a general overview of the compression workflow is provided – a detailed description of the methods used to compress DNA sequences, quality scores and read identifiers can be found in the **Supplementary Methods**. The workflow is depicted on Fig. 4 and can be briefly described as follows.

Reads clustering

Reads clustering stage is a 2-step process, consisting of reads binning (Fig. 4B) and reads matching (Fig. 4C). During binning, for each read from the input FASTQ file(s) (Fig. 4A), its sequence signature (i.e., the lexicographically smallest k-mer, known also as minimizer, but with some restrictions) is being sought. The signature is being used as an identifier of the bin which the read is placed into. During this step, some statistics are gathered related to the observed DNA bases, base quality scores and read identifiers. At the end of the binning, these statistics are used to compute the quality scores quantizers, which are stored

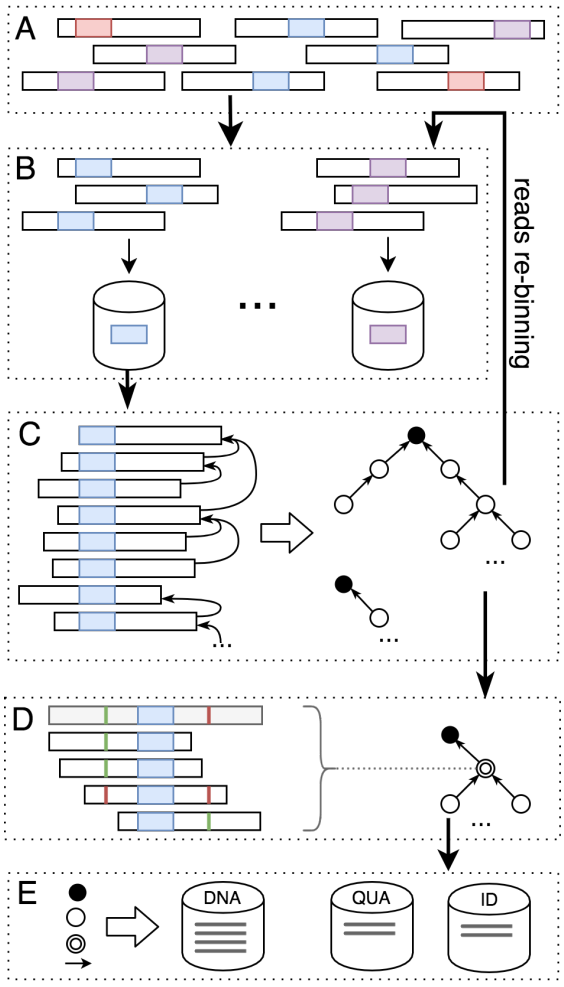

Figure 4: A general compression workflow of FaStore. (A) Raw FASTQ reads are (B) distributed into bins. (C) The reads are matched, giving as a result a reads similarity graph. Optionally, the reads follow further re-distribution and matching. (D) With the final similarity graph, the reads are assembled into contigs. (E) The reads data are encoded either in contigs or differentially, depending on the matching result.

in a global codebook (when using QVZ mode). Analogously, based on the observed tokens in the read identifiers, a token dictionary is build. These data will be used during compression stage.

After the binning all the reads, we perform matching of them, independently per each bin. The goal is to find for each read, a referential one, which has possibly the lowest "encoding cost", i.e. a number of operations required to transform one sequence to another and under some user-specified constrains. In

order to do so, we firstly reorder the reads, so that the reads with sequences possibly originating from the same genomic region, will be placed close to each other. Then, we iterate over the reordered reads and, for each sequence, we search in a window of m previous ones for the best match. A read can be matched as a *normal match*, an *exact match* (an identical sequence was found) or as a *hard read* (when no satisfactory reference was found). The result of reads matching is represented as a similarity graph, where each node represents a read (DNA sequence) and the edge represents the type of match. More specifically, the result is a collection of trees, where each hard read represents a tree root (a tree can also consist of only a root node). With such graph we can already proceed to the compression stage (as in C0 mode).

Reads re-clustering

However, in order to improve the clustering between the sequences, a number of optional reads re-clustering steps can be performed. The goal is to create larger clusters of highly similar (groups of) reads to possibly bring the reads from the same genomic regions close to each other, by re-distributing the reads. To do so, we firstly define a new subset of signatures, which will be used as a filter, to select the bins into which the reads can be moved. Then, for each tree, we select a new root node, which has a new signature residing at the beginning or at the end of its sequence. The trees are re-balanced and moved into bins (similarly as in Fig. 4B) denoted by their root signatures, where each tree is represented in the new bin as a single read (its root). This allows to improve the clustering between the reads, by performing an additional matching of them (as in Fig. 4C) and, as a result, building larger trees of similar reads. If needed, another re-distribution step(s) can be performed (in C1 mode we perform 3), otherwise we proceed to the compression stage.

The compression stage is a two-step process, consisting of assembling the reads sequences (Fig. 4D) into contigs and encoding the reads (Fig. 4E), using the previously built reads similarity graph. Firstly, we traverse each tree and try to assemble the reads into possibly large contigs. The goal is to encode the reads with respect to the built consensus sequences, encoding only the variants (if present) in the contigs. While assembling a contig, for each read, we try to anchor it into the consensus sequence using the position of its signature (which resides at the "center" of the consensus). To add the read to the contig, we assess whether it does not introduce too many variants into the current consensus sequences, as they will need to be encoded by the other reads already present in the contig. When no more reads can be added to the contig, its final consensus sequence is determined by majority voting. As a result, in the graph some of the nodes are replaced with the contig nodes, updating the connections between nodes accordingly.

Reads compression

Finally, we proceed to encode the reads data (Fig. 4E), storing the result in a number of streams, separately for DNA sequences, quality scores, and read identifiers. The read sequences are encoded either in contigs (encoding differentially versus consensus sequences) or differentially versus each other, depending on the matching result. To encode the quality scores using QVZ we use the quantizers from the previously created codebook. Alternatively, when using Illumina 8-level binning or binary thresholding, we encode the transformed quality values. In parallel, we encode the read identifiers using the previously built dictionary. The streams are compressed using custom arithmetic coder or a general-purpose compressor PPMd.

Variant calling

To investigate side effects of applying a lossy compression for base quality scores, we firstly prepared a set of test WGS-14x and WGS-42x FASTQ files, which come from deep sequencing of NA12878 *H.sapiens* individual. These files included: (a) original input files (lossless), (b) original input files with lossy compressed (or discarded) quality scores, (c) FaStore-shuffled reads with lossy compressed quality scores. Moreover, using WGS-14x dataset we tested the effect of reordering the reads using an additional set of test FASTQ files (but without applying any compression). These included: (d) original input files with randomly shuffled reads, (e) FaStore-shuffled reads. A detailed description of the FASTQ files preparation steps can be found in **Supplementary Methods**.

With such prepared input FASTQ files, we followed the GATK (McKenna et al. 2010) Best Practices (Auwera et al. 2013) pipeline to assess the variants. We used BWA-MEM (Li & Durbin 2009; Li 2013) to map the reads to the human genome assembly GRCh37. Following a number of alignments post-processing steps, we called the variants using GATK HaplotypeCaller (GATK-HC). For assessing the variant calling performance, we used as a "gold standard" the variants from GIAB (Zook et al. 2014) and benchmarked our results using Illumina Haplotype comparison tools pipeline (https://github.com/Illumina/hap.py). This pipeline is also recommended by Global Alliance for Genomics and Health (GA4GH) as one of benchmarking standard protocols. We reported precision and recall results for the obtained SNPs. For completeness, we also filtered the variants using GATK Variant Quality Scores Recalibration (VQSR). Both the SNPs and INDELs calling results, with and without VQSR filtering, are available in **Supplementary Worksheet W2**.

Software availability

FaStore can be downloaded from https://github.com/refresh-bio/FaStore.

Acknowledgments

We would like to thank Ivo Gut for supporting the project and Marcos Fernández for helpful discussions and technical insights.

This work was supported by: National Science Centre, Poland [under project DEC-2015/17/B/ST6/01890 to S.D.]; European Union's Seventh Framework Programme (FP7/2007-2013) [under grant agreement No. 305444 (RD-Connect) to Ł.R.];

Competing financial interests

The authors declare no competing financial interests.

References

Auwera, G. *et al.* 2013. From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline *Current protocols in bioinformatics*

Benoit, G. *et al.* 2015. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph *BMC Bioinformatics* **16**, 288.

Bonfield, J.K., Mahoney, M.V. 2013. Compression of FASTQ and SAM format sequencing data *PLOS ONE* **8**, e59190.

Clarke, L. *et al.* 2012. The 1000 Genomes Project: data management and community access *Nat. Methods* **9**, 459–462.

Cock, P.J., Fields C.J., Goto, N., Heuer, M.L. & Rice, P.M. 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants *Nucleic acids research* **38**, 1767–1771.

Cox, A.J. *et al.* 2012 Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform *Bioinformatics* **28**: 1415–1419.

Deorowicz, S., Grabowski, Sz. 2011. Compression of DNA sequence reads in FASTQ format *Bioinformatics* **27**, 860–862

Eberle M. *et al.* 2013. A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree *Genome Research* **27**: 157–164.

Firtina, C., Alkan, C. 2016. On genomic repeats and reproducibility *Bioinformatics*

Grabowski, S., Deorowicz, S., Roguski, Ł. 2015. Disk-based compression of data from genome sequencing *Bioinformatics* **31**, 1389–1395.

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

Hernaez, M., Ochoa, I., Weissman, T. 2016. A cluster-based approach to compression of quality scores In *Proc. of Data Compression Conference*, pp. 261–270.

Jones, D.C., Ruzzo, W.L., Peng, X. & Katze, M.G. 2012. Compression of next-generation sequencing reads aided by highly efficient de novo assembly *Nucleic Acids Res.* **40**, e171.

Kavak, P. *et al.* 2015. Robustness of massively parallel sequencing platforms *PLOS ONE* **10**, e0138259.

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

Li, H. and Durbin, R. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform *Bioinformatics* **25**: 1754–1760.

Li, H. *et al.* 2009. The sequence alignment/map format and SAMtools *Bioinformatics* **25**, 2078–2079.

Malysa, G. *et al.* 2015. QVZ: lossy compression of quality scores *Bioinformatics* **31**: 3122–3129.

McKenna, A. *et al.* 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data *Genome research* **20**: 1297–1303.

Numanagić, I. *et al.* 2016. Comparison of high-throughput sequencing data compression tools *Nat. Methods* **13**, 1005–1008.

Ochoa, I. *et al.* 2016. Effect of lossy compression of quality scores on variant calling *Brief. Bioinform.* **18**, 183–194.

Patro, R. & Kingsford, C. 2015. Data-dependent bucketing improves reference-free compression of sequencing reads *Bioinformatics* **31**, 2770–2777.

Roguski, Ł. & Deorowicz, S. 2014. DSRC 2 – Industry-oriented compression of FASTQ files *Bioinformatics* **30**, 2213–2215.

Schadt, E.E. *et al.* 2010. Computational solutions to large-scale data management and analysis *Nature Reviews Genetics* **11**: 647–657.

Stephens, Z.D. *et al.* 2015. Big Data: Astronomical or Genomical. *PLOS Biol.* **13**, e1002195.

The 100,000 Genomes Project 2017. https://www.genomicsengland.co.uk/the-100000-genomes-project-by-numbers/.

Illumina 2014 Reducing Whole-Genome Data Storage Footprint https://www.illumina.com/documents/products/whitepapers/whitepaper_datacompression.pdf.

Yu, Y.W., Yorukoglu, D., Peng, J., Berger, B. 2015. Quality score compression improves genotyping accuracy *Nature Methods* **33**: 240–243.

Zook, J.M. *et al.* 2014 Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls *Nature biotechnology* **32**: 246–251.

3.4 CARGO: effective format-free compressed storage of genomic information

Roguski Ł, Ribeca P. CARGO: effective format-free compressed storage of genomic information. Nucleic Acids Res. 2016 Jul 8;44(12):e114. DOI: 10.1093/nar/gkw318

3.5 Brief summary

3.5.1 Context

High-throughput sequencing data is usually represented either as raw reads in FASTQ format and stored in compressed form using gzip or represented as aligned reads in SAM format and stored in compressed form in BAM format. The HTS data represented in SAM format can be considered as a superset of the data stored in FASTQ format, where the SAM alignments contain a significant amount of intermediate data generated during the sequence mapping and the data post-processing stages. Information, such as the possible mapping location of the read in the reference genome and the indicator of the confidence of the mapping, is used in further genomic data analysis steps, allowing it to efficiently access the alignments by their mapping position. Usually, the backward conversion from the aligned reads in SAM format to the raw reads in FASTQ format is possible (given that the information carried with raw reads was not discarded during the mapping and post-processing steps), by using different tools, such as Picard or SAMTools. Therefore, different solutions to storing HTS data are available, the application of which depends on the used genomic data processing pipelines.

During our research we have designed, developed, and tested a set of novel methods and approaches for storing and compressing HTS data. As presented in the publications included in this chapter, we performed a number of analyses compressing both FASTQ and SAM files originating from a variety of different sequencing experiments, using different sequencing platforms and coming from different organisms. At the same time, one can also observe other novel methods being developed. Therefore, the aim of this brief summary is to compare methods that we have developed with the current state-of-the-art using a concise and small dataset.

Data categories

As mentioned in Section 2.1, when compressing raw reads in FASTQ format we can distinguish three main categories of data: read identifiers (denoted as *ID*), DNA base sequences (*SEQ*) and base quality scores (*QUA*). When compressing alignments in SAM format, we will consider four main categories of compressed data: query template names (*ID*), sequence

alignment data (*ALN*), quality scores (*QUA*), and optional information (*OPT*). The SAM fields which compose the *ALN* data category include: the alignment flags (*FLAG* field), the segment sequence (*SEQ* field), the pairing information between the reads (*RNEXT*, *PNEXT*, and *TLEN* fields), and the *CIGAR* field. Here, we also consider the values of the mapping quality (*MAPQ* field) assigned by the mapper as optional information (storing them in *OPT* category alongside the content of the optional fields), because they are not required to losslessly compress or decompress the sequence data. One needs to note, however, that the above-mentioned classification is arbitrary, but it allows for a non-ambiguous and practical separation of the data categories – other classification may be also possible.

Regarding the possible pairing information between the reads, we decided not to introduce any additional category for representing it, since, in the current SAM format data representation, it is not possible to fully decouple the pairing data from the data stored in the other categories. For example, in addition to the pairing information stored in *RNEXT*, *PNEXT*, and *TLEN* fields, the *FLAG* field also stores bit-wise the information about pairing, such as, which read is considered as the first one from the pair or whether both reads are aligned. Moreover, in case of unaligned reads, some of the fields storing the pairing data may be left with an empty/dummy value. Hence, the pairing between the unaligned reads is preserved by the (unique) query template names. Similarly, in the case of FASTQ files, the representation of pairing information between the reads is not clearly defined, as it can be preserved either by the read identifiers or on the file level (see the formats description in Section 1.1.3).

Datasets

To briefly compare our solutions with the current state-of-the-art, we selected as input data two datasets, namely *WEX* and *WGS*, which have been also used in the tests performed in Section 3.3. We compare the results of compressing these datasets both in their raw (FASTQ) and aligned (SAM) representations. These datasets were obtained from a whole-exome sequencing (WEX; ~240× coverage) and from a whole-genome sequencing (WGS; ~42× coverage) experiment of *H. Sapiens* individuals. Both were sequenced in a paired-end library setup and using the Illumina HiSeq

platform. These represent a common use-case for using sequencing in clinical diagnostics.

The WEX dataset comes from GIAB [222] and is originally available in BAM format (aligned to the reference human genome assembly version 37 (*GRCh37*) and sorted by position). Hence, it requires conversion to FASTQ format in order to test compression methods of raw reads. The conversion from BAM to SAM format is straightforward. Both operations can be accomplished by using SAMTools or Picard tools. The WGS dataset comes from Illumina Platinum Genomes[1] and is available in FASTQ format through the SRA platform. Therefore, in order to provide its representation in SAM format, we need to map the raw reads to the reference genome (using the same human genome assembly as in the WEX dataset) using BWA-MEM [108] following the GATK Best Practices [11] (as explained in Section 1.1.3) – the used command lines with the used versions of the tools can be found in the Appendix A. Important to note, since the mappers usually remove the comments present in the FASTQ read identifiers line, we also removed them before testing the compression of the FASTQ files (in the case of WEX dataset, they have already been removed).

The selected information about the datasets, including the sizes of different kinds of data (in GB) both in FASTQ and SAM formats are presented in Table 3.1. What is interesting, for both datasets the number of records (alignments) in SAM format is greater than the number of records (raw reads) in FASTQ format (WEX: ~0.1 M, WGS: ~3.3 M). The greater number of SAM records may signify the existence of chimeric reads or multiple possible alignments of reads (or pairs). As mentioned in Section 1.1.3, in case of reporting multiple alignments, some of the content present in the main alignment will be duplicated (such as query template names, DNA sequences, or quality scores) represented in SAM format. Moreover, although the number of reported SAM alignments is greater than the number of FASTQ reads, the reported size of the data of *ID* category in SAM format is smaller than its equivalent in FASTQ. This is due to the fact that the read identifiers in FASTQ format start with '@' symbol, which is removed in SAM query template names. The links to obtain the datasets with the explanation of the protocol used to perform the conversion between the formats can be found in Appendix A.

[1] https://www.illumina.com/platinumgenomes.html

Table 3.1: Summary of the WEX and WGS datasets with a comparison between the sizes of each class of data in FASTQ and SAM formats.

Dataset		FASTQ format					SAM format					
Name	Read len. (bp)	Recs. (M)	Size (GB)	Data class SEQ	ID	QUA	Recs. (M)	Size (GB)	Data class ALN	ID	QUA	OPT
WEX	126	150.4	44.7	18.9	6.1	18.9	150.5	55.5	23.3	5.9	19.0	5.0
WGS	101	1304.5	295.5	131.8	25.4	131.8	1307.8	391.0	169.8	24.2	131.9	45.4

Methods overview

In our brief analysis, we have used the solutions, which have been previously analyzed and tested in the publications included in this chapter, but, possibly, in the newest and officially supported versions. Moreover, we only consider solutions that fully support lossless compression of the up-to-date SAM and FASTQ formats – they have been described in more detail in Chapter 2. Following the classification introduced in Section 2.1 (FASTQ format) and Section 2.2 (SAM format), we can distinguish the following categories of compression methods.

When compressing raw reads in FASTQ format, we consider the general (DSRC2 in *FAST* and *MAX* compression modes, Fqzcomp [19] in *STD* and *MAX* modes, Quip [85] in *STD* mode, and *CARGO*-based family of compressors), the assembly-based (Quip in *MAX* mode, *LEON* [15]) and the reads-reordering-based (*SCALCE* [70], FaStore in *C0* and *C1* modes). Unfortunately, as with the authors of [151], when benchmarking different FASTQ compression solutions, we encountered difficulties in successfully running reference-based compression methods and, as a result, they have been excluded from our brief analysis. As a side note, for the current general and assembly-based compression methods, it does not matter, whether the input data have been generated in the single-end or paired-end mode – they process the data as they are, not altering the reads order. By contrast, the reads-reordering solutions SCALCE and FaStore provide explicitly the paired-end processing mode, to possibly preserve the pairing between the reads on the file-level. As our input data has been generated from a paired-end library, we will primarily focus on the paired-end compression mode provided by these tools (suffix *-PE*). However, for comparison purposes only, we will also include the results obtained when compressing the dataset in the single-end mode (suffix *-SE*; losing the pairing information on the file-level after the decompression).

When compressing alignments in SAM format, we can distinguish two main categories of compression methods, namely: the non-reference-based ones (Quip, SCRAMBLE [18], and DeeZ [71] methods run in *NOREF* mode; CARGO SAM format compressors in *STD* and *EXT* variants) and the reference-based ones (Quip, SCRAMBLE, DeeZ run in *REF* modes; *CARGO-SAM-REF*). Moreover, when performing reference-based compression, SCRAMBLE, implementing the CRAM format (both in the version 2.0 the newest one – 3.0; the version 2.0 was tested in Section 3.4), also offers an option to embed the provided reference sequence in the compressed archive when run in *EMBREF* mode, hence not requiring the reference to be provided externally during the decompression. As a side note, we also included in our tests the non-reference-based SCRAMBLE compressor implementing CRAM format in version 2.0, but only as a reference point, since it does not preserve all the SAM optional fields.

Finally, in our tests, we also included the gzip-based compression solutions, which being the most commonly used solutions are considered as the *de facto* standard. These are: pigz[2] (a parallel version of the gzip compressor) and the SCRAMBLE-based implementation of BAM format.

All the applications were run in multithreading mode using 8 threads, if supported. Only Quip and Fqzcomp do not support compression with an externally provided number of processing threads – when run in multithreading mode (set by default) they compress each kind of the data in a separate thread. Moreover, some methods, such as the CARGO-based family of FASTQ format compressors or a simple *CARGO-SAM-STD* have been provided as a proof-of-concept, being semi-automatically generated. Hence, a direct and complete comparison with format-specific and optimized state-of-the-art compressors is not an easy task.

3.5.2 Results

Overview

The tests were performed on the cluster in Centro Nacional de Análisis Genómico[3] (CNAG), which consists of more than 100 compute nodes each one having two Intel Xeon Quad Core 2.93 GHz processors with 48

[2] http://zlib.net/pigz/
[3] http://www.cnag.cat

GB of RAM. It has about 3 PB of network-distributed hard-drive storage mounted as a Lustre parallel file system[4]. Inter-node communication and data transfer is performed via a dedicated Infiniband network.

In Table 3.2 we show the results of compressing raw reads in FASTQ format for both WEX and WGS datasets. Analogously, in Table 3.3 we show the results of compressing alignments in SAM format. Some of these results are best visualized as a picture – Fig. 3.1 shows the comparison between different compressed sizes of the WEX dataset that each of the methods achieves, both for the data represented in FASTQ and SAM formats. Analogously, Fig. 3.1 shows these results for the WGS dataset.

The tested compression solutions, which are currently considered as the *de facto* standard are *PIGZ* (FASTQ) and *SCRAMBLE-BAM* (SAM). *PIGZ* managed to compress the input WEX dataset in FASTQ format initially of 44.8 GB in size to 11.7 GB (28% of its original size) and the WGS dataset of 295.5 GB in size to 98.2 GB (33%). In comparison, *SCRAMBLE-BAM* compressed the WEX dataset in SAM format of 55.5 GB in size to 9.8 GB (18%) and the WGS dataset of 391 GB in size to 92.5 GB (24%). Interestingly, the aligned reads stored in SAM format occupy significantly more space than their equivalent raw representation in FASTQ format, but compressed as a BAM file they occupy less space than stored in FASTQ file and compressed using gzip. This is primarily due to the fact that the highly similar DNA sequences (mapping to the same or neighboring positions in the reference genome) reside much closer to each other in the file. Hence, the possible overlappings between sequences can be more easily encoded by the dictionary-based compression method used in BAM format. Moreover, when compressing FASTQ files using the *CARGO-FQ-GZIP* method, which decouples records data into three separate streams and compresses them using the same compression algorithm as *PIGZ*, a significant improvement in compression ratio can be achieved compared to *PIGZ*. *CARGO-FQ-GZIP* managed to compress the FASTQ datasets WEX to 10.1 GB (23%) and WGS to 86.2 GB (29%), which, in the case of the WGS dataset, this also gives a higher compression result than when compressing it in SAM format using *SCRAMBLE-BAM*. This simple experiment clearly shows the importance of decoupling the records data prior to the actual compression.

Nonetheless, the highest overall compression ratio for the FASTQ format

[4] http://lustre.org/

Table 3.2: Summary of the results of compressing WEX and WGS datasets in FASTQ format.

Method	Total			Speed		DNA				ID				QUA			
	Size	Ratio	Inv.	Comp.	Dec.	Size	Ratio	Inv.	Frac.	Size	Ratio	Inv.	Frac.	Size	Ratio	Inv.	Frac.
WEX																	
FASTORE-SE-C1	4383	10.21	10%	5.2	83.6	624	30.37	3%	14%	729	8.35	12%	17%	3030	6.25	16%	69%
FASTORE-SE-C0	4462	10.03	10%	38.1	102.8	697	27.19	4%	16%	734	8.30	12%	16%	3031	6.25	16%	68%
FASTORE-PE-C1	4443	10.07	10%	5.2	100.3	1020	18.58	5%	23%	380	16.02	6%	9%	3043	6.23	16%	68%
FASTORE-PE-C0	4462	10.03	10%	35.1	104.5	1061	17.86	6%	24%	371	16.41	6%	8%	3030	6.25	16%	68%
SCALCE-SE	5746	7.79	13%	42.8	80.0	1291	14.68	7%	22%	1261	4.83	21%	22%	3194	5.93	17%	56%
SCALCE-PE	6336	7.06	14%	51.6	92.2	1883	10.06	10%	30%	1260	4.83	21%	20%	3193	5.93	17%	50%
LEON	6127	7.30	14%	16.3	49.2	1713	11.06	9%	28%	251	24.26	4%	4%	4163	4.55	22%	68%
QUIP-FQ-MAX	7465	5.99	17%	50.6	23.6	4195	4.52	22%	56%	316	19.27	5%	4%	2954	6.41	16%	40%
FQZCOMP-MAX	5291	8.46	12%	18.0	16.0	2087	9.08	11%	39%	314	19.39	5%	6%	2890	6.56	15%	55%
FQZCOMP-STD	7700	5.81	17%	74.3	56.6	4440	4.27	23%	58%	314	19.39	5%	4%	2946	6.43	16%	38%
DSRC-MAX	7924	5.65	18%	159.2	135.2	4534	4.18	24%	57%	332	18.34	5%	4%	3058	6.20	16%	39%
DSRC-FAST	8599	5.20	19%	306.4	588.7	4731	4.01	25%	55%	328	18.56	5%	4%	3540	5.35	19%	41%
CARGO-FQ-PPMD	8404	5.32	19%	133.5	118.0	4663	4.06	25%	55%	432	14.09	7%	5%	3308	5.73	17%	39%
CARGO-FQ-LZMA	8847	5.06	20%	8.0	261.6	4921	3.85	26%	56%	351	17.35	6%	4%	3574	5.30	19%	40%
CARGO-FQ-BZIP2	9266	4.83	21%	81.5	154.3	5105	3.71	27%	55%	593	10.27	10%	6%	3567	5.31	19%	38%
CARGO-FQ-GZIP	10080	4.44	23%	16.7	514.2	5296	3.58	28%	53%	654	9.31	11%	6%	4129	4.59	22%	41%
PIGZ	11694	3.83	26%	12.6	165.7	—	—	—	—	—	—	—	—	—	—	—	—
WGS																	
FASTORE-SE-C1	42142	7.01	14%	6.1	78.6	3439	38.31	3%	8%	4859	5.23	19%	12%	33844	3.89	26%	80%
FASTORE-SE-C0	42788	6.91	14%	25.9	79.0	4119	31.99	3%	10%	4858	5.23	19%	11%	33811	3.90	26%	79%
FASTORE-PE-C1	44936	6.58	15%	8.5	82.0	8524	15.46	6%	19%	2466	10.31	10%	5%	33946	3.88	26%	76%
FASTORE-PE-C0	45720	6.46	15%	29.7	81.9	9402	14.01	7%	21%	2433	10.45	10%	5%	33885	3.89	26%	74%
SCALCE-SE	56214	5.26	19%	10.4	69.6	12345	10.67	9%	22%	7015	3.62	28%	12%	36854	3.58	28%	66%
SCALCE-PE	63710	4.64	22%	36.1	79.9	19897	6.62	15%	31%	6984	3.64	27%	11%	36829	3.58	28%	58%
LEON	60293	4.90	20%	13.3	38.1	13206	9.98	10%	22%	16	>1000	0%	0%	47071	2.80	36%	78%
QUIP-FQ-MAX	63008	4.69	21%	34.3	21.2	28497	4.62	22%	45%	2	>1000	0%	0%	34509	3.82	26%	55%
FQZCOMP-MAX	56893	5.19	19%	7.7	7.0	24035	5.48	18%	42%	2	>1000	0%	0%	32856	4.01	25%	58%
FQZCOMP-STD	63154	4.68	21%	53.9	43.1	29041	4.54	22%	46%	2	>1000	0%	0%	34111	3.86	26%	54%
DSRC-MAX	63684	4.64	22%	121.3	106.0	29960	4.40	23%	47%	21	>1000	0%	0%	33703	3.91	26%	53%
DSRC-FAST	73769	4.01	25%	304.7	432.4	32931	4.00	25%	45%	2	>1000	0%	0%	40836	3.23	31%	55%
CARGO-FQ-PPMD	69581	4.25	24%	93.0	84.6	31776	4.15	24%	46%	1371	18.54	5%	2%	36433	3.62	28%	52%
CARGO-FQ-LZMA	74391	3.97	25%	8.7	230.8	32369	4.07	25%	44%	456	55.75	2%	1%	41565	3.17	32%	56%
CARGO-FQ-BZIP2	76316	3.87	26%	73.5	124.9	34457	3.82	26%	45%	1475	17.24	6%	2%	40383	3.26	31%	53%
CARGO-FQ-GZIP	86164	3.43	29%	15.5	433.2	36120	3.65	27%	42%	3297	7.71	13%	4%	46745	2.82	35%	54%
PIGZ	98182	3.01	33%	18.3	136.2	—	—	—	—	—	—	—	—	—	—	—	—

Notes: The sizes are reported in gigabytes. The compression and decompression performance speeds are in MB/s. *Ratio* is the ratio between the size of original file or data class and its compressed size, whereas *Inv.* is the inverse relation. *Frac.* denotes the fraction of the total compressed size the data class occupies.

Table 3.3: Summary of the results of compressing WEX and WGS datasets in SAM format.

Method	Total			Speed		ALN				ID				QUA				OPT			
	Size	Ratio	Inv.	Comp.	Dec.	Size	Ratio	Inv.	Frac.	Size	Ratio	Inv.	Frac.	Size	Ratio	Inv.	Frac.	Size	Ratio	Inv.	Frac.
WEX																					
SCRAMBLE-CRAM3-REF	4700	11.80	8%	183.0	273.2	416	56.11	2%	9%	804	7.39	14%	17%	3353	5.65	18%	71%	124	39.94	3%	3%
SCRAMBLE-CRAM2-REF	5350	10.37	10%	181.8	270.5	460	50.75	2%	9%	804	7.39	14%	15%	3945	4.81	21%	74%	136	36.41	3%	3%
CARGO-SAM-REF	4765	11.64	9%	120.8	209.3	602	38.78	3%	13%	668	8.90	11%	14%	3344	5.67	18%	70%	148	33.46	3%	3%
DEEZ-REF	5026	11.03	9%	16.3	21.2	866	26.95	4%	17%	621	9.57	10%	12%	3178	5.96	17%	63%	359	13.79	7%	7%
QUIP-SAM-REF	4854	11.43	9%	38.5	30.2	882	26.47	4%	18%	689	8.63	12%	14%	2954	6.42	16%	61%	327	15.14	7%	7%
SCRAMBLE-CRAM3-EMBREF	5417	10.24	10%	181.8	296.6	1124	20.77	5%	21%	804	7.39	14%	15%	3353	5.65	18%	62%	124	39.94	3%	2%
SCRAMBLE-CRAM2-EMBREF	6133	9.04	11%	140.0	280.1	1244	18.76	5%	20%	804	7.39	14%	13%	3945	4.81	21%	64%	136	36.41	3%	2%
SCRAMBLE-CRAM3-NOREF	5562	9.97	10%	198.8	301.4	1121	20.82	5%	20%	804	7.39	14%	14%	3353	5.65	18%	60%	280	17.69	6%	5%
SCRAMBLE-CRAM2-NOREF	6648	8.34	12%	119.3	283.2	1759	13.27	8%	26%	804	7.39	14%	12%	3945	4.81	21%	59%	136	36.41	3%	2%
CARGO-SAM-STD	5496	10.09	10%	67.9	191.2	1197	19.50	5%	22%	668	8.90	11%	12%	3348	5.66	18%	61%	280	17.69	6%	5%
CARGO-SAM-EXT	5602	9.90	10%	84.7	179.5	1276	18.29	5%	23%	668	8.90	11%	12%	3344	5.67	18%	60%	311	15.92	6%	6%
DEEZ-NOREF	5401	10.27	10%	17.6	23.0	1241	18.81	5%	23%	621	9.57	10%	11%	3178	5.96	17%	59%	359	13.79	7%	7%
QUIP-SAM-NOREF	9063	6.12	16%	37.1	29.8	5091	4.59	22%	56%	689	8.63	12%	8%	2954	6.42	16%	33%	327	15.14	7%	4%
SCRAMBLE-BAM	9765	5.68	18%	194.6	269.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
WGS																					
SCRAMBLE-CRAM3-REF	50347	7.77	13%	170.7	263.4	3648	46.56	2%	7%	5245	4.61	22%	10%	39796	3.31	30%	79%	1656	27.44	4%	3%
SCRAMBLE-CRAM2-REF	59656	6.55	15%	172.3	238.1	3909	43.45	2%	7%	5245	4.61	22%	9%	48773	2.70	37%	82%	1726	26.33	4%	3%
CARGO-SAM-REF	51065	7.66	13%	131.6	184.4	5634	30.15	3%	11%	4615	5.24	19%	9%	39024	3.38	30%	76%	1789	25.42	4%	4%
DEEZ-REF	53589	7.30	14%	16.3	21.5	7936	21.40	5%	15%	4734	5.11	20%	9%	36987	3.57	28%	69%	3930	11.57	9%	7%
QUIP-SAM-REF	51027	7.66	13%	34.5	27.2	7452	22.79	4%	15%	4828	5.01	20%	9%	34571	3.82	26%	68%	4175	10.89	9%	8%
SCRAMBLE-CRAM3-EMBREF	52484	7.45	13%	169.1	277.9	4362	38.94	3%	8%	5245	4.61	22%	10%	39796	3.31	30%	76%	3079	14.76	7%	6%
SCRAMBLE-CRAM2-EMBREF	60436	6.47	15%	169.6	238.4	4688	36.23	3%	8%	5245	4.61	22%	9%	48773	2.70	37%	81%	1726	26.33	4%	3%
SCRAMBLE-CRAM3-NOREF	55427	7.05	14%	167.4	264.8	7283	23.32	4%	13%	5245	4.61	22%	9%	39817	3.31	30%	72%	3079	14.76	7%	6%
SCRAMBLE-CRAM2-NOREF	68711	5.69	18%	98.8	248.9	12964	13.10	8%	19%	5245	4.61	22%	8%	48773	2.70	37%	71%	1726	26.33	4%	3%
CARGO-SAM-STD	55486	7.05	14%	60.8	169.6	8772	19.36	5%	16%	4613	5.24	19%	8%	39085	3.38	30%	70%	3012	15.09	7%	5%
CARGO-SAM-EXT	56315	6.94	14%	74.5	163.8	9346	18.18	6%	17%	4613	5.24	19%	8%	39027	3.38	30%	69%	3327	13.67	7%	6%
DEEZ-NOREF	54387	7.19	14%	16.4	21.6	8733	19.45	5%	16%	4734	5.11	20%	9%	36987	3.57	28%	68%	3930	11.57	9%	7%
QUIP-SAM-NOREF	78203	5.00	20%	33.3	26.3	34628	4.91	20%	44%	4828	5.01	20%	6%	34571	3.82	26%	44%	4175	10.89	9%	5%
SCRAMBLE-BAM	92466	4.23	24%	137.6	271.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Notes: The sizes are reported in gigabytes. The compression and decompression performance speeds are in MB/s. *Ratio* is the ratio between the size of original file or data class and its compressed size, whereas *Inv.* is the inverse relation. *Frac.* denotes the fraction of the total compressed size the data class occupies.

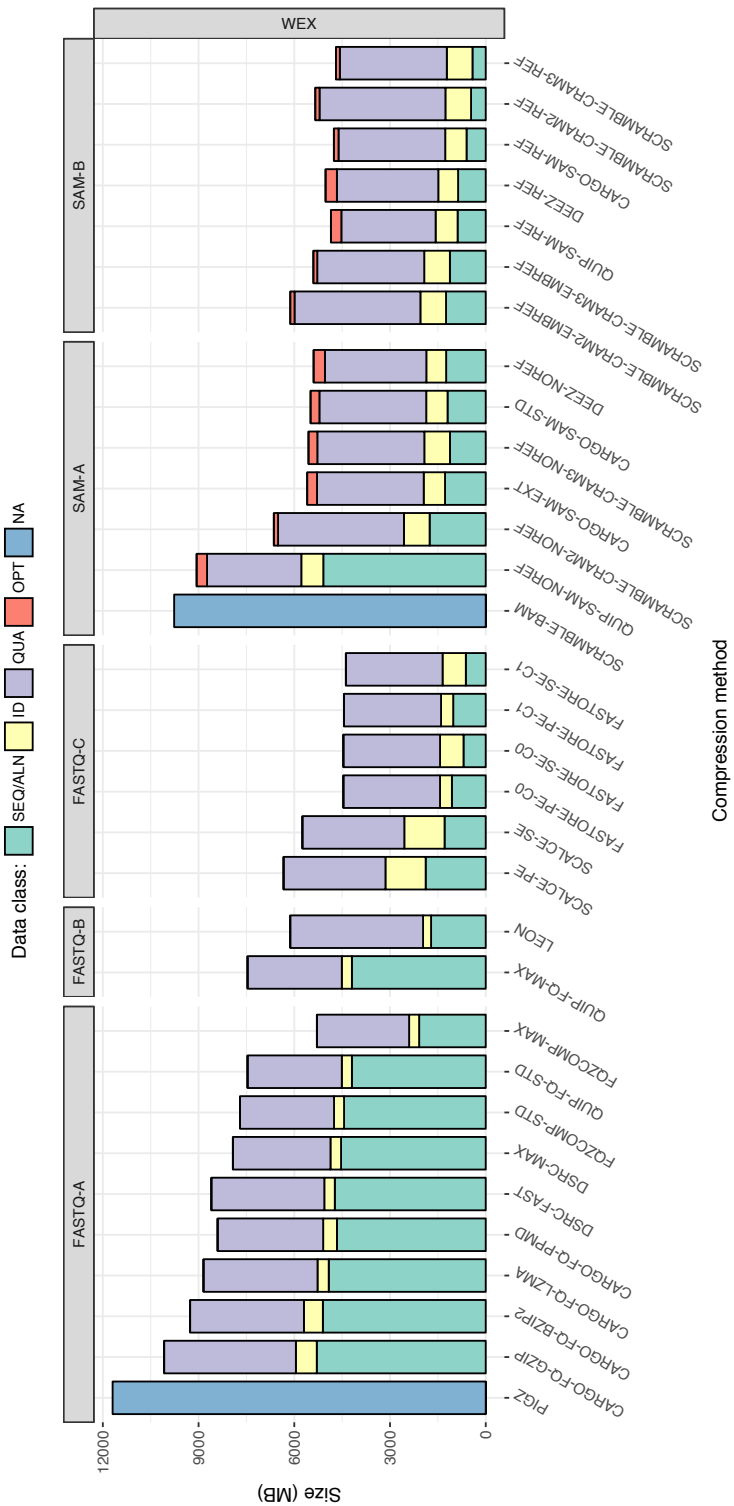

Figure 3.1: Comparison of compressed file sizes and data streams between different compression methods for the WEX dataset represented in FASTQ and SAM formats. *PIGZ* and *SCRAMBLE-BAM* solutions do not decouple the records content when compressing the data, hence we only report the size of the whole compressed file denoted as *NA*. The categories of compression methods are respectively: general ones (*FASTQ-A*, *SAM-A*), assembly-based (*FASTQ-B*), reads-reordering-based (*FASTQ-C*), and reference-based (*SAM-B*).

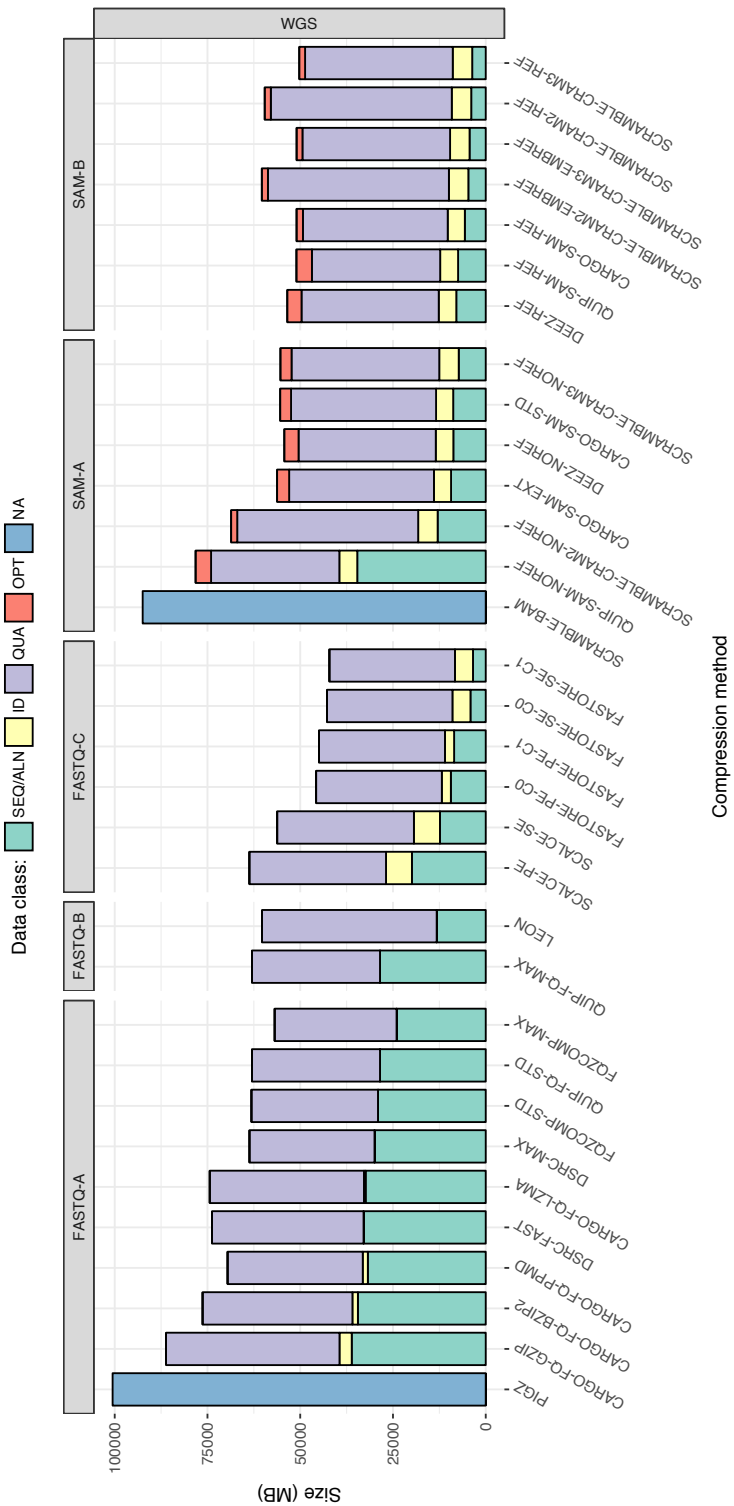

Figure 3.2: Comparison of compressed file sizes and data streams between different compression methods for the WGS dataset represented in FASTQ and SAM formats. *PIGZ* and *SCRAMBLE-BAM* solutions do not decouple the records content when compressing the data, hence we only report the size of the whole compressed file denoted as *NA*. The categories of compression methods are respectively: general ones (*FASTQ-A*, *SAM-A*), assembly-based (*FASTQ-B*), reads-reordering-based (*FASTQ-C*), and reference-based (*SAM-B*).

was achieved by the reads-reordering method FaStore. In *C1* mode it managed to compress the WEX dataset to 4.4 GB (10% of its original size) and the WGS dataset to 44.9 GB (15%). In comparison to SAM format, the highest overall compression ratio was achieved by the reference-based compression methods. *SCRAMBLE-CRAM3-REF* managed to compress the SAM WEX dataset to 4.7 GB (8%) and the WGS dataset to 50.3 GB (13%). Similar results have been been achieved by the reference-based version of the CARGO-based compressor, followed by Quip and DeeZ. What is interesting are the high compression results achievable by simple proof-of-concept SAM format CARGO-based compressors which show that there is still a significant room for improvement in the current state-of-the-art.

Compression of quality scores

As can be clearly seen from Fig. 3.1 and 3.2, for the majority of the solutions, the largest amounts of space occupy the quality scores – up to 70 – 74% (WEX) or 80 – 82% (WGS) of the resulting compressed file. These are the most difficult kind of data to compress, primarily due to their high entropy and containing a significant degree of noise. Nonetheless, in the case of compressing quality scores in the FASTQ files, the best result was achieved by Fqzcomp, which managed to store them in 2.9 GB (15% of their original size) for the WEX dataset and in 32.9 GB (25%) for the WGS. Similarly, in the case of compressing SAM files, Quip provided the best compression results, storing the quality scores in the case of the WEX dataset in 2.9 GB (16%) and for the WGS dataset in 34.6 GB (26%).

Compression of the sequence and alignment data

After the quality scores, the next type of data occupying most of the space is the sequence (FASTQ) and alignment (SAM) data. When compressing the sequence data in FASTQ format, the best compression ratio was achieved by FaStore, which managed to compress the input datasets by up to 6% of their original size. For comparison, LEON, which does not reorder the reads (but performs a reads preprocessing step before their assembly), managed to reduce the size of the sequence data by up to 9 – 10% of their original size. Similarly, *FQZCOMP-MAX*, a general method, managed to reduce the sequence size by up to 11% and 18% for the WEX and WGS datasets respectively. The compressed sequence data can occupy by up

to 23 – 58% (WEX) or 19 – 47% (WGS) of the resulting file. As a side note, when running FaStore in the single-end mode (i.e., discarding the pairing information between the reads on the file level), the sequence data can be further reduced up to half of the one achieved in the paired-end mode – by up to 3% of their original size for both datasets.

When compressing alignments in SAM format and sorted by their mapping position (the operation can be also treated as a form of reads reordering transformation), the best compression was achieved by the non-reference-based methods. *SCRAMBLE-CRAM3-REF* achieved the best result, for both datasets it managed to reduce the alignment data by up to 2% of their original size. Similar results were achieved by the reference-based CARGO solution. Considering now non-reference based methods, the best compression result was also achieved by *SCRAMBLE-CRAM3-NOREF*, which managed to reduce the alignment size by up to 5% (WEX) and 4% (WGS) of its size. Here, analogously, comparable results were achieved by CARGO and DeeZ. Interestingly, the solutions which implement both reference and non-reference compression modes, when run in non-reference mode, report the size of the compressed data of *ALN* kind as almost twice the size as the one obtained in reference-based mode. Important to mention, when the solutions are not run in reference-based mode, some optional fields need to be stored explicitly in the resulting file and cannot be regenerated automatically without access to the reference. This brings additional, significant savings. For example, when run with access to the reference sequence both SCRAMBLE and the CARGO solutions can reduce the size of the optional fields content by up to 3 – 4% of their original size compared to 6 – 7% when not using the reference. Finally, what is interesting, when running SCRAMBLE in the mode of embedding the reference sequence in the resulting compressed archive (the *EMBREF* suffix), the size overhead of storing the reference sequence (included in the *ALN* class) compared to when run in non-reference-based compression is negligible. In the case of WGS dataset, the compression results are even better than when run without the reference sequence. The possible improvement (or decrease) in compression ratio, in fact, strictly depends on the depth and the uniformity of the sequence coverage of the compressed dataset. As a side note, running *SCRAMBLE* with embedding the reference sequence also allows it to re-generate some of the optional fields during decompression, hence, not requiring them to be stored.

Compression of the read identifiers

The significant reduction achieved above in the compressed size of data in *SEQ* and *ALN* categories is primarily thanks to reordering the reads. The reads can be either compressed in groups sharing a large degree of sequence similarity between the consecutive reads (FASTQ) or compressed as mapped to a reference genome and sorted (reordered) by the mapping position (SAM). However, the significant gains in sequence and alignment data compression achievable by these methods come at the cost of decreasing the compression efficiency of the read identifiers. This is due to the fact that in the initial FASTQ files, the consecutive read identifiers are more likely to exhibit some sort of order, e.g., the numerical values associated with some tokens (such as arbitrary read numbers or flowcell coordinates) tend to appear in an increasing order. Therefore, the read identifiers in the reordered FASTQ files are more difficult to compress – this can be more clearly seen on the Figures 3.1 and 3.2.

This problem is especially visible for the WGS dataset, where the identifiers can be compressed up to less than 1% of the original size by a number of different methods, whereas FaStore compresses them up to 10%. Similarly, all the SAM solutions compress them up to $19-22\%$ of their original size with the best result achievable by CARGO-based methods on this dataset. One needs to note, however, that the WGS dataset contains the identifies following the SRA format (i.e., *@<library_name>.<read_number> <more_data_stored_as_comment>*), which, in addition, were trimmed before compression (as explained in the previous subsection). Therefore, when considering the WEX dataset, possibly containing the original identifiers generated by the sequencing machine, these differences in compressed size are less significant. FaStore, in paired-end mode can compress the identifiers by up to 6% of their original size. For comparison, LEON achieves the best compression results for this category of data and compresses them up to 4%. All the SAM solutions compress the identifiers by up to $10-14\%$, with the best result achievable by DeeZ. However, when considering sizes of compressed data from *SEQ* and *ID* (FASTQ) or *ALN* and *ID* (SAM) categories together, it can be seen that the best compression ratio is achievable either by reads-reordering-based methods (*FASTORE-C1*) or reference-based methods (*SCRAMBLE-CRAM3-REF*, *CARGO-SAM-REF*).

Compression speed comparison

Regarding the compression and decompression speeds offered by different solutions – the best way to visualize them is as a picture. Fig. 3.3 and 3.4 show the compression speeds of different solutions in a 2D space for FASTQ and SAM format respectively.

When compressing FASTQ files, *DSRC2-FAST* achieves the best performance speed results – more than 300 MB/s when compressing and more than 400 MB/s when decompressing. Comparable decompression speeds (over 400 MB/s) were also achieved by *CARGO-FQ-GZIP*. However, what is interesting, although *CARGO-FQ-GZIP* and *PIGZ* achieve comparable compression speeds (~15 MB/s), the decompression speeds offered by *CARGO-FQ-GZIP* are at least twice that offered by *PIGZ*, while using the same compression settings and the number of processing threads. Moreover, it can be noted that some solutions offer a higher compression ratio at the cost of spending more computational resources during the data compression stage. They perform different reads preprocessing operations, such as: reordering the reads (FaStore, SCALCE), performing the assembly of the reads (LEON) or performing an exhaustive search for sequence matches of different streams (*CARGO-FQ-GZIP*, *CARGO-FQ-LZMA*). Nonetheless, with the resulting reduced output file size and a reasonably lightweight decompression algorithm they can also achieve relatively high decompression speeds.

In the case of SAM format, the newest SCRAMBLE-based implementation of CRAM format provides for both datasets the highest performance speeds, offering compression speeds ~170 MB/s and decompression speeds ~270 MB/s. Similarly, when transcoding SAM files into BAM format, it achieves ~130 MB/s (WGS) or ~190 MB/s (WEX) with a decompression speed of ~270 MB/s for both datasets. By comparison, *CARGO-SAM-REF* method is left just behind offering compression speeds ~120 – 130 MB/s and decompression speed ~190 – 210 MB/s. The non-reference based compression versions of the above mentioned methods achieve lower compression speeds, yet, they still achieve relatively high performance speeds alongside all the tools tested. As a side note, regarding a potential comparison of the compression and decompression speeds offered by different methods for both FASTQ and SAM formats, they cannot be directly compared – to measure the compression speed of SAM

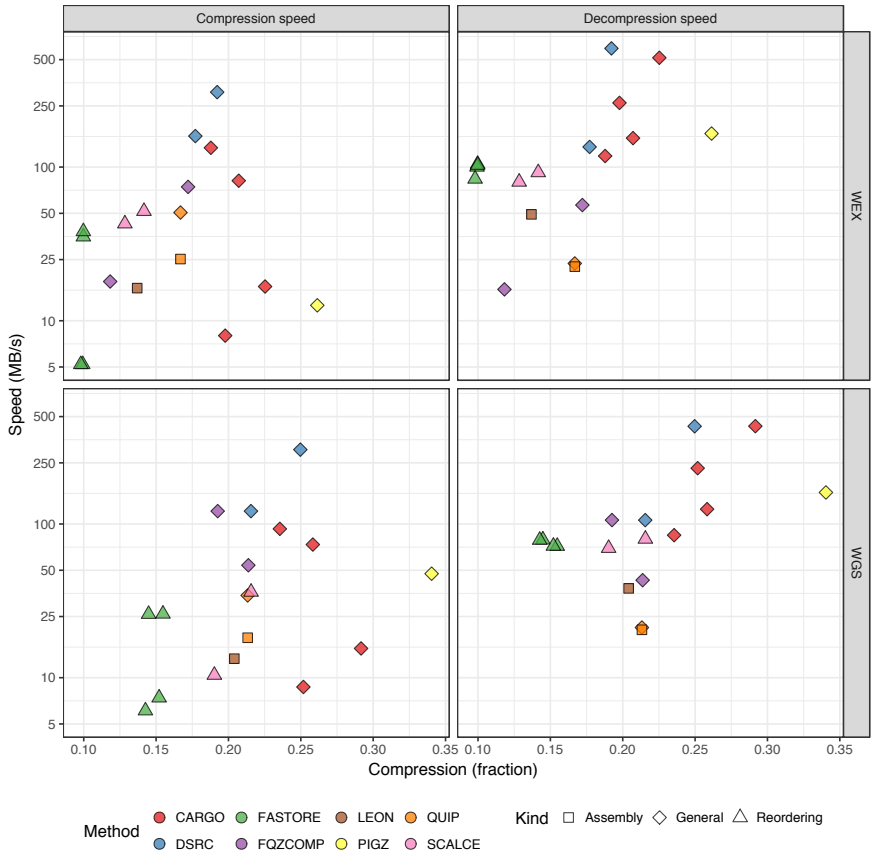

Figure 3.3: Comparison of performance speed between different FASTQ compression methods. *Fraction* denotes the ratio between the compressed and original file.

compression methods in a fair way, we would need to include the time spent on aligning the reads to the reference sequence and time spend on sorting the alignments by position taken by external tools. Moreover, in the case of FASTQ files, although different compression methods have been compared, the ones which apply specialized reads preprocessing operations will usually achieve a superior compression ratio compared to general-purpose methods. These improvements in compression ratio come, however, at a cost of spending additional resources for the reads preprocessing step.

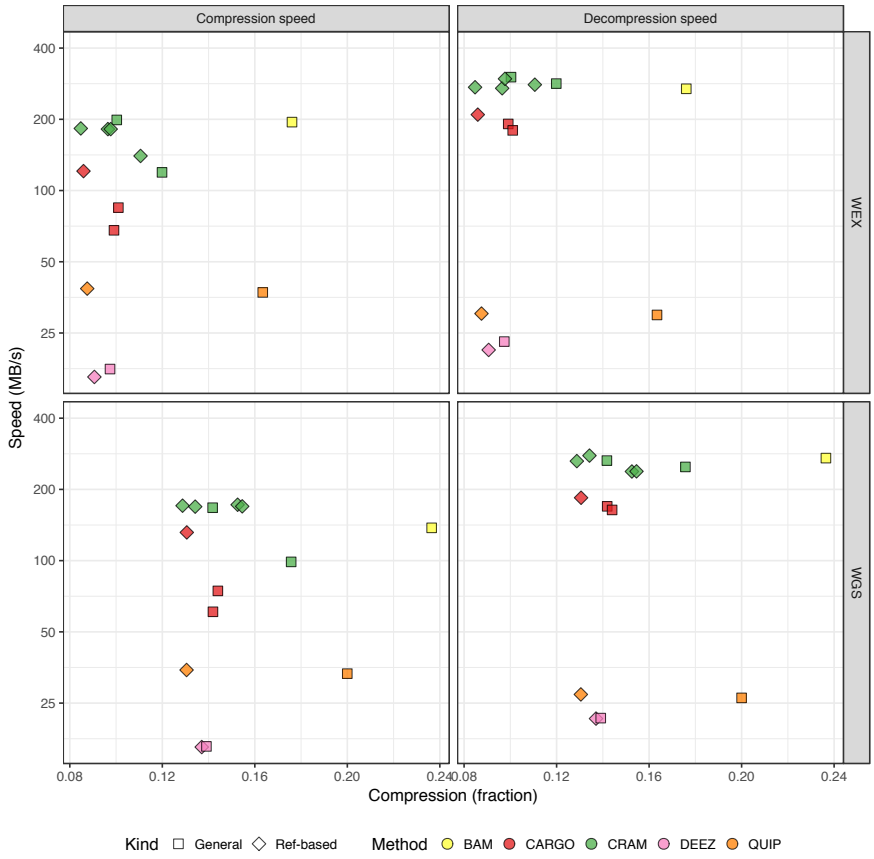

Figure 3.4: Comparison of performance speed between different SAM compression methods. *Fraction* denotes the ratio between the compressed and original file.

CHAPTER 4

DISCUSSION AND OUTLOOK

4.1 HTS data compression workflow

The approaches to storing and compressing HTS data, as described in Chapter 2, share a substantial number of underlying ideas. In general, all the methods can be divided in two main categories: general ones and ones based on reads preprocessing. The main difference between these two approaches is that in the former, the reads (primarily stored in FASTQ format) can be compressed "on-the-fly", generating a compressed archive almost immediately. In the latter, the reads follow a number of preprocessing steps in order to reduce the entropy of the input data, possibly casting the data into a different and more easily compressible form. During this process, they also possibly produce a number of temporary files and require a number of intermediate synchronization points between the separable data processing stages before the actual compression can be started. The HTS data compression process can as a whole be compared to the general data compression workflow as presented in Fig. 1.8, where each of the different read preprocessing steps can be perceived of as a form of the data transformation step.

Moreover, aligning raw reads to a reference genome, producing alignments in SAM format and sorting the resulting alignments by their mapping position (and generating a new file as a result) can also be seen as two combined complex data transformation steps. Although the primary purpose behind generating the intermediate mapping results in SAM format is usually to perform an analysis of possible genomic variants (as explained in Section 1.1.3), the SAM compression methods can also be used as an efficient way to store the HTS data. Therefore, different approaches to compress HTS data can be applied depending on the data formats used in

the genomic data analysis pipelines, storage capabilities and data processing requirements. Finally, storing the information from a re-sequencing experiment as a set of variants in VCF format, following the re-sequencing data analysis workflow as depicted in Fig. 1.3, can also be perceived of as a form of HTS data compression, albeit lossy, as recovering the raw reads is not possible.

4.2 Lossless compression of HTS data

4.2.1 Compression of sequence and alignment data

When compressing the raw reads in FASTQ format, the most commonly used methods are still the general ones. Some of their main advantages include usually implementing computationally lightweight algorithms, while providing a satisfactory compression ratio and achievable compression speeds. These allow to compress the HTS data as they are while being generated "on-the-fly". These methods usually also provide a relatively high decompression speed, allowing for an efficient ingestion of the data in the following data processing steps, such as mapping. Moreover, no additional information is required in order to compress or to decompress the data, which can sometimes be a big asset. Unfortunately, the reads represented in FASTQ format are still in the majority of cases compressed using general purpose text compression methods, such as gzip. These methods, although being easily paralellizable, provide a non-satisfactory compression ratio as has been shown in a number of experiments in the previous chapter. Perhaps, the most striking drawback of compressing FASTQ files by using gzip is that all the reads data are lumped together and compressed in chunks. As shown in the previous chapter, just splitting the reads content into different data streams and compressing them separately can already provide a significant reduction in the storage space, even when using the same compression algorithm. Hence, the general solutions we developed, such as DSRC2 or CARGO-based ones, can achieve a significantly better compression ratio than the commonly used gzip. Moreover, they are also easily parallelizable and do not sacrifice performance speed at the cost of providing high compression ratio as do the other state-of-the-art solutions. Therefore, these solutions can be easily exchanged with the existing gzip-based compression tools and adopted to genomic data

processing pipelines.

Moreover, compressing FASTQ files using read-reordering-based (e.g., FaStore, SCALCE [70]) or assembly-based methods (e.g., LEON [15]), usually allows to achieve a superior compression ratio compared to the general ones. These methods can exploit the high sequence redundancy present in the HTS data, which increases together with the sequencing depth of the sample. The possible improvements in the space savings come at the cost of performing an additional pass or passes over the input data prior to performing the actual compression. Because these methods spent more resources during compression stage, they allow for reducing significantly size of the data. Moreover, they usually do not apply any additional data transformations on the decompressed reads (e.g., reordering of the reads), providing relatively high decompression speeds. Therefore, such methods can be especially useful for data-sharing scenarios, when the raw reads can be compressed once into the smallest size possible with the goal of being shared efficiently. Alternatively, they can be also a good solution for a long-term storage. However, it is important to mention, that these solutions perform read preprocessing operations on a "global" scope and, hence, their scalability may depend on the sequencing depth of the input dataset.

Considering the reads mapped to the reference sequence and stored as alignments in SAM format. The produced SAM files occupy a significant amount of space, more than the input raw FASTQ file (as shown in Tab. 3.1), where the reads are annotated with the alignment information and some content originating from the raw reads may also appear duplicated (as in the case of reporting alternative alignments). Therefore, the data stored in the resulting SAM files can be considered as a superset of the data contained in the input FASTQ file(s). However, what is interesting, compressing such annotated reads sorted by their mapping position, using gzip-based BGZF algorithm and storing them in BAM format, gives better compression results than compressing the input FASTQ just by using gzip (both methods are considered as the *de facto* standard). These gains in space savings are primarily due to the locality of the sequence data, where the sequences sharing a high degree of similarity reside close to each other in the file, ordered by their mapping position in the reference genome. Nonetheless, one needs to remember that storing the alignments in BAM format is, still, a highly inefficient approach, as the alignment data is not

being decoupled, and, similarly as when compressing FASTQ files using gzip, all the data is lumped together and compressed. Storing the content of each of the SAM fields in a separate stream and compressing each stream independently using a more appropriate compressor already provides a significant reduction in the size of the resulting file.

The above-mentioned SAM format compression methods, although compressing reads mapped to a reference genome, do not require the reference sequence to be present neither during compression nor decompression. They are considered as general methods, compressing the alignment data as it is. However, when the reference sequence (which was used during the sequence mapping process) is available, using the reference-based compression methods can further reduce the file size, especially reducing the sequence and alignment data overhead. With the sequence mapping information, the sequence with possible mismatches can be encoded differentially with respect to the reference sequence and so is not required to be stored verbatim. The newest SCRAMBLE-based implementation of CRAM format [18] and the CARGO-based solution show the gains that can be achieved. Another advantage of using a reference-based compression method is that some of the optional fields which were generated during the mapping stage, such as *MD* (string denoting observed mismatching positions) or *NM* (string denoting edit distance with respect to the reference sequence), can be automatically re-generated while decompressing, allowing for additional reductions in storage costs. Therefore, these methods can be used as an efficient way to store the HTS data in compressed SAM format.

However, the significant improvement in compression of alignment data achievable by reference-based methods comes at a cost. Mainly, the decompression process requires the exact reference sequence used while compressing (and mapping) to be present, otherwise it is not possible to recover the compressed data. Therefore, explicit mechanisms or protocols to identify, manage and access the reference sequence in a non-ambiguous way are required, especially if the data is to be distributed. Nonetheless, as a partial solution for this problem, CRAM format supports embedding the reference sequence in the compressed archive, which does not degrade the compression ratio for the datasets with high coverage. Otherwise, non-reference based solutions should be considered. As a side note, in CARGO-based solutions, embedding a reference sequence could also be

possible, since datasets of different type and format can be stored inside a single CARGO container. Therefore, storing a compressed reference sequence as one, e.g., FASTA dataset and referencing it by another single or multiple SAM datasets would be an option definitely worth considering.

4.2.2 Compression of read identifiers

Apart from possibly improving the compression ratio of the sequence (and alignment) data, changing the initial order of the reads also has a significant impact on the compression ratio of the reads identifiers. DNA sequences present in the reads in the initial FASTQ files as generated by the sequencer are assumed to be stored in random order (but preserving the information about pairing between the sequences originating from the same fragment). Only the content present in the consecutive read identifiers in the same FASTQ file exhibits some sort of order, where, e.g., the numerical values associated with some tokens (such as arbitrary read number or flowcell coordinates) tend to appear in an increasing order. Therefore, as has been shown in the previous chapter, methods compressing the read identifiers of the raw reads stored in the non-altered order allow to achieve the best compression results compared to the ones that reorder the reads. The same applies in the case of compressing SAM files, where the reads are reordered according to their mapping position in the reference genome. However, there is a trade off to consider, in that, although reordering the reads can achieve superior compression results of sequence and alignment data, it does so at the cost of hampering the compression of the reads identifiers.

Therefore, in FaStore to mitigate the problem of compressing reordered read identifiers, when compressing FASTQ files generated from a paired-end library, we keep one read identifier per pair of reads, only encoding the possible differences between them (which usually would be token values of 1 or 2 at some position in the identifier identifying the read from the pair). Possibly, a similar way of handling paired reads could be implemented both in DSRC2 or in CARGO-based FASTQ compression solutions.

4.2.3 Potential challenges when compressing reordered reads

As has been previously noted, read preprocessing methods can provide the best compression gains for the sequence and alignment data, but at the cost of degrading compression ratio for the reads identifiers. Moreover, altering the order of the reads should not influence the compression of quality scores in general. Nonetheless, there is another issue which should be addressed when reordering the reads. As previously mentioned, the initial order of the DNA sequences generated by the sequencing machine (and stored in FASTQ files) is random. This is an important assumption, which helps to ensure the correctness of the reads mapping process. More specifically, some mappers, such as BWA-MEM [108] or GEM [127], when processing the FASTQ reads chunk-by-chunk in multi-threaded mode, try to estimate the paired-end library insert size (see Section 1.1.2) by analyzing the reads locally inside the chunks of data. In order to possibly achieve the same mapping results, the reads after decompression should be also possibly generated in a random order (assessing the randomness of the reads order *per se* already can be a challenging problem). As been pointed in [56], for some mappers randomly re-shuffling the reads in the input FASTQ files can already lead to reporting different alignments (and variant calling results). These differences, however, are mostly due to the reads which map to difficult genomic regions and which contain a high degree of sequence repetitions. Therefore, if after decompression the reads in FASTQ files are clustered in groups in an invalid way (e.g., the reads originating from a complex genomic region are stored sorted by mapping position) then some mappers may estimate the insert size incorrectly. Hence, as a hint, a theoretical fragment size (if available) set during the library preparation step could possibly be used by such mappers to help assess of the correctness of the mapping process. Otherwise, performing an additional step of read reordering after the decompression can be considered, such as random reordering or sorting the reads by read identifier (if available) to recover their order as in the initial FASTQ files generated by the sequencer. As a side note, in order to retrieve properly paired raw FASTQ reads from the alignments in SAM format, when using SAMTools [111] one needs to first sort the SAM alignments by their read identifiers.

4.3 Exploring lossy compression methods for HTS data

4.3.1 Compression of read identifiers

There are multiple possibilities that one can consider when storing HTS data with a controlled degree of information loss, where the selection of kinds of data to preserve depends on the use-cases, the data processing pipelines and storage requirements. The only kind of data which is required to be stored in a lossless manner is the DNA sequence. However, losing the initial order of the reads as produced by a sequencer should not be considered as information loss, as long as the pairing information between the reads is preserved. Unfortunately, both in FASTQ and SAM formats, the representation of reads pairing is not clearly defined. As mentioned in Section 1.1.3, in FASTQ files, the pairing is primarily carried by the reads identifiers, where, in addition, the reads reside in the same lines in two separate files or are stored interleaved in one file. In the case of SAM format, the pairing information is represented in a number of fields, but only for aligned reads – for unaligned reads it is kept on the read identifiers. Therefore, when considering lossy compression of the read identifiers, preserving pairing information between the reads must be ensured. In principle, preserving the pairing information between the reads in the identifier field would require the pair of reads to have a unique identifier shared between them.

Nonetheless, some data present in initial FASTQ read identifiers can be discarded during the data processing stages in genomic pipelines. For example, when performing reads mapping, mappers by default trim the read identifiers by removing the comments, i.e., the content after the first whitespace symbol. Such a decision is required to produce non-ambiguous query template names when generating SAM alignments – a whitespace symbol present in a query template name could break the SAM format parsing algorithms, which split the SAM record line into a number of fields using whitespace characters as separators. However, some sequencing protocols which use barcoding, can provide the barcode sequence stored in the read comment field. In such cases, mappers, such as BWA-MEM [108], provide an option to store the present barcode sequence as one of the optional fields (using *BC* tag). Moreover, after the mapping, some genomic data analysis protocols may use some data from the read

identifier during the alignments post-processing stage. For example, in the case of the HTS data produced by Illumina platforms, sometimes, when looking for sequence optical duplicates (see Section 1.1.3 for more details), the machine-specific flowcell coordinates can be stored in the main part of the read identifier. Nonetheless, after performing the reads mapping and further alignment post-processing steps (and possibly removing the read duplicates), the most important feature to assess is to keep a unique identifier per read or a pair of reads (which is also the requirement for preserving the pairing information between the reads using the current formats).

With the above in mind, in FaStore we primarily focused on the lossy compression of read identifiers implementing two methods (apart from the lossless one). In the first one, we remove the comments in the identifiers, in a way that is similar to that performed by mappers by default, which already gives significant savings in the storage space. In the second case, we do not store the identifiers, generating only a unique string per read (and a unique string per pair of reads in the paired-end mode) while decompressing the data – the generated identifiers will be different from the initial ones. Therefore, skipping storing identifiers allows for great savings in storage space. Alternatively, in DSRC2 we provide an option to select only a subset of fields (or tokens) from the identifiers, which will be preserved. In this way, for example, preserving only the unique read number and flowcell coordinates to be stored can be easily set by the user. Although in CARGO we did not explore lossy compression methods for the reads identifiers, adding a possible filtering of the read identifiers fields (both for SAM and FASTQ formats) should be a straightforward task, involving minimal modifications in the records parsing code.

4.3.2 Compression of quality scores

As has been shown in a number of experiments in Chapter 3 and pointed out by numerous researches [19, 126], quality scores are the most challenging type of data to compress, being characterized by a high entropy and containing a non-negligible amount of signal noise. They occupy the majority of the compressed space, both in FASTQ and SAM formats, greatly hampering the efficient sharing of the meaningful information carried with the sequencing reads.

As mentioned in Section 1.1.3, FASTQ format allows for representing up to 93 distinct quality score values (encoded as human-readable ASCII symbols). However, in practice usually no more than 60 values are used. Moreover, when performing analysis of HTS re-sequencing data which has been sequenced with high coverage, having such a vast resolution of values for representing quality scores becomes unnecessary. The information about DNA sequence alignments plays the most important role, as a significant number of reads should be available to support the evidence of possible occurrences of variants. As has been shown shown in Chapter 3, the efficiency of compressing DNA sequences increases with the sequencing coverage of the datasets. By contrast, the efficiency of compressing quality scores basically remains unchanged. Considering the fact that, per-base quality scores serve only as an auxiliary information in variant calling and are not frequently used by the mappers[1], expressing the probability of the erroneous base call (and not covering any other possible errors, which could occur during the sequencing process) sacrificing such a vast amount of space to store them is unnecessary.

Moreover, Illumina, starting with their sequencing platform HiSeq 2500, introduced an option to generate quality scores in a reduced resolution. This scheme, known also as "Illumina bining" [80], reduces the available range of quality values to only 8 values, significantly improving the compression ratio. In addition, it was also shown in [80] that producing quality scores in this reduced resolution does not degrade the efficiency of variant calling. Therefore, this strategy has been applied by default in the newer sequencers, such as HiSeq 3000, NextSeq, or HiSeq X family. As a proof-of-concept, in DSRC2, we implemented this bining strategy showing how much space can be saved by compressing FASTQ files with the reduced resolution of quality scores. Similarly, in CARGO, we applied the same strategy when compressing SAM files. In FaStore we went a bit further and explored a number of different quality compression schemes. In addition to the lossless and bining schemes, we integrated a method proposed by [126], which aims to reduce the mean square error of the quality values, significantly improving the compression ratio. Moreover, we implemented a more restrictive scheme, namely, binary thresholding, in which quality scores are represented as either "good" or "bad" (depending on the specified threshold) and which allows for the size of the compressed data to be

[1] Apart from being used as one of the factors to calculate the overall quality of mapping.

greatly reduced. The schemes implemented in FaStore could also be easily implemented in the CARGO framework either as separate codecs or data transformation steps – the possible integration of the novel compression methods into the CARGO framework will be discussed below.

Therefore, having the lossy compression methods in mind, to reduce the storage requirements for the HTS data, one would need to select an appropriate lossy quality scores compression scheme depending on the experiment set up or which genomic data analysis pipeline is used. However, this can be a very challenging task due to the absence of "ground truth" data sets which the results of the analyses using lossily-processed files can be compared with. Nonetheless, some recent initiatives, such as GIAB [222], provide a limited set of high-confidence SNP and INDEL variant calls, which can already be used as a starting point.

4.3.3 Compression of SAM optional fields

During the reads mapping stage and during the post-processing stage of alignments, some auxiliary alignment data can be generated. This extra data is primarily stored in the SAM optional fields. Some examples include: read group names (*RG* tag), edit distance to the reference sequence (*NM* tag), observed mismatching positions (*MD* tag) and a set of fields related to different quality scores. During the base quality scores recalibration step, as performed using GATK [44, 139], the alignments can be annotated with recalibrated quality values corresponding to the insertion (optional field with *BI* tag) or deletion (*BD* tag) type of variant (if any of them appear). Moreover, when modifications of the original quality scores (represented in SAM field *QUAL*) have been applied during any alignment post-processing step, the original quality values may be stored in the optional field denoted with the tag *OQ*. Therefore, at the end of alignment post-processing, different quality values can be stored in multiple fields, greatly increasing the size of the file. As a side note, in our comparison tests performed in Section 3.5, the SAM alignments contain only the original quality scores as the input raw reads, as they have been generated directly by the mapper and only sorted by the mapping position.

As previously mentioned, most of the intermediate data generated during the mapping and alignment post-processing stages is stored in the

optional fields and is primarily used as supporting data for variant calling. The raw FASTQ reads, mapped to a reference sequence and later annotated with intermediate data in SAM format can occupy up to twice the initial space (or even more when storing different data related to quality scores), both represented in raw text format or compressed in BAM format. Therefore, the decision to preserve the optional fields should primarily depend on the data storage and data processing use-cases. For example, when considering the archival or long-term storage of the data with the possibility of performing alignments post-processing in the future, a significant amount of the optional data can be easily discarded. Moreover, some of the optional fields such as *MD* and *NM* can be re-generated during the decompression stage by some reference-based SAM format compressors. Therefore, to possibly reduce the storage requirements of the alignments data to the minimum, the alignments could be stored in the form as initially generated by the mapper and by using efficient reference-based SAM format compressors such as ones based on CARGO or by tools implementing the newest version of CRAM format.

4.4 Integration of the developed methods

4.4.1 Integration between the methods

During our research we have designed and developed a number of HTS data compression methods both lossless and lossy with results comparable to or even better than the state-of-the-art. Some of these methods have been implemented as standalone compressors, such as DSRC2, ORCOM (compression of DNA sequences only), or FaStore and they are used to compress raw reads in FASTQ format. Moreover, by designing the CARGO framework to enable efficient representation, storage and processing of HTS data, many arbitrary format-free compression solutions can be easily and automatically generated. As a proof-of-concept, CARGO-based FASTQ and SAM format compression tools have been developed. Therefore, as a natural consequence, a possible integration between the designed methods should be considered.

In general, there are two possible ways to perform the integration among the developed solutions. In the first one, the compression methods implemented in DSRC2, ORCOM, and FaStore could be integrated into the

CARGO framework. More specifically, the sequence, quality, and read identifiers compression algorithms could be implemented as standalone codecs in CARGO. When generating CARGO-based compression solutions, these codecs (available alongside the current ones: gzip, bzip2, LZMA, and PPMd) could be selected explicitly by the user during the record data type definition step. This way, when processing the data, the new codecs could be applied to compressing the specified data streams. Having a rich set of different codecs would also allow for a rapid prototyping and fast exploration of novel compression solutions aimed at storing HTS data in an efficient way. For example, one could easily generate a CARGO-based SAM format compressor, utilizing the read identifiers or quality scores compression algorithms from DSRC2 or FaStore.

However, in case of integrating ORCOM or FaStore DNA sequence compression methods into CARGO, providing a special read reordering stage would be required, since FaStore uses a minimizer-based read reordering technique prior to the actual data compression. Fortunately, the CARGO framework provides the functionality of a custom user-defined transformation of the records applied prior to compression (or, also, directly after decompression). In this way, a read-reordering-based transformation as defined in ORCOM (and implemented as the bining stage) or FaStore (the binning and, possibly, re-binning stages) can be implemented in the generated records transformation module. In the next step, the actual DNA sequence compression codec based on FaStore or ORCOM could be selected to compress the preprocessed reads. Moreover, FaStore-based read reordering and compression methods could also be applied to compressing the unaligned reads when using a reference-based SAM format compression solution generated by CARGO, further improving the compression ratio.

A second idea for the integration of the methods developed could be to use the CARGO framework as a data storage back-end for DSRC2, ORCOM or FaStore. More specifically, all the (possibly compressed) data streams can be stored inside CARGO containers. Multiple datasets of different formats can be stored in one container. Each dataset is treated as a collection of records of a predefined uniform type (by the user), where the content present in the records is decoupled into separate data streams (following the classical approach as described in Chapter 2). Each of the streams can be compressed independently and stored as a collection of (possibly

compressed) blocks inside the container. All these data separation and compression steps are performed automatically by CARGO. In this way, for example, when compressing FASTQ files using DSRC2, the read identifiers, DNA sequences and quality scores data could be stored in separate CARGO streams. Moreover, storing the encoded reads data in streams as in ORCOM or FaStore would be equivalent to storing the data in CARGO streams. Therefore, the functionality of CARGO streams and containers could be used by other external solutions to store HTS data, not requiring them to define their own archive format.

4.4.2 Integration through a common API

To facilitate the storage of data inside containers, CARGO defines a simplified API layer, namely TypeAPI (implemented in the C++ programming language). It exposes the functionality of CARGO streams and allows for a transparent decomposition of the records' content, storing the data in streams. This way, given a high-level record type definition in CARGO meta-language provided by the user, a corresponding low-level definition of the record in C++ and in TypeAPI can be automatically generated. This definition is then used to automatically generate a CARGO-based application template code in C++, which could be customized by the user. Although the current API definition and offered functionality is still in its infancy and shows several limitations (primarily, as it used when generating standalone applications linked with the CARGO framework), the generated C++ record definition with the framework could be possibly used in developing external applications to store the data inside CARGO containers.

In DSRC2 we also introduced a simple, high-level Application Programming Interface (API) implemented both in C++ and Python programming languages. Some of the main features include compressing/decompressing single FASTQ records or whole FASTQ files/DSRC archives with manually (or automatically) specified compression setting by the user. It allows for an easy integration of DSRC2 compressor functionality with other external applications written in these languages. However, what is more important, when using this API to compress or decompress the data, the applications can operate directly on the buffered data when it is residing in memory, reducing the amount of resources needed to be spent on

streaming the data or handling temporary files.

As a good reference, a more complete API has been proposed in HTSlib and HTSjdk[2] libraries, which provide support for working with HTS data stored in SAM, BAM, CRAM, VCF, and BCF formats. For example, they provide functionality for reading, writing, and searching (by genomic position) performed both on the (compressed) files and on single (or multiple) records. These libraries are also used by the most popular genomic data processing and analysis tools such as SAMTools, Picard, or GATK. Clearly, reducing the amount of CPU time spent on reading/writing operations from/to disk and for performing data transcodding/parsing when accessing the HTS data is highly desirable, since the data already occupies a large amount of space. Therefore, refining the API in CARGO and integrating it as a simple proof-of-concept with other developed solutions would be a good exercise to encourage integration of other external applications, not only oriented to data compression. As a side note, some solutions, such as ADAM [136] or Goby [24] provide their own data processing ecosystem (ADAM) or data management framework (Goby) to store and access the HTS data. However, the problematic aspect of these solutions is that the external applications need to be built around the specific ecosystem or framework. This, combined with their own internal HTS data representations, can limit their practical usability in current genomic data processing pipelines, as additional CPU time will also be spent on inter-format conversions.

4.4.3 Integration of the methods with the genomic pipelines

Most commonly, HTS data are stored in compressed form in a number of files, e.g., multiple FASTQ files coming from one experiment. To work with the data, the files can be either fully decompressed or their contents can be decompressed "on-the-fly" and directly streamed out to other applications requesting it. Streaming, by default, is supported by the majority of the commonly used HTS data processing applications. Therefore, the possible integration of the developed solutions with current genomic data pipelines can be easily accomplished by exchanging the used compression applications with the developed ones. This is the most flexible approach, since the developed solutions can be integrated into current genomics pipelines in a transparent way – it does not require any inference into the

[2] http://www.htslib.org

external application source code, e.g., to handle working with the data through some API.

For example, DSRC2 can be used as a perfect replacement for gzip, allowing it to compress and decompress raw FASTQ reads "on-the-fly" or as files. When processing the data in the "fast" mode it offers high compression and decompression speeds, while providing a significant reduction in the compressed data compared to gzip. Alternatively, for a closer integration with external applications it also offers an API to work directly with DSRC2 archives and FASTQ files. For long-term storage of raw FASTQ reads, FaStore is the perfect option. It offers a superior compression ratio compared to the other available FASTQ state-of-the-art solutions, while providing relatively high decompression speeds (at the expense of slower compression speeds). Moreover, with a broad range of extra options to perform controlled lossy compression of the non-sequence data, additional savings in storage space can be substantial.

When considering storage and compression of aligned reads in SAM format, CARGO-based solutions also provide one of the highest achievable compression results. The solutions can be easily applied to compressing the alignments either when the reference sequence (used during the mapping stage) is available or not, being a suitable replacement for BAM and CRAM format-based compression solutions. Similarly, CARGO-based FASTQ format compressors can also be integrated into current genomic data processing pipelines in a transparent way. Moreover, one of the most important features of the CARGO framework is the independence from the HTS data format, offering a high degree of flexibility when working with HTS data. Possibly, any data format can be modeled and an efficient compressor can be semi-automatically generated (as has been done for FASTQ and SAM formats). Therefore, if any external application such as a mapper or assembler would need to output some intermediate results in a form not fitting into any of the currently available formats, CARGO could easily provide a way to represent and store the data in a compressed form. It can be especially useful when designing and developing pipelines, not being tied to any specific genomic data representation or file format. Moreover, multiple datasets of different format can be stored in one container, allowing for an easy encapsulation of data originating from a project or experiment. With a possible integration of all the developed methods and refinement of the current API, such a framework could provide an even more complete

solution to supporting the majority of genomic data-intensive workflows in an efficient way.

4.5 Future directions

As has already briefly mentioned in the previous section, one of the most important next steps would be the integration of the developed solutions. During the integration, providing a new and refined API would also be a very important feature for allowing easier integration of external tools. Moreover, it would be also worthwhile to design the API to be compatible with the one being currently developed by the Global Alliance for Genomics and Health (GA4GH[3]) [22] organization. The main goal of GA4GH is to define and develop a set of unified methods and protocols to enable responsible, privacy-preserving and effective sharing of genomic and clinical data among institutions and organizations around the world. These methods and protocols are independent of the underlying data format. Therefore, for our integrated solutions, designing a robust API compatible with the one provided by GA4GH API would be a solid step forward into the future of efficient genomic data sharing.

Another idea worth exploring could be an alternative (and better structured) representation of raw and aligned reads, where FASTQ and SAM formats are currently used as a *de facto* standard. Unfortunately, these formats, as has been shown in Sections 1.1.3 and 3.5, are greatly limited and cumbersome, hampering the efficiency of compression, as some information can not be properly decoupled. For example, in FASTQ format, a possible additional meta-data information or read annotation can be stored only in the read identifiers' line, which itself is a free format field and does not follow any standard. Moreover, in both formats, the pairing information between the reads is preserved in an ill-defined way. In addition, in SAM format, multiple different alignment information is stored together in one field, in non-meaningful binary flags, which could be easily decoupled into a number of different fields (e.g., of boolean type). This is also one of the primary motivations behind the ongoing work of Motion Pictures Experts Group (MPEG) from the International Standardization Organization (ISO) who in the past developed a number of popular video

[3] https://www.genomicsandhealth.org/

and audio formats used worldwide. Today, the MPEG group is exploring a new data representation and format for storing raw and aligned reads which can also possibly provide a higher compression than the current state-of-the-art.

Moreover, as has been shown in Section 2.4, neither FASTQ nor SAM formats are suitable for representing the data produced by the emerging Third Generation Sequencing platforms. The generated long reads data are accompanied by a significant amount of meta-data with additional sequencing-process-specific information, which cannot be feasibly represented either in FASTQ or SAM formats in a lossless manner. For example, in order to store the sequencing data generated by the PacBio platform and aligned to a reference genome, a number of extensions to the existing SAM format specification has been added. However, the conversion to SAM format is still a lossy process, discarding a significant amount of data. A similar problem applies to data generated by the Oxford Nanopore platform. Therefore, it would be worthwhile to explore alternative data representations in the CARGO framework, which could enable efficient storage (in a compressed form) and processing of the data generated by third-generation sequencing platforms. Moreover, a CARGO-based solution could also provide equivalent converters to FASTQ and SAM formats or an API to access only a specific subset of stored data.

CONCLUSIONS

With the rapid development of sequencing technologies, the inception of a number of nation-wide and international-scale sequencing projects, and the advent of personal genome re-sequencing, we are experiencing a flood of genomic data. This is why the main interest of the research described in this work was to explore and develop efficient techniques to compress and store the data generated by high-throughput sequencing platforms.

To summarize, the main contributions of this research are as follows.

- We developed DSRC2, a general, high-performance compressor for files in FASTQ format, based on the methods proposed in [42] with major improvements. DSRC2 can be used either as a standalone compressor or as a library enabling integration with third party applications and pipelines, allowing such applications to store FASTQ data in compressed form. On average, this allows us to reduce the size of FASTQ files by up to 20% of their initial size, and provides compression and decompression speeds reaching 500 MB/s when using 8 threads.

- We developed ORCOM, a proof-of-concept compressor for storing short-read sequence data which focuses on providing maximum compression ratio. ORCOM only supports compression of DNA sequences, and aims to exploit the significant sequence redundancy present in the data generated from deep sequencing experiments. ORCOM allowed us to compress the short-read data in a *H. sapiens* sample dataset of 134 GB (coverage ~42×) to only 4.3 GB (3.2%). This provided the best compression ratio achieved so far, which is significantly better than the results obtained by the existing state-of-the-art solutions.

- We developed FaStore, a complete solution for compressing FASTQ files and which supports short-read data sequenced both in single-end and paired-end configuration. FaStore extends the DNA compression methods introduced in ORCOM and adds a sequence assembly step, further improving the compression ratio. The DNA reads from the same *H. sapiens* dataset mentioned above can be compressed from 134 to only 3.4 GB (2.6%), or to 8.5 GB (6.5%) when preserving the pairing information between the reads. On average, it can compress FASTQ files by up to 12.5% in lossless mode. When compared to state-of-the-art methods FaStore achieves the best compression ratio, both for storing DNA sequences and full FASTQ reads. With FaStore, we also explored different lossy compression methods to encode read identifiers and base quality scores, showing how much space one can save by selecting an appropriate lossy scheme.

- We developed CARGO, a framework and a general binary format for compressed data, which allows one to semi-automatically create compression systems to store HTS data in configurable containers. One can generate CARGO-based solutions for any genomic file format: the user defines the record data type by using an abstract domain-specific language, alongside custom data transformations and querying and compression methods. As a proof of concept, we generated a family of compressors for the FASTQ format. Some of them achieve compression ratio and/or speed comparable to those provided by the specialized FASTQ compressors. Then, we created a second family of CARGO-based compressors, this time for the SAM format. They achieve compression results comparable to or better than those obtained by current state-of-the-art SAM compressors. Using our reference-based SAM format compressor with lossy base quality scores we managed to compress a 17-TB subset of SAM alignments from the 1000 Genomes Project [1] to 1.44 TB (8.4%), while still retaining the possibility to range-query the stored data by chromosome and position.

- The research-and-development stages for the projects above happened at different times. As a result, each method was typically tested in a different computational environment, and using different input datasets. Across the years we also experienced constant

improvements in the state of the art provided by our competitors. This is why, in one of the previous chapters, we performed an additional brief comparison of all our solutions within the context of the current literature, using a concise dataset and the same computational environment. We focused on our lossless compression methods for short-read data, represented as both raw FASTQ reads and aligned SAM records. We showed that such methods achieve the best results, or results comparable to those obtained by other state-of-the-art methods, in terms of both compression ratio and processing speed.

In short, we developed a variety of compression methods covering a set of different use-cases that can be found whenever genomic data is processed by bioinformatics pipelines or stored for later use. However, some of the methods we proposed are format-independent, and can be possibly reused for applications not considered here – for instance, to store data produced by emerging third-generation sequencing platforms. In general, all the solutions developed here allow for straightforward integration with existing genomic data processing pipelines, and can serve as a practical replacement for file-based data compression. We believe these solutions represent significant and reasonable steps toward a more accomplished approach to properly incorporating data compression into bioinformatics pipelines.

BIBLIOGRAPHY

[1] 1000 Genomes Project Consortium et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010. [cited at p. 13, 18, 68, 81, and 172]

[2] 1000 Genomes Project Consortium et al. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012. [cited at p. 13 and 18]

[3] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015. [cited at p. 13]

[4] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. *The design and implementation of modern column-oriented database systems*, volume 5. Now Publishers, Inc., 2013. [cited at p. 79]

[5] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. [cited at p. 22]

[6] K. Adhikari, T. Fontanil, S. Cal, J. Mendoza-Revilla, M. Fuentes-Guajardo, J.-C. Chacón-Duque, F. Al-Saadi, J. A. Johansson, M. Quinto-Sanchez, V. Acuña-Alonzo, et al. A genome-wide association scan in admixed Latin Americans identifies loci influencing facial and scalp hair features. *Nature communications*, 7:10815, 2016. [cited at p. 2]

[7] T. S. Alioto, I. Buchhalter, S. Derdak, B. Hutter, M. D. Eldridge, E. Hovig, L. E. Heisler, T. A. Beck, J. T. Simpson, L. Tonon, et al. A comprehensive assessment of somatic mutation detection in cancer using whole-genome sequencing. *Nature communications*, 6, 2015. [cited at p. 27]

[8] C. Alkan, B. P. Coe, and E. E. Eichler. Genome structural variation discovery and genotyping. *Nature Reviews Genetics*, 12(5):363–376, 2011. [cited at p. 27]

[9] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061–1067, 2009. [cited at p. 22 and 23]

[10] S. J. Aronson and H. L. Rehm. Building the foundation for genomics in precision medicine. *Nature*, 526(7573):336–342, 2015. [cited at p. 14 and 15]

[11] G. A. Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, et al. From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline. *Current protocols in bioinformatics*, pages 11–10, 2013. [cited at p. 18, 24, 25, 26, 27, and 138]

[12] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. [cited at p. 23]

[13] T. Batu and S. C. Sahinalp. Locally consistent parsing and applications to approximate string comparisons. In *International Conference on Developments in Language Theory*, pages 22–35. Springer, 2005. [cited at p. 62]

[14] M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *Annual Symposium on Combinatorial Pattern Matching*, pages 219–231. Springer, 2011. [cited at p. 63]

[15] G. Benoit, C. Lemaitre, D. Lavenier, E. Drezen, T. Dayris, R. Uricaru, and G. Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC bioinformatics*, 16(1), 2015. [cited at p. 59, 62, 65, 67, 139, 155, and 203]

[16] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986. [cited at p. 49]

[17] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. Assembling large genomes with single-molecule sequenc-

ing and locality-sensitive hashing. *Nature biotechnology*, 33(6):623–630, 2015. [cited at p. 11]

[18] J. K. Bonfield. The Scramble conversion tool. *Bioinformatics*, 30(19):2818–2819, 2014. [cited at p. 71, 72, 75, 76, 77, 78, 140, 156, and 206]

[19] J. K. Bonfield and M. V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PloS one*, 8(3), 2013. [cited at p. 54, 57, 58, 59, 60, 61, 64, 65, 67, 72, 73, 76, 77, 78, 139, and 160]

[20] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006. [cited at p. 62]

[21] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, I. Birol, S. Boisvert, J. A. Chapman, G. Chapuis, R. Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1), 2013. [cited at p. 21]

[22] J. Burn. A federated ecosystem for sharing genomic, clinical data. *Science*, 352(6291):1278–1280, 2016. [cited at p. 168]

[23] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994. [cited at p. 52]

[24] F. Campagne, K. C. Dorff, N. Chambwe, J. T. Robinson, and J. P. Mesirov. Compression of structured high-throughput sequencing data. *PLoS One*, 8(11), 2013. [cited at p. 84 and 166]

[25] R. Cánovas and A. Moffat. Practical compression for multi-alignment genomic files. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135*, pages 51–60. Australian Computer Society, Inc., 2013. [cited at p. 70, 73, and 76]

[26] M. J. Chaisson, J. Huddleston, M. Y. Dennis, P. H. Sudmant, M. Malig, F. Hormozdiari, F. Antonacci, U. Surti, R. Sandstrom, M. Boitano, et al. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*, 517(7536):608–611, 2015. [cited at p. 11]

[27] K. Chen, J. W. Wallis, M. D. McLellan, D. E. Larson, J. M. Kalicki, C. S. Pohl, S. D. McGrath, M. C. Wendl, Q. Zhang, D. P. Locke, et al.

BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods*, 6(9):677–681, 2009. [cited at p. 27]

[28] X. Chen, O. Schulz-Trieglaff, R. Shaw, B. Barnes, F. Schlesinger, M. Källberg, A. J. Cox, S. Kruglyak, and C. T. Saunders. Manta: rapid detection of structural variants and indels for germline and cancer sequencing applications. *Bioinformatics*, 32(8):1220–1222, 2016. [cited at p. 27]

[29] K. Cibulskis, M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology*, 31(3):213–219, 2013. [cited at p. 27]

[30] R. Cijvat, S. Manegold, M. Kersten, G. W. Klau, A. Schönhuth, T. Marschall, and Y. Zhang. Genome sequence analysis with MonetDB. *Datenbank-Spektrum*, 15(3):185–191, 2015. [cited at p. 79]

[31] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature nanotechnology*, 4(4):265–270, 2009. [cited at p. 11]

[32] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984. [cited at p. 41]

[33] G. Cochrane, C. E. Cook, and E. Birney. The future of DNA sequence archiving. *GigaScience*, 1(1), 2012. [cited at p. 54]

[34] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2010. [cited at p. 16, 28, 29, and 60]

[35] E. M. Coonrod, J. D. Durtschi, R. L. Margraf, and K. V. Voelkerding. Developing genome and exome sequencing for candidate gene identification in inherited disorders: an integrated technical and bioinformatics approach. *Archives of pathology & laboratory medicine*, 137(3):415–433, 2013. [cited at p. 15]

[36] A. Cornish-Bowden. IUPAC-IUB symbols for nucleotide nomenclature. *Nucleic Acids Res*, 13(3021):30, 1985. [cited at p. 28]

[37] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012. [cited at p. 63]

[38] C. Cruchaga, C. M. Karch, S. C. Jin, B. A. Benitez, Y. Cai, R. Guerreiro, O. Harari, J. Norton, J. Budde, S. Bertelsen, et al. Rare coding variants in the phospholipase D3 gene confer risk for Alzheimer/'s disease. *Nature*, 505(7484):550–554, 2014. [cited at p. 2]

[39] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011. [cited at p. 18, 33, and 35]

[40] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier. Halvade: scalable sequence analysis with MapReduce. *Bioinformatics*, 31(15):2482–2488, 2015. [cited at p. 18]

[41] S. Deorowicz. Universal lossless data compression algorithms. *Philosophy Dissertation Thesis, Gliwice*, 2003. [cited at p. 38 and 43]

[42] S. Deorowicz and S. Grabowski. Compression of DNA sequence reads in FASTQ format. *Bioinformatics*, 27(6):860–862, 2011. [cited at p. 58, 64, 93, and 171]

[43] S. Deorowicz and S. Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8(1), 2013. [cited at p. 54 and 56]

[44] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics*, 43(5):491–498, 2011. [cited at p. 18, 25, 26, and 162]

[45] K. C. Dorff, N. Chambwe, Z. Zeno, M. Simi, R. Shaknovich, and F. Campagne. GobyWeb: simplified management and analysis of gene expression and DNA methylation sequencing data. *PLoS One*, 8(7), 2013. [cited at p. 84]

[46] S. Dorok. The relational way to dam the flood of genome data. In *Proceedings of the 2015 ACM SIGMOD on PhD Symposium*, pages 9–13. ACM, 2015. [cited at p. 80]

[47] J. Duda. Asymmetric numeral systems. *arXiv preprint arXiv:0902.0271*, 2009. [cited at p. 48 and 71]

[48] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *Picture Coding Symposium (PCS), 2015*, pages 65–69. IEEE, 2015. [cited at p. 48]

[49] A. Dutta, M. M. Haque, T. Bose, C. Reddy, and S. S. Mande. FQC: A novel approach for efficient compression, archival, and dissemination of fastq datasets. *Journal of bioinformatics and computational biology*, 13(03), 2015. [cited at p. 59, 60, 64, and 67]

[50] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009. [cited at p. 10]

[51] R. Ekblom and J. B. Wolf. A field guide to whole-genome sequencing, assembly and annotation. *Evolutionary applications*, 7(9):1026–1042, 2014. [cited at p. 21]

[52] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. [cited at p. 43]

[53] A. D. Ewing, K. E. Houlahan, Y. Hu, K. Ellrott, C. Caloian, T. N. Yamaguchi, J. C. Bare, C. P'ng, D. Waggott, V. Y. Sabelnykova, et al. Combining tumor genome simulation with crowdsourcing to benchmark somatic single-nucleotide-variant detection. *Nature methods*, 12(7):623–630, 2015. [cited at p. 27]

[54] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome research*, 8(3):186–194, 1998. [cited at p. 19]

[55] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000. [cited at p. 22 and 53]

[56] C. Firtina and C. Alkan. On genomic repeats and reproducibility. *Bioinformatics*, 32(15):2243–2247, 2016. [cited at p. 62 and 158]

[57] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011. [cited at p. 82]

[58] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome research*, 21(5):734–740, 2011. [cited at p. 74 and 76]

[59] R. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978. [cited at p. 46]

[60] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012. [cited at p. 26 and 27]

[61] Genome of the Netherlands Consortium et al. Whole-genome sequence variation, population structure and demographic history of the Dutch population. *Nature Genetics*, 46(8):818–825, 2014. [cited at p. 14]

[62] T. C. Glenn. Field guide to next-generation DNA sequencers. *Molecular ecology resources*, 11(5):759–769, 2011. [cited at p. 8 and 9]

[63] S. Golomb. Run-length Encodings (Corresp.). *IEEE Trans. Inf. Theor.*, 12(3):399–401, 2006. [cited at p. 49]

[64] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966. [cited at p. 43]

[65] S. Goodwin, J. D. McPherson, and W. R. McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, 2016. [cited at p. 8, 9, and 11]

[66] S. Grabowski, S. Deorowicz, and Ł. Roguski. Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9):1389–1395, 2015. [cited at p. 93]

[67] A. L. Greninger, S. N. Naccache, S. Federman, G. Yu, P. Mbala, V. Bres, D. Stryke, J. Bouquet, S. Somasekar, J. M. Linnen, et al. Rapid metagenomic identification of viral pathogens in clinical samples by real-

time nanopore sequencing analysis. *Genome medicine*, 7(1), 2015. [cited at p. 11]

[68] M. Griffith, C. A. Miller, O. L. Griffith, K. Krysiak, Z. L. Skidmore, A. Ramu, J. R. Walker, H. X. Dang, L. Trani, D. E. Larson, et al. Optimizing cancer genome sequencing and analysis. *Cell systems*, 1(3):210–223, 2015. [cited at p. 15]

[69] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature methods*, 7(8):576–577, 2010. [cited at p. 22]

[70] F. Hach, I. Numanagić, C. Alkan, and S. C. Sahinalp. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012. [cited at p. 62, 65, 66, 76, 139, 155, and 203]

[71] F. Hach, I. Numanagic, and S. C. Sahinalp. DeeZ: reference-based compression by local assembly. *Nature methods*, 11(11):1082–1084, 2014. [cited at p. 70, 72, 75, 76, 78, 140, and 207]

[72] D. Haussler, D. A. Patterson, M. Diekhans, A. Fox, M. Jordan, A. D. Joseph, S. Ma, B. Paten, S. Shenker, T. Sittler, et al. A million cancer genome warehouse. Technical report, DTIC Document, 2012. [cited at p. 54]

[73] M. Henderson. Genetic mapping of babies by 2019 will transform preventive medicine. *The Times. 9th February*, 2009. [cited at p. 14]

[74] D. S. Hirschberg and D. A. Lelewer. Context modeling for text compression. In *Image and Text Compression*, pages 113–144. Springer, 1992. [cited at p. 40 and 41]

[75] M. Howison. High-throughput compression of FASTQ data with SeqDB. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 10(1):213–218, 2013. [cited at p. 56 and 84]

[76] D. A. Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. [cited at p. 44]

[77] S. Hwang, E. Kim, I. Lee, and E. M. Marcotte. Systematic comparison of variant calling pipelines using gold standard personal exome

variants. *Scientific reports*, 5, 2015. [cited at p. 26]

[78] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al. MonetDB: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012. [cited at p. 79]

[79] Illumina. Estimating Sequencing Coverage. Technical report, 2014. [cited at p. 6]

[80] Illumina. Reducing Whole-Genome Data Storage Footprint. Technical report, 2014. [cited at p. 65, 66, 77, and 161]

[81] International Human Genome Sequencing Consortium et al. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004. [cited at p. 5]

[82] L. Janin, G. Rosone, and A. J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, 30(1):24–30, 2013. [cited at p. 67]

[83] L. Janin, O. Schulz-Trieglaff, and A. J. Cox. BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics*, 30(19):2796–2801, 2014. [cited at p. 63]

[84] P. D. Johnson Jr, G. A. Harris, and D. Hankerson. *Introduction to information theory and data compression*. CRC press, 2003. [cited at p. 43]

[85] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22), 2012. [cited at p. 59, 60, 62, 65, 72, 73, 76, 139, and 202]

[86] C. Kandoth, M. D. McLellan, F. Vandin, K. Ye, B. Niu, C. Lu, M. Xie, Q. Zhang, J. F. McMichael, M. A. Wyczalkowski, et al. Mutational landscape and significance across 12 major cancer types. *Nature*, 502(7471):333–339, 2013. [cited at p. 2]

[87] C. Kingsford and R. Patro. Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928, 2015. [cited at p. 63]

[88] L. Klein and A. Taaheri. HDF-EOS5 Data Model, File Format and Library. Technical Report July 2006, 2007. [cited at p. 83]

[89] D. E. Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985. [cited at p. 45]

[90] D. C. Koboldt, K. Chen, T. Wylie, D. E. Larson, M. D. McLellan, E. R. Mardis, G. M. Weinstock, R. K. Wilson, and L. Ding. VarScan: variant detection in massively parallel sequencing of individual and pooled samples. *Bioinformatics*, 25(17):2283–2285, 2009. [cited at p. 27]

[91] D. C. Koboldt, Q. Zhang, D. E. Larson, D. Shen, M. D. McLellan, L. Lin, C. A. Miller, E. R. Mardis, L. Ding, and R. K. Wilson. VarScan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome research*, 22(3):568–576, 2012. [cited at p. 27]

[92] Y. Kodama, M. Shumway, and R. Leinonen. The Sequence Read Archive: explosive growth of sequencing data. *Nucleic acids research*, 40(D1):D54–D56, 2012. [cited at p. 54]

[93] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing genomic sequence fragments using SlimGene. *Journal of Computational Biology*, 18(3):401–413, 2011. [cited at p. 64 and 74]

[94] V. Kuleshov, C. Jiang, W. Zhou, F. Jahanbani, S. Batzoglou, and M. Snyder. Synthetic long-read sequencing reveals intraspecies diversity in the human microbiome. *Nature biotechnology*, 34(1):64–69, 2016. [cited at p. 12]

[95] D. Laehnemann, A. Borkhardt, and A. C. McHardy. Denoising DNA deep sequencing data—high-throughput sequencing errors and their correction. *Briefings in bioinformatics*, 17(1):154–179, 2016. [cited at p. 18, 20, and 64]

[96] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001. [cited at p. 10]

[97] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, and W. e. a. FitzHugh. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001. [cited at p. 6 and 12]

[98] E. S. Lander and M. S. Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–239, 1988. [cited at p. 5]

[99] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984. [cited at p. 46]

[100] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012. [cited at p. 22 and 23]

[101] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3), 2009. [cited at p. 22]

[102] R. M. Layer, C. Chiang, A. R. Quinlan, and I. M. Hall. LUMPY: a probabilistic framework for structural variant discovery. *Genome biology*, 15(6), 2014. [cited at p. 27]

[103] R. Leinonen, H. Sugawara, and M. Shumway. The sequence read archive. *Nucleic acids research*, 39:D19–D21, 2010. [cited at p. 81]

[104] M. Lek, K. J. Karczewski, E. V. Minikel, K. E. Samocha, E. Banks, T. Fennell, A. H. O'Donnell-Luria, J. S. Ware, A. J. Hill, B. B. Cummings, et al. Analysis of protein-coding genetic variation in 60,706 humans. *Nature*, 536(7616):285–291, 2016. [cited at p. 13]

[105] M. J. Levene, J. Korlach, S. W. Turner, M. Foquet, H. G. Craighead, and W. W. Webb. Zero-mode waveguides for single-molecule analysis at high concentrations. *Science*, 299(5607):682–686, 2003. [cited at p. 10]

[106] D. Levy, G. B. Ehret, K. Rice, G. C. Verwoert, L. J. Launer, A. Dehghan, N. L. Glazer, A. C. Morrison, A. D. Johnson, T. Aspelund, et al. Genome-wide association study of blood pressure and hypertension. *Nature genetics*, 41(6):677–687, 2009. [cited at p. 2]

[107] S. Levy, G. Sutton, P. C. Ng, L. Feuk, A. L. Halpern, B. P. Walenz, N. Axelrod, J. Huang, E. F. Kirkness, G. Denisov, et al. The diploid genome sequence of an individual human. *PLoS Biol*, 5(10), 2007. [cited at p. 2, 5, and 14]

[108] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013. [cited at p. 22, 23, 138, 158, 159, and 201]

[109] H. Li. Towards better understanding of artifacts in variant calling from high-coverage samples. *Bioinformatics*, 30(20):2843–2851, 2014. [cited at p. 25 and 26]

[110] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. [cited at p. 22 and 84]

[111] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009. [cited at p. 17, 18, 23, 26, 30, 33, 158, and 200]

[112] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010. [cited at p. 22]

[113] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008. [cited at p. 22]

[114] P. Li, X. Jiang, S. Wang, J. Kim, H. Xiong, and L. Ohno-Machado. HUGO: Hierarchical mUlti-reference Genome cOmpression for aligned reads. *Journal of the American Medical Informatics Association*, 21(2):363–373, 2014. [cited at p. 72 and 75]

[115] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008. [cited at p. 22]

[116] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. [cited at p. 22]

[117] Y. Li, H. Zheng, R. Luo, H. Wu, H. Zhu, R. Li, H. Cao, B. Wu, S. Huang, H. Shao, et al. Structural variation in two human genomes mapped at single-nucleotide resolution by whole genome de novo assembly. *Nature biotechnology*, 29(8):723–730, 2011. [cited at p. 27]

[118] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, et al. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012. [cited at p. 75]

[119] K. E. Lohmueller, T. Sparsø, Q. Li, E. Andersson, T. Korneliussen, A. Albrechtsen, K. Banasik, N. Grarup, I. Hallgrimsdottir, K. Kiil, et al. Whole-exome sequencing of 2,000 Danish individuals and the role of rare coding variants in type 2 diabetes. *The American Journal of Human Genetics*, 93(6):1072–1086, 2013. [cited at p. 14]

[120] N. J. Loman, J. Quick, and J. T. Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature methods*, 12(8):733–735, 2015. [cited at p. 11]

[121] N. J. Loman and A. R. Quinlan. Poretools: a toolkit for analyzing nanopore sequence data. *Bioinformatics*, 30(23):3399–3401, 2014. [cited at p. 90]

[122] R. Luo, Y.-L. Wong, W.-C. Law, L.-K. Lee, J. Cheung, C.-M. Liu, and T.-W. Lam. BALSA: integrated secondary analysis for whole-genome and whole-exome sequencing, accelerated by GPU. *PeerJ*, 2, 2014. [cited at p. 18]

[123] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967. [cited at p. 67]

[124] M. Mahoney. Data compression explained. *mattmahoney. net, updated May*, 7, 2012. [cited at p. 41]

[125] M. V. Mahoney. Adaptive weighing of context models for lossless data compression. Technical report, 2005. [cited at p. 59]

[126] G. Malysa, M. Hernaez, I. Ochoa, M. Rao, K. Ganesan, and T. Weissman. QVZ: lossy compression of quality values. *Bioinformatics*, 31(19):3122–3129, 2015. [cited at p. 67, 76, 160, and 161]

[127] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, 9(12):1185–1188, 2012. [cited at p. 22, 23, and 158]

[128] E. R. Mardis. Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet.*, 9:387–402, 2008. [cited at p. 9]

[129] E. R. Mardis. A decade/'s perspective on DNA sequencing technology. *Nature*, 470(7333):198–203, 2011. [cited at p. 7]

[130] E. R. Mardis. Next-generation sequencing platforms. *Annual review of analytical chemistry*, 6:287–303, 2013. [cited at p. 9]

[131] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, L. A. Bemben, J. Berka, M. S. Braverman, Y.-J. Chen, Z. Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005. [cited at p. 9]

[132] N. Mark. Arithmetic Coding+ Statistical Modeling= Data Compression. *Dr. Dobb's Journal*, 1991. [cited at p. 41]

[133] M. Marmor, K. Hertzmark, S. M. Thomas, P. N. Halkitis, and M. Vogler. Resistance to HIV infection. *Journal of urban health*, 83(1):5–17, 2006. [cited at p. 2]

[134] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, 1979. [cited at p. 48]

[135] C. E. Mason, P. Zumbo, S. Sanders, M. Folk, D. Robinson, R. Aydt, M. Gollery, M. Welsh, N. E. Olson, and T. M. Smith. Standardizing the next generation of bioinformatics software development with BioHDF (HDF5). In *Advances in Computational Biology*, pages 693–700. Springer, 2010. [cited at p. 83]

[136] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013*, 207, 2013. [cited at p. 86 and 166]

[137] A. M. Maxam and W. Gilbert. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences*, 74(2):560–564, 1977. [cited at p. 3]

[138] R. S. McBean, C. A. Hyland, and R. L. Flower. Approaches to determination of a full profile of blood group genotypes: single nucleotide variant mapping and massively parallel sequencing. *Computational and structural biotechnology journal*, 11(19):147–151, 2014. [cited at p. 2]

[139] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al.

The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research*, 20(9):1297–1303, 2010. [cited at p. 18, 26, and 162]

[140] I. Mendizabal, O. Lao, U. M. Marigorta, A. Wollstein, L. Gusmão, V. Ferak, M. Ioana, A. Jordanova, R. Kaneva, A. Kouvatsi, et al. Reconstructing the population history of European Romani from genome-wide data. *Current Biology*, 22(24):2342–2349, 2012. [cited at p. 2]

[141] M. L. Metzker. Sequencing technologies—the next generation. *Nature reviews genetics*, 11(1):31–46, 2010. [cited at p. 9]

[142] A. E. Minoche, J. C. Dohm, and H. Himmelbauer. Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and genome analyzer systems. *Genome biology*, 12(11), 2011. [cited at p. 64]

[143] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on communications*, 38(11):1917–1921, 1990. [cited at p. 41]

[144] M. Morey, A. Fernández-Marmiesse, D. Castiñeiras, J. M. Fraga, M. L. Couce, and J. A. Cocho. A glimpse into past, present, and future DNA sequencing. *Molecular genetics and metabolism*, 110(1):3–24, 2013. [cited at p. 5 and 9]

[145] N. Nagarajan and M. Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, 2013. [cited at p. 16, 20, and 21]

[146] G. Narzisi and B. Mishra. Comparing de novo genome assembly: the long and short of it. *PloS one*, 6(4), 2011. [cited at p. 21]

[147] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. [cited at p. 23]

[148] S. Nik-Zainal, H. Davies, J. Staaf, M. Ramakrishna, D. Glodzik, X. Zou, I. Martincorena, L. B. Alexandrov, S. Martin, D. C. Wedge, et al. Landscape of somatic mutations in 560 breast cancer whole-genome sequences. *Nature*, 534(7605):47–54, 2016. [cited at p. 2]

[149] F. Nothaft. Scalable Genome Resequencing with ADAM and avocado. Technical report, Tech. Report No.: UCB/EECS-20IS-6S, UC Berkeley,

2015. [cited at p. 86]

[150] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, et al. Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 631–646. ACM, 2015. [cited at p. 86]

[151] I. Numanagić, J. K. Bonfield, F. Hach, J. Voges, J. Ostermann, C. Alberti, M. Mattavelli, and S. C. Sahinalp. Comparison of high-throughput sequencing data compression tools. *Nature Methods*, 2016. [cited at p. 56 and 139]

[152] I. Ochoa, H. Asnani, D. Bharadia, M. Chowdhury, T. Weissman, and G. Yona. QualComp: a new lossy compressor for quality scores based on rate distortion theory. *BMC bioinformatics*, 14(1), 2013. [cited at p. 67 and 76]

[153] I. Ochoa, M. Hernaez, R. Goldfeder, T. Weissman, and E. Ashley. Effect of lossy compression of quality scores on variant calling. *Briefings in bioinformatics*, 18(2):183–194, 2016. [cited at p. 65]

[154] M. V. Olson and A. Varki. Sequencing the chimpanzee genome: insights into human evolution and disease. *Nature Reviews Genetics*, 4(1):20–28, 2003. [cited at p. 2]

[155] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M. R. Speicher, J. Zschocke, and Z. Trajanoski. A survey of tools for variant analysis of next-generation genome sequencing data. *Briefings in bioinformatics*, 15(2):256–278, 2014. [cited at p. 15, 18, 20, and 26]

[156] Pacific Biosciences. bas.h5 Reference Guide. Technical report, 2013. [cited at p. 89]

[157] R. K. Patel and M. Jain. NGS QC Toolkit: a toolkit for quality control of next generation sequencing data. *PloS one*, 7(2), 2012. [cited at p. 19]

[158] R. Patro and C. Kingsford. Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17):2770–2777, 2015. [cited at p. 63]

[159] J. Peterson, S. Garges, M. Giovanni, P. McInnes, L. Wang, J. A. Schloss, V. Bonazzi, J. E. McEwen, K. A. Wetterstrand, C. Deal, et al. The NIH human microbiome project. *Genome research*, 19(12):2317–2323, 2009. [cited at p. 81]

[160] M. Pirooznia, F. S. Goes, and P. P. Zandi. Whole-genome CNV analysis: advances in computational approaches. *Frontiers in genetics*, 6, 2015. [cited at p. 27]

[161] M. Pirooznia, M. Kramer, J. Parla, F. S. Goes, J. B. Potash, W. R. McCombie, and P. P. Zandi. Validation and assessment of variant calling pipelines for next-generation sequencing. *Human genomics*, 8(1), 2014. [cited at p. 25]

[162] N. Popitsch and A. von Haeseler. NGC: lossless and lossy compression of aligned high-throughput sequencing data. *Nucleic acids research*, 41(1), 2013. [cited at p. 72, 73, 77, and 78]

[163] K. Prüfer, F. Racimo, N. Patterson, F. Jay, S. Sankararaman, S. Sawyer, A. Heinze, G. Renaud, P. H. Sudmant, C. De Filippo, et al. The complete genome sequence of a Neanderthal from the Altai Mountains. *Nature*, 505(7481):43–49, 2014. [cited at p. 2]

[164] M. J. Puckelwartz, L. L. Pesce, V. Nelakuditi, L. Dellefave-Castillo, J. R. Golbus, S. M. Day, T. P. Cappola, G. W. Dorn, I. T. Foster, and E. M. McNally. Supercomputing for the parallelization of whole genome analysis. *Bioinformatics*, 30(11):1508–1513, 2014. [cited at p. 24]

[165] J. Quick, N. J. Loman, S. Duraffour, J. T. Simpson, E. Severi, L. Cowley, J. A. Bore, R. Koundouno, G. Dudas, A. Mikhail, et al. Real-time, portable genome sequencing for Ebola surveillance. *Nature*, 530(7589):228–232, 2016. [cited at p. 11]

[166] C. Raczy, R. Petrovski, C. T. Saunders, I. Chorny, S. Kruglyak, E. H. Margulies, H.-Y. Chuang, M. Källberg, S. A. Kumar, A. Liao, et al. Isaac: ultra-fast whole-genome secondary analysis on Illumina sequencing platforms. *Bioinformatics*, 29(16):2041–2043, 2013. [cited at p. 18]

[167] T. Rausch, T. Zichner, A. Schlattl, A. M. Stütz, V. Benes, and J. O. Korbel. DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339, 2012. [cited at p. 27]

[168] K. Reinert, B. Langmead, D. Weese, and D. J. Evers. Alignment of Next-Generation Sequencing Reads. *Annual review of genomics and human genetics*, 16:133–151, 2015. [cited at p. 22 and 23]

[169] A. Rhoads and K. F. Au. PacBio sequencing and its applications. *Genomics, proteomics & bioinformatics*, 13(5):278–289, 2015. [cited at p. 11]

[170] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. Twigg, A. O. Wilkie, G. McVean, G. Lunter, WGS500 Consortium, et al. Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics*, 46(8):912–918, 2014. [cited at p. 26 and 27]

[171] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of research and development*, 20(3):198–203, 1976. [cited at p. 46]

[172] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. [cited at p. 63]

[173] J. T. Robinson, H. Thorvaldsdóttir, W. Winckler, M. Guttman, E. S. Lander, G. Getz, and J. P. Mesirov. Integrative genomics viewer. *Nature biotechnology*, 29(1):24–26, 2011. [cited at p. 84]

[174] Ł. Roguski and S. Deorowicz. DSRC 2—Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215, 2014. [cited at p. 93 and 203]

[175] Ł. Roguski and P. Ribeca. CARGO: effective format-free compressed storage of genomic information. *Nucleic acids research*, 44(12), 2016. [cited at p. 93 and 204]

[176] M. N. Sakib, J. Tang, W. J. Zheng, and C.-T. Huang. Improving transmission efficiency of large sequence alignment/map (SAM) files. *PloS one*, 6(12), 2011. [cited at p. 72 and 78]

[177] D. Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004. [cited at p. 46, 47, 48, and 52]

[178] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, et al. GAGE: A

critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012. [cited at p. 21]

[179] F. Sanger, A. Coulson, T. Friedman, G. Air, B. Barell, N. Brown, J. Fiddes, C. Hitchinson III, P. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage ϕD X174 DNA. *Journal of molecular biology*, 2(125), 1978. [cited at p. 3]

[180] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977. [cited at p. 3]

[181] C. T. Saunders, W. S. Wong, S. Swamy, J. Becq, L. J. Murray, and R. K. Cheetham. Strelka: accurate somatic small-variant calling from sequenced tumor–normal sample pairs. *Bioinformatics*, 28(14):1811–1817, 2012. [cited at p. 25 and 27]

[182] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics*, 11(9):647–657, 2010. [cited at p. 16]

[183] R. Schmieder and R. Edwards. Quality control and preprocessing of metagenomic datasets. *Bioinformatics*, 27(6):863–864, 2011. [cited at p. 19]

[184] G. I. Shamir and N. Merhav. Low-complexity sequential lossless coding for piecewise-stationary memoryless sources. *IEEE transactions on information theory*, 45(5):1498–1519, 1999. [cited at p. 43]

[185] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001. [cited at p. 41]

[186] D. Shkarin. PPM: One step to practicality. In *Data Compression Conference, 2002. Proceedings. DCC 2002*, pages 202–211. IEEE, 2002. [cited at p. 41]

[187] D. Sims, I. Sudbery, N. E. Ilott, A. Heger, and C. P. Ponting. Sequencing depth and coverage: key considerations in genomic analyses. *Nature Reviews Genetics*, 15(2):121–132, 2014. [cited at p. 6 and 15]

[188] L. M. Smith, J. Z. Sanders, R. J. Kaiser, P. Hughes, C. Dodd, C. R. Connell, C. Heiner, S. Kent, and L. E. Hood. Fluorescence detection

in automated DNA sequence analysis. *Nature*, 321(6071):674–679, 1985. [cited at p. 4]

[189] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981. [cited at p. 23]

[190] J. A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., 1988. [cited at p. 52]

[191] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982. [cited at p. 52]

[192] P. H. Sudmant, T. Rausch, E. J. Gardner, R. E. Handsaker, A. Abyzov, J. Huddleston, Y. Zhang, K. Ye, G. Jun, M. H.-Y. Fritz, et al. An integrated map of structural variation in 2,504 human genomes. *Nature*, 526(7571):75–81, 2015. [cited at p. 13]

[193] P. J. Talmud, J. A. Cooper, R. W. Morris, F. Dudbridge, T. Shah, J. Engmann, C. Dale, J. White, S. McLachlan, D. Zabaneh, et al. Sixty-five common genetic variants and prediction of type 2 diabetes. *Diabetes*, page DB_141504, 2014. [cited at p. 2]

[194] H. Tang, E. Lyons, and C. D. Town. Optical mapping in plant comparative genomics. *GigaScience*, 4(1), 2015. [cited at p. 21]

[195] L. Tattini, R. D'Aurizio, and A. Magi. Detection of genomic structural variants from next-generation sequencing data. *Frontiers in bioengineering and biotechnology*, 3, 2015. [cited at p. 27]

[196] W. Tembe, J. Lowey, and E. Suh. G-SQZ: compact encoding of genomic sequence and quality data. *Bioinformatics*, 26(17):2192–2194, 2010. [cited at p. 56]

[197] J. A. Tennessen, A. W. Bigham, T. D. O'Connor, W. Fu, E. E. Kenny, S. Gravel, S. McGee, R. Do, X. Liu, G. Jun, et al. Evolution and functional impact of rare coding variation from deep sequencing of human exomes. *science*, 337(6090):64–69, 2012. [cited at p. 2]

[198] S. G. Tringe and E. M. Rubin. Metagenomics: DNA sequencing of environmental samples. *Nature reviews genetics*, 6(11):805–814, 2005. [cited at p. 2]

[199] UK10K Consortium et al. The UK10K project identifies rare variants in health and disease. *Nature*, 526(7571):82–90, 2015. [cited at p. 14]

[200] K. R. Veeramah and M. F. Hammer. The impact of whole-genome sequencing on the reconstruction of human population history. *Nature Reviews Genetics*, 15(3):149–162, 2014. [cited at p. 2]

[201] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001. [cited at p. 5]

[202] D. F. Voytas and C. Gao. Precision genome engineering and agriculture: opportunities and regulatory challenges. *PLoS Biol*, 12(6), 2014. [cited at p. 2]

[203] N. Waddell, M. Pajic, A.-M. Patch, D. K. Chang, K. S. Kassahn, P. Bailey, A. L. Johns, D. Miller, K. Nones, K. Quek, et al. Whole genomes redefine the mutational landscape of pancreatic cancer. *Nature*, 518(7540):495–501, 2015. [cited at p. 2]

[204] R. Wan, V. N. Anh, and K. Asai. Transformations for the compression of FASTQ quality scores of next-generation sequencing data. *Bioinformatics*, 28(5):628–635, 2012. [cited at p. 66]

[205] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, C. G. A. R. Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature genetics*, 45(10):1113–1120, 2013. [cited at p. 14]

[206] T. A. Welch. A technique for high-performance data compression. *Computer*, 6(17):8–19, 1984. [cited at p. 52 and 75]

[207] D. A. Wheeler, M. Srinivasan, M. Egholm, Y. Shen, L. Chen, A. McGuire, W. He, Y.-J. Chen, V. Makhijani, G. T. Roth, et al. The complete genome of an individual by massively parallel DNA sequencing. *nature*, 452(7189):872–876, 2008. [cited at p. 2]

[208] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012. [cited at p. 86]

[209] R. N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference, 1991. DCC'91.*, pages 362–371. IEEE, 1991. [cited at p. 52]

[210] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. [cited at p. 46]

[211] A. R. Wood, T. Esko, J. Yang, S. Vedantam, T. H. Pers, S. Gustafsson, A. Y. Chu, K. Estrada, J. Luan, Z. Kutalik, et al. Defining the role of common variation in the genomic and biological architecture of adult human height. *Nature genetics*, 46(11):1173–1186, 2014. [cited at p. 2]

[212] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu. Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, 32(11):1632–1642, 2016. [cited at p. 23]

[213] X. Yang, S. P. Chockalingam, and S. Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66, 2013. [cited at p. 20]

[214] Y. Yang, B. Xie, and J. Yan. Application of next-generation sequencing technology in forensic science. *Genomics, proteomics & bioinformatics*, 12(5):190–197, 2014. [cited at p. 3]

[215] K. Ye, M. H. Schulz, Q. Long, R. Apweiler, and Z. Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, 2009. [cited at p. 27]

[216] Y. W. Yu, D. Yorukoglu, and B. Berger. Traversing the k-mer landscape of NGS read datasets for quality score sparsification. In *International Conference on Research in Computational Molecular Biology*, pages 385–399. Springer, 2014. [cited at p. 76]

[217] Y. W. Yu, D. Yorukoglu, J. Peng, and B. Berger. Quality score compression improves genotyping accuracy. *Nature biotechnology*, 33(3):240–243, 2015. [cited at p. 68 and 76]

[218] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. [cited at p. 86]

[219] Y. Zhang, L. Li, Y. Yang, X. Yang, S. He, and Z. Zhu. Light-weight reference-based compression of FASTQ data. *BMC bioinformatics*, 16(1), 2015. [cited at p. 59 and 61]

[220] M. Zhao, Q. Wang, Q. Wang, P. Jia, and Z. Zhao. Computational tools for copy number variation (CNV) detection using next-generation sequencing data: features and perspectives. *BMC bioinformatics*, 14(11), 2013. [cited at p. 26 and 27]

[221] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977. [cited at p. 50]

[222] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit. Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls. *Nature biotechnology*, 32(3):246–251, 2014. [cited at p. 29, 31, 34, 138, 162, and 199]

APPENDIX A

SUPPLEMENTARY MATERIALS

In this Appendix we provide supplementary materials used when performing the brief analysis in Chapter 3. The majority of the presented information can be found in the supplementary materials for the publications presented in Chapter 3, hence we do not include them here.

A.1 Datasets

A.1.1 WEX dataset

WEX dataset consists of a whole-exome sequencing of a *H. Sapiens* individual – a son from the Ashkenazim trio experiment, which was performed by GIAB [222]. It was sequenced using Illumina HiSeq platform from a paired-end library with coverage ~240. The dataset is available as aligned reads in SAM (BAM) format and can be downloaded from:

```
ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/AshkenazimTrio/HG
  002_NA24385_son/OsloUniversityHospital_Exome
  /151002_7001448_0359_AC7F6GANXX_Sample_HG002-EEogPU_v02-KIT-Av
  5_AGATGTAC_L008.posiSrt.markDup.bam
```

After downloading the file it was renamed to WEX.bam.

A.1.2 WGS dataset

WGS dataset consists of a whole-genome sequencing of a *H. Sapiens* individual from CEPH 1463 family, which was provided by Illumina Platinum Genomes[1]. It was sequenced using Illumina HiSeq platform from a paired-end library with coverage ~235. The dataset is available as 36 FASTQ files

[1] https://www.illumina.com/platinumgenomes.html

(stored in 18 pairs). However, for our test, we only used 3 pairs of files, giving an approximate coverage of 42. The files are available to download from:

```
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174310/ERR174310_1.
  fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174310/ERR174310_2.
  fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174311/ERR174311_1.
  fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174311/ERR174311_2.
  fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174312/ERR174312_1.
  fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174312/ERR174312_2.
  fastq.gz
```

A.2 Data preparation

A.2.1 FASTQ files

WEX dataset

We use SAMTools [111] (in version 1.3) to transcode WEX dataset from BAM format to FASTQ:

```
# sort the file by read names before transcoding
samtools sort -n WEX.bam > WEX-sorted.bam

# transcode to FASTQ
samtools fastq -1 WEX_1.fastq -2 WEX_2.fastq -0 WEX.fastq WEX.
  bam
```

The results of conversion is stored as 2 FASTQ files – WEX_1.fastq and WEX_2.fastq. There are no unpaired reads – in that case, they would be stored in WEX.fastq file. As a side note, the resulting FASTQ files have no comment content in the read identifier, as it was most probably removed during the mapping process.

WGS dataset

We decompress all the downloaded files and concatenate them pairwise. During decompression, we also remove the comment content from the read identifiers, which can be done as follows:

```
gunzip -c ERR17431*_1.fastq.gz | awk '{print $1}' >> WGS_1.fastq
```

```
gunzip -c ERR17431*_2.fastq.gz | awk '{print $1}' >> WGS_2.fastq
```

The result is stored as two FASTQ files: WGS_1.fastq and WGS_2.fastq.

A.2.2 SAM files

WEX dataset

The WEX dataset is originally available in form of reads mapped to the reference sequence (human genome assembly version 37 (*GRCh37*)) and sorted by position. As it is stored in compressed BAM format, it only requires conversion to SAM format:

```
samtools view -H WEX.bam > WEX.sam
```

The result is the SAM file WEX.sam.

WGS dataset

The WGS dataset is available only as unaligned raw reads in FASTQ format, hence, it requires mapping to the reference sequence and sorting by mapping position. To perform mapping, we use BWA-MEM [108] (in version 0.7.10) and use the same version of reference sequence as in case of WEX. It can be downloaded from:

```
ftp://ftp.ncbi.nlm.nih.gov/1000genomes/ftp/technical/reference/
  human_g1k_v37.fasta.gz
```

We decompress the reference sequence using *gzip* and name the resulting file as REF.fasta.

The mapping of raw reads and sorting by position is as follows:

```
# index the reference sequence and to it
bwa index REF.fasta
bwa mem -M REF.fasta WGS_1.fastq WGS_2.fastq > WGS-raw.sam

# convert to BAM format in order to perform sorting
samtools view -b -h WGS-raw.sam > WGS-raw.bam
samtools sort -O sam WGS-raw.bam > WGS.sam
```

The resulting SAM file will be stored as WGS.sam.

A.3 Running tests

Whenever a compressor supports multi-threading, it is run using 8 processing threads. Program execution time was measured using Linux command 'time', running the programs as:

```
/usr/bin/time <program>
```

A.3.1 FASTQ format

Standalone tools

When running tools, we use placeholders: *IN.fastq* to specify the input FASTQ file name, *COMP.** to specify the compressed file name and *OUT.fastq* to specify the output (decompressed) FASTQ file.

PIGZ We tested PIGZ in version 2.3.3, using the following commands:

- To compress:
  ```
  pigz -9 -p 8 -c IN.fastq > COMP.gz
  ```
- To decompress:
  ```
  pigz -d -p 8 -c COMP.gz > OUT.fastq
  ```

FQZCOMP We tested FQZCOMP [85] in version 4.6, which was downloaded from https://sourceforge.net/projects/fqzcomp/. We used the following commands:

- To compress using *FQZCOMP-STD*:
  ```
  fqz_comp c IN.fastq COMP.fqz
  ```
- To compress using *FQZCOMP-MAX*:
  ```
  fqz_comp -n2 -q3 -s8+ -b IN.fastq COMP.fqz
  ```
- To decompress:
  ```
  fqz_comp -d COMP.fqz OUT.fastq
  ```

QUIP We tested QUIP [85] in version 1.1.8, which was downloaded from https://github.com/dcjones/quip. We used the following commands:

- To compress using *QUIP-FQ-STD*:
  ```
  quip -v -c IN.fastq > COMP.qp
  ```
- To compress using *QUIP-FQ-MAX*:
  ```
  quip -a -v -c IN.fastq > COMP.qp
  ```
- To decompress:
  ```
  quip -d -c COMP.qp OUT.fastq
  ```

SCALCE We tested SCALCE [70] in version 2.8, which was downloaded from http://sfu-compbio.github.io/scalce/. We used the following commands:

- To compress using *SCALCE-SE*:
  ```
  scalce -T 8 -o COMP.sc IN.fastq
  ```
- To decompress using *SCALCE-SE*:
  ```
  scalce -d -T 8 -o OUT.fastq COMP.sc_1.scalcen
  ```
- To compress using *SCALCE-PE*:
  ```
  scalce -T 8 -r -o COMP.sc IN.fastq
  ```
- To decompress using *SCALCE-PE*:
  ```
  scalce -d -r -T 8 -o OUT.fastq COMP.sc_1.scalcen
  ```

LEON We tested LEON [15] in version 1.0.0, which was downloaded from http://gatb.inria.fr/software/leon/. We used the following commands:

- To compress:
  ```
  leon -c -file IN.fastq -nb-cores 8 -lossless
  ```
- To decompress:
  ```
  leon -d -file IN.leon -nb-cores 8
  ```

DSRC2 We tested DSRC2 [174] in version 2.1.0, which was downloaded from https://github.com/lrog/dsrc. We used the following commands:

- To compress using *DSRC-FAST*:
  ```
  dsrc c -m0 -t 8 -v IN.fastq COMP.dsrc
  ```
- To compress using *DSRC-MAX*:
  ```
  dsrc c -m2 -t 8 -v IN.fastq COMP.dsrc
  ```
- To decompress:
  ```
  dsrc d -t 8 COMP.dsrc OUT.fastq
  ```

FASTORE We tested FASTORE in the development version 0.8. The compression process consists of: (a) binning reads, (b) optional re-binning (only in C1 mode) and (c) compression. Therefore, a compression using any of the *FASTORE-** solutions is a multi-step process, running a set of sub-applications: *fastore_bin*, *fastore_rebin* in C1 mode and *fastore_pack*.

- To bin the reads in single-end mode:
  ```
  fastore_bin e -i"IN.fastq" -o"__tmp.bin"
      -p8 -s10 -H -q0 -t8
  ```
- To bin the reads in paired-end mode:
  ```
  fastore_bin e -i"IN_1.fastq IN_2.fastq" -o"__tmp.bin"
      -p8 -s10 -H -q0 -t8 -z
  ```
- To compress in C0 mode:

```
fastore_pack e -i"__tmp.bin" -o"COMP.pack"
    -f256 -c10 -d8 -w256 -t8 [PE]
```

where *PE* specifies a paired-end mode ('-z' switch) – in single-end mode it is left empty.

- To re-bin the reads in C1 mode:

```
fastore_rebin e -i"__tmp.bin"   -o"__tmp_2.bin"
    -p2 -w1024 -t8 [W] [PE]
fastore_rebin e -i"__tmp_2.bin" -o"__tmp_4.bin"
    -p4 -w1024 -t8 [W] [PE]
fastore_rebin e -i"__tmp_4.bin" -o"__tmp_8.bin"
    -p8 -w1024 -t8 [W] [PE]
```

where *W* specifies the size of the matching window for paired-end mode ('-W1024' switch) – in single-end mode it is left empty.

- To compress in C1 mode:

```
fastore_pack e -i"__tmp_8.bin" -o"COMP.pack"
    -f256 -c10 -d8 -w256 -t8 [W] [PE]
```

- To decompress in single-end mode:

```
fastore_pack d -i"COMP.pack" -o"OUT.fastq" -t8
```

- To decompress in paired-end mode:

```
fastore_pack d -i"COMP.pack" -o"OUT_1.fastq OUT_2.fastq"
    -t8 -z
```

CARGO-based solutions

We tested CARGO-based FASTQ compressor solutions [175] using CARGO framework in version 0.7.1, which was downloaded from https://biocargo.sourceforge.net. Apart from the placeholders *IN.fastq* and *OUT.fastq* introduced in the previous section, when working with CARGO-based solutions, we additionally use: *CONTAINER* to which specify the name of the CARGO container, and *DATASET* to specify the name of the dataset under which the data is stored in the container.

Container creation Before running tests, we created temporary CARGO containers to store the compressed data, separately for WEX and WGS datasets. For WEX dataset we created a container able to hold compressed data up to 11.3 GB in size, by running command:

```
cargo_tool --create-container --container-file=CONTAINER
    --large-block-size=8 --large-block-count=20
    --small-block-size=256 --small-block-count=32
```

Such created container consists of 20×64 large blocks, each of 8 MiB in size, and 32×64 small blocks, each 256 KiB in size. The total available size

will equal to ~10.75GiB (11 274 289 152 B).

Similarly, we created a large container for storing WGS dataset, able to compressed hold data up to 86.4 GB in size, by running command:

```
cargo_tool --create-container --container-file=CONTAINER
    --large-block-size=8 --large-block-count=160
    --small-block-size=256 --small-block-count=32
```

As a side note, after compressing the files the container can be optionally shrinked to adapt its size to the size of the compressed content, by removing unused blocks:

```
cargo_tool --shrink-container --container-file=CONTAINER
```

Moreover, after performing a single test (compression and decompression), the container can be cleared, removing all the stored datasets:

```
cargo_tool --clear-container --container-file=CONTAINER
```

Running CARGO All the generated *CARGO-FQ-** solutions are invoked in a similar way, that is:

- To compress:
  ```
  cargo_fastqrecord_toolkit-* c -v -c CONTAINER -n DATASET
      -t 8 -b 8 -i IN.fastq
  ```
- To decompress:
  ```
  cargo_fastqrecord_toolkit-* d -v -c CONTAINER -n DATASET
      -t 8 -o OUT.fastq
  ```

A.3.2 SAM format

Reference sequence

In some scenarios, a sequence reference file *REF.fasta* is used – it is the same file as the one used during the FASTQ reads mapping stage, as presented in the previous section.

Moreover, when running SCRAMBLE in reference-based compression mode (*SCRAMBLE-REF-** solutions), the reference sequence needs needs to be indexed prior to compression, by using e.g. SAMTools. It needs to be done only once, by running:

```
samtools faidx REF.fasta
```

Analogously, when compressing SAM files using *CARGO-SAM-REF* solution, a reference sequence needs to be previously indexed, by running

(only once):

```
cargo_samrecord_toolkit-ref r -i REF.fasta -o REF.fasta.bff
```

Running standalone tools

When running tools, we use placeholders: *IN.sam* to specify the input SAM file name, *COMP.** to specify the compressed file name and *OUT.sam* to specify the output (decompressed) SAM file.

SCRAMBLE We tested SCRAMBLE [18] in versions 1.13.10 and 1.14.9, implementing CRAM format in versions 2 (*SCRAMBLE-CRAM2-**) and 3 (*SCRAMBLE-CRAM3-**) respectively. The solution was downloaded from https://sourceforge.net/projects/staden/. The compressed sizes were reported using program *cram_dump*. We run *SCRAMBLE-CRAM2-** using binary *scramble-v2* and *SCRAMBLE-CRAM3-** using binary *scramble-v2*, with following command lines:

- To compress using *SCRAMBLE-BAM*:
  ```
  scramble-v2 -I sam -O bam -m -t 8 IN.sam > COMP.bam
  ```
- To decompress using *SCRAMBLE-BAM*:
  ```
  scramble-v2 -I bam -O sam -m -t 8 COMP.bam > OUT.sam
  ```
- To compress using *SCRAMBLE-CRAM2-REF*:
  ```
  scramble-v2 -I sam -O cram -m -r REF.fasta -t 8 IN.sam >
    COMP.cram
  ```
- To decompress using *SCRAMBLE-CRAM2-REF*:
  ```
  scramble-v2 -I cram -O sam -m -r REF.fasta -t 8 COMP.cram >
     OUT.sam
  ```
- To compress using *SCRAMBLE-CRAM3-REF*:
  ```
  scramble-v3 -I sam -O cram -p -P -m -r REF.fasta -t 8 IN.
    sam > COMP.cram
  ```
- To decompress using *SCRAMBLE-CRAM3-REF*:
  ```
  scramble-v3 -I cram -O sam -p -P -m -r REF.fasta -t 8 COMP.
    cram > OUT.sam
  ```
- To compress using *SCRAMBLE-CRAM3-NOREF*:
  ```
  scramble-v3 -I sam -O cram -x -p -P -t 8 IN.sam > COMP.cram
  ```
- To decompress using *SCRAMBLE-CRAM3-NOREF*:
  ```
  scramble-v3 -I cram -O sam -x -p -P -t 8 COMP.cram > OUT.
    sam
  ```
- To compress using *SCRAMBLE-CRAM3-EMBREF*:
  ```
  scramble-v3 -I sam -O cram -e -p -P -m -r REF.fasta -t 8 IN
    .sam > COMP.cram
  ```

- To decompress using *SCRAMBLE-CRAM3-EMBREF*:
```
scramble-v3 -I cram -O sam -e -p -P -m -t 8 COMP.cram > OUT
  .sam
```

DEEZ We tested DEEZ [71] in version 1.1, which was downloaded from http://sfu-compbio.github.io/deez/. We run following commands:

- To compress using *DEEZ-REF*:
```
deez -t 8 -r REF.fasta IN.sam -o COMP.dz -v
```
- To decompress using *DEEZ-REF*:
```
deez -t 8 -r REF.fasta COMP.dz -o OUT.sam
```
- To compress using *DEEZ-NOREF*:
```
deez -t 8 IN.sam -o COMP.dz -v
```
- To decompress using *DEEZ-NOREF*:
```
deez -t 8 COMP.dz -o OUT.sam
```

QUIP-SAM-REF We used the same QUIP binary as when compressing FASTQ files. We run the following commands:

- To compress:
```
quip -r REF.fa -c IN.sam > COMP.qp
```
- To decompress:
```
quip -d -r REF.fasta -c COMP.qp >OUT.sam
```
- To print the sizes of compressed streams:
```
quip -l -v COMP.qp
```

CARGO-based solutions

We used the same CARGO framework as when compressing FASTQ files.

Container creation Before running tests, we create temporary CARGO containers to store the compressed data, separately for the WEX and WGS datasets. For WEX we create a container able to store compressed data up to 5.9 GB in size, by running command:
```
cargo_tool --create-container --container-file=CONTAINER --large
  -block-size=8 --large-block-count=10 --small-block-size=256 --
  small-block-count=32
```

Similarly, we create a large container for the WGS dataset, able to store compressed data up to 59.6 GB in size, by running command:
```
cargo_tool --create-container --container-file=CONTAINER --large
  -block-size=8 --large-block-count=110 --small-block-size=256
  --small-block-count=32
```

Analogously, as in the case of compressing FASTQ files, after compressing the files the container can be shrinked to adapt its size to the size of the content or cleared.

Running CARGO The CARGO-based *CARGO-SAM-** solutions were run as follows:

- To compress using *CARGO-SAM-STD* and *CARGO-SAM-EXT*:
```
cargo_samrecord_toolkit-* c -v -c CONTAINER -n DATASET -t 8
   -b 8 -i IN.sam
```
- To decompress using *CARGO-SAM-STD* and *CARGO-SAM-EXT*:
```
cargo_samrecord_toolkit-* d -v -c CONTAINER -n DATASET -t 8
   -o OUT.sam
```
- To compress using *CARGO-SAM-REF*:
```
cargo_fastqrecord_toolkit-* c -v -c CONTAINER -n DATASET -t
   8 -b 8 -a -i IN.fastq
```
- To decompress using *CARGO-SAM-REF*:
```
cargo_fastqrecord_toolkit-* d -v -c CONTAINER -n DATASET -t
   8 -a -f REF.fasta.bff -o OUT.fastq
```