Modern systems for large-scale genomics data processing in the cloud

Sergei Yakneen
EMBL
Meyerhofstrasowse 1,
69117 Heidelberg, Germany

09/11/2018

Contents

Chapter 1

Introduction

1.1 Context and Motivation

In the 17 years since the publication of the first draft human genome[94] the fields of genomics and molecular biology have undergone a major shift. The direction of this shift is towards an increasing adoption of computational approaches alongside experimental methods, bringing both of these fields of study into the realm of information science. This transition has been facilitated by two major factors - the advent of next generation sequencing[166], and the development of the Internet and cloud computing[20]. Next generation sequencing has been responsible for bringing down the cost of DNA sequencing to the point where it has become possible to sequence and study entire populations of individuals[72], while the Internet and cloud computing are democratising access to large-scale computational resources such that computation on big datasets, which was previously only accessible to large institutions, is becoming tractable to a growing group of researchers and citizen scientists.

The continued appetite for sequencing of larger and larger cohorts of individuals by the research community is driven by the desire to better understand the evolutionary history of the human species[86], to identify causes and mechanisms of action of rare genetic diseases that affect a very small proportion of the population[17], and to elucidate and potentially target the genetic component of more common diseases such as cancer[195], heart disease[18], or dementia[168] that place a heavy burden on our society. All of these factors together mean that the need for the generation and interpretation of genomic data is growing at an unprecedented scale.

Yet, the analysis of DNA sequencing data to study human genomes remains a largely unsolved problem. The protein coding sequence of the human genome, its *exome*, constitutes roughly 1% of the human DNA and successful studies have carried out exome-based analyses on cohorts at the scale of tens of thousands of individuals[102]. However, the other 99% of the human genome, it's non-coding regions, contain crucial information such as gene regulatory elements[29] that are essential to our full understanding of the mechanisms and processes that are underlying the human genetic landscape. Given current technologies, Whole Genome Sequencing

(WGS) is considerably more expensive and generates data-set sizes at the petabyte (PB) scale that are challenging for even the largest international consortia to tackle [176]. WGS studies at 100,000 participants scale that are planned for the coming years[47] will further increase data-set size and complexity by several orders of magnitude, a challenge that is presently unanswered by the current generation of bioinformatics infrastructures and algorithms.

A bigger and more distant challenge is the development of clinical sequencing and genomics which will truly bring whole-genome sequencing applications to population scale. Currently DNA sequencing has limited adoption within the clinical practice with applications limited to rare Mendelian disorders[101] and certain types of cancers[160] where a small set of genomic loci is interrogated via a gene panel[10] with a set of well-delineated disease sub-types based on these genetic markers. The use of whole-genome sequencing for clinical applications is presently nearly non-existent due to its high cost compared to the clinical utility of its findings, yet the potential for the impact of this approach remains substantial as certain genomic variants such as Structural Variations (SVs) typically have a large effect on an individual's phenotype due to their size[152], but are generally not amenable to interrogation via gene panels.

The magnitude of the opportunity for improvement in the space of DNA sequencing and genomics is thus clear to us - we seek a way to improve the current methods of DNA data analysis such that it becomes tractable and cost-effective to undertake whole-genome sequencing studies within research and clinical contexts at the scale of hundreds of thousands to millions of human genomes.

1.2 Challenges and Problem Statement

Let's examine the key challenges that need to be addressed in order to enable efficient genomic data analysis at the scale that is desired by the research and clinical communities.

Several broad groups of challenges are identified below and further examined throughout this thesis:

Data Set Size - The size of the raw genomic data generated by population-scale studies will be hundreds to thousands of petabytes making it impractical to move and make copies of the data[177].

Data Retention - The cost of generating the data is significantly higher than the cost of storing the data, thus making it impractical to throw away the raw data after initial analysis[137].

Data Formats - The data formats used for storing genomic data are primarily large size character and binary files (FASTA, SAM, BAM, VCF)[117, 38] that have loose specifications and scale poorly to large cohort sizes. File indexing

structures typically support indexing by genomic coordinate only, thus limiting queryability.

Data Fragmentation - The data will be generated at multiple sequencing centres located in different jurisdictions with a wide variety of genomic data handling requirements. Data processing must proceed at multiple locations that respect the requirements of each jurisdiction[136].

Data Type Diversity - Comprehensive characterization of a person's genome that is useful in a clinical setting implies the collection and integrative analysis of many diverse data types - including germline[122] and somatic[68] genomic variants, transcriptomics[193], epigenomics[87], metabolomics[190], and clinical information. Uniform collection, processing, and integration of these data types is required to successfully associate the role of this genomic variation on disease phenotypes[160].

Data Processing Stages - Data processing for genomics analysis proceeds through a sequence of stages from base-calling, to quality-control, to genome alignment, to variant calling, to annotation, to downstream analysis[42]. Each stage typically has non-trivial computational requirements needing several days on a multi-core machine to complete with increased failure risk as a function of data set size. Intermediate results from one stage are often required as input for downstream stages. Fully sequential processing makes inefficient use of the data by redundantly loading and interrogating the data in memory over a series of passes through the sample.

Toolset Fragmentation - Although comprehensive genomic characterisation of each sample is typically of interest to researchers, specific bioinformatics tools only provide solutions to a limited subspace of the overall problem, thus requiring integration of multiple tools that may produce incongruent outputs and compete for resources producing computational bottlenecks.

Having listed these challenges we attempt to restate the problem in simpler terms before providing a high level overview of the types of approaches and solutions that will be developed and considered in detail in the body of this thesis in order to deliver a conceptual and practical framework for the effective management of genomic data at the desired scale.

Our problem statement is then as follows:

Human genomic data sets will, in the future, be generated for analysis in various locations throughout the world, at the aggregate rate of multiple petabytes of data per day in the context of disease and clinical practice. The desired outcome of these analyses is the comprehensive characterisation of genomic features and their association with phenotypic variables of interest[198]. The goal of the research community is in capturing the maximum number of samples – N, with high accuracy – A, to increase statistical power of studies[82], while the interest of clinicians is to capture specific individuals with high accuracy – A, and in the shortest possible time – T, in order to inform clinical decision making[191]. Both parties wish to do so at

minimal possible total cost – $C = c_g + c_s + c_a + c_r$, taking into account the cost of data generation, cost of data storage, cost of data analysis, and cost of subsequent data retrieval. Because of the high cost of generating this data each time, the data, once generated, will need to be stored for the foreseeable future. The overwhelming data set size prevents data movement between locations, requiring analysis algorithms to be colocated with the data.

The analysis is hampered by reliance on data formats that have not been designed for operation at such large scale and the necessity to execute a variety of computational algorithms[113, 209, 9, 132] on the data that have been individually developed by different authors within an academic context, using different technologies that compete with each other for computational resources, and at-times produce contradictory results that require human intervention to integrate. The underlying assumption of genomic coordinate-sorted ordering and traversal of the data made by most algorithms limits the modes of reasoning about the dataset to a series of pre-processing steps, followed by another series of coordinate-wise traversals through the data, which impose severe processing time costs, such as the requirement to have generated, seen and sorted all of the data, before an analysis can proceed as well as the inability to stop and interpret analysis results mid-processing.

The optimization problem of maximising N, and A, while minimizing C for research purposes remains unsolved for values of N above 3000 samples when it comes to high-coverage whole genome sequencing, while the problem of maximizing A, and minimising T, and C is presently not solved in the clinical setting for any sample size. It is our proposed solution for tackling these issues that we turn to next.

1.3 Proposed Solution

We assume that N, the number of samples that can be successfully sequenced will depend almost entirely on the total cost C, which itself, among other factors, is determined by the desired accuracy and processing time. We thus focus most of our efforts on the joint optimization of cost, accuracy, and time as necessary conditions for the maximisation of effective sample size N and enablement of whole genome sequencing for clinical practice.

We note that the cost of data generation C is dependent on the sequencing technology used, the underlying chemistry, and the cost of the reagents[135]. Improving these characteristics falls outside the scope of our discussion, and we assume the cost of the data generation component c_g of C to be constant throughout this thesis.

We establish and discuss at length the characteristics of the optimization function in the body of the thesis but here note briefly that analysis accuracy A is evaluated along the usual dimensions of sensitivity and specificity and can generally be improved by generating more data for a given sample up to a theoretical maximum inherent in the sequencing technology used and the nature of the analysis algorithms employed. Generation of more data naturally leads to increased analysis time T and cost C. The time to accomplish the analysis can be reduced by either giving up ac-

curacy (by looking at less data, or using faster but less accurate algorithms[114]), by increasing the level of parallelisation within the computational pipeline i.e. parallelising steps that are currently sequential[95], or by utilising additional computational resources, thereby increasing costs. The various components of cost, in turn, can be optimized by improved data storage and retrieval structures[145] (via multi-level caches and hybrid storage media, for example), by improved-efficiency analysis algorithms, and by reduction of analysis accuracy and increase of analysis time (via cheaper hardware).

It is clear from the discussion above that cost, accuracy, and processing time are not orthogonal concerns i.e. changes in one may lead to changes in the other two. It thus appears that no optimization effort is likely to simultaneously satisfy the requirements of all parties that are interested in large scale genomic analysis, and a successful computational framework for delivering such analyses must allow efficient and dynamic optimization of these parameters to fit the needs of the end user. This is typically not the case for present day genomics frameworks because of the sequential way they look at data[131, 186] i.e. all of the data is generated before it is processed by downstream tools, and accuracy and processing time need to be decided on before launching a set of tools because they step through the genome in coordinate-wise manner.

To address these challenges we develop and describe within this thesis a new computational framework, called Rheos, that is based on the concepts of data streaming, cloud computing, and service orientation to provide a comprehensive toolset for genomic data analysis that can potentially scale to processing of millions of genomes while arming its users with the capability to make timely, responsive, and principled decisions about the tradeoffs between analysis cost, accuracy, and duration.

Three distinct characteristics set Rheos apart from current generation genomic analysis frameworks and each of these allows us tackle some of the issues and challenges described in Section **??**. These are:

- Service Orientation
- Event and Data Streaming
- Random Data Ordering

Service orientation[48] allows us to decompose the overall problem of comprehensively reasoning about genomic data into a set of small loosely-coupled components, each of which is optimized to tackle a particular well-defined subset of the complete set of requirements of the system. Each service has a contract that it makes with its clients, it has an explicit set of inputs that it knows how to process, it has an interface that defines the modes of communication it supports, it has a set of outputs that it produces according to its capabilities, and it has a set of operational characteristics that makes explicit commitments about the service's reliability, speed, etc[146]. This has a number of benefits - a service can be small enough that it optimizes the solution to a particular problem without being subject to the same

competing constraints that larger tools are subject to, which provides opportunities for improved performance and hardware utilization. As long as the service respects its input and output commitments it is free to maintain arbitrary internal representations of the data enabling optimization of data storage and query costs (c_s and c_r). A service can be monitored such that hardware is allocated elastically up and down based on demand to ensure optimal utilization, as well as providing a continued measure of whether the service is meeting its operational reliability requirements to its clients[33]. This is especially useful in contexts where demand for certain calculations is highly variable.

The issue of inter-service communication is of major importance because of the large size of the data-set and the potential for various difficult-to-debug run-time race and error conditions inherent in a distributed system[62]. Currently, most bioinformatics tools do not communicate with each other directly via an API, instead they use popular file formats such as SAM/BAM/CRAM[117, 60], and VCF[38], as well as a myriad of more esoteric file formats not only as a storage medium but also as a means of communicating information between each other. This paradigm hurts the ultimate scalability of the entire system because of the necessity to write data to disk and possibly move it over the network in order to enable communication across tools. Furthermore, a file-based information exchange mechanism forces a coarse-grained, sample-level, communication between components that wish to avoid tight coupling between each other, even though most of the reasoning about genomic data occurs at locus, or small locus-neighbourhood, levels[45].

Rheos adopts a data and event stream approach to accomplish scalable fine-grained communication between services[138]. This approach allows each service to listen to and produce data at the level of granularity that it needs to make decisions, and that its downstream dependencies are interested in (for instance at read, locus, or breakpoint levels). When primary data is ingested into the Rheos system (from a sequencer, or a data repository) the data stream can start to be analyzed immediately[73], unlike file-based systems that need to wait for the entire sample to transmit before beginning. This approach can potentially enable real-time analysis given sufficient allocation of computational resources[6]. Since the raw data is extremely large, it is advantageous to move this data between machines, and between disk and RAM as little as possible, thus instead of passing the raw data around the network various services pass around events of interest about the raw data amongst themselves[49]. When a particular service needs the raw data (rather than the corresponding events) for its decision-making it can be shipped this data as necessary, or it can be instructed to run on the host that has already cached this data in memory. Data streaming allows for extreme scalability, but a key challenge when dealing with data streams is that one is no longer guaranteed to ever be able to see "all of the data" for a particular sample, at least in any meaningful amount of time[61]. Because genomic algorithms frequently make use of various summary statistics accumulated over the data-set[148, 104], not being able reason over all the data at once means that approximations for these summary statistics are required. Rheos uses approximations calculated within time windows over the data stream[40, 12] and we consider their properties in detail in the body of this thesis.

A key assumption made by nearly all algorithms in the genomics space that

participate in variant calling and reason over sequence reads is that the reads are coordinate-sorted with respect to the reference genome to which they are aligned[117, 64, 24, 156]. The algorithms then proceed by traversing the genome in coordinate-wise fashion from the beginning of chromosome 1 to the end of chromosome Y interrogating each locus in turn by examining the set of reads that overlap that locus (a read pileup)[117]. Getting the reads into a state that is usable by these algorithms then requires, at a minimum, that all the reads for a given sample have been generated, have gone through QC[199], have been aligned[114], have been investigated for PCR duplicates[186], and have been sorted[186]. Each of these steps can take hours or even days to complete, especially on high coverage whole genome samples. We take a different approach with Rheos by relaxing the requirement for the reads to have been sorted before any variant calling can take place, and instead develop a set of variant calling algorithms that do not assume any particular order within the data that they observe. This allows Rheos to make use of sequence data as soon as it comes off the sequencing machine, thereby dramatically reducing the total time T required to process genomic data compared to the current generation of algorithms. Rheos accomplishes this by employing the service- and stream-based approaches discussed above to process each read on-the-fly as it moves through the system. The read is first assessed for quality, then aligned to a reference genome by the alignment service. This service emits an event with a coordinate that corresponds to the alignment. Variant calling services listen to this event stream and incorporate the evidence for genomic variation supplied by this read into their models of the genomic features that exist at that particular locus for that sample via a statistical framework based on an iterated application of Bayes' rule[206, 14].

Because current generation tools can see all of the data for a particular locus at once they can incorporate all of the evidence supplied by this data in a minimal number of calculations, corresponding to each particular algorithm[117, 64]. Rheos, on the other hand, to incorporate the same amount of evidence will need to perform a larger number of calculations in a redundant manner, incorporating the data as it is observed. This cost is compensated for, however, by the fact that Rheos can immediately incorporate new data about a particular locus when it becomes available without the need to have accumulated all of the data for all of the loci, generating significant time savings. Furthermore, because data arrives in no particular order the set of variant calls produced by Rheos at any given point in time represent a comprehensive characterisation of the sample as if the sample was sequenced at an average coverage consistent with the amount of data that has been observed so far. Observing more data is equivalent to raising the average coverage uniformly throughout the genome, thereby improving call accuracy[8]. This provides us with a framework to actively and dynamically trade off call-set accuracy A for processing time T and cost C as actual data is being observed thereby enabling novel applications whereby sequencing is abandoned early when issues such as sample-swap[52], or contamination[23] are detected. In addition, when sufficient accuracy is reached based on observed data at a particular locus, the framework may choose to stop looking at further data, whereas current generation approaches necessitate committing to a particular sequencing depth a-priori. Furthermore, because current methods iterate through the data in a coordinate-wise manner, their partial results are not really usable until the entire data-set has been traversed (as they represent only

a particular region of the genome), whereas Rheos call-sets represent progressive elaboration of a complete genomic characterisation and are thus usable at any level of accuracy that is fit for the purposes of the underlying analysis. We develop the details of the statistical framework used by Rheos and compare its theoretical and real performance to current generation frameworks in the body of this thesis.

We conceived of Rheos as a modern bioinformatics framework that aims to enable the large scale genomics studies of the future[47, 88, 127] in both research and clinical contexts by providing a toolset that allows for interpretation and comprehensive characterisation of high coverage human whole genome samples at the scale of millions of samples. In order to meet the diverse requirements of its users the framework allows users to make informed and dynamic tradeoffs along the optimization dimensions of of cost, accuracy, and time. Rheos unique abilities rest upon three characteristics that set it apart from current generation tools, these are: service orientation, data streaming, and random data ordering. Taken together these characteristics enable Rheos to perform at unprecedented levels of scale while retaining call-set accuracy and reducing per-sample processing time. We dedicate the main body of this thesis to the development of the theoretical framework underlying Rheos, exploring its characteristics, benefits and tradeoffs, discussing its implementation, and evaluating and comparing Rheos' performance to the current best practices in genomics algorithms on real data.

1.4 Thesis Outline

This thesis focuses on the development of the conceptual framework behind the Rheos platform, the theoretical properties of the various algorithms employed by Rheos, and the implementation and experimental validation of the framework on real data. Chapter 2 provides an introduction to the fields of genomics, including cancer genomics and clinical genomics, a survey of the main tools and algorithms that are commonly used in genomics is provided including the details of the underlying statistical models and operational characteristics. The chapter concludes with a look at workflow frameworks that tie individual algorithms together into computational pipelines. Chapter 5 sets up and describes the conceptual framework underlying Rheos based on the approaches of Service Orientation, Data Streaming, and Random Data Ordering, mentioned above. We describe the overall architecture as well as the model behind individual services that comprise Rheos and investigate the theoretical properties of the algorithms that underlie Rheos-based genomic analysis. Chapter **??** describes the actual implementation of the Rheos framework's components and investigates their operational characteristics. Chapter **??** is dedicated to the experimental evaluation of Rheos in comparison to other extant frameworks and algorithms using real genomic data. We conclude this work in Chapter 6 with a discussion of the results and an examination of the future direction of Rheos development.

Chapter 2

Background and Related Work

2.1 Genomics

The field of genomics is closely related to, yet distinct, from the field of genetics, which itself stems from the work of such seminal figures as Charles Darwin[39] and Gregor Mendel[70]. While genetics largely focuses on the study of single (or relatively small numbers of) genes - the *genotype*, and how genetic variation and mutation affect the physical traits of a given cell or organism - its *phenotype*, genomics focuses on larger scale events and mechanisms that tend to act on the entirety of an organism's genome, shaping its architecture and ultimately affecting its survival.

2.1.1 History of Genomics

Each living cell is a bio-chemical machine that carries out a number of complex behaviours such as interactions with the surrounding environment, motility, metabolism, and reproduction, that are necessary for its survival and proliferation, based on a genetic program that is encoded within the cell's DNA. The DNA is nominally subdivided into functionally distinct areas known as *genes*. The cell utilizes the program within each gene by first *transcribing* the DNA into an intermediary information-carrier molecule called RNA, and then *translating* this RNA into molecules called *proteins* that are utilized by the cell to carry out the majority of its functions. Understanding and interpretation of the underlying genetic program thus underpins our ability to comprehend the entirety of the different behaviours that each cell undertakes.

The success of this undertaking is contingent, first and foremost, on our ability to effectively read off the information encoded in the DNA, an activity known as *sequencing*. We are able to sequence DNA thanks to the pioneering work of researchers Rosalind Franklin[59], James Watson, and Francis Crick[194] who first elucidated the physical structure of DNA, then followed by the work of Fred Sanger[164, 163] who devised the first effective DNA sequencing method. The sequencing method

allows us to transform information that is physically encoded on the DNA molecule via a sequence of four distinct types of *basepairs* - Adenine, Cytosine, Guanine, and Thymine into a string stored on a computer using a four-letter alphabet - A,C,T, and G, thus turning DNA interpretation into a digital information processing problem.

While the entire length of the DNA of an organism ranges from several hundred thousand basepairs for simple organisms like viruses and bacteria, to about 3,000,000,000 basepairs for a human, to over 150,000,000,000 for certain plants[149] the limitations of Sanger DNA sequencing technology are such that the sequencing machine can only produce DNA fragment strings, known as *reads* that are 800 - 1,000 basepairs long[164]. Reconstituting the original complete DNA sequence from partial overlaps between reads is thus a costly, time consuming, and computationally intensive problem known as *de-novo assembly*[208]. Once one such full sequence (known as a *reference* sequnce) is assembled however, sequencing other individuals of the same (or closely related) species becomes a significantly easier undertaking. Rather than assembling the sequence *de-novo* one can search for a position on the reference sequence that provides the best matching *alignment* between the reference and each read obtained for the specimen under study. This technique is known as *genomic alignment*[113] or *mapping* and yields for each fragment a coordinate that represents where on the reference sequence the fragment maps to. Furthermore, because the DNA of any two organisms of the same species is largely identical, with differences occurring at about 0.1% of all sites (although this depends on DNA mutation rate)[140] researchers are able to significantly reduce the amount of information that is required to fully represent the genome of a specimen by retaining only the information that describes the sites where that specimen is different from the reference sequence for that species.

A general approach has thus emerged, where each new species of interest undergoes a relatively costly *de-novo* assembly process for the first genome, which then becomes the reference genome for that species. The sequencing of further individuals of that species utilizes, relatively cheaper, *alignment* and identification of *variants* (sites where the individual differs from the reference) to investigate the effect these variants may have on different phenotypes of interest such as disease susceptibility and survival[124].

Although genomicists study many different types of organisms the study of human genomes garners by far the most attention and research funding[**needcitation**] due to the natural desire of humans to better understand ourselves and influence, where possible, genetic factors impacting human longevity and health. Subsequent to the development of DNA sequencing methods by Fred Sanger one of the most audacious and crucial projects for the development of genomics as a branch of science has been The Human Genome Project[94] - an international effort to sequence and *de-novo* assemble the first complete human genome consisting of chromosomes 1-22, X, and Y (as well as mitochondrial DNA) and totalling approximately 3 billion basepairs. The project ran for over 10 years, completing in 2001, and cost more than $3 billion USD. Although the main project effort was completed using the Sanger sequencing method, a competing version of the human genome was simultaneously published by a commercial company led by JC Venter[188], using a new sequencing method called shotgun sequencing[189], a method that formed the basis for a new revolution

in sequencing technology, now termed Next Generation Sequencing[166].

2.1.2 Next Generation Sequencing

The Next Generation Sequencing methodology[126] relies on fragmenting the DNA of a subject into millions of fragments that are between 100-500 basepairs (bp) in length, then sequencing all of the short fragments and aligning all the reads to the reference with the aid of a relatively fast algorithm[114]. Because NGS sequencing methods are prone to certain errors and biases[44], it is necessary to sequence enough DNA fragments to overlap (or cover) every location in the genome several times (typically 10-30), in order to build a statistical model that will be able to determine the underlying sequence, known as *genotyping*[141], with a high degree of confidence. Thus, at present, a single sequenced DNA sample will typically contain 1 billion reads with a file size of 150GB when compressed.

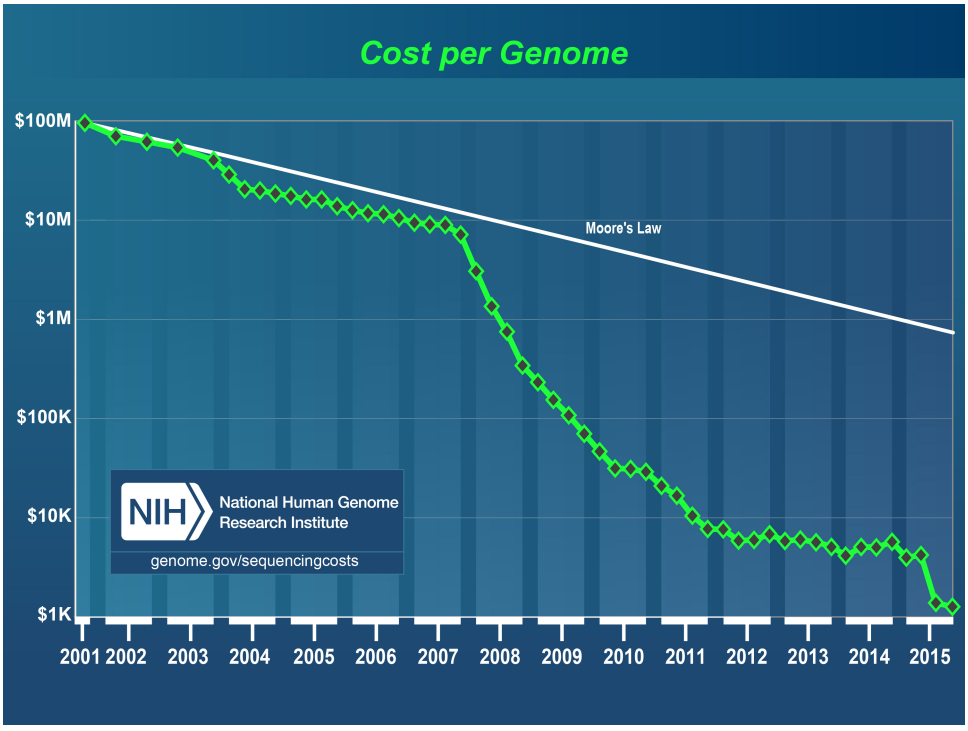


Figure 2.1: Cost of DNA sequencing[182]

Figure 2.1 shows the change in the cost of DNA sequencing over the course of the past 15 years. The precipitous drop in sequencing cost observed since 2008 coincides with wide adoption of NGS methodologies. This drop in price has made tractable a new set of large scale genomics sequencing projects that aim to characterize human genetic diversity at population scale, projects such as the 1000 Genomes Project[28], and the large scale sequencing of the Icelandic population[72].

2.1.3 Genomics Studies

2.1.4 Cancer Genomics

Cancer is a genetic disease that has an extremely high burden on the human population. In 2012, the global incidence of new cases worldwide has been estimated as 14.1 million, and deaths at 8.2 million[183]. The economic cost of cancer to the European Union has been estimated at 126 billion euro in 2009[121], and in the US $124.5 billion USD in 2010[202]. Because of the genetic nature of the disease studying genomes of cancer patients helps uncover the mechanisms behind the development and evolution of cancer[179].

Cancerous tumours arise from a single cell which over time accumulates a series of somatic mutations that cause it to exhibit properties such as: increased mutation rate, increased proliferation, anchorage independent growth, and resisting cell death[74] . Only certain mutations, however, contribute to the development of cancer, while others are benign. Cancer genomics studies aim to identify and characterize those mutations that are cancer drivers and play a role in the formation or progression of tumours[179].

Studying cancer genomes is more complex and expensive than studying the genomes of healthy individuals because each patient requires that two DNA samples are collected - that of the normal tissue, and that of the tumour. This is necessary to identify those mutations that are somatic - i.e. only occur in the tumour cell population[158].

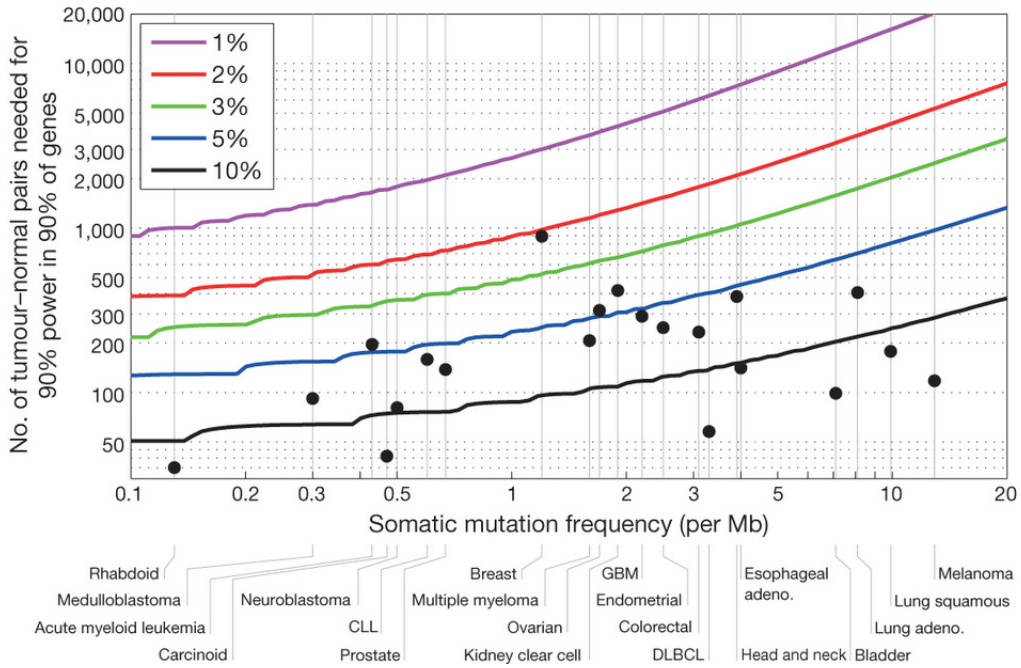


Figure 2.2: Sequencing sample size required by mutation rate[99].

Although there is a large number of identified mutations that are implicated in

cancer (2,002,811 SNV, 10,534 gene fusions, 61,299 genome rearrangements, 695,504 CNV segments in COSMIC v70; August 2014)[57], each mutation has a low chance of being present in any given tumour. Figure 2.2 demonstrates the sample size required to have 90% statistical power to identify 90% of the variants that occur with a set frequency in tumours with varying background mutation rates. Thus, identifying 90% of the mutations occurring with a frequency of at most 1% in Lung Adenocarcinoma requires a sample size of at least 10,000 patients. The necessity to sequence large cohorts of patients in order to be able to comprehensively detect cancer related genomic variants has led to the creation of several large scale cancer sequencing studies.

2.1.5 Clinical Genomics

2.2 Computational Methods for Next Generation Sequencing

Because the size of a typical genome is millions to billions of basepairs long, and current DNA sequencing technology frequently generates errors during the sequencing process, requiring multiple samples of each genomic location to be generated, the amount of data required to be examined in order to characterize even a single sample is well beyond the capabilities of any human. Thus, a multitude of computational approaches are required in order to make the task tractable for individual samples as well as cohorts, and entire populations.

The task of comprehensive characterization of genomic data for an individual is typically decomposed into a series of computational steps, each with its own data representation, and typically developed by a separate research group, which are then assembled into computational pipelines and executed by workflow engines on diverse computing environments. Our goal in this section is to enumerate and describe the individual steps and to provide a survey of the key computational tools and data formats that presently form the set of best practices in this rapidly evolving branch of science. Since Rheos is designed to improve upon these best practices we identify in each section the key mathematical and algorithmic ideas that underpin each approach in order to adapt and translate them into the Rheos framework.

The data that is used in virtually all modern genomics studies is generated on a next generation DNA sequencing machine. Several types of sequencers have been developed but the most frequently used ones are made by Illumina. The raw data produced by such a sequencer is a set of image files, where the color of each pixel represents the corresponding nucleotide base in a DNA strand that is being sequenced in each micro-well of a flowcell, representing the sample of interest. The succession of images produced by each cycle of sequencing then results in a set of reads, a collection of randomly ordered DNA fragments that are further analyzed by downstream tools. The first challenge in generating these reads is the accurate interpretation of pixel colors and mapping them to the corresponding nucleotide

bases, known as base-calling. Because all of the currently available DNA sequencing methodologies are imperfect at reading the underlying DNA sequence a number of errors is introduced into the process at various stages and special QA software is required in order to detect and assess the location and severity of the errors. A typical output of the QA process is a filtered set of reads where the lowest quality reads have been filtered out and each base within each read is assigned a quality score which represents the best current estimate of the probability that the base has been called incorrectly. The currently most frequently used file format for storing DNA sequence reads along with their read qualities is a text file known as fastq.

Depending on whether the organism under study has previously been sequenced there may already exist a reference sequence for it i.e. a file that for each genomic location describes the most frequently occurring nucleotide for that species at that location. Humans, and many other species of organisms already have reference sequences available. If the reference sequence for the organism under study is available then the next processing step involves searching for the position in the reference sequence that best matches each read that has been generated for the sample under study in the previous step. The coordinate of the best match is then assumed to be the location in the genome where that particular read has originated from. This process is known as genome alignment and it is very resource intensive for species with large genomes such as humans (3 billion bases) because a typical sequencing effort will generate at least 1 billion reads for a single sample, and each read needs to be mapped to the reference genome. This problem is made more difficult by the fact that an organism's genome typically has a large proportion of repeated sequence fragments and thus the generated reads do not uniquely align to a single location on the reference. A list of matching positions is generated instead, where each match needs to be scored and the highest scoring match is assumed to be the true origin of the read. Many alignment algorithms exist but the most accurate and fast ones use a two step process of indexing, implemented via hash tables or prefix/suffix tries, to generate a short list of promising match locations, followed by a more exact local alignment that uses dynamic programming to generate a best match. The alignment process is further complicated by the presence of sequencing errors, various genomic variants, and disease state such as cancer, all of which generate significant (and sometimes drastic) differences between the obtained reads and the reference genome, thus necessitating inexact matching approaches. The best algorithms that are currently available have a typical runtime of 24-48 hours on a modern 8-core machine. The most widely adopted standard for storing the alignment data on disk is the SAM[117] (and its binary and indexed counterpart BAM) format developed in the context of the 1000 Genomes Project. In addition to the sequence data and base qualities that are already available in fastq, the SAM format adds a reference coordinate to each read, an overall mapping quality for the read, and whether each position in the read matches the reference sequence, along with other useful metadata.

When a reference sequence does not exist, or when it is undesirable to use one, genome alignment tools are inapplicable and a different approach, called de-novo assembly, is used. Under this approach each read is broken into smaller subsequences called k-mers (of length k), these k-mers are then used to build a graph structure

called a de Bruijn graph. Unique paths through the graph represent possible arrangements of reads that correspond to the underlying sequence and the highest scoring path is chosen as the true sequence. Using the de-novo assembly approach has some advantages over alignment-based methods because it models the structure of the organism's genome directly as it is observed rather than in relation to a reference. This is because no reference is perfect, but instead each reference has its own set of errors that were introduced in its construction. Furthermore, genomic structural variants, which represent large (hundreds to millions of basepairs long) sequences that may be deleted, duplicated, or inverted within a given genome challenge alignment software because of the alignment errors that they introduce and require sophisticated algorithms to later detect, whereas in the de-novo assembly approach these variants are directly modelled as they occur in the underlying sequence and are thus easier to identify. De-novo assembly has its own set of challenges however related to difficulties dealing with repetitive sequences that are found within the genome, as well as the extremely high resource requirements of de-novo assembly algorithms, especially when it comes to memory. The de Bruijn graph is typically built in memory and can be multiple terabytes in size, thus requiring computers with extremely high memory to process. Since, even when using in-memory graph construction the runtime for a single sample is typically several days, it is impractical to move the graph representation to disk without dramatically increasing the algorithm runtime to the point where its duration becomes unreasonable. In practice whole genome de-novo assembly is currently rarely used for processing human genomic data because of the challenges described above. Instead, modern algorithms supplement read alignment with local assembly of particular genomic regions of interest in order to reap some of the benefits offered by assembly-based methods without incurring all of the costs.

Once the reads have been aligned they are typically sorted by genomic coordinate so that all of the reads that overlap a given coordinate can be examined together at once. This is an expensive sortation step that does not lend itself well to parallelization and takes several hours to complete per sample. Subsequent to the sortation step is another round of data QA which aims to throw out low quality reads that poorly align to the reference. Care must be taken however, because these low quality reads may not only signal underlying data or sequencing issues like sample contamination, or lane-swap, but may also signal the presence of structural variants or integration of retrovirus DNA into the host under study, both of which are of high interest to properly identify. Thus, it is common to split the sample into reads of high quality that are further assessed with one set of algorithms and a set of reads that map with low quality, or fail to map at all, to be assessed with a different set of algorithms.

At this point the data is ready to begin the process of variant calling, that is, identifying the genomic features of the sample that are different from the reference sequence for that organism (i.e. mutations). It is important to distinguish germline variant calling from somatic variant calling at this time. In germline variant calling we are trying to identify the set of variants that have been passed to the individuals under study from their parents and are thus present in every cell of the organism forming the underlying genetic background of that individual where some variants

may be neutral to the organism's survival, some may be beneficial, and some may be deleterious. Comprehensively identifying and classifying these is of significant research and clinical interest as they confer susceptibility or resistance to certain disease vectors as well as potential medical remedies and may act as biomarkers to predict disease prognosis or response to treatment within the groups of patients that harbour them.

Somatic mutations are those that each individual cell accumulates over its lifetime and they are of especial interest in the context of cancer where a certain set of mutations accumulated in a particular sequence and over a period of time disrupt the normal cell lifecycle and result in the formation of a malignant tumour. In this context researchers typically sequence both healthy cells (such as those drawn from the patient's blood) and cancerous cells. Mutations are identified in both and the difference between these sets of mutations is then stipulated to be the set of somatic mutations present within that tumour. Just like in the germline case, not all of the somatic mutations contribute to the formation of the cancer and the appropriate identification and classification of those mutations that do (so-called cancer drivers) is an important question of significant clinical and research importance which we consider further below. From a technical standpoint calling somatic variants is significantly more complex than calling germline variants because healthy cells generally conform to the underlying genetic characteristics of the organism, such as the number of chromosomes and ploidy (23 chromosomes, diploid, for humans), whereas in the cancer cells these characteristics can be severely disrupted with entire chromosomes missing or present in amplified copy number, requiring different and more complex statistical models to accurately identify. An additional complexity that is unique to somatic variant calling is the concept of sub-clonal mutations. These are mutations that have been acquired only by some of the cells within a tumour. Since sequencing samples data from a large number of cells within a tumour the reads from which are all pooled together, only a comparatively low number of reads will contain information about sub-clonal mutations, thus making them more difficult to detect, even though such mutations may have a significant impact on the tumour phenotype and thus would be very important to properly identify.

We typically think of three classes of genomic variants that are identified by different methods and oftentimes by separate tools. The simplest to accurately detect, and most frequently occurring are Single Nucleotide Polymorphisms (SNPs), in the germline case, and Single Nucleotide Variants (SNVs), in the somatic case. These are single basepair substitutions where the germline genome differs from the reference sequence by a single letter (for SNPs), or the somatic genome differs from the germline genome by a single letter (for SNVs). SNPs are quite common in humans and occur at the rate of approximately 1 per 1,000 bases on average, or, equivalently, 3 million per individual. Somatic SNVs have a widely varying incidence rate depending on the type of cancer involved with typical rates between (INSERT RATES HERE). For humans, which are diploid (i.e. have two copies of each of the chromosomes, except for the sex chromosomes X and Y), we classify SNPs and SNVs as being either heterozygous (with one reference allele and one variant allele) or homozygous (with both alleles being variant). Methods to detect and accurately genotype SNPs and SNVs typically rely on counting the reads that overlap a given

genomic position and evaluating a statistical model that contrasts the probability of the site being reference versus the probability of the site being variant in the face of potential sequencing errors which are expressed as base quality scores and mapping quality scores (as previously described). The models employed for somatic SNV detection and genotyping are significantly more complex than the models for germline variant detection because of the possibility of sub-clonal mutations (as previously described) as well as regions of amplified copy number (i.e. regions where the organism is no longer diploid but can have any number of additional copies of a chromosomal region, or an entire chromosome). More advanced methods output not only lists of variant sites for a sample but calculate a distribution of genotype likelihoods, i.e. all the possible genotypes at a given variant site along with their relative probabilities so that these can be integrated into the models of downstream statistical analyses in a comprehensive manner.

Indels represent sequence insertions and deletions that are anywhere from 1 base-pair (bp) to about 50 basepairs long. There is no strict upper bound on the length of an indel and individual tools typically decide on their own cutoffs for length although pretty much all tools place their cutoff at a length that is smaller than the typical read length (150 - 500 bp presently). Indel callers typically look for several mismatched bases in a row between the reference and the sample under study and classify the entire length of the mismatched sequence as an insertion or deletion correspondingly. Other indel callers borrow some of the methodology from structural variant callers which are similar to indels, only typically bigger in size, and are potentially more complex.

Structural variants (SVs) are more large scale genomic rearrangements that occur in both germline and somatic genomes and can have a very drastic effect on the organism's phenotype because they can affect a large number of genes at once, resulting in the loss of function of particular important genes, or the creation of gene fusions where, because of a rearrangement, one gene comes under the programmatic control of another gene thereby disrupting important cellular processes. The most common types of structural variants include insertions, deletions, segmental duplications, inversions, and translocations. Simpler structural variants sometimes combine to produce more complex events that are especially difficult to detect properly. The methods for calling and genotyping of structural variants typically rely on looking at the reads that are deemed low quality for the SNP calling process. These are reads that fail to map to the reference genome, split-reads, which are reads where one part of the read maps to one location on the reference and another part of the read maps to another location, and divergently mapped reads (sequencing is frequently done on read pairs where two ends of a DNA fragment of standard size are sequenced in the opposite directions generating a pair of reads with a standard distance, called insert size, inbetween them), with a shorter or longer than expected insert size. SV callers break down these reads into smaller fragments (k-mers) and attempt to map these k-mers to the reference sequence. The goal is to determine the location of breakpoints, which are positions on the sample genome where a DNA strand break is thought to have occurred as part of the genomic rearrangement that has taken place. Once a list of breakpoints is obtained the algorithm attempts to reconstruct the most likely event sequence that these breakpoints could have arisen

from, pairing up adjacent breakpoints that are the result of a sequence deletion, for example. Thus, each pair of breakpoints typically gives rise to a single SV call in the final output of the caller. SV calling is a complex and error-prone process that generates double-digit false-positive and false-negative rate, especially in the somatic case, where patient genomes can undergo drastic rearrangements as a result of cancer-related processes such as chromothripsis and are thus extremely difficult to resolve with accuracy.

Once variants (SNPs/SNVs, Indels, and SVs) have been comprehensively called, a filtering step is necessary because callers are typically initially tuned for highest sensitivity in order to detect the most variants, thus admitting an increased number of false positive calls. Additionally, because calling of SNPs and SVs typically occurs separately by different tools there can be significant call-set overlap where the SNP caller sees a region as a group of SNPs, whereas the SV caller will see it as a single breakpoint. These overlaps need to be resolved in order to avoid redundant calls. A number of filtering approaches exist, some of which rely on heuristics such as strand bias, or read support to filter out low quality variants. Other filtering approaches rely on curated variant databases or machine learning methods in order to reduce the number of false positive calls. One popular filtering approach involves ensemble calling where several different variant calling methods are used on the same dataset and a variant is excluded unless it is called by multiple tools. These methods are typically able to reduce the false positive rate of the call-set by 5-10% while only nominally affecting the false negative rate.

When a filtered high quality call-set has been prepared it is of interest to determine which of the variants are likely to have an effect on the organism's phenotype and which variants are likely to have no consequence. This is accomplished via variant annotation. The annotation process consults a database of known genes and other genomic elements (promoters, enhancers, etc.) to determine the likely consequence of each variant based on the type of mutation that it represents i.e. a synonymous mutation (that doesn't change the underlying amino acid) is likely to have no phenotypic effect, whereas a stop gain mutation inside the coding region of a known gene may indicate a potential loss of function of that gene and may thus have a considerable effect on the observed phenotype. When annotating somatic mutations it is important to consider known cancer genes and delineate whether mutations are "passengers" or "drivers" depending on whether they are thought to be driving the carcinogenesis process by constitutively activating a cancer gene or deactivating a tumour suppressor, or they are simply acquired as part of the genomic instability that is induced by carcinogenesis. An outcome of the variant annotation process then, is a list of somatic or germline variants accompanied by a designation of the known genomic features that they fall in, along with an assigned functional impact. This is typically the last step of an NGS analysis pipeline after which the variant call-set is considered completed and can be used for any number of downstream analyses depending on the particular research question or clinical application being considered. For instance, the variants may be used as input into a Genome Wide Association Study (GWAS), a Quantitative Trait Locus (QTL) analysis, a rare variant association study, or as input into the computation of a clinical biomarker.

2.2.1 File Formats

DNA sequencing studies generate large amounts of data - a single whole-genome sample sequenced at 30x coverage on a modern Illumina sequencer generates roughly 10^9 reads which are strings of length 150 characters and take 100 GB of space on disk when compressed with gzip. Thus, even a moderately-sized study of several thousand individuals needs to grapple with the efficient management of hundreds of terabytes of data. Because of the size of these datasets data storage, access, and exchange formats play a major role in determining the speed, cost, and efficiency with which large-scale analyses can be undertaken. The field of genomics has developed in bursts associated with the major international projects that have been undertaken in the past 30 years, including the Human Genome Project[94], the HapMap Project[30], and the 1000 Genomes Project[28]. It is the latter project that has given rise to most of the currently adopted file format standards in use today, including FASTA/FASTQ for raw sequence data, SAM/BAM for reference-mapped sequence data, and VCF for representing genomic variants. Because these file formats have become the primary information exchange medium in the field of genomics they have a large influence on software tools and data access patterns that are in common use today and thus warrant a closer look.

FASTA/FASTQ

The FASTA/FASTQ file format is a text file format developed at The Sanger Institute for representing genomic sequencing reads[25]. Each read in the file consists of four lines:

- The first is a unique identifier (that encodes some information about the sequencing process). For example - HWUSI-EAS100R:6:73:941:1973#0/1
- The second is the read sequence itself: $S = \{s_i : s \in \{A, C, G, T, N\}\}$. For example - ACGTCCCGTCCCTNTCCA
- The third is a + sign acting as a separator.
- The fourth is a set of per-base quality scores, represented on the Phred scale, that represent an estimate of the probability that the base has been called correctly. If the error probability is defined as ϵ then $Q_{phred} = -10 \times \log_{10}(\epsilon)$ and conversely $\epsilon = 10^{\frac{-Q_{phred}}{10}}$. In the actual file Phred scores are represented with ASCII characters from !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFG HIJKLMNOPQRSTUVWXYZ[\]_^àbcdefghijklmnopqrstuvwxyz{|}~ordered by increasing quality, where ! is the lowest possible quality and ~is the highest.

```
@809R9ABXX101220:5:48:9875:59863/1
TTTCTTTTCTTTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTTTTGAGAAGAGATTTCTCACCTTTCTCCAACGCTCC
+
2?>DCBC=CD>@?BECAD:DBAB?BA>@AA>C?CCBCCCB###################################
@809R9ABXX101220:5:61:10970:200284/1
TTTCTTTTCTTTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTTTTGGGATGGGGTTTTCCCCCTGTTTCCCGGGCGG
+
BABEECCCEEDCDCFEDFFEDDDDDDDDDDDDDDDDDDDDDCC#################################
@809R9ABXX101220:5:66:9966:82226/2
TCTTTTCTTTTCTTTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTTTGAATGGGGTTCTCCTCCTTTTCCCCCGGGT
+
>AB@AADDBBCEDCCCCEECEEECCCCCCCCCCCCCCCCCCCA>>#############################
@809R9ABXX101220:5:1:3553:170934/1
CTTTTCTTTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTTTTGTAGTGGTGTTTCGTCCTGGTTCCCTCGCCTGCGCTC
+
2B;5>><;=;A@:6:>DAABB=6?6AB:6CA6>C7>C#######################################
@809R9ABXX101220:5:24:11915:38264/1
CTTTTCTTTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTTTTGGGATGGGGTTTCGCTCCTTTTTCCCAGGCTGGGGTGC
+
BBBCCEECCCDFECFFEDDDDDDDDDDDDDDDDDDDDCC#####################################
@809R9ABXX101220:5:65:20977:60085/2
TTTTCTTCCTTTTTTTTTTTTTTTTTTTTTTGAATTGACGTTCCGCACCTGTCGCACGGCTTGCTCGCACCGTCGCC
+
=;;<=/;94??AABAA8BCABB7BA###################################################
@809R9ABXX101220:5:27:2026:120477/1
TTTTTTTTTTTTTTTTTTTTGAGAGGGGGTTTCGCCCCTGTTGCCCAGGCTGGACTGCCCCGTCCCCGGCCAAAAA
+
>>>>?@@@@@@@@@@@@AA#########################################################
```

Figure 2.3: Excerpt from a FASTQ file. /1 and /2 at the end of read ID indicate whether read is first or second in pair.

Since FASTQ is a large text file with highly repetitive content it is typically stored in a compressed manner.

SAM/BAM

The Sequence Alignment Map (SAM) text file format and its accompanying binary compressed version BAM was created for sequencing data storage and analysis in the context of the 1000 genomes project[117]. These files provide additional fields on top of the ones available in FASTA/FASTQ in order to relay information related to sequence alignment to a reference genome, and are the most widely used format for storing DNA sequencing data today. A SAM file consists of a header and a body. These are described below. All tables, descriptions, and definitions in this section are reproduced or adapted from the SAM file specification at https://samtools.github.io/hts-specs/SAMv1.pdf.

SAM Header Each line of the header begins with the character '`@`' followed by a header record (see Table 2.1). Each line is TAB-delimited and, apart from `@CO` lines, each data field follows the format '`TAG:VALUE`' where `TAG` is a two-character string that defines the format and content of `VALUE`.

Table 2.1: SAM file header record and column definition. Tags listed with '*' are required. (adapted from https://samtools.github.io/hts-specs/SAMv1.pdf)

Tag	Description
`@HD`	The header line. The first line if present.
`VN`*	Format version. *Accepted format*: `/^[0-9]+\.[0-9]+$/`.
`SO`	Sorting order of alignments. *Valid values*: `unknown` (default), `unsorted`, `queryname` and `coordinate`.
`GO`	Grouping of alignments, indicating that similar alignment records are grouped together but the file is not necessarily sorted overall. *Valid values*: `none` (default), `query` (alignments are grouped by QNAME), and `reference` (alignments are grouped by RNAME/POS).
`SS`	Sub-sorting order of alignments. Valid values are of the form *sort-order*`:`*sub-sort*, where *sort-order* is the same value stored in the `SO` tag and *sub-sort* is an implementation-dependent colon-separated string further describing the sort order. *Regular expression*: `(coordinate|queryname|unsorted)(:[A-Za-z0-9_-]+)+`
`@SQ`	Reference sequence dictionary. The order of `@SQ` lines defines the alignment sorting order.
`SN`*	Reference sequence name. The `SN` tags and all individual `AN` names in all `@SQ` lines must be distinct. The value of this field is used in the alignment records in RNAME and RNEXT fields. Regular expression: `[!-)+-<>-~][!-~]*`
`LN`*	Reference sequence length. *Range*: $[1, 2^{31} - 1]$
`AH`	Indicates that this sequence is an alternate locus.
`AN`	Alternative reference sequence names.
`AS`	Genome assembly identifier.
`DS`	Description. UTF-8 encoding may be used.
`M5`	MD5 checksum of the sequence.
`SP`	Species.
`UR`	URI of the sequence.
`@RG`	Read group. Unordered multiple `@RG` lines are allowed.
`ID`*	Read group identifier. Each `@RG` line must have a unique `ID`.
`BC`	Barcode sequence identifying the sample or library.
`CN`	Name of sequencing center producing the read.
`DS`	Description. UTF-8 encoding may be used.
`DT`	Date the run was produced (ISO8601 date or date/time).
`FO`	Flow order. The array of nucleotide bases that correspond to the nucleotides used for each flow of each read.
`KS`	The array of nucleotide bases that correspond to the key sequence of each read.
`LB`	Library.
`PG`	Programs used for processing the read group.
`PI`	Predicted median insert size.
`PL`	Platform/technology used to produce the reads. *Valid values*: `CAPILLARY`, `LS454`, `ILLUMINA`, `SOLID`, `HELICOS`, `IONTORRENT`, `ONT`, and `PACBIO`.
`PM`	Platform model. Free-form text providing further details of the platform/technology used.
`PU`	Platform unit (e.g. flowcell-barcode.lane for Illumina or slide for SOLiD). Unique identifier.
`SM`	Sample. Use pool name where a pool is being sequenced.
`@PG`	Program.
`ID`*	Program record identifier. Each `@PG` line must have a unique `ID`. The value of `ID` is used in the alignment `PG` tag and `PP` tags of other `@PG` lines. `PG` IDs may be modified when merging SAM files in order to handle collisions.
`PN`	Program name
`CL`	Command line. UTF-8 encoding may be used.
`PP`	Previous `@PG-ID`. Must match another `@PG` header's `ID` tag. `@PG` records may be

Table 2.3: SAM file alignment record mandatory column definition. (adapted from https://samtools.github.io/hts-specs/SAMv1.pdf)

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	`[!-?A-~]{1,254}`	Query template NAME
2	FLAG	Int	$[0, 2^{16} - 1]$	bitwise FLAG
3	RNAME	String	`\*|[!-()+-<>-~][!-~]*`	Reference sequence NAME
4	POS	Int	$[0, 2^{31} - 1]$	1-based leftmost mapping POSition
5	MAPQ	Int	$[0, 2^{8} - 1]$	MAPping Quality
6	CIGAR	String	`\*|([0-9]+[MIDNSHPX=])+`	CIGAR string
7	RNEXT	String	`\*|=|[!-()+-<>-~][!-~]*`	Ref. name of the mate/next read
8	PNEXT	Int	$[0, 2^{31} - 1]$	Position of the mate/next read
9	TLEN	Int	$[-2^{31} + 1, 2^{31} - 1]$	observed Template LENgth
10	SEQ	String	`\*|[A-Za-z=.]+`	segment SEQuence
11	QUAL	String	`[!-~]+`	ASCII of Phred-scaled base QUALity+33

SAM Body The body of a SAM file contains alignment records (see Section 2.2.2 on details of alignment algorithms). Each record has 11 mandatory fields. These fields always appear in the same order and must be present, but their values may be '0' or '*' (depending on the field) if the corresponding information is unavailable. Table 2.3 lists the mandatory fields in the SAM format: Alignment records represent the information contained in a sequencing read subsequent to it being aligned to a reference genome (see Figure 2.4).

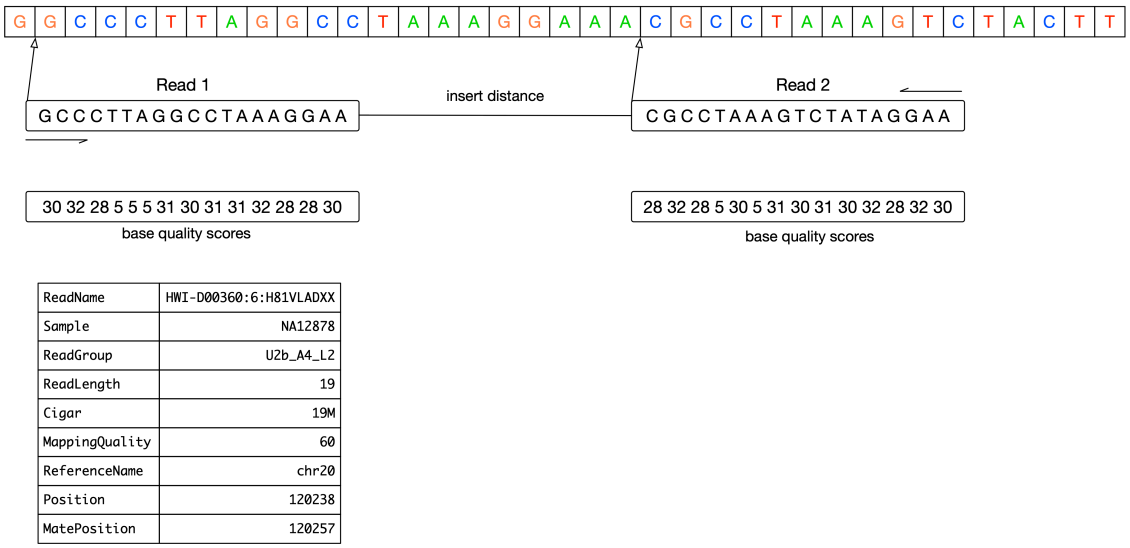


Figure 2.4: A read-pair that is aligned to the reference.

1. QNAME: Query template NAME. Reads/segments having identical QNAME are regarded to come from the same template. A QNAME '*' indicates the information is unavailable. In a SAM file, a read may occupy multiple alignment lines, when its alignment is chimeric or when multiple mappings are given.

2. FLAG: Combination of bitwise FLAGs. Each bit is explained in the following table:

Bit		Description
1	0x1	template having multiple segments in sequencing
2	0x2	each segment properly aligned according to the aligner
4	0x4	segment unmapped
8	0x8	next segment in the template unmapped
16	0x10	SEQ being reverse complemented
32	0x20	SEQ of the next segment in the template being reverse complemented
64	0x40	the first segment in the template
128	0x80	the last segment in the template
256	0x100	secondary alignment
512	0x200	not passing filters, such as platform/vendor quality controls
1024	0x400	PCR or optical duplicate
2048	0x800	supplementary alignment

- For each read/contig in a SAM file, it is required that one and only one line associated with the read satisfies 'FLAG `& 0x900 == 0`'. This line is called the *primary line* of the read.
- Bit 0x100 marks the alignment not to be used in certain analyses when the tools in use are aware of this bit. It is typically used to flag alternative mappings when multiple mappings are presented in a SAM.
- Bit 0x800 indicates that the corresponding alignment line is part of a chimeric alignment. A line flagged with 0x800 is called as a *supplementary line*.
- Bit 0x4 is the only reliable place to tell whether the read is unmapped. If 0x4 is set, no assumptions can be made about RNAME, POS, CIGAR, MAPQ, and bits 0x2, 0x100, and 0x800.
- Bit 0x10 indicates whether SEQ has been reverse complemented and QUAL reversed. When bit 0x4 is unset, this corresponds to the strand to which the segment has been mapped. When 0x4 is set, this indicates whether the unmapped read is stored in its original orientation as it came off the sequencing machine.
- Bits 0x40 and 0x80 reflect the read ordering within each template inherent in the sequencing technology used.[1] If 0x40 and 0x80 are both set, the read is part of a linear template, but it is neither the first nor the last read. If both 0x40 and 0x80 are unset, the index of the read in the template is unknown. This may happen for a non-linear template or when this information is lost during data processing.
- If 0x1 is unset, no assumptions can be made about 0x2, 0x8, 0x20, 0x40 and 0x80.
- Bits that are not listed in the table are reserved for future use. They should not be set when writing and should be ignored on reading by current software.

[1]For example, in Illumina paired-end sequencing, first (0x40) corresponds to the R1 'forward' read and last (0x80) to the R2 'reverse' read. (Despite the terminology, this is unrelated to the segments' orientations when they are mapped: either, neither, or both may have their reverse flag bits (0x10) set after mapping.)

3. RNAME: Reference sequence NAME of the alignment. If `@SQ` header lines are present, RNAME (if not '*') must be present in one of the `SQ-SN` tag. An unmapped segment without coordinate has a '*' at this field. However, an unmapped segment may also have an ordinary coordinate such that it can be placed at a desired position after sorting. If RNAME is '*', no assumptions can be made about POS and CIGAR.

4. POS: 1-based leftmost mapping POSition of the first CIGAR operation that "consumes" a reference base (see table below). The first base in a reference sequence has coordinate 1. POS is set as 0 for an unmapped read without coordinate. If POS is 0, no assumptions can be made about RNAME and CIGAR.

5. MAPQ: MAPping Quality. It equals $-10\log_{10}\Pr\{\text{mapping position is wrong}\}$, rounded to the nearest integer. A value 255 indicates that the mapping quality is not available.

6. CIGAR: CIGAR string. The CIGAR operations are given in the following table (set '*' if unavailable):

Op	BAM	Description	Consumes query	Consu refere
`M`	0	alignment match (can be a sequence match or mismatch)	yes	yes
`I`	1	insertion to the reference	yes	no
`D`	2	deletion from the reference	no	yes
`N`	3	skipped region from the reference	no	yes
`S`	4	soft clipping (clipped sequences present in SEQ)	yes	no
`H`	5	hard clipping (clipped sequences NOT present in SEQ)	no	no
`P`	6	padding (silent deletion from padded reference)	no	no
`=`	7	sequence match	yes	yes
`X`	8	sequence mismatch	yes	yes

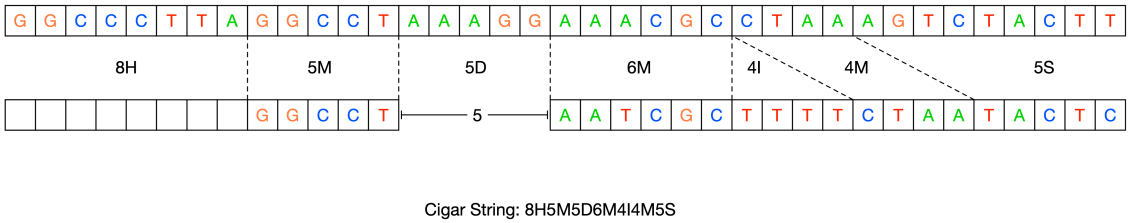


Figure 2.5: Example of an alignment CIGAR string.

- "Consumes query" and "consumes reference" indicate whether the CIGAR operation causes the alignment to step along the query sequence and the reference sequence respectively.
- `H` can only be present as the first and/or last operation.
- `S` may only have `H` operations between them and the ends of the CIGAR string.

- For mRNA-to-genome alignment, an `N` operation represents an intron. For other types of alignments, the interpretation of `N` is not defined.
- Sum of lengths of the `M/I/S/=/X` operations shall equal the length of SEQ.

7. RNEXT: Reference sequence name of the primary alignment of the NEXT read in the template. For the last read, the next read is the first read in the template. If `@SQ` header lines are present, RNEXT (if not '*' or '=') must be present in one of the `SQ-SN` tag. This field is set as '*' when the information is unavailable, and set as '=' if RNEXT is identical RNAME. If not '=' and the next read in the template has one primary mapping (see also bit 0x100 in FLAG), this field is identical to RNAME at the primary line of the next read. If RNEXT is '*', no assumptions can be made on PNEXT and bit 0x20.

8. PNEXT: 1-based Position of the primary alignment of the NEXT read in the template. Set as 0 when the information is unavailable. This field equals POS at the primary line of the next read. If PNEXT is 0, no assumptions can be made on RNEXT and bit 0x20.

9. TLEN: signed observed Template LENgth. If all segments are mapped to the same reference, the unsigned observed template length equals the number of bases from the leftmost mapped base to the rightmost mapped base. The leftmost segment has a plus sign and the rightmost has a minus sign. The sign of segments in the middle is undefined. It is set as 0 for single-segment template or when the information is unavailable.

10. SEQ: segment SEQuence. This field can be a '*' when the sequence is not stored. If not a '*', the length of the sequence must equal the sum of lengths of `M/I/S/=/X` operations in CIGAR. An '=' denotes the base is identical to the reference base. No assumptions can be made on the letter cases.

11. QUAL: ASCII of base QUALity plus 33 (same as the quality string in the Sanger FASTQ format). A base quality is the phred-scaled base error probability which equals $-10 \log_{10} \Pr\{\text{base is wrong}\}$. This field can be a '*' when quality is not stored. If not a '*', SEQ must not be a '*' and the length of the quality string ought to equal the length of SEQ.

BAM Files BAM files are SAM files that have been compressed with block gzip compression[110]. If the reads inside the BAM file are sorted in increasing order of reference coordinate, the BAM file supports random access via a supplementary BAM index (**.bam.bai*) file. Each block inside a BAM file is a separate gzip archive up to 64Kb in size with extra metadata to encode the read positions contained inside the block. The BAM index file contains offsets into the BAM file that correspond to particular genomic coordinate ranges. The BAM file this provides a binary compressed structure that supports indexed search on genomic coordinates. No other indexing, including by ID, is currently supported. The majority of DNA sequencing data in the world is currently stored as BAM files.

VCF Files

The Variant Call Format (VCF)[38] is a type of text file that was created in the context of the 1000 Genomes project to allow the representation of various types of genetic variation that are discovered via NGS experiments. VCF files describe variants in a refererence-relative manner - i.e. all genetic variation is shown with respect to a given reference genome build. VCF files have a tabular form and allow one to describe genetic variants as they occur in (or are absent from) a entire cohort of samples using a single file. This file format, despite several significant limitations, has become the widely adopted standard for representing genetic variation. All descriptions in this section have been reproduced or adapted from the 2011 Danecek et al. paper and the VCF specification ()

A VCF file consists of the following sections - Meta-information section describes the format and content of the data contained in the file, header section specifies the columns present in the file, data section contains the actual variant calls that are being described.

Meta-information The meta-information section contains a number of lines that describe the data section. The first line of this section is a *fileformat* line that specifies what version of the VCF spec the file adheres to. There can be any number of lines describing INFO fields. These fields are populated into the INFO column of the data section. A single INFO field may describe a single value or a tuple of values. These values can be Integer, Float, Flag, Character, or String. All of the INFO fields thus described are to be placed into the INFO column as a string of semi-colon-separated key-value pairs. The FILTER field describes any filters that have been applied to the variants. The FORMAT field describes the format of the genotype columns that are specified in the data section. The ALT field describes symbolic alternate alleles, that result from variants that have been called but not accurately genotyped. The *contig* field lists the contigs that the variants specified in the file have been called relative to (typically these are chromosome names from the reference sequence). The SAMPLE field specifies the samples that the variants map to. The PEDIGREE field specifies pedigree relationships between samples in the file.

Header The header section is a single tab-delimited line that lists what columns are present in the body. The mandatory columns are:

- #CHROM
- POS
- ID
- REF
- ALT

- QUAL
- FILTER
- INFO

If the VCF file contains genotypes then the INFO column is followed by a FORMAT column, followed by one column for each present sample where each column name is the respective sample ID and all sample IDs are unique within the file.

Data The data section of a VCF file contains the actual list of variants and their genotypes in all samples in tabular form where the columns align with the header section. The values are tab-separated. Any missing values are indicated with a ”.” (dot). The line contents are as follows:

CHROM - An identifier of the chromosome where the variant resides. The ID of the chromosome should match one of the *contig* entries in the meta-information section. All of the variants that belong to a single CHROM should exist in a single contiguous block of rows in a VCF file.

POS - 1-based position of the variant with respect to the reference chromosome specified in CHROM. Variant positions should be sorted numerically in increasing order.

ID - A semi-colon separated list of unique identifiers, where they exist. If no identifier exists a missing value should be indicated.

REF - the reference bases corresponding to the variant location where each base $b \in \{A, C, G, T, N\}$.

ALT - a comma separated list of alternative alleles. The alleles do not have to called in any of the samples. Value can be either a String of $b \in \{A, C, G, T, N, *\}$ or a missing value (when there is no variant).

QUAL - the PHRED scale variant quality. When ALT is present QUAL is $-10 \log_{10}(novariant)$ and when ALT is missing QUAL is $-10 \log_{10}(variant)$. If QUAL is unknown then missing value must be specified.

FILTER - encodes the filter status. If the variant passes all QC filters the value should be PASS. Otherwise there should be a semi-colon separated list of codes for filters that have not passed. If FILTER information is not available there should be a missing value.

INFO - A list of key-value pairs for the additional fields encoded for each variant as specified in the INFO lines of the metadata section. Keys without corresponding values may be used to indicate group membership. There is a number of frequently used reserved keys (see Table 2.4).

Key	Number	Type	Description
AA	1	String	Ancestral allele
AC	A	Integer	Allele count in genotypes, for each ALT allele, in the same order as listed
AD	R	Integer	Total read depth for each allele
ADF	R	Integer	Read depth for each allele on the forward strand
ADR	R	Integer	Read depth for each allele on the reverse strand
AF	A	Float	Allele frequency for each ALT allele in the same order as listed (estimated from primary data, not called genotypes)
AN	1	Integer	Total number of alleles in called genotypes
BQ	1	Float	RMS base quality
CIGAR	A	String	Cigar string describing how to align an alternate allele to the reference allele
DB	0	Flag	dbSNP membership
DP	1	Integer	Combined depth across samples
END	1	Integer	End position (for use with symbolic alleles)
H2	0	Flag	HapMap2 membership
H3	0	Flag	HapMap3 membership
MQ	1	Float	RMS mapping quality
MQ0	1	Integer	Number of MAPQ == 0 reads
NS	1	Integer	Number of samples with data
SB	4	Integer	Strand bias
SOMATIC	0	Flag	Somatic mutation (for cancer genomics)
VALIDATED	0	Flag	Validated by follow-up experiment
1000G	0	Flag	1000 Genomes membership

Table 2.4: Reserved INFO keys (from https://samtools.github.io/hts-specs/VCFv4.3.pdf).

If genotype information is present, the exact same definition is used for all samples. The FORMAT field specifies a column seprated list of the data types and their order that are present in the genotype columns. This is followed by one data block for each sample that contains the genotype data as described in the FORMAT column. All keys are optional but missing values should be indicated. There is a number of frequently used and reserved keys (see Table 2.5).

Field	Number	Type	Description
AD	R	Integer	Read depth for each allele
ADF	R	Integer	Read depth for each allele on the forward strand
ADR	R	Integer	Read depth for each allele on the reverse strand
DP	1	Integer	Read depth
EC	A	Integer	Expected alternate allele counts
FT	1	String	Filter indicating if this genotype was "called"
GL	G	Float	Genotype likelihoods
GP	G	Float	Genotype posterior probabilities
GQ	1	Integer	Conditional genotype quality
GT	1	String	Genotype
HQ	2	Integer	Haplotype quality
MQ	1	Integer	RMS mapping quality
PL	G	Integer	Phred-scaled genotype likelihoods rounded to the closest integer
PQ	1	Integer	Phasing quality
PS	1	Integer	Phase set

Table 2.5: Reserved genotype keys (from https://samtools.github.io/hts-specs/VCFv4.3.pdf).

The following keys are most important and frequently used:

GT - The genotype, encoded as allele values separated by / for unphased genotypes and | for phased genotypes. The values are 0 for reference allele, 1 for first allele listed in ALT, 2 for the second, and so on (for instance 0/1 for heterozygous variant in a diploid sample). When a call cannot be made the missing value is used (for instance ./.).

GL - Genotype likelihoods. A comma separated list of log_{10} likelihoods for all possible genotypes given the set of REF and ALT alleles at the locus.

GP - Genotype posterior probabilities, in the same order as GL field.

GQ - Genotype Quality in PHRED scale. Probability the genotype call is wrong given that the site is variant.

AD - Allele Depth. Per-sample read depth for each allele.

DP - Read Depth. $\sum_i AD_i$.

See Listing 21 for an example of a VCF file.

Remarks

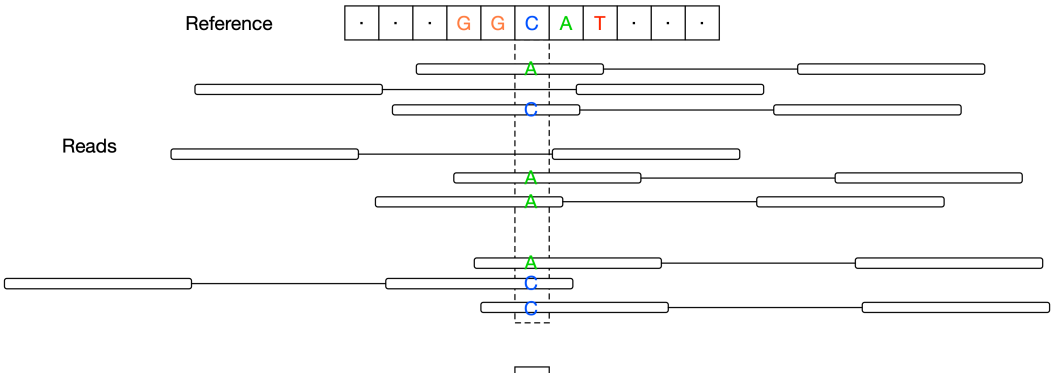
The SAM/BAM and VCF file formats have bencome the de-facto standards for representing sequencing read data and genetic variation respectively. Because of this status most computational tools that exist in this space either consume or produce one of these data-types and the data analysis process is influenced by the file access patterns inherent in these standards. This introduces a number of challenges and limitations that have held back scalability of the existing tools to larger data sets. Specifically:

- The focus on files for information storage and retrieval implies that sophisticated file management schemes must be deployed for successful management of large cohorts of samples. This includes concerns of data security, and data migration that must be implemented at the file-system level.
- Storage of sequence data at sample level of granularity in SAM/BAM files creates very large files that are typically greater than 150 GB in size per sample and are thus quite cumbersome to work with.
- Lack of usage of databases implies only basic indexing and querying schemes are possible for sequencing and variant data. Large amounts of data thus need to be loaded into memory in order to perform basic queries.
- Indexing only by genomic coordinate implies a coordinate-based data traversal mechanisms that process a genome linearly from beginning to the end.
- The multi-sample VCF format makes it easy to interpret all carriers for a single variant (by reading a single row), but is not well suited for interrogating all variants for a single individual (scanning all rows for a single column) in a text file.
- Storage of sequencing data in multiple per-sample files does not take advantage of extreme sequence similarity between samples at a given genomic locus, thus reducing the opportunities for data compression and driving up project costs for large scale sequencing efforts where data set size exceeds 1PB.

The stream-based approach adopted by the Rheos framework desribed in this work attempts to address many of the above issues.

2.2.2 Alignment

Genome alignment (also called mapping) is the process that, given a DNA sequencing read, finds the location in a reference genome that the read best matches to. This is an important process in a next generation sequencing pipeline because it provides a way of ordering the otherwise unordered collection of raw reads (by refrence coordinate) that can be used to find locations where the sample genome is different from the reference (see Figure 2.6).



Locus

A	
C	

Chromosome	2
Position	120238
Genotype	0/1
Genotype Quality	30
Genotype Likelihoods	-323.03,-99.29,-802.53
Ref	C
Alt	A
Read Depth	7
Ref Count	3
Alt Count	4
Sum of Ref Qual	92
Sum of Alt Qual	120

Figure 2.6: Collection of reads aligned to a reference genome and evaluated at a locus that contains a heterozygous SNP.

This process is equivalent to substring search where the string being searched is 3×10^9 characters long and there are 10^9 patterns of length 150 to be found. This process is complicated by the fact that reads from a sample have both sequencing errors and genuine genetic variation that make them differ from the reference, and the fact that certain regions of the genome can be highly repetititve, with the same sequence pattern occurring hundreds of times, making unique mapping challenging.

There are several key applications of alignment that involve sequences of varying length, require different properties, and may not all be best accomplished by the same algorithms. These are:

- Alignment of individual single-end and paired-end reads to a human reference sequence. Where most of the read is expected to match successfully. This is the most abundant use case and the one towards which most alignment algorithms and software implementations are geared.
- Alignment of split-reads to a reference sequence. Split-reads are those where a part of the read maps to one position in the reference sequence and another part of the read maps to a different part of the reference indicating that the read spans a genomic rearrangement. Reads that admit a split alignment may

come out unmapped from a normal alignment stage. This type of alignment is important for both germline structural variant calling and somatic structural variant calling where genomic aberrations are more common.

- Alignment of reads to a group of reference genomes for common pathogens (viruses, bacteria), as well as other common sample contaminating species.

- Alignment of assembled contigs to the reference sequence after local assembly. Some variant calling methods will perform local assembly of reads into a group of potential haplotypes and will perform alignment of these haplotypes to the reference sequence to identify where the actual variants are. The haplotypes may be anywhere from hundreds to millions of bases long.

- Alignment of reads to a group of alternative haplotypes. Variant callers may generate a list of candidate alternative haplotypes and reads need to be aligned to all of these in order to determine which haplotype is best supported by the reads data.

Initially, algorithms for mapping sequencing reads (such as Maq[115] and SOAP[118]) have used a hash table approach[154] (building up tables of subsequences) for finding promising approximate locations for a read within the reference, and then using the Smith-Waterman[175] dynamic programming algorithm for selecting the best matching location. These approaches, however, have proven to be quite slow with typical runtimes exceeding 72 hours on a single high coverage whole genome sample. For the past decade the best available genome alignment approaches in terms of balanced accuracy and speed of processing have been based on the Burrows-Wheeler Transform (BWT)[19] and the FM index[56]. Two of the most popular and widely used are Bowtie[97] and BWA[113]. With recent improvements in computer power, hash table based approaches, such as minimap2[108] and SNAP[207] have made a comeback and are again becoming competitive in the alignment space. A commercial tool called Novoalign (http://www.novocraft.com/products/novoalign/) has consistently been a top performer in terms of speed and accuracy, but the method is not publically available.

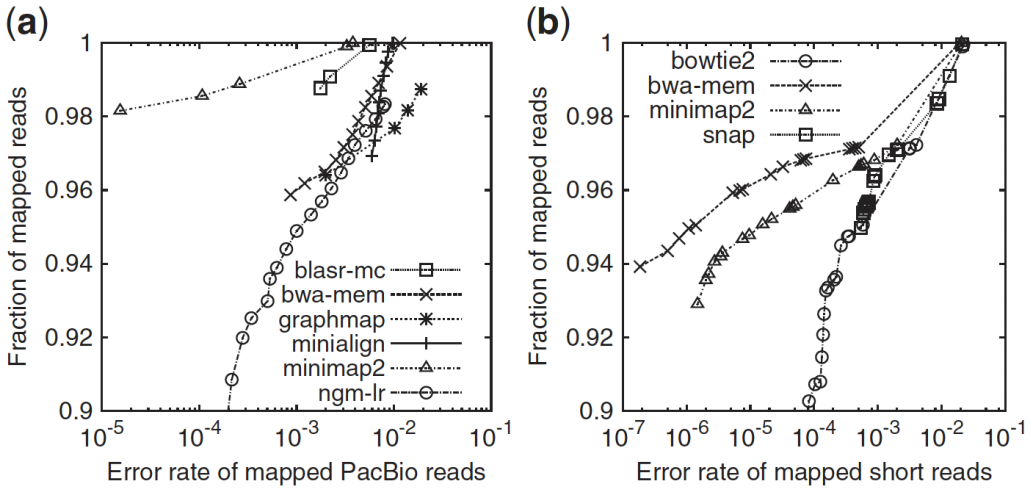


Figure 2.7: Comparison of several aligners on simulated a)long reads >1000 bp, and b) short reads 150bp.[108]

String search is a well studied field, and many genome aligners use the equivalence relationship between suffix trees, suffix arrays, and the BWT (see Figure 2.8) to construct searchable and compressed data structures suitable for genome aligment applications.

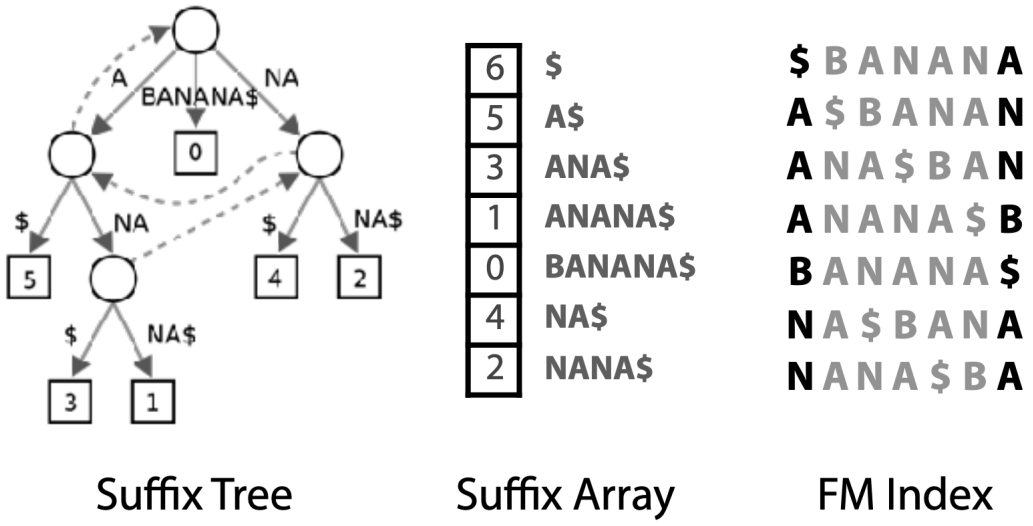


Figure 2.8: Equivalence between a Suffix Tree, Suffix Array, and the BWT matrix of the string BANANA.[1]

Suffix Tree A suffix *trie* T is a tree data structure that given a string $s = c_1, c_2, ..., c_n$ over an alphabet Σ, such that $c_i \in \Sigma$ encodes all of the suffixes of s from the root to the leaves. s is terminated with a sentinel character \$. For any character $c \in \Sigma$ and node $n_i \in T$, n_i has at most one child labeled c. Every leaf is the sentinel character \$. See Figure 2.9 for example.

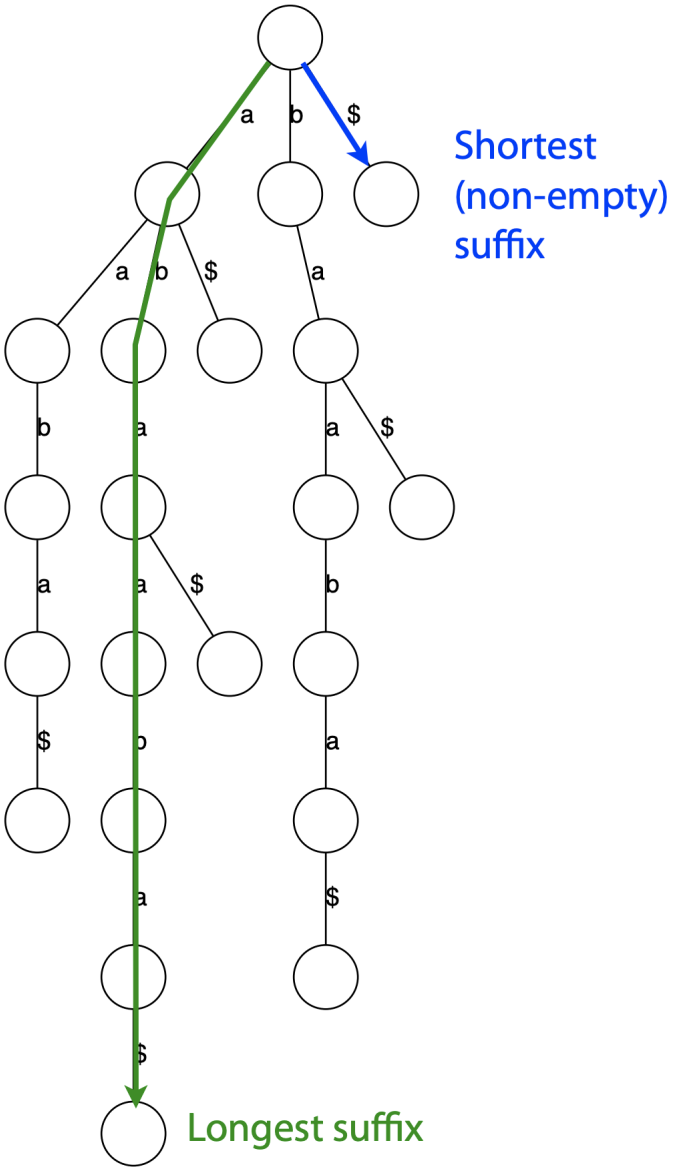


Figure 2.9: Suffix trie of string *abaaba*\$.[1]

Because every substring of s is a prefix of some suffix of s, given a search query q one can check whether q is a substring of s in $\mathcal{O}(|q|)$ time by progressively matching each character of q from the root of T until either all of q is matched (hence q is a substring of T) or at some point a character of q does not have a matching node in T, indicating that q is not a substring of T. The size (number of nodes) of the trie can grow with $\mathcal{O}(|s|^2)$

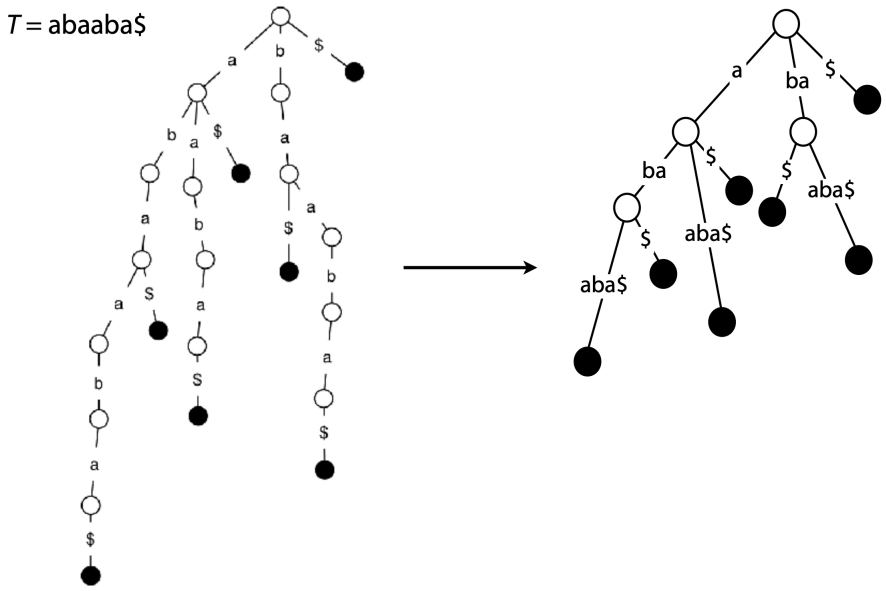


Figure 2.10: Create suffix tree by coalescing non-branching paths into single nodes.[1]

The size of the tree can be reduced by coalescing all non-branching subpaths of T into single nodes (see Figure 2.10).

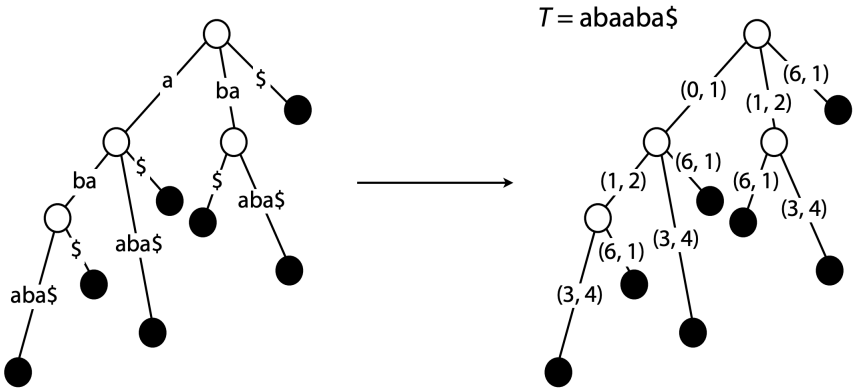


Figure 2.11: Reduce size by replacing substrings by offsets into the original string.[1]

The size can be further reduced to $\mathcal{O}(|s|)$ by replacing string nodes with pairs of offsets into the original string s (see Figure 2.11).

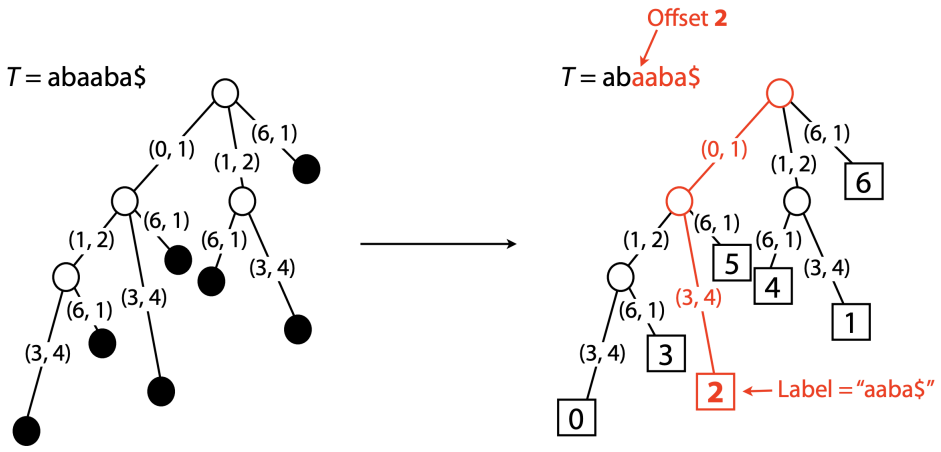


Figure 2.12: Aid search by storing index of substring in leaves.[1]

Searching can be aided by storing offsets of suffixes in the leaf node of the path spelling out that suffix. See Figure 2.12. The online construction algorithm by Ukkonen[185] allows creation of suffix trees in $\mathcal{O}(|s|)$ time and space. Although search against the suffix tree is very fast - $\mathcal{O}(|q|)$, the tree size is still an issue when considering applications to human genome sequencing. A suffix tree for a human would take occupy more than 45 GB of memory[1]. Furthermore, alignment of real reads is made more complex because reads have errors and encode genetic variation, thus inexact searching mechanisms are required.

Suffix Array Suffix arrays were developed by Manber and Myers[123] as a data structure that is equivalent to a suffix tree but occupies less space. Given a string $s = c_1, c_2, ..., c_n$, and letting $s[i, j]$ be a substring of s between indexes i and j, the suffix array SA of s contains integers providing the starting positions of suffixes of s sorted in lexicographical order, s.t. $\forall i \in [1, n] : s[SA[i-1], n] < s[SA[i], n]$. See Figure 2.8 for an exaple suffix array representing the string *banana*. Because the array is sorted, a simple binary search is possible that finds or rules out a match of query q in s in $\mathcal{O}(|q| \log |s|)$, although a more sophisticated search algorithm in $\mathcal{O}(|q| + \log |s|)$ is possible[123]. A suffix array can be constructed by a depth-first traversal of a suffix tree in $\mathcal{O}(|s|)$ time, with the best current algorithm due to Kärkkäinen[89]. Using the array can bring the space requirements for a human-sized genome search index down to about 16GB[1].

The BWT (see Figure 2.13) is constructed from an original string s by appending a special termination character \$ that is lexicographically smaller than all other characters and does not occur in s. Then a matrix of all cyclic rotations of $s\$$ is constructed and sorted in ascending lexicographical order (Figure 2.13 a). The last column of this matrix is the BWT, which can be used to reconstruct the original string and perform substring searches on it. This mechanism relies on the fact that all common prefixes of substrings occupy a contiguous block of rows in the BWT matrix (because of lexicographical ordering, Figure 2.13 c)), and the LF (last-first) property of the BWT. Namely, the character in the last column of the BWT directly

precedes the character in the first column of the BWT in the original string. And, the index of the occurrence of a character in the last column of the BWT is the same as the index of the occurrence of the same character in the first column of the BWT (see Figure 2.13 b)) i.e. the first occurrence of g in the last column corresponds to the first occurence of g in the first column. The third occurrence of a in the last column corresponds to the third occurrence of a in the first column, etc.

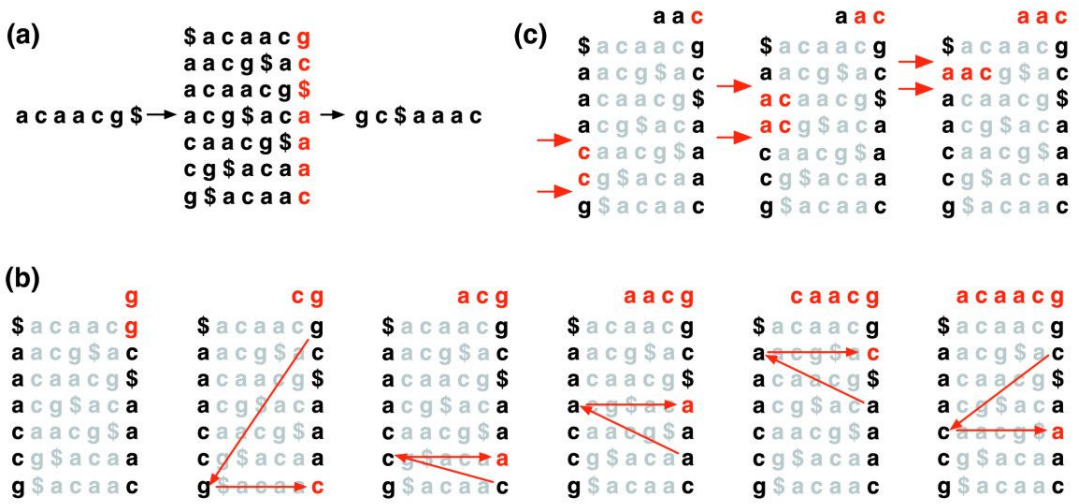


Figure 2.13: a) Creating the BWT of a string. b) All substrings with the same prefix occupy adjacent rows. c) The original string can be recovered from the BWT by using the LF rule.[97]

Alternatively the BWT can be constructed from the suffix array via:

$$BWT[i] = \begin{cases} s[SA[i]-1], & \text{if } SA[i] > 1 \\ \$, & \text{otherwise} \end{cases} \tag{2.1}$$

, per [173]

The entire original string can be recovered from the BWT in the following manner. The last column is the BWT. The first column is the lexicographically sorted BWT. Find the row that starts with \$. The character in the last column in that row is the last character of original string s (g in the example above). Find the first occurrence of that character in the first column. Look up the character in the last column of that row (c in the example above). This is the second-last character of s. Note the index of its occurrence in the last column (it's a second c in the example). Find the row with the same character and the same index in the first column (second c in the first column). The last character in that row is the second-last character of s. Repeat this process until the last character in a row is \$. This recovers the original string.

FM Index

Bowtie Bowtie is a popular aligner developed by Ben Langmead in 2009, with a focus on fast alignment of short reads[97], later followed up and ehnanced with Bowtie 2 for longer reads[95], and adapted massively parallel deployment on the AWS cloud with Crossbow[96]. Since the <50bp long short-reads that the original Bowtie was created for are no longer seen we focus our exposition on Bowtie 2.

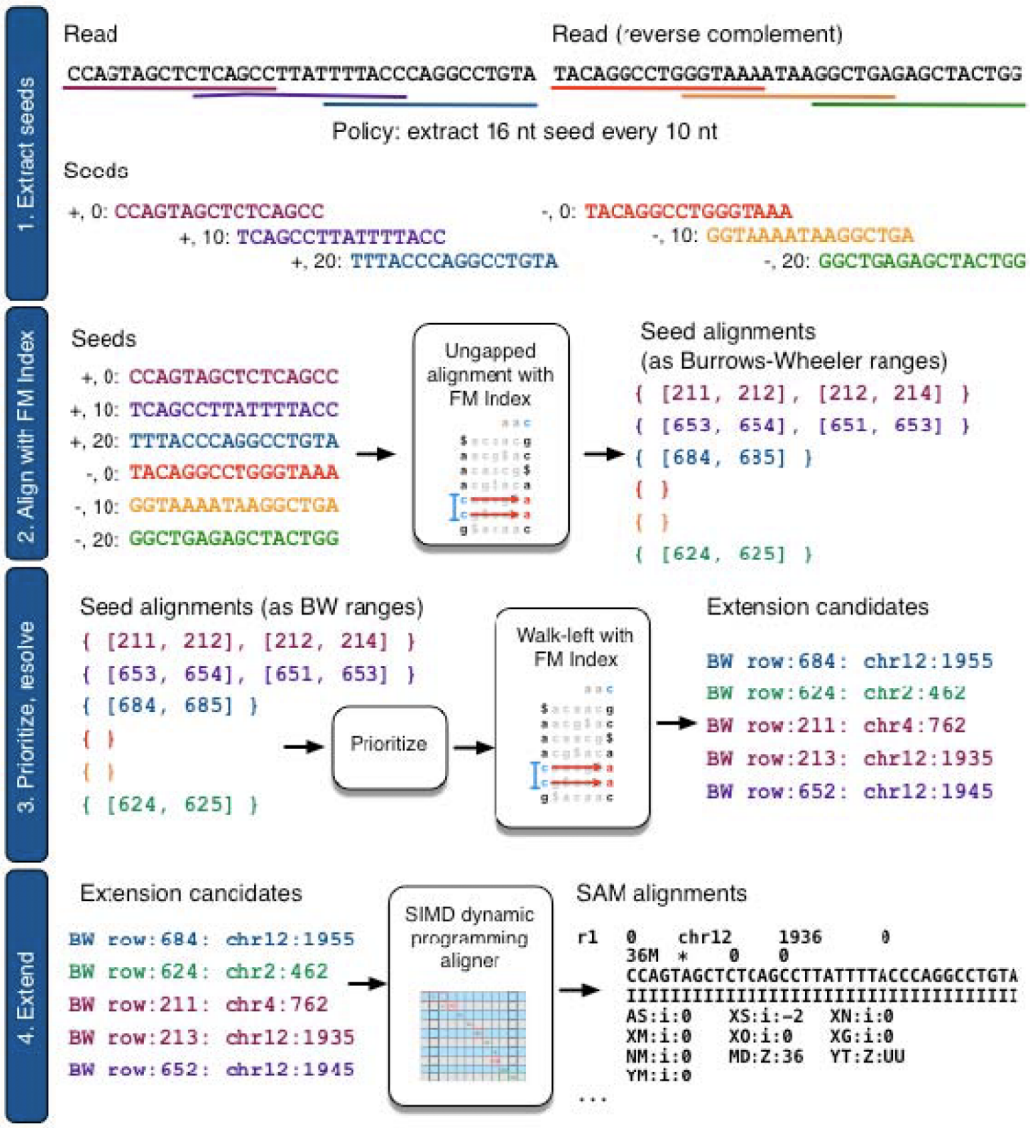


Figure 2.14: S
teps used by Bowtie 2 alignment.[95]

Figure 2.14 shows the sequence of steps used by the Bowtie 2 alignment process. One of the big challenges for aligners is the ability to deal with gapped alignment i.e. sequences in the reads that are inserted or deleted with respect to the reference. These may be the result of sequencing errors or genuine genetic variation present in the sample. Since the read sequence cannot be matched to the reference exactly, gapped alignment drastically increases the search space of possible matches that need to be considered, corresponding to different possible gap sizes. In order to mitigate the degrading effect of gapped alignment on performance, Bowtie adopts a seed-

and-extend mechanism where it performs ungapped exact alignments of substrings of the read that become seeds, followed by a gap-aware dynamic programming-based seed extension to obtain the final alignment. The index is built using a variant of [89] and occupies 2.2 GB of disk space for the human reference.

For alignment every read and its reverse complement are divided into overlapping subsequences that become the potential seeds. Each seed is aligned to the reference using the FM index in an ungapped fashion, but allowing a configurable number of mismatches. For a given read, if the FM index search fails to match the seed to the refrence at some location i it backtracks and attempts a base substitution at that position that can be successfully matched and in a way that maximizes the overall alignment quality. It performs up to k (configurable, default 1) such substitutions per read. Since the substitutions are made in a greedy manner the alignment result may not be globally optimal. To avoid potentially excessive backtracking Bowtie builds both a forward and a mirror index of the reference genome and attempts to match both ends of the read one after the other, with the forward and mirror index respectively attempting to obtain a high scoring alignment of both ends. The number of backtracking steps is further cut off at a hard threshold for performance reasons.

The FM index search produces sets of ranges between which each seed matches. Seeds are assigned priorities based on the number of potential matches, where seeds that have fewer potential matches (i.e. map more uiquely) are given higher priority. Seeds are then selected in priority order for exact resolution. Using the BWT LF property and the FM occurrence array the selected range is resolved to an actual maching location for a given seed. The seeds are then extended in both directions using an adaptation of SIMD-accellerated Smith Waterman alignment via dynamic programming. The striped Smith-Waterman algorithm[54] that Bowtie 2's alignment is based on allows end-to-end alignment in the neighbourhood of the seed with arbitrary numbers of mismatches and gaps and is base quality aware. When reads align to repetitive regions there may be many thousands of match locations for the seeds. To more accurately position these reads Bowtie employs a reseeding strategy where the seeds are selected from a read multiple times using a sliding window in an attempt to find a more accurate seed location.

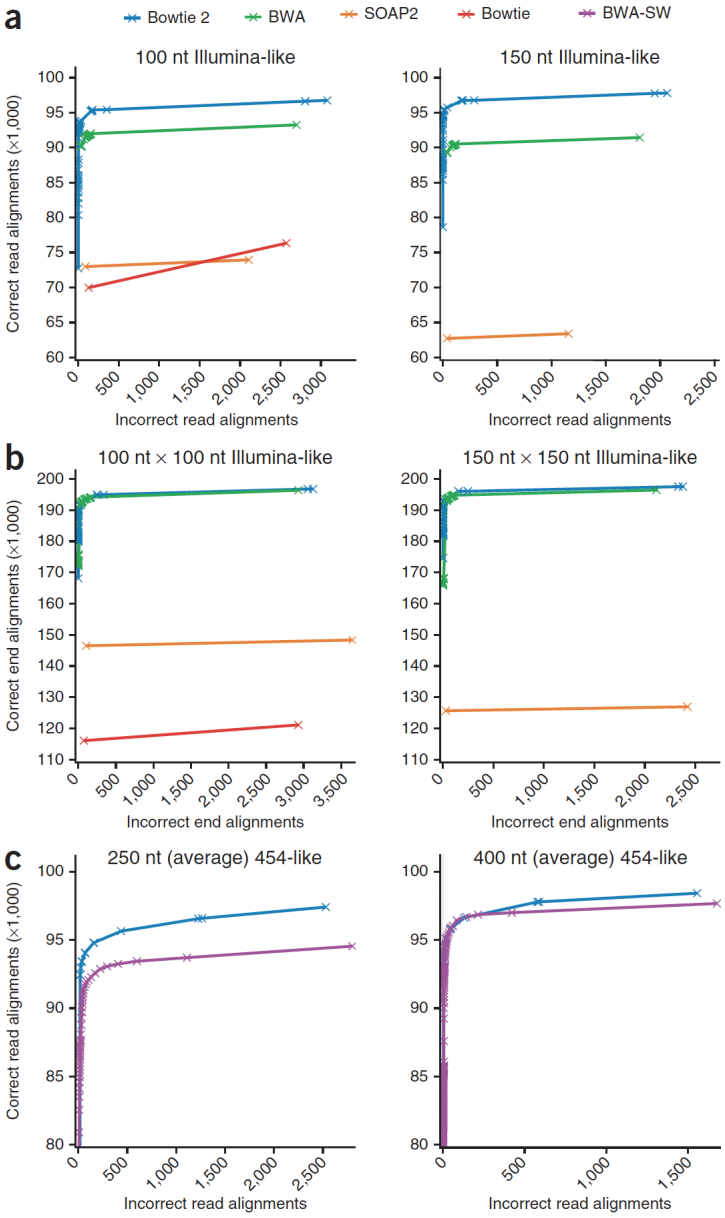


Figure 2.15: Performance comparison of Bowtie 2 to other callers on simulated data a) 100 and 150 bp unpaired, b) 100x100 and 150x150 paired, and c) 250 and 400 long reads.[95]

Bowtie uses information about paired-end reads to attempt to perform a more accurate alignment of the pairs. When the user supplies an expected DNA fragment length and read orientation and one read in a pair is fully aligned, Bowtie calculates a window of reference coordinates where the other read in the pair would be expected to reside and attempts to align the other read in that window using dynamic programing. If it's not able to produce a high quality alignment in this fashion it will attempt to align the other read using the full normal procedure.

A performance comparison by the tool's author Ben Langemead from [95] is shown in Figure 2.15. Bowtie 2 can be seen to be the same or slightly better than BWA and significantly better than Bowtie and SOAP2 on the simulated dataset.

BWA BWA is a tool by Heng Li that implements a series of algorithms for genome alignment based on the BWT and the FM index and described in a series of papers - [113], [112], [104]. Although the original BWA algorithm was focused towards reads that are fewer than 50 basepairs long and is no longer relevant, the subsequent BWA-SW, and BWA-MEM algorithms remain very popular. Since BWA-MEM is seen as the ultimate successor to the previous BWA algorithms we focus our discussion on this implementation of alignement. The following common notation is adopted (see Table **??**)

Table 2.6: BWA Common notation[106],[104]

Symbol	Description
$\Sigma = \{\$, A, C, G, T, N\}$	alphabet of DNA strings with lexicographical order $\$ < A < C < G < T < N$
$\$$	Is a sentinel character
N	Is an ambiguous DNA base
T	String: $T = a_0 a_1 \dots a_{n-1}$ with $a_{n-1} = \$$
$\|T\|$	Length of T including sentinels: $\|T\| = n$
$T[i]$	The i-th symbol in string T: $T[i] = a_i$
$T[i, j]$	Substring: $T[i, j] = a_i \dots a_j$
T_i	Suffix: $T_i = T[i, n-1]$
S	Suffix array; $S(i)$ is the position of the i-th smallest suffix
B	BWT: $B[i] = T[S(i) - 1]$ if $S(i) > 0$ or $B[i] = \$$ otherwise
$C(a)$	Accumul. count array: $C(a) = \|\{0 \leq i \leq n-1 : T[i] < a\}\|$
$O(a, i)$	Occurrence array: $O(a, i) = \|\{0 \leq j \leq i : B[j] = a\}\|$
$P \circ W$	String concatenation of string P and W
Pa	String concatenation of string P and symbol a: $Pa = P \circ a$
$\overline{P}$	Watson-Crick reverse complement of DNA string P

The *suffix array interval* $l^l(P), l^u(P)$ of a string P is defined as:

$$l^l(P) = \min \{k : P \text{ is the prefix of } T_{S(k)}\}$$
$$l^u(P) = \max \{k : P \text{ is the prefix of } T_{S(k)}\}$$
$$l^s(P) = l^u(P) - l^l(P) + 1 \text{ - the interval size}$$

Based on the definition of the FM index:

$$I^l(aP) = C(a) + O(a, I^l(P) - 1) \quad (2.2)$$
$$I^u(aP) = C(a) + O(a, I^u(P)) - 1 \quad (2.3)$$

and $I^l(aP) \leq I^u(aP)$ if and only if aP is a substring of T.

A string that terminates with a \$ is called a *text*. A text may have multiple sentinels and every sentinel has a different lexicographical rank i.e. given a text T if there exist $T[i] = \$$ and $T[j] = \$$, then $T[i] < T[j]$ iff $i < j$. Given an ordered set of texts their ordered string concatenation is called a *collection*. Given a set of DNA texts $R_0, \ldots, R_n$, let $T = R_0\overline{R_0}R_1\overline{R_1}, \ldots, R_{n-1}\overline{R_{n-1}}$ be a bidirectional collection of R. The FM index of T is called the FMD index, and the *bi-interval* of P is $[l^l(P), l^l(\overline{P}), l^s(P)]$. Using the fact that we can compute the suffix array interval of aP via Equation 2.2, and that $I^s(\overline{cP}) = I^s(cP)$ the full bi-interval of aP can be derived. This can be used, as in Algorithms 2 and 1 to bi-directionally extend a substring match P and its complement $\overline{P}$ in either direction.

Algorithm 1: Backward extension[106]

Input: Bi-interval $[k, l, s]$ of string W and a symbol a
Output: Bi-interval of string aW

Function BackwardExt($[k, l, s], a$) **begin**
  for $b \leftarrow 0$ **to** 5 **do**
    $k_b \leftarrow C(b) + O(b, k-1)$ $s_b \leftarrow O(b, k+s-1) - O(b, k-1)$
  $l_0 \leftarrow l$
  $l_4 \leftarrow l_0 + s_0$
  for $b \leftarrow 3$ **to** 1 **do**
    $l_b \leftarrow l_{b+1} + s_{b+1}$
  $l_5 \leftarrow l_1 + s_1$
  return $[k_a, l_a, s_a]$

Algorithm 2: Forward extension[106]

Input: Bi-interval $[k, l, s]$ of string W and a symbol a
Output: Bi-interval of string Wa

Function ForwardExt($[k, l, s], a$) **begin**
  $[l', k', s'] \leftarrow$BackwardExt($[l, k, s], \overline{a}$)
  return $[k', l', s']$

A MEM (*maximal exact match* is an exact pattern match to the index that cannot be extended in either direction, and a SMEM (*supermaximal exact match*) is a MEM that is not contained in other MEMs of the pattern. Building on match extension, Algorithm 3[106] provides a mechanism for finding SMEMs of a read in the reference sequence. BWA-MEM generates SMEMs that cover each position in a read, it then performs greedy SMEM filtering and chaining, linking together SMEMs that are located close to each other and filtering out chains that are contained in other longer chains. The list of seeds generated through this process is ranked by the length of its chain and the length of the seed itself.

Algorithm 3: Finding super-maximal exact matches (SMEMs)[106]

Input: String P and start position i_0; $P[-1] = 0$
Output: Set of bi-intervals of SMEMs overlapping i_0

Function SUPERMEM1(P, x) **begin**
  Initialize Curr, Prev *and* Match *as empty arrays*
  $[k, l, s] \leftarrow [C(P[i_0]), C(\overline{P[i_0]}), C(P[i_0]+1) - C(P[i_0])]$
  for $i \leftarrow i_0 + 1$ **to** $|P|$ **do**
    if $i = |P|$ **then**
      Append $[k, l, s]$ *to* Curr
    else
      $[k', l', s'] \leftarrow$FORWARDEXT($[k, l, s], P[i]$)
      if $s' \neq s$ **then**
        Append $[k, l, s]$ *to* Curr
      if $s' = 0$ **then**
        break
      $[k, l, s] \leftarrow [k', l', s']$
  Swap array Curr *and* Prev
  $i' \leftarrow |P|$
  for $i \leftarrow i_0 - 1$ **to** -1 **do**
    Reset Curr *to empty*
    $s'' \leftarrow -1$
    for $[k, l, s]$ **in** Prev **do**
      $[k', l', s'] \leftarrow$BACKWARDEXT($[k, l, s], P[i]$)
      if $s' = 0$ **or** $i = -1$ **then**
        if Curr *is empty* **and** $i + 1 < i' + 1$ **then**
          $i' \leftarrow i$
          Append $[k, l, s]$ *to* Match
      if $s' \neq 0$ **and** $s' \neq s''$ **then**
        $s'' \leftarrow s'$
        Append $[k, l, s]$ *to* Curr
    if Curr *is empty* **then**
      break
    Swap Curr *and* Prev
  return Match

The seeds are extended using a banded affine gap-penalty dynamic programming alignment implementation. A number of heuristics is deployed to limit the search space. For paired-end mapping, BWA-MEM performs Smith-Waterman alignment in a window of $[\mu - 4\sigma, \mu + 4\sigma]$ from a mapped read, when its mate in unmapped. When selecting how to match paired alignments BWA-MEM uses a score of $S_{i,j} = S_i + S_j - \min\{-a \log_4 P(d_{i,j}, U\}$, where S_i and S_j are individual read alignment scores, $d_{i,j}$ is the insert distance between two reads, $P(d_{i,j})$ is the probability of observing $d_{i,j}$ under a normal model of insert sizes, and U is a sensitivity threshold. Figure 2.16 shows a comparison performed by BWA-MEM author between BWA-MEM and other popular aligners. BWA-MEM is seen to outperform all aligners except the commercial tool Novoalign on accuracy and be in the top two tools for speed.

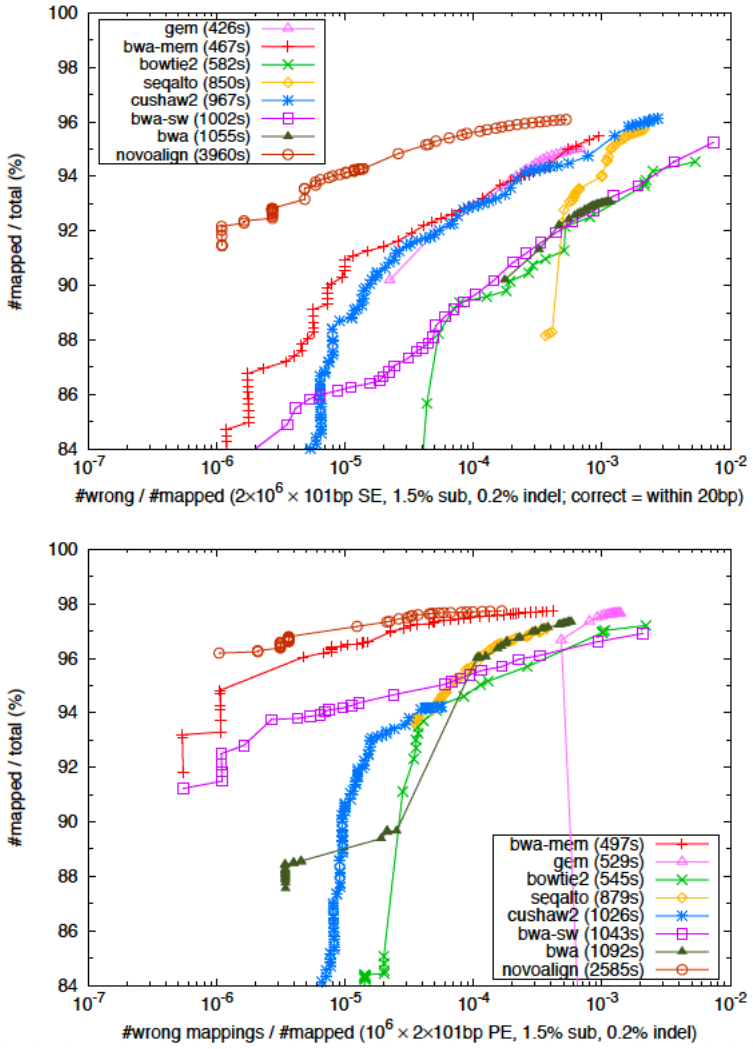


Figure 2.16: P
erformance of multiple aligners on simulated single-end and paired-end 101 bp reads.[104]

Minimap2 Bowtie 2 and BWA-MEM that have been presented so far represent state-of-the-art algorithms for the alignment of short (<500 bp) reads to a reference genome using an FM-index. There are, however, several legitimate cases for the alignment of much longer reads to one or several references, and for finding overlaps between groups of long reads. New DNA sequencing technologies, including Single Molecule Real-Time (SMRT)[155] from Pacific Biosciences Inc., and Oxford Nanopore Technologes' (ONT)[119], have been producing reads that range from 1000 bp to $> 10^6$ bp albeit with a much higher error rate than those from Illumina short reads. These long reads can be used to resolve structural variation encountered in repeat-enriched areas of the genome that are not possible to uniquely resolve with shorter Illumina-based reads[159]. Additionally, tools such as the GATK HaplotypeCaller (see Section 2.2.4) create locally assembled haplotypes (that can be thousands of basepairs long) and align these to the reference sequence to detect variants. Yet, tools that have been developed primarily for short read alignment (like BWA-MEM) either crash or perform exceedingly slow when aligning long read

sequences. Minimap 2[108] has been created to solve the efficient alignment problem for long reads using hashmap-based approach. All of the formulas in this section have been reproduced from the 2018 paper by Heng Li[108].

Minimap2 uses a hashmap approach based on building a database of representative k-mers, called minimizers, for a reference sequnce, and searching against this database. Minmizers have been introduced by Roberts et al.[157] as a way of reducing the storage required for comparing biological sequences. The intuitive notion is that if one is comparing two strings with a significant overlap, extracting and comparing a set of representative k-mers from both strings will produce a match between the k-mers that can then be used as the basis for a seed-and-extend algorithm for more accurate matching. In this case, rather than storing all k-mers, one may store only the minimizers and use them when searching for seeds.

Given a string $T_i = a_1a_2...a_n$ where $a_i \in \Sigma = \{A, C, G, T, N\}$, a k-mer triple is a tuple (s, i, p) that stores the k-mer string s, index i that identifies T_i and p the start index of s in T_i. Assume there exists an order on elements of Σ. If we consider w consecutive k-mers, covering $w+k-1$ letters of T_i, then the smallest k-mer is called a (w, k) minimizer of T_i (See Figure 2.17). If two strings have a substring of length $w + k - 1$ in common, then they have a (w, k) minimizer in common.

Position	1	2	3	4	5	6	7	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	2	3	1	0	3	4	3	4	2	6	4	7	2	8	1	4	7	5	1
k-mers	2	3	1					4	2	6	4	7	2	8					
with		3	1	0					**2**	**6**	**4**	**7**	**2**	**8**	**1**				
minimizer			1	0	3					6	4	7	2	8	1	4			
in				**0**	**3**	**4**					4	7	2	8	1	4	7		
bold					3	4	3					7	2	8	1	4	7	5	
	(a)							(b)					2	8	1	4	7	5	1

Figure 2.17: E
xamples of minimizers - a) (5,3)-minimizer, b) (6,7)-minimizer.[157]

Minimap2 builds a list of of minimizers of the reference sequence and stores them in a hash table where the key is the hash of the minimizer string and the value is the list of offsets into the reference where that minimizer is found. For each query sequence that needs to be aligned, minimap2 builds minimizers of the query and searches the reference hash table for them. The matches that are found become alignment anchors and sets of anchors that are in the same order and orientation in the query and the referece are joined into chains. The gaps between anchors in a chain are then filled using dynamic programming with a 2-piece affine gap penalty to produce a final base-level alignment.

On simulated long reads (Figure 2.7 a)), minimap2 is more accurate than other aligners and is up to 30 times faster. On real SMRT reads minimap2 was 70 times faster than short-read aligners. On short reads (Figure 2.7 b)), minimap2 was less accurate than BWA-MEM but was up to 3 times faster. Using data from a synthetic diploid cell-line[116] minimap2 shows a higher false negative rate compared to BWA-MEM (FNR 2.6% vs. 2.3%), but a lower false positives per million bases (FPPM,

7.0 vs. 8.8) for SNPs, and similar performance for indels.

2.2.3 Raw Data QC

The data that is generated as part of NGS experiments can have widely varying quality, and may suffer from systematic biases introduced by the experimental protocol employed, the technology used, as well as individual events such as sample contamination[98, 7, 44, 23]. The effect of low quality data such as sequencing errors on downstream analysis can be quite significant, not only because it may introduce false positive variants in the final output, but also because it can produce a large number of potentially variant sites that need to be evaluated and will consume a large amount additional computational resources to eventually rule out. A wide variety of computational methods exist for examining the data after it has been generated in order to identify and fix or filter out low quality data. The most widely used tools include Picard[151], FastQC[11], QC Toolkit[148], QC-Chain[210], and FASTX-Toolkit[67]. These tools evaluate data at two granularities - read-level, and sample-level. Some of these can act on sequencing data pre-alignment, while others either work post-alignment or actively make use of a built-in aligner. The QC process is generally organized into a QC pipeline (see Figure 2.18). Furthermore, some tools, like FastQC and Picard tend to only collect various QC metrics and generate reports, leaving the filtering based on these metrics up to the user, while other tools, such as QC Toolkit and QC-Chain perform the actual filtering themselves.

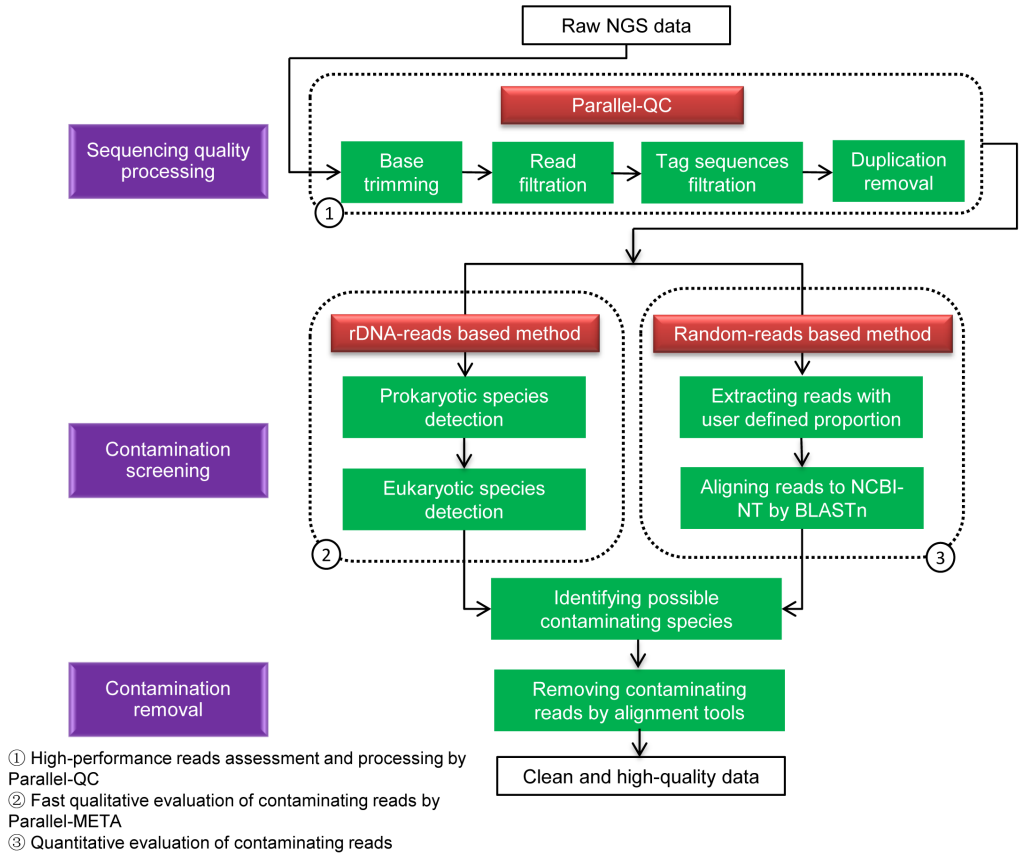


Figure 2.18: The QC-Chain pipeline involves trimming low quality bases, removal of adapter sequences, removal of duplicate reads, and contamination detection.[210].

At individual read level the following QC measures are of interest:

Base Quality Distribution - Reads in FASTQ format have a base-quality score associated with each nucleotide which serve as estimates of the probability that the base has been called correctly by the base-caller software. Bases towards the end of Illumina reads tend to suffer from deteriorating quality([44] and may not be usable for downstream analysis.

Adapter Sequence Presence - The NGS library preparation protocol involves ligating an adapter sequence to the end of DNA fragments in order to bind the fragment to the sequencing flowcell. Sometimes the sequencing process does not stop at the end of the actual DNA molecule and also sequences the adapter. It is necessary to detect and trim out these adapter sequences as they do not belong to the genome[16].

Duplicate Detection - in NGS libraries that use PCR amplification a particular DNA fragment may be sequenced multiple times resulting in increased apparent, and depleted actual coverage of that region of the genome, possibly influencing downstream variant calling efforts. Both Picard[151] and Samtools[103] have utilities for detecting PCR duplicates, although the additional benefit of this processing step may be marginal[46].

Sample Contamination/Sample Swap - Sample contamination during library preparation may result in the presence of foreign DNA in the generated sequence. The foreign sequence may originate from the same or from foreign species. Furthermore, entire samples may be swapped or mislabelled (for instance, cancer samples with tumour from one patient, but normal sample from another patient). Contamination may be detected by aligning reads to a panel of potentially contaminating species' reference genomes[210], or when a surprising number of variants is found in a given genome.

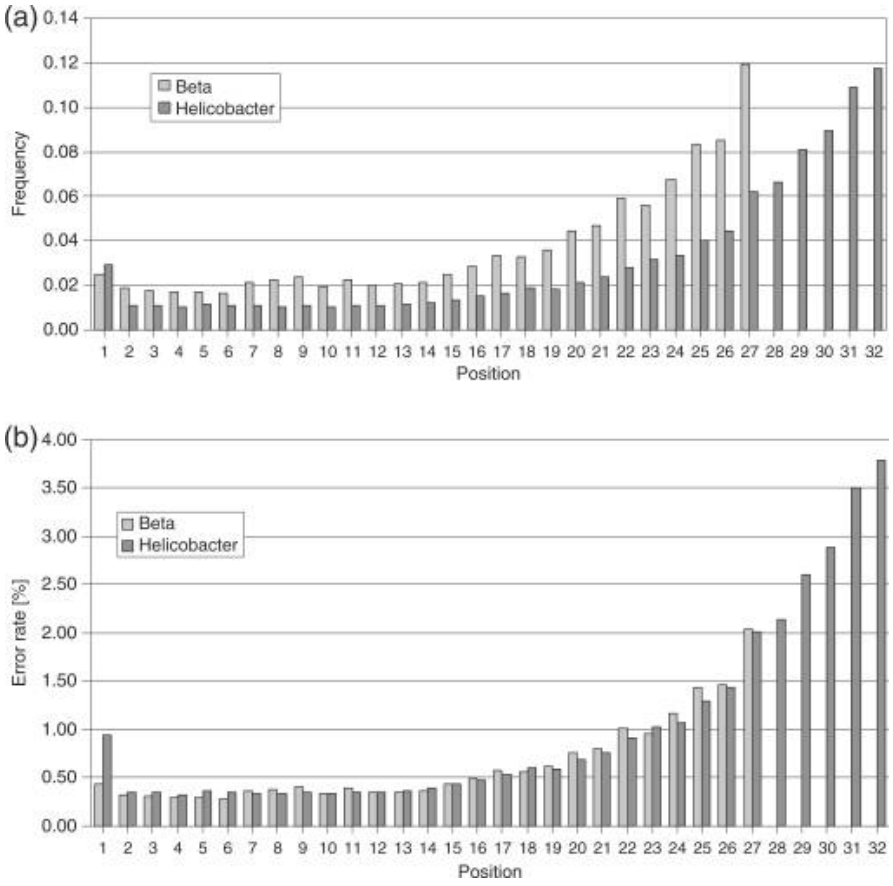


Figure 2.19: Frequency of wrong base calls in Illumina reads based on position in the read from 5' to 3'.[44].

At the sample level the following QC measures are important:

Insert Distribution - summary statistics (mean, median, standard deviation) related to the distribution of the distance between paired-end reads.

Per-base Quality Distribution - distribution of base qualities for each position in a read, aggregated over all reads.

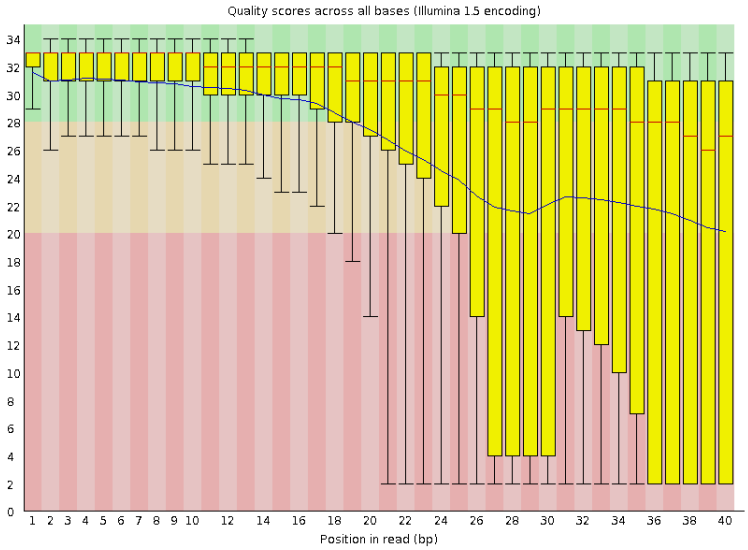


Figure 2.20: from https://www.bioinformatics.babraham.ac.uk/projects/fastqc.

Per-read Quality Distribution - distribution of average read qualities.

Per-base Sequence Content Distribution - distribution of nucleotide frequencies for each position in a read, aggregated over all reads.

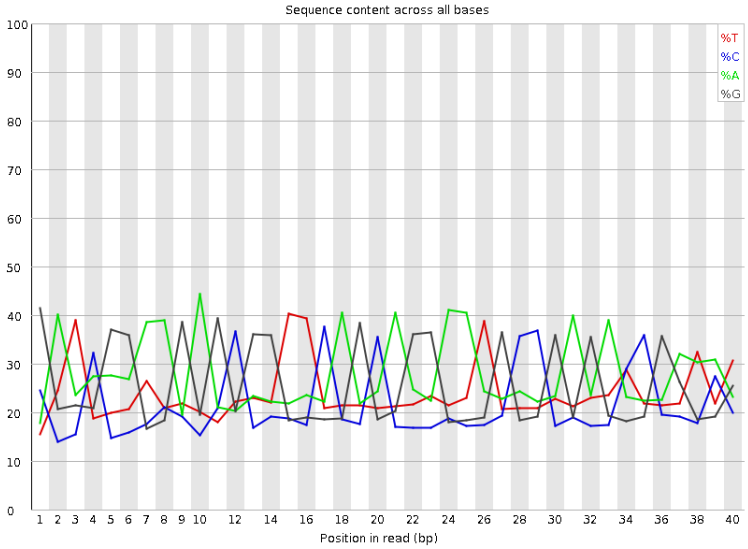


Figure 2.21: from https://www.bioinformatics.babraham.ac.uk/projects/fastqc.

Per-read GC Content Distribution - distribution of average GC content per read.

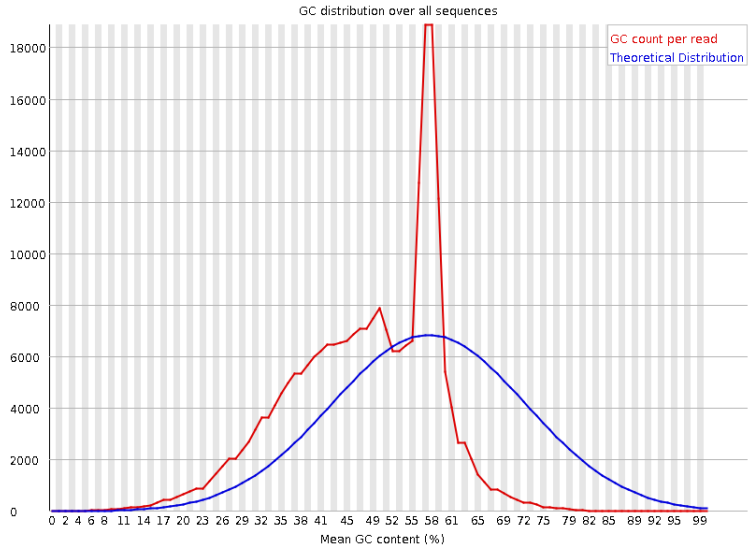


Figure 2.22: from https://www.bioinformatics.babraham.ac.uk/projects/fastqc.

Read-length Distribution - summary statistics of read lengths.

Per-base N Content - distribution of uncalled bases for each position in a read, aggregated over all reads.

Sequence Duplication Distribution - distribution of counts for duplicated sequences.

Overrepresented Sequence Distribution - frequency of sequence fragments that occur more frequently that expected.

Per-base Adapter Content Distribution - frequency of adapter sequence presence for each position in a read, aggregated over all reads.

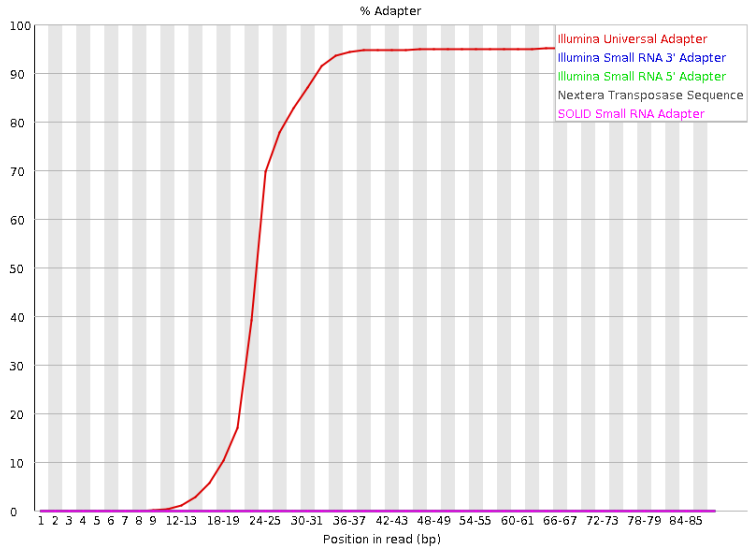


Figure 2.23: from https://www.bioinformatics.babraham.ac.uk/projects/fastqc.

Many other QC metrics can be of interest and are listed on the individual tools' pages (for instance https://broadinstitute.github.io/picard/picard-metric-definitions.html). Once collected, these metrics can be used for visualization, manual curation, threshold-based filtering, or machine learning approaches to QC.

2.2.4 Germline SNP Calling

Single Nucleotide Polymorphisms or SNPs are locations in an individual's genome where that individual differs from the reference sequence at a single position. The reference sequence is haploid i.e. it provides a single base (for instance T) at every genomic location, whereas the human genome is diploid (there are two copies of each chromosome, and thus two bases at each location) for chromosomes 1-22, and chromosome X for females, while being haploid for chromosomes X and Y for males. Thus, at each genomic location, the human genome may be:

Homozygous Reference - When both alleles carried by the individual at that location match the reference.

Heterozygous - When one allele matches the reference and one is different from the reference.

Homozygous Alternate - When both alleles are the same and different from the reference.

Multiallelic - When both alleles are different from the reference and are different from each other[79].

SNPs are the most common type of genomic variant, with every individual carrying over 3 million SNPs on average[171]. Furthermore, the presence of certain SNPs is strongly associated with disease[31], where some SNPs are known to be causative[84], while others, are merely associated with a disease phenotype[165]. A large number of scientific studies[78] and clinical practice[203] is thus enabled by efficient and comprehensive characterization of the gamut of human SNPs to assess their contribution to disease risk, see Figure 2.24.

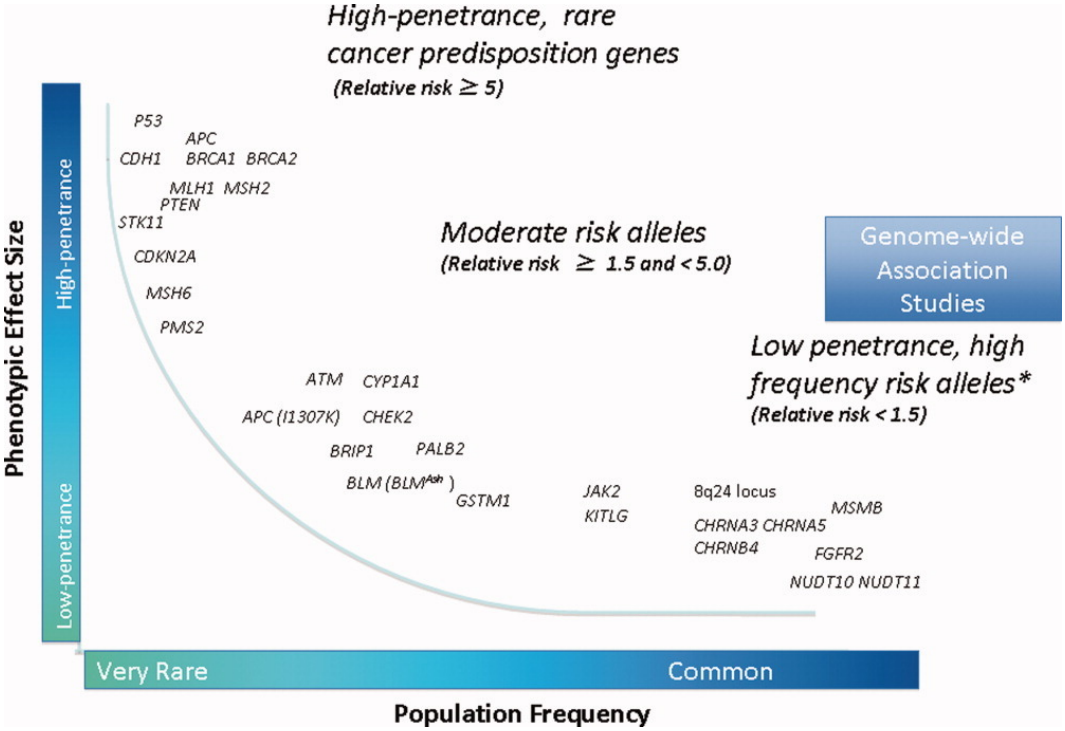


Figure 2.24: Distribution of mutations by population frequency against phenotypic effect size.[196].

There are a number of methods that have been used for assessing SNPs with the aid of microarry technology[76], but here we focus on methods that make use of Next Generation Sequencing (NGS). Since the primary data type generated by NGS is a sequencing read, most presently used methods for SNP detection rely on investigating the collection of sequencing reads that overlap each genomic locus and comparing the observed data to the reference sequence. It is important to distinguish two typically separate activities that take place as part of SNP calling - variant calling, and genotyping. Variant calling attempts to locate positions in the sample genome where that sample is different from the reference, whereas genotyping attempts to assign an actual genotype (e.g. homozygous-alternate), along with a measure of confidence, to each putative variant. We present several of the key computational methods currently used in SNP calling with additional detail. These are:

- samtools
- freebayes
- GATK
- platypus

These tools have been selected because they have been developed independently, at different institutions, and have been repeatedly demonstrated to produce consistent and high-quality results. See Figure 2.25 for a recent comparison.

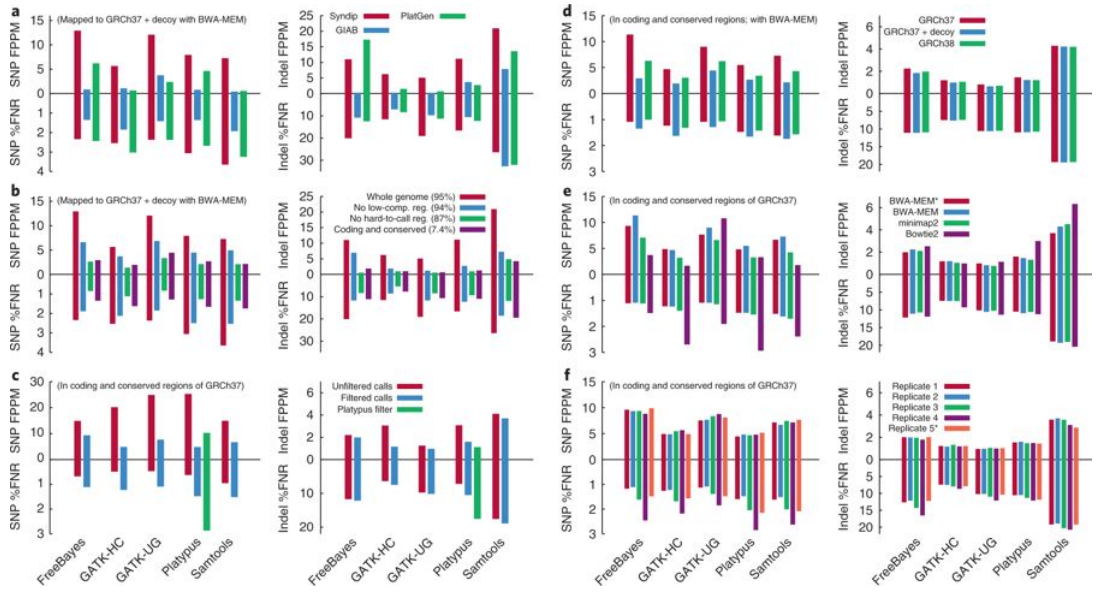


Figure 2.25: Comparison of samtools,freebayes, GATK, and platypus on three benchmark data sets - Syndip, GIAB, and PlatGen. Here FPPM - number of false positives per megabase of sequence, and FNR - false negative rate = 100xFN/(TP+FN).[116].

samtools

Samtools[117] is a software package for genomic data processing developed by Heng Li et al. in the context of the 1000 Genomes Project[28] and implemented as a C program with a CLI interface. This tool has enjoyed continued and widespread use in the bioinformatics community for the purposes of small variant calling (including SNPs). All of the mathematical results in this section are reproduced from the 2011 paper by Li[103] that describes the method, as well as a set of mathematical notes made available separately by Li[107] in 2010.

Although the samtools framework could be extended to support calling multi-allelic sites, the framework, as-published, has been developed for calling only bi-allelic variants. Table 2.7 contains commonly used definitions.

It is additionally assumed that there are n individuals being sequenced with the i-th individual having ploidy m_i (typically 2 in practice). At a particular genomic locus, the sequence read data for the i-th individual is d_i and the genotype is g_i, an integer in $[0, m_i]$, counting the number of reference alleles in the individual at that locus. Furthermore, it is assumed for simplicity that data at individual genomic loci are independent (which isn't necessarily true), as are sequencing and mapping errors between loci and individuals.

Because of the above independence assumptions the joint likelihood function of the data observed for all individuals factors as a product of individual likelihood

Table 2.7: Samtools common definitions

Symbol	Description
n	Number of samples
m_i	Ploidy of the i-th sample ($1 \leq i \leq n$)
M	Total number of chromosomes in samples: $M = \sum_i m_i$
d_i	Sequencing data (bases and qualities) for the i-th sample
g_i	Genotype (the number of reference alleles) of the i-th sample ($0 \leq g_i \leq m_i$)[1]
ϕ_k	Probability of observing k reference alleles ($\sum_{k=0}^{M} \phi_k = 1$)
$P(A)$	Probability of an event A
$\mathcal{L}_i(\theta)$	Likelihood function for the i-th sample: $\mathcal{L}_i(\theta) = P(d_i\|\theta)$

functions:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} \mathcal{L}_i(\theta) \tag{2.4}$$

Suppose that a single sample i represents an individual of ploidy m_i and a given locus is covered by k reads. The sequencing data d_i is composed of an array of bases where each element has value 1 representing the reference allele and is 0 otherwise.

$$d_i = (b_1, \ldots, b_k) = (\underbrace{1, \ldots, 1}_{l}, \underbrace{0, \ldots, 0}_{k-l})$$

The error probability of the j-th base is ϵ_j, which is taken to be the larger between sequencing and mapping errors for that read. Under the independence assumptions above:

$$\mathcal{L}_i(0) = P(d_i|0) = \prod_{j=1}^{l} \epsilon_j \prod_{j=l+1}^{k} (1 - \epsilon_j) = \left(1 - \sum_{j=l+1}^{k} \epsilon_j + o(\epsilon^2)\right) \prod_{j=1}^{l} \epsilon_j \tag{2.5}$$

$$\mathcal{L}_i(m_i) = P(d_i|m_i) = \left(1 - \sum_{j=1}^{l} \epsilon_j + o(\epsilon^2)\right) \prod_{j=l+1}^{k} \epsilon_j \tag{2.6}$$

For $0 < g_i < m_i$:

$$\begin{aligned}
\mathcal{L}_i(g_i) = P(d_i|g_i) &= \sum_{a_1=0}^{1} \cdots \sum_{a_k=0}^{1} \Pr\{d_i|B_1 = a_1, \ldots, B_k = a_k\} \Pr\{B_1 = a_1, \ldots, B_k = a_k|g\} \\
&= \sum_{\vec{a}} \left(\frac{g}{m}\right)^{\sum_j a_j} \left(1 - \frac{g}{m}\right)^{k - \sum_j a_j} \cdot \prod_j p_j(a_j) \\
&= \left(1 - \frac{g}{m}\right)^k \prod_j \sum_{a=0}^{1} p_j(a) \left(\frac{g}{m-g}\right)^a \\
&= \left(1 - \frac{g}{m}\right)^k \prod_{j=1}^{l} \left(\epsilon_j + \frac{g}{m-g}(1-\epsilon_j)\right) \prod_{j=l+1}^{k} \left(1 - \epsilon_j + \frac{\epsilon_j g}{m-g}\right) \\
&= \left(1 - \frac{g}{m}\right)^k \left\{ \left(\frac{g}{m-g}\right)^l + \left(1 - \frac{g}{m-g}\right) \left(\sum_{j=1}^{l} \epsilon_j - \sum_{j=l+1}^{k} \epsilon_j \right) + o(\epsilon^2) \right\}
\end{aligned}$$

where $a = \sum_j a_j$ and

$$p_j(a) = \begin{cases} \epsilon_j & a = 1 \\ 1 - \epsilon_j & a = 0 \end{cases}$$

In particular, for a diploid sample ($m = 2$), the likelihoods for $g = 0, 1, 2$ are

$$\mathcal{L}(0) = \prod_{j=1}^{l} \epsilon_j \prod_{j=l+1}^{k} (1 - \epsilon_j) \tag{2.7}$$

$$\mathcal{L}(1) = \frac{1}{2^k} \tag{2.8}$$

$$\mathcal{L}(2) = \prod_{j=1}^{l} (1 - \epsilon_j) \prod_{j=l+1}^{k} \epsilon_j \tag{2.9}$$

For instance, taking $g_i = 2$ (i.e. the true genotype is homozygous-reference) as an example, and under above independence assumptions, the likelihood of observing the data d_i is the likelihood of sampling l reads without error (the reads match the reference) and $k - l$ reads with error (the reads do not match the reference).

Let $\Phi = \{\phi_k\}$ for $k \in [0, m_i]$ be a prior distribution of genotype probabilities (a model from population genetics, such as Wright-Fisher, can be used, or an empirical distribution from antoher study), the actual genotype for individual i at the given locus is estimated via Bayes' Rule as:

$$\hat{g}_i = \operatorname*{argmax}_{g_i} \Pr\{G_i = g_i|d_i, \Phi\} = \operatorname*{argmax}_{g_i} \frac{P(d_i|g_i)\phi_k}{\sum_{h_i} P(d_i|h_i)\phi_h} = \operatorname*{argmax}_{g_i} \frac{\mathcal{L}(g_i)\phi_k}{\sum_{h_i} \mathcal{L}(h_i)\phi_h} \tag{2.10}$$

The variant quality is defined as:

$$Q_{\text{var}} = -10 \log_{10} \Pr\{G = m_i|d_i, \Phi\} \tag{2.11}$$

i.e. the Phred-scaled probability that the locus is homozygous reference given the observed data. The locus is called *variant* if the variant quality exceeds a certain pre-defined threshold.

Equations 2.10 and 2.11 thus represent the key computations corresponding to the activities of SNP genotyping and variant calling that are performed by samtools for each genomic locus in a sample.

freebayes

freebayes[64] is a C software package implemented by E. Garrison for discovery and genotyping of SNPs, indels, and other small variants that builds on a previous method by G. Marth[128]. Where small means that the variant length is smaller than the size of a sequencing read. Unlike samtools, which looks at all genomic loci independently, freebayes uses local sequence context to guide detection and genotyping of variants, additionally freebayes builds variant haplotypes[37] i.e. groups of variants that are inherited together on the same DNA molecule. All of the figures and methematical formuls in this section are reproduced from the 2012 preprint by Garrison et al[64].

Table 2.8: Freebayes common definitions

Symbol	Description
n	Number of samples
m_i	Copy number of the i-th sample ($1 \leq i \leq n$)
M	Total number of copies of each locus in all samples: $M = \sum_i m_i$
$\{b_j : j \in [1, K]\}$	Set of K distinct alleles present among M copies
$\{C_j : j \in [1, K]\}$	Corresponding set of allele counts for each b_K
$\{f_j : j \in [1, K]\}$	Corresponding set of allele frequencies for each b_K
G_i	Unphased genotype of individual i
$\{b_{i_j} : i \in [1, n], j \in [1, k_i]\}$	Set of k_i distinct alleles of G_i
$\{c_{i_j} : i \in [1, n], j \in [1, k_i]\}$	Set of k_i allele counts for each b_{i_j}
$\{f_{i_j} : i \in [1, n], j \in [1, k_i], f_i = c_i/k_i\}$	Set of allele frequencies of G_i
$B_i : \lvert B_i \rvert = m_i$	Multiset of alleles equivalent to G_i
$R_i = \{r_j : j \in [1, s_i]\}$	Set of s_i sequencing observations for each sample i s.t. there are $\sum_{i=1}^{n} \lvert R_i \rvert$ reads at a given locus
$\{q_j : j \in [1, s_i]\}$	Mapping quality, i.e. probability that read r_j is mis-mapped against the reference

For a set of n individuals the conditional probability of a combination of genotypes

given the observed data is assessed simultaneously as:

$$P(G_1, \ldots, G_n | R_1, \ldots, R_n) = \frac{P(G_1, \ldots, G_n) P(R_1, \ldots, R_n | G_1, \ldots, G_n)}{P(R_1, \ldots, R_n)} \tag{2.12}$$

$$P(G_1, \ldots, G_n | R_1, \ldots, R_n) = \frac{P(G_1, \ldots, G_n) \prod_{i=1}^{n} P(R_i | G_i)}{\sum_{\forall G_1, \ldots, G_n} P(G_1, \ldots, G_n) \prod_{i=1}^{n} P(R_i | G_i)} \tag{2.13}$$

For a single sample at a particular locus there are R_i reads, and k_i observed alleles - $B'_i = b'_1, \ldots, b'_{k_i}$, which correspond to $b_1, \ldots, b_i$ underlying alleles represented at the given locus. For each observed allele b'_i there is a corresponding count of observations o_f s.t. : $\sum_{j=1}^{k_i} o_j = s_i$ and each b'_i corresponds to a true allele b_i. The probability of a single observation b'_i given a genotype in a single sample is:

$$P(b'_i | G) = \sum_{\forall (b_i \in G)} f_i P(b'_i | b_i) \tag{2.14}$$

where f_i is the allele frequency of b_i in G. The authors introduce the following approximation:

$$P(b' | b) = \begin{cases} 1 & \text{if } b' = b \\ P(error) & \text{if } b' \neq b \end{cases} \tag{2.15}$$

where $P(error)$ is derived from the base quality score from the sequencing read. Using the above approximation the probability of observing the data R_i given the genotype is estimated as:

$$P(R_i | G) \approx \binom{s_i}{o_1, \ldots, o_{k_i}} \prod_{j=1}^{k_i} f_{i_j}^{o_j} \prod_{l=1}^{s_i} P(b'_l | b_l) \tag{2.16}$$

In order to evaluate the posterior probability of a particular combination of genotypes given the data, the authors derive:

$$P(G_1, \ldots, G_n | f_1, \ldots, f_k) = \binom{M}{c_1, \ldots, c_k}^{-1} \prod_{i=1}^{n} \binom{m_i}{c_{i_1}, \ldots, c_{i_{k_i}}} \tag{2.17}$$

counting the number of ways of selecting a set of unphased genotypes G_i given the allele frequency spectrum f_k, and

$$P(f_1, \ldots, f_k) = P(a_1, \ldots, a_M) = \frac{M!}{\theta \prod_{z=1}^{M-1} (\theta + z)} \prod_{j=1}^{M} \frac{\theta^{a_j}}{j^{a_j} a_j!} \tag{2.18}$$

using Ewens' sampling formula[53], where θ is the population mutation rate and allele frequencies f_i are transformed to frequency counts $a_1, \ldots, a_M : \sum_{c=1}^{M} ca_c = M$ where each a_f is the number of alleles in $b_1, \ldots, b_k$ whose allele count in the sample set is c.

Once variants are initially detected using Equation 2.13, the method continues to build local haplotypes grouping variants that are inherited together in dynamically-sized windows based on a distance threshold between successive variants. Each group of variants is combined into a haplotype observation H_i, with an assigned quality score that is the minimum of the supporting reads' mapping quality and the minimum base quality of the variant allele observations (bases that span the variants). The size of the window is determined via an iterative process where an initial variant is used as the seed, and all of the reads that overlap it are added. If these reads overlap any other variants, then these are added, along with any reads that overlap them, and so on, until no new variants can be added.

Once the window is determined, the method uses a gradient descent algorithm to find a MAP estimate of the genotype for each sample. It starts with the maximum likelihood estimate for each sample's genotype given the observed data, and then attempts to find a genotype assignment that has a higher posteriror probability across all samples.

GATK

The GATK[42] is not actually a tool that's built solely for SNP calling, but is instead a comprehensive framework for genomic data analysis that includes tools for data pre-processing and QA, SNP and indel calling, CNV calling, and SV calling, as well as post-processing and filtering. It is implemented as a Java program and the latest version makes use of the in-memory computing engine Apache Spark for efficient computation over large data sets. Here we focus on the SNP calling aspects of the framework. There are two components in the GATK that deal with SNP calling, the UnifiedGenotyper, and the HaplotypeCaller. The UnifiedGenotyper is an older component that has been superceded by the HaplotypeCaller for all practical purposes and we focus our attention on it. All of the mathematical formulas and figures, as well as some of the descriptions, in this section have been reproduced from the 2010 manuscript by McKenna et al.[131], the 2011 manuscript by DePristo et al.[42], and the GATK website[2]

The GATK Best Practices pipeline (see Figure 2.26) is a set of best practices published by The Broad Institute that describe how to best use their software. In the context of this section we are primarily interested in the processes that occur in the middle panel of this figure that deal with processing aligned reads to call and genotype variants that will then be used for post-processing and filtering.

The HaplotypeCaller which is the tool that is primarily used for SNP calling takes a continued refinement approach, where the data is processed in the following sequential steps:

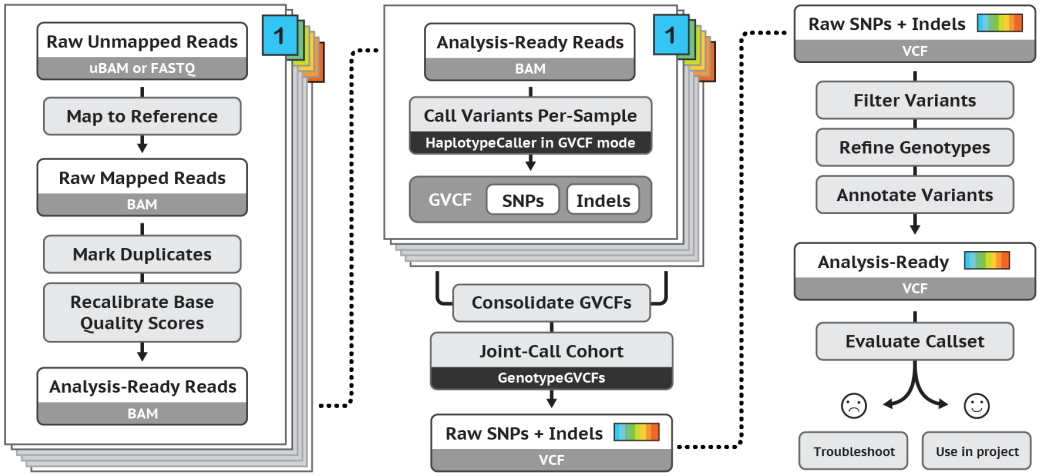


Figure 2.26: GATK Best Practices pipeline[2]

Define active regions - The program determines which regions of the genome it needs to operate on (active regions), based on the presence of evidence for variation.

Determine haplotypes by assembly of the active region - For each active region, the program builds a De Bruijn-like graph to reassemble the active region and identifies what are the possible haplotypes present in the data. The program then realigns each haplotype against the reference haplotype using the Smith-Waterman algorithm in order to identify potentially variant sites.

Determine likelihoods of the haplotypes given the read data - For each active region, the program performs a pairwise alignment of each read against each haplotype using the PairHMM algorithm. This produces a matrix of likelihoods of haplotypes given the read data. These likelihoods are then marginalized to obtain the likelihoods of alleles for each potentially variant site given the read data.

Assign sample genotypes - For each potentially variant site, the program applies Bayes' rule, using the likelihoods of alleles given the read data to calculate the likelihoods of each genotype per sample given the read data observed for that sample. The most likely genotype is then assigned to the sample.

Active regions are defined by targeting loci with high quality reads that are different from the refernce and surrounded by soft-clipped bases (bases in the read that couldn't be aligned by the alignment algorithm). The region is then smoothed by a Gaussian kernel and passed onto variant calling if its profile score exceeds a pre-defined threshold.

For each active region the GATK performs local assembly by first building a de-Bruijn graph[27] using just the reference sequence for the active region. The graph is enhanced by comparing each read from the active region and creating nodes in the graph where the read differs from the graph. Edge weights between pairs of nodes are increased when a read passes through that edge. The resulting graph must

meet complexity requirements based on the ratio of non-unique to unique kmers and presence of cycles (that are a sign of repetitive sequence). When the graph does not meet complexity requirements it is automatically rebuilt with successively increased kmer sizes. If a sufficiently complex graph cannot be built, the region is discarded because the method is not able to produce a high quality variant call in this case. Once the graph is built it is refined by pruning out paths that are not supported by a sufficient number of reads. By traversing the paths in the graph and selecting those with a high aggregate weight the algorithm builds a list of possible local haplotypes within the active region. Each candidate haplotype is then re-aligned to the reference sequence using Smith-Waterman[175] alignment to produce a list of potential variant sites that includes SNPs and other potential variants.

Reads from the active region are aligned to each haplotype using a Hidden Markov Model via the PairHMM algorithm[45] to produce a likelihood of each read given a haplotype. The set of haplotype likelihoods is transformed to a set of likelihoods per allele where for each allele at a given site the the highest scoring likelihood for a given read and a given haplotype that supports that allele is selected. Under a read independence assumption the total likelihood of the allele is computed as the product of read likelihoods.

For each variant locus the probability of each genotype is calculated using:

$$P(G|D) = \frac{P(G)P(D|G)}{\sum_i P(G_i)P(D|G_i)} \tag{2.19}$$

where D is the set of reads, G_i is the set of possible genotypes, and under the assumptions of a diploid sample, independent read observations, and independent errors. The prior probability of genotypes $P(G)$ is assumed to be uniform.

Based on the diploid and independence assumptions:

$$P(D|G) = \prod_j \left(\frac{P(D_j|H_1)}{2} + \frac{P(Dj|H_2)}{2} \right) \tag{2.20}$$

where $P(D_j|H_n)$ is the likelihood of read given the haplotype perviously obtained.

The genotype with the highest $P(G|D)$ is emitted as the genotype for that variant.

Platypus

Platypus[156] is a Python and C package by Rimmer et al., that provides SNP, indel, and other small and medium (upt to 1kb) variant calling capabilities utilizing a Bayesian statistical framework built on top of reference-free local assembly of haplotypes. All of the figures and methematical formuls in this section are reproduced from the 2014 Nature Genetics manuscript by Rimmer et al. See Figure 2.27 for the general sequence of data processing steps in platypus.

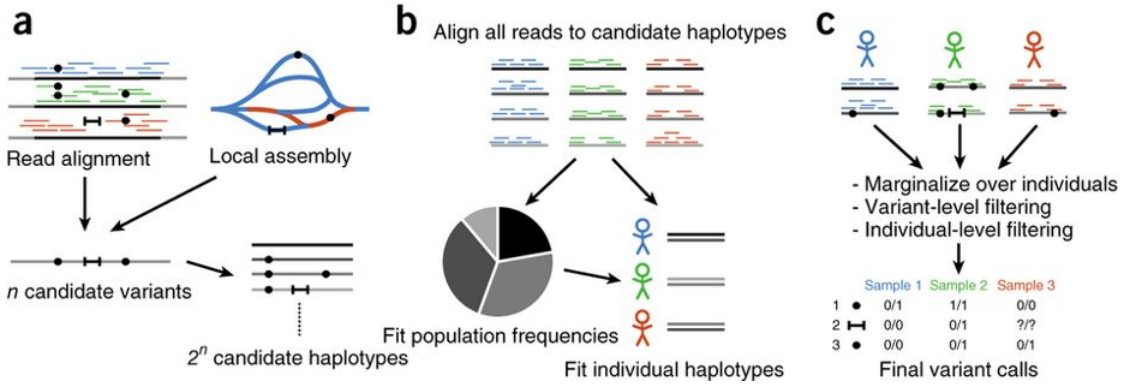


Figure 2.27: Three stages of data processing in Platypus[156]

Platypus works by loading batches of reads into memory from a BAM file. 100 kb (kilobases) are loaded at a time. Low quality reads are filtered out. Sites that are flagged as potential variants by the aligned become candidate variants. The local assembly step operates on successive windows of 1.5 kb in size, including all reads that map to the window as well as their mates, irrespective of where or if they map. As a first step, the algorithm builds a colored de Bruijn graph[85] from the reads and the reference sequence. Candidate variant haplotypes are produced by finding paths that diverge from the reference and come back to the reference by performing a Depth First Search from the graph node that diverges and until the reference is reached again. All such paths are recorded and contain putative SNPs, MNVs, indels, and larger rearrangements that are up to the window-size in length. A prior of 0.33×10^{-3} is used for SNPs under the assumption that SNPs occur in humans at a density of 10^{-3} per base and every non-reference allele is equally likely. Smaller windows are created by combining groups of candidate variants that are within 15 basepairs of each other, as long as window size is smaller than read-length and there are fewer than 8 variants in a group.

Candidate haplotypes are generated by taking all possible combinations of candidate variants (assuming diploid sample), resulting in 2^n haplotypes. For a window with 8 variants there are 256 possible haplotypes. The haplotype likelihood $P(r|h)$ is calculated by aligning reads to each haplotype using a Hidden Markov Model. Platypus uses the Viterbi algorithm[58] to compute the most likely path through the HMM as an approximation of the actual likelihood as a matter of optimization. After haplotype likelihoods are computed for all combinations of reads and haplotypes Platypus uses Expectation Maximization to estimate the frequency of each haplotype under the following model:

$$\mathcal{L}(R|\{h_i, f_i\}_{i=1...a}) = \prod_{\substack{samples,\\ s}} \sum_{\substack{haplotypes,\\ i,j}} f_i f_j \prod_{\substack{reads,\\ r \in R_s}} \left(\frac{1}{2} p(r|h_i) + \frac{1}{2} p(r|h_j) \right) \tag{2.21}$$

where f_i is frequency of haplotype h_i, a is the number of considered alleles, R is all reads, R_s reads in sample s.

The posterior probability of a variant v given the data is computed as:

$$P(v|R) = \frac{P(v)\mathcal{L}(R|\{h_i, f_i\}_{i=1...a})}{P(v)\mathcal{L}(R|\{h_i, f_i\}_{i=1...a}) + (1 - P(v))\mathcal{L}(R|\{h_i, \frac{f_i}{1-F_v}\}_{i \in I_v})} \tag{2.22}$$

here the likelihood of data R given all haplotypes is compared to the likelihood of R given those haplotypes that do not include v. I_v is the set of haplotype indices such that h_i does not contain v, and $F_v = \sum_{i \in I_v} f_i$. A variant is called when its posterior exceeds a pre-defined threshold. Genotype likelihoods for a variant are calculated by marginalizing over the genotypes at other variants within the window, and the best likelihood is selected as the genotype.

Remarks

Although samtools, freebayes, GATK, and Platypus are not the only tools that have been successfully used for germline SNP calling in the past 10 years, they have enjoyed sustained popularity and continued use in large scale genomics projects such as 1000 Genomes Project, ExAC[102], PCAWG[176] and others. They thus can serve as primary examples of how the problem of calling germline SNPs is currently approached within the field, as well as providing a benchmark for new methods that attempt to solve similar problems. There are several key distinctions in the approaches that these tools take to the overall problem, yet a number of key similarities exist that we outline here and adopt as a general framework for tackling the SNP calling problem in our work.

Site Selection The problem of selecting sites in the genome for detecting variants has variety of approaches from evaluating every single site independently as one that potentially harbors a variant (as in samtools) to the varied windowing approaches used by the other variant callers. Looking at sites independently has the benefit of simplicity, while the windowing approach, at the expense of additional computation, allows a more comprehensive evaluation of a genomic region that is not limited to SNPs but can also support the detection of other types of variants. Thus, even though an independent site approach may be acceptable in an initial implementation, some form of windowing is desired in order to benefit from knowledge about the surrounding sequence context. In general, the interestingness of a site for the purposes of detecting a potential variant is universally linked to the presence of high quality reads spanning a particular locus that disagree with the reference sequence at that locus.

Haplotype Construction Samtools has the simplest model here as it does not attempt to construct haplotypes at all. This limits the ability of samtools to accurately represent genomic architecture, and prevents it from being able to supply

phasing information for variants, which is of interest. Freebayes has the next simplest model, where putative haplotypes are constructed directly from the observed read sequences with the observation window. GATK and Platypus actually perform local assembly in the window to come up with an arrangement of reads that is free of artifacts[111] associated with alignment to the reference. Although local assembly appears to improve the ability of these callers to accruately represent sequence variants (especially non-SNPs) this processing step introduces a significant impact on the overall processing cost, mostly incurred from the resource-intensive HMM-aided alignment of reads to the putative haplotypes.

Allele Frequency Spectrum Estimation A distribution of allele frequencies in a population is of interest as it can be used as a prior in the calculation of genotype likelihoods for the samples under analysis and local and population-specific allele frequencies can vary significantly from values implied by generic population-genetics models. Most callers provide capabilities for estimating the Allele Frequency Spectrum by Expectation Maximization or similar approaches (Equation (5) in [103], not covered here, equation 2.18 for freebayes above). Alternatively, a non-informative prior (as in GATK) can be used.

Genotyping Pretty universally across the methods, genotyping is set up as a Bayesian inference selecting a Maximum Likelihood Estimate or a Maximum A Posteriori estimate of the genotype (for a single site or a haplotype) given the reads data, and taking into account the probability of errors dervied from read base quality and mapping quality scores supplied by the aligner. See Eq. 2.10 for samtools', Eq. 2.13 frebayes', Eq. 2.19 GATK's, Eq. 2.21 Platypus' model setups. This model thus remains highly relevant for any new development in the space.

2.2.5 Germline Indel Calling

2.2.6 Germline Structural Variant Calling

Structural Variants (SVs) are medium- to large-size alterations (typically >50bp in length) of the genomic sequence that fall into a several broad categories (see Figure 2.28) including insertions, deletions, tandem and interspersed duplications, inversions, translocations, mobile element insertions, as well as complex rearrangements that constitute a combination of the above classes or are otherwise difficult to classify.

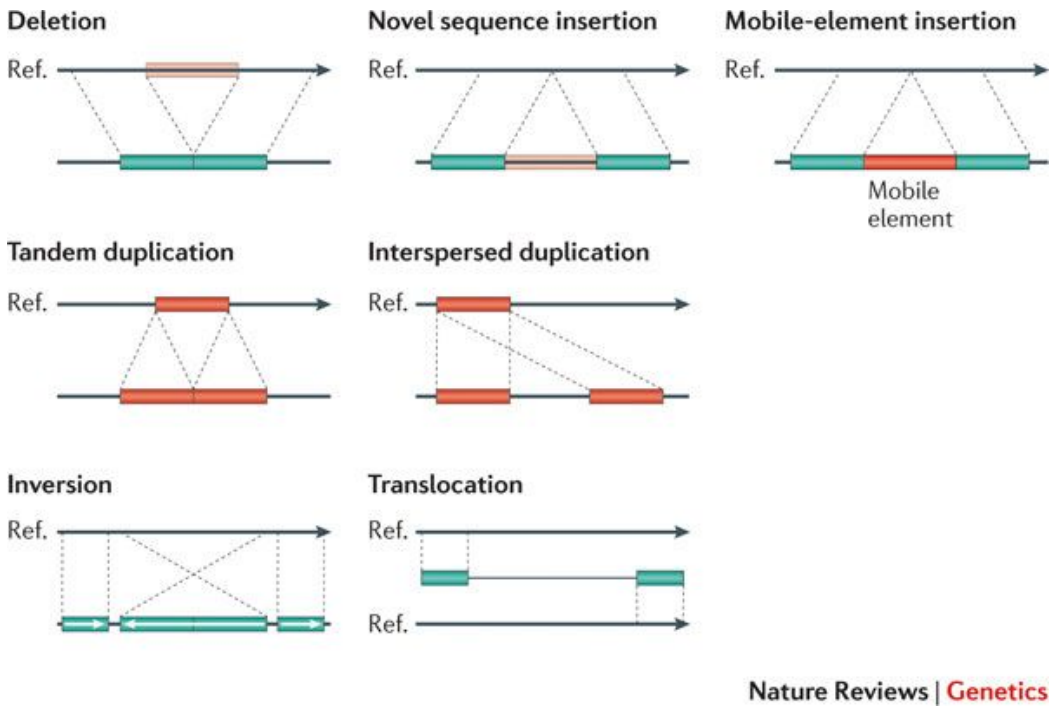


Figure 2.28: Classes of structural variation.[9].

Accurate detection of SVs remains an open challenge since both sensitivity and specificity performance of SV calling is an order of magnitude worse than for SNP calling. For instance, in the latest data release of the 1000 Genomes Project an estimated sensitivity of 88% was achieved for deletions, 65% for duplications, 32% for inversions[180] and False Discovery Rate of up to 89%autocitemills2011mapping. A key challenge that impacts calling accuracy is small read size relative to the size of the variants. Since SVs can be hudreds of kilobases in length and typical Illumina reads are only 150-500 baspeairs long, accurate reconstruction relies on an agglomeration of reads in order to produce the variants. Structural Variant detection approaches typically make use of several sources of evidence (see Figure 2.29) for the detection of "breakpoints" - regions of the genome where a DNA double-strand break is detected and sequence is either inserted or excised. The breakpoints are then interpreted to produce the most likely variants that they represent.

Discordantly-mapped read pairs are pairs of sequencing reads that the alignment algorithm has mapped at a distance that is statistically significantly larger or smaller than the average read-pair distance for that sample, or that have an unusual read orientation, since read-pairs are supposed to be sequencing from two ends of a molecule towards each other. Reads that map closer than they are supposed to are indicative of an insertion, reads that map farther are indicative of a deletion, and reads that map in an unusual orientation are indicative of an inversion.

Split-reads are read-pairs where one of the reads maps properly but the aligner is not able to map the other read because different pieces of the read map to different locations in the genome indicating that this read spans a breakpoint. Distance

between the split read pieces and their mapping orientation are informative of the type of breakpoint that the read spans.

Read-depth Regions have a higher than normal or lower than normal read depth (count of reads spanning a region) are indicative of increased or reduced copy number respectively. Care must be taken to distinguish actual SVs from areas of repetitive sequence where the same simple sequence pattern is repeated many times. Typically aligners are not able to accurately resolve such areas and map many reads to the same set of coordinates making it appear like a duplication.

Assembly-based approaches perform local assembly in a reference-free manner to reconstruct the variants encoded in the sample sequence that the aligner may struggle with resolving because of a large number or increased complexity of rearrangements.

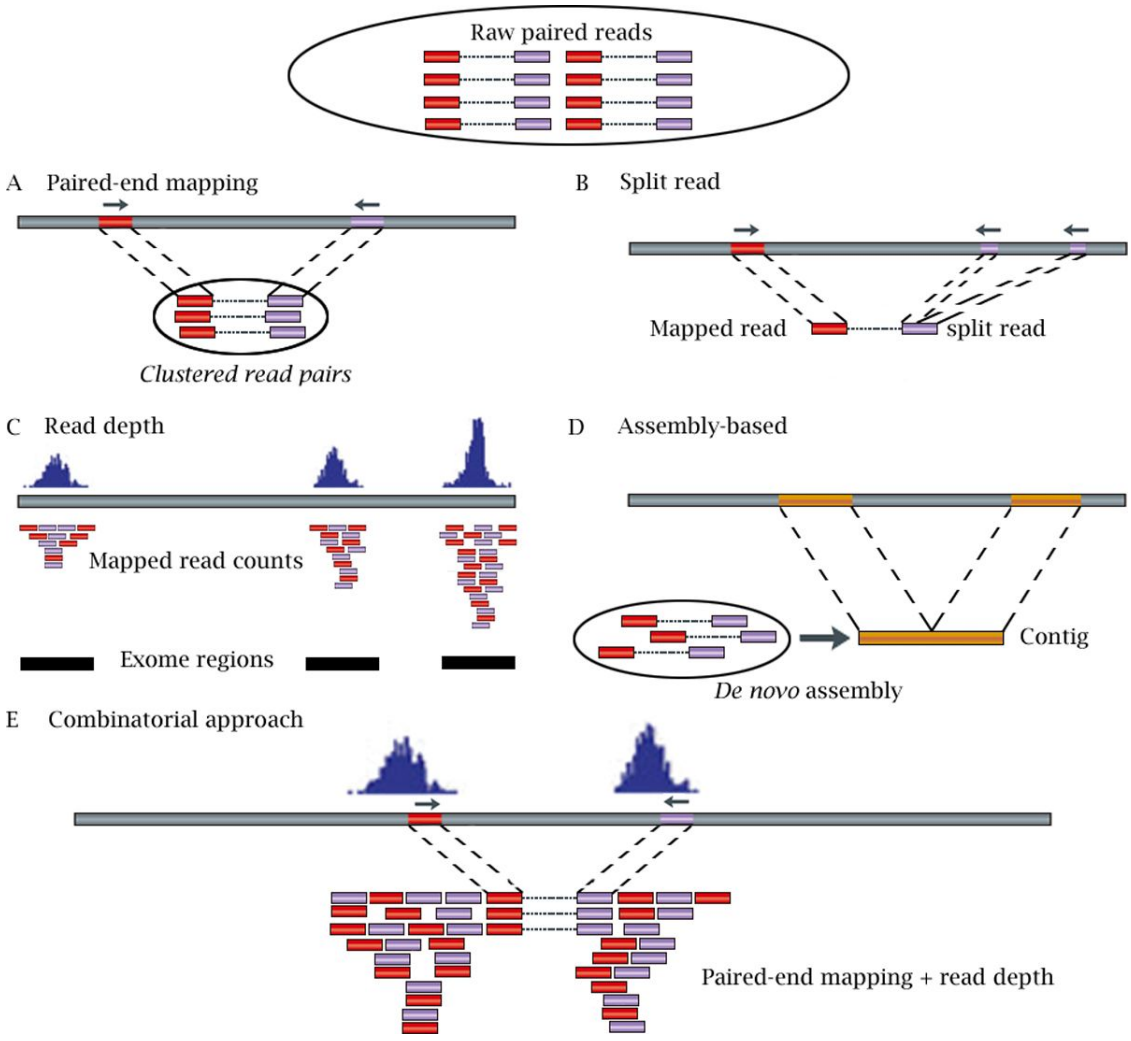


Figure 2.29: Sources of evidence for the presence of Structural Variants.[209].

We look at several SV callers in closer detail.

Delly

Delly[153] is a structural variant calling tool built in 2011 by T. Rausch et al. in the context of the 1000 Genomes Project. Delly uses discordantly-mapped reads and split-reads as sources of evidence for the presence of SVs, is able to characterize a broad set of variants and has been implemented in C++. All of the mathematical formulas and figures in this section are reproduced from the 2012 mauscript by Raush et al.

The discordantly-mapped reads component of Delly begins by computing the median and standard deviation of the read insert size (distance between the ends of the two reads in a pair), as well as the default read orientation by sampling a pre-defined number of reads from the BAM file. Read-pairs that have an insert size farther than 3 standard deviations from the median are considered as evidence for structural variation. This induces a minimum SV size detectable by Delly. See Figure 2.30 for the variant types implied by each insert size deviation and read-pair orientation.

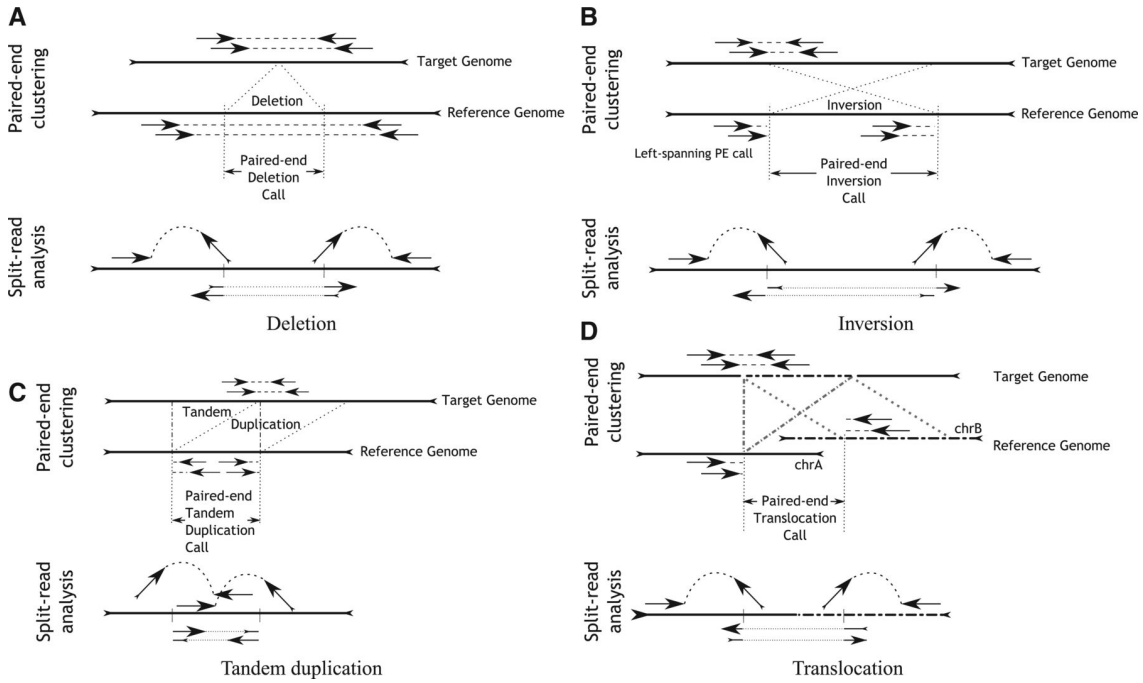


Figure 2.30: Variant classes detectable by Delly based on their read-pair and split-read signatures.[153].

Using the list of discordantly-mapped read-pairs Delly builds an undirected, weighted graph to indicate which read-pairs support the same variant. In the graph $G(V, E)$, a read-pair p_i corresponds to a node $v_i \in V$ and an edge $e_{v_i,v_j} \in E$ connects two nodes that support the same SV. The weight $w(e_{v_i,v_j})$ is the absolute value of the difference between the SV sizes predicted by v_i and v_j. v_i and v_j support the same variant when the corresponding read pairs have the same read orientation and the absolute difference between the left and right ends of the two reads is less than the expected insert size. Variants are identified by computing connected components C_i

of G. When components are not fully connected, Delly sorts the edges in each such component by weight and attempts to find a maximal clique M_i within C_i using edge with the smallest weight as the seed of the clique. The clique is extended by searching for the next smallest edge for which one of the vertices is already in the clique M_i, and requiring that $M_i \cup \{v_l, v_m\}$ is also a clique, until no further vertices can be added. The vertices in M_i are reported as the read pairs supporting that SV. Each rearrangement type is analyzed independently in this manner.

All of the rearrangements identified in the discordantly-mapped read analysis are refined using split-read analysis. The reads used for split-read analysis are those where one read in the pair maps to the reference and the other is unmapped. All such reads within a distance of 2 standard deviations of the median insert size from the breakpoint are considered up to a configurable maximum of 1000. Delly splits each unmapped read into kmers and aligns the kmers to the reference sequence spanning the SV.

Lumpy

Lumpy[100] is an SV caller developed in 2013 by R. Layer et al. and is a package implemented in C++. All of the mathematical formulas and figures presented in this section are reproduced from the 2014 publication by R. Layer et al.

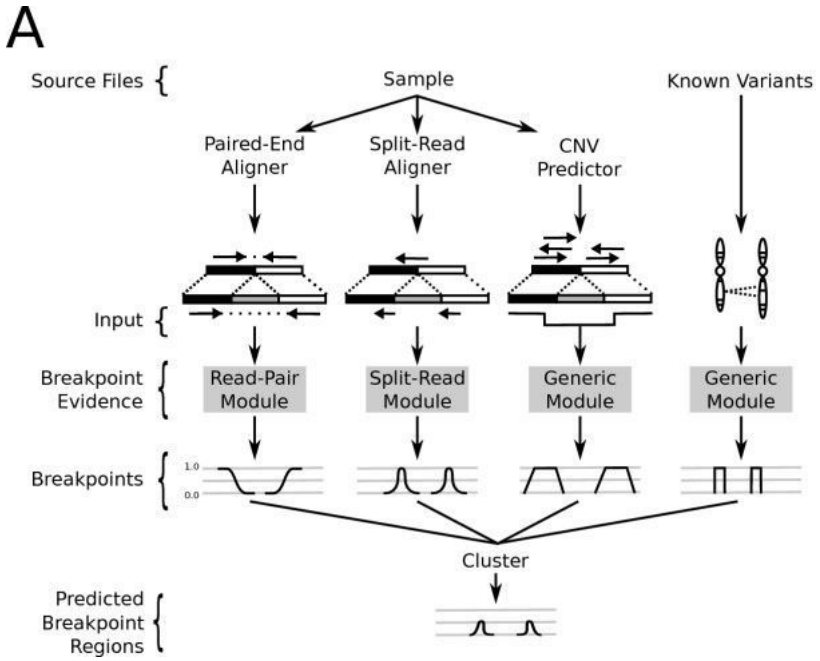


Figure 2.31: Lumpy calling model integrates several signals from a single sample.[100].

Lumpy defines an SV breakpoint as a pair of bases that are adjacent in the sample under study but not in the reference genome. Furthermore, each breakpoint is represented as a pair of probability distributions that span the predicted breakpoint regions and represent the uncertainty about the precise location of the breakpoint.

Lumpy integrates multiple signals (see Figure 2.31) to update the probability distributions that represent each breakpoint based on different kinds of evidence provided. A breakpoint is a tuple $b = \langle E, l, r, v \rangle$ where E is the evidence, $b.l$ and $b.r$ are the left and right intervals each having start and end coordinates, $b.l.s$ and $b.l.e$ for example. $b.l.p$ is a vector of length $b.l.e - b.l.s$ where each $p[i]$ is the probablity that $b.l.s + i$ is the true location of the breakpoint. $b.v$ is the breakpoint class (insertion, deletion, etc.). Overlapping breakpoints are merged together. The intervals that contain 95% of the probability density, as well as the Maximum Likelihood Estimates of the location of each variant are returned.

The paired-end analysis looks at read pairs $\langle x, y \rangle$ where each read is aligned to the reference genome as $R(x) = \langle c, o, s, e \rangle$ where c is the chromosome, $o \in +, -$ indicates alignment orientation, s and e represent the alignment start and end respectively. It is assumed that x and y align uniquely and that $R(x).s < R(x).e < R(y).s < R(y).e$. $S(x) = \langle o, s, e \rangle$ is defined to be the alignment of x with respect to the sample's (unknown) genome. Read pairs are assumed to be aligned with $R(x).o = +, R(y).o = -$ and having $R(y).e - R(x).s$ approximately equal to the library preparation fragment size. Read pairs that align outside the expected parameters for orientation and distance constitute evidence for structural variant breakpoints, in particular reads with the same or switched orientation, and pairs that align at a distance shorter or longer than the fragment size. Expected fragment length is estimated from the mean fragment length in the sample, along with its standard deviation. Breakpoint variety is determined as follows:

When $R(x).c = R(y).c$, the variety is inferred from read orientation. $R(x).o = R(y).o$ implies an inversion. $R(x).o = -, R(y).o = +$ implies a tandem duplication, $R(x).o = +, R(y).o = -$ implies a deletion. Insertions are not presently supported. When $R(x).c \neq R(y).c$ the breakpoint is labelled an interchromosomal rearrangement.

$\langle x, y \rangle$ are mapped to $b.l$ and $b.r$ as follows. By convention x is assumed to map to l and y to r. It is assumed that there exists a single breakpoint b between x and y. Orientation of x determines whether l is upstream or downstream from x. Thus, if $R(x).s = +$, then the breakpoint begins after $R(x).e$. The length of the breakpoint interval is proportional to expected fragment length and its standard deviation. The distance of one end of the breakpoint from x is assumed to be less than expected insert size L plus $v_f s$ - a configurable number of standard deviations. The probability that position i in the breakpoint interval l is part of the actual breakpoint is estimated as the probability that x and y span i, which is true when the fragment that had produced $\langle x, y \rangle$ is longer than the distance from x to i. Thus, the quantity of interest is $P(S(y).e - S(x).s > i - R(x).s)$ for $R(x).o = +$ and $P(S(y).e - S(x).s > R(x).e - i)$ for $R(x).o = -$. This quantity is estimated from the empirical distribution of fragment sizes collected from the sample.

Lumpy does not perform its own split-read alignment instead relying on the results of a split-read aligner like YAHA[55]. Every split-read is a DNA fragment X that consits of two or more sub-fragments $x_i, ..., x_j$ that do not map to adjacent locations of the reference. Each sub-fragment pair $\langle x_i, x_{i+1} \rangle$ is evaluated independently depending on how it maps with respect to the reference $R(x)$. When

$R(x_i).o = R(x_{i+1}).o$ the event is marked as either a deletion or a tandem duplication, where $R(x_i).s < R(x_{i+1}).s$ implies a deletion, and otherwise a tandem duplication. When orientations do not match, the event is marked an inversion. To account for potential inaccuracies in the location of the breakpoint with respect to the split-read pair, each split-read pair is mapped to two breakpoint intervals l and r that are centered at the split. The probability is modeled to be highest at the split with an exponential decrease towards the edges, with a configurable width parameter.

SvABA

SvABA[192] is a germline and somatic SV caller by J. Walla et al., created in 2016. Svaba is implemented in C++ and relies on genome-wide local assembly for variant detection. Internally SvABA makes use of SGA[174] and BWA-MEM[113] for assembly and read mapping respectively.

Figure 2.32: The SvABA variant callng method[192].

SvABA performs local de-novo assembly in 25-kbp windows tiled across the genome with a 2kb overlap. First reads that may be indicative of variation are extracted from a BAM file, these include reads with high-quality soft-clipped bases, discordant reads (with non-standard orietnation or with insert size greater than four standard deviations away from the sample mean insert size, determined using a sample of 5 million reads with top and bottom 5% of insert sizes truncated), unmapped reads, reads with unmapped mates, and reads with insertions or deletions indicated in the CIGAR string. Low quality reads that are marked as PCR duplicates, have failed QC, and reads with repeats of >20 basepairs are filtered out. Discordant reads are realigned to the reference with BWA-MEM and those with an available nondiscordant alignment of >70% of the maximum alignment score are discarded. Reads with many different (>20) high quality candidate alignments are also discarded. Remaining discordant reads are clustered based on orientation and insert-size and assembled using SGA.

The contigs produced by SGA are aligned to the reference genome using BWA-MEM and examined for potential variants. Contigs with an alignment that has fewer than 30 nonaligned bases and no alignment gaps are considered reference. Indels are called when the alignment has gaps, and SVs called when the resulting alignment is multi-part. In order to evaluate read support for the putative variants, reads from the windows are aligned to both the reference and the assembled contigs using BWA-MEM. Reads are considered matching to the contig when the alignment score for the read to the contig is greater than the alignment of the read to the reference and is >90% of the length of the match. Reads that have an alignment that is up to 8 bases to the left or right of a putative variant are consdered supporing the variant.

Remarks

We looked at Delly, Lumpy, and SvABA in the context of germline SV calling. Although there are some differences in the details of the approaches taken by the tools' authors there are consistent similarities as well, which additionally carry over to other SV callers such as Pindel[204], and Manta[22], that have not been presented here. These approaches rely heavily on paired-end reads and select those that are mapped uncommonly far apart or close together. These are clustered or optionally locally assembled to produce putative breakpoints. Breakpoint locations are further refined by split-read analysis, using those reads that are unmapped by regular read mapping software. These reads are broken down into kmers and each kmer is aligned separately to non-adjacent locations in the genome. Alternative haplotypes may be constructed using these reads and alignments to the reference and the alternative haplotypes scored for genotyping purposes. A new method for structural variant calling should thus focus on making the best use of these two data types (discordant and split reads), while possibly also making use of read depth information to be competitive with the current generation of best callers.

2.2.7 Variant Filtering

2.2.8 Somatic SNP Calling

Mutect - Muse - Pindel -

2.2.9 Somatic Indel Calling

2.2.10 Somatic Structural Variant Calling

2.2.11 Germline Variant Annotation

Annovar - Variant Effect Predictor -

2.2.12 Somatic Variant Annotation

2.2.13 de-novo Assembly

Velvet - Abyss -

2.3 High Performance , High Throughput, and Cloud Computing

The practice of performing large scale scientific computation on supercomputers or clusters of commodity hardware can be split into two notions - High Performance Computing (HPC) and High Throughput Computing (HTC).

The European Grid Infrastructure defines these as follows[65]:

> HPC - A computing paradigm that focuses on the efficient execution of compute intensive, tightly-coupled tasks. Given the high parallel communication requirements, the tasks are typically executed on low latency interconnects which makes it possible to share data very rapidly between a large numbers of processors working on the same problem. HPC systems are delivered through low latency clusters and supercomputers and are typically optimised to maximise the number of operations per seconds. The typical metrics are FLOPS, tasks/s, I/O rates.

> HTC - A computing paradigm that focuses on the efficient execution of a large number of loosely-coupled tasks. Given the minimal parallel communication requirements, the tasks can be executed on clusters or physically distributed resources using grid technologies. HTC systems are typically optimised to maximise the throughput over a long period of time and a typical metric is jobs per month or year.

Although early High Performance Computing efforts (1960's - 1980's) relied on supercomputers with a shared memory model[162], where all of the memory was shared between multiple processors, by the late 1980's machines with a distributed memory model[142], where each processor has its own memory, started gaining ground, forming the basis for the modern HPC cluster.

The software interface that the user has to a HPC/HTC cluster typically takes the shape of a queueing system such as PBS[77] or LSF[211] where the user writes a script that submits a series of jobs to the queueing system. The jobs can invoke software that is installed by the IT department that manages the cluster. The user is not able to install any software and has limited visibility into the runtime performance characteristics of the jobs they submit.

Cloud computing has emerged in the early 2000's enabled by improvements in hardware virtualization which was driven by the adoption of Virtual Private Networks, and the desire to commercialize access to compute capacity as a utility[20].

The National Institute of Standards and Technology provides a standard definition of cloud computing that encompasses several areas of this domain - Essential Characteristics, Service Models, and Deployment Models[133].

The Essential Characteristics of a cloud are as follows:

On-demand self-service - End-user can independently manage infrastructure without involving the service provider.

Broad network access - Cloud resources are available on the network via a set of standard protocols.

Resource pooling - Service providers dynamically assign virtual infrastructure in a multi-tenant environment based on consumer demand.

Rapid elasticity - Resources can be elastically provisioned and discarded according to customer requirements.

Measured service - Resource usage by end users is measured and transparently provided back to the user by the service provider.

It is the self-service and broad network access characteristics that set cloud computing apart from traditional HPC computing the most.

Service Models include:

Infrastructure as a Service (IaaS) - This service allows the the user to provision and control virtualized infrastructure such as VMs and networks.

Platform as a Service (PaaS) - This service allows the user to deploy their application onto virtualized hardware but not to control the management of the infrastructure.

Software as a Service (SaaS) - This service allows the user to make use of applications that are deployed on virtualized hardware but not to manage the applications or the infrastructure itself.

The Deployment Models covered by the NIST definition are as follows:

Private Cloud - Operated privately by a single organization and not accessible on a public network.

Community Cloud - Established for use by a particular community of users with a common interest.

Public Cloud - Established for general use by the public.

Hybrid Cloud - A collection of cloud entities that use one of the other deployment models but allow application portability.

The first publicly available commercial cloud computing platform has been developed by Amazon.com and launched in August, 2006 in the form of two services - Elastic Compute Cloud (EC2), and Simple Storage Service (S3). Cloud offerings by

Microsoft and Google followed in 2010, and 2012. This early lead has allowed Amazon to capture the majority of the public cloud computing market, earning $2.57 billion USD in Q1 2016 revenue.

One of the main drawbacks, however, of using Amazon's or another proprietary cloud solution is the issue of "vendor lock-in" i.e. inability to easily switch infrastructure providers should the customer wish to do so, because of the amount of software relying on the proprietary cloud provider protocols. Another key reason for avoiding public clouds is the necessity to store sensitive data. This issue applies both to the commercial enterprise (with industries such as banking, and payments) and scientific domains (especially genomics and medicine) where handling of sensitive patient data is restricted based on both technical security, as well as ethical considerations[91].

To help alleviate these concerns an open-source cloud platform called Openstack was launched in 2010 jointly by Rackspace Hosting and NASA[167]. Openstack provides most of the same features that are provided by Amazon Web Services and other commercial cloud providers as free open-source tools. These include:

- Infrastructure
- Networking
- Identity Management
- Block Storage
- Object Storage
- Managed Databases
- Queues
- Monitoring

Openstack deployments for the basis for most academic private and community clouds such as EBI Embassy Cloud[32], University of Chicago Open Science Data Cloud[71], Cancer Genome Collaboratory[205], and Helix Nebula[129]. Because these clouds implement the security measures necessary when handling patient data they are a system of choice for large scale bioinformatics analyses.

2.4 Workflow Systems

The focus on workflow stems the work of Frederick Taylor (1856-1915) and Henry Gantt (1861-1919) on the improvement and automation of industrial processes, also known as "scientific management"[181]. One of the key techniques that were devised at the time and served as the prototype for future workflows were "time and motion

studies"[13] where employees were observed as they performed repetitive cycles of work in order to determine standard execution times and sequences of steps. As this field evolved over the course of the 20th century it gave rise to several other related fields such as Operations Management, Business Process Management, and Lean Manufacturing.

In 1993 an international consortium was formed with the purpose of defining the standards related to workflows and workflow management systems. This consortium is called the Workflow Management Coalition (WfMC). One of the key specifications produced by the WMC in 1995 is The Workflow Reference Model[80]. This document provides two basic definitions that illuminate the scope and purpose of workflow systems:

> Workflow - The computerised facilitation or automation of a business process, in whole or part.

> Workflow Management System - A system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic.

A number of standards have been produced for workflow definition, many of them are XML-based[169]. Notable examples include:

XPDL - Was developed by the WfMC, currently at version 2.2, as of 2012. Uses an XML dialect to express process definitions

BPML - Developed by the Object Management Group (OMG) using XML. Deprecated as of 2008 in favour of BPEL.

BPEL/BPEL4WS - Developed by Organization for the Advancement of Structure Information Standard (OASIS). Uses XML format. Adopted by Microsoft and IBM for their workflow products -

Graphically, workflow definitions are typically expressed using a Petri-Net[150] or Business Process Model and Notation (BPMN), the latter borrowing its structure from UML activity diagrams. A set of workflow definition design patterns exists to guide workflow creation[43]. A workflow engine is responsible for ingesting workflow definitions, generating their graphical representation, and allowing the user to execute the workflow definitions on suitable hardware.

As initially the focus of workflow systems research and development has been on process improvement within commercial enterprises there exists a large pool of workflow engine implementations targeted at that sector. Some of these are:

jBPM - An open-source workflow engine that is based on the Java platform and is currently owned by Red Hat.

Activiti - An open-source workflow engine that has been developed by previous jBPM developers.

Oracle BPEL Process Manager - A commercial workflow engine acquired by Oracle from Collaxa in 2004, now integrated into the rest of the Oracle portfolio.

Websphere Process Server - Commercial workflow engine that is part of IBM's Business Process Manager suite.

Although these tools have gained wide adoption in the enterprise community they have had limited success within scientific circles. Instead, several open-source workflow management systems exist that have been purpose-built for the scientific domain, and especially bioinformatics. These include:

Kepler[120] - A Java-based WfMS built on top of the Ptolemy II[41] execution engine.

Taverna[143] - A Java-based WfMS originally built by myGrid, currently under incubation at Apache Software Foundation.

Galaxy[66] - A Python-based WfMS developed specifically for bioinformatics applications with a focus on GUI-driven development of workflows.

Curcin et al[36] provide a head-to-head comparison of six scientific workflow systems including Taverna and Kepler, whereby Taverna is described as primarily being aimed at researchers who wish to build scientific workflows from web services utilizing a proprietary XML dialect called SCUFL which implements a DAG model of workflows. The primary execution environment for a Taverna workflow is on a grid or an HPC cluster. Kepler implemented a different methodology, whereby workflow modelling, which is taken on by Actors, is separated from workflow execution, taken on by Directors. An Actor knows only about its inputs, the computation that it needs to perform, and the output that it needs to produce, while Directors provide different models of execution, such as Synchronous Data Flow, Process Network, Continuous Time, and Discrete Event.

The Galaxy workflow framework has a specific focus on bioinformatics analyses and comes with a large library of community-developed bioinformatics workflows. The user creates and executes workflows via a web-based GUI where pre-installed tools and scripts can be laid out into a pipeline. The primary deployment environment for Galaxy is on an institutional HPC cluster although a separate component allows the deployment of a Galaxy instance on Amazon Web Services[5].

2.5 Service Oriented Architectures

2.6 Stream-based Systems

Chapter 3

The Butler Framework - Requirements and Architecture

Chapter 4

The Butler Framework - Implementation and Experimental Validation

Chapter 5

The Rheos Framework

In this chapter we describe a software framework called Rheos, which demonstrates an approach for reasoning about large genomic datasets utilizing concepts of service-orientation and data streaming in contrast with traditional genomic data analysis frameworks[42] that take a procedural batch-based approach. Rheos' focus on service-orientation and streaming allow the users to make active tradeoff decisions between analysis time, cost, and quality as well as setting up precise operational Service Level Agreements, both between Rheos components, and between Rheos and external systems, as we describe in detail below.

5.1 General Framework Design

As already discussed in Chapters 1 and 2, the general problem consists of collecting DNA samples from a population of individuals under study, sequencing these samples using Next Generation Sequencing techniques, identifying the mutations that are present, annotating their functional impact and utilizing the obtained data in a downstream data analysis with research or clinical decision-making goals. While there is a great variety of possible downstream analyses that may be performed depending on the individual goals of the analyst, there is a fairly well established set of steps for processing of the raw NGS data into a set of annotated variants, and it is these steps that we target with this work. The typical approach that is in widespread use today is to collect a batch of samples and then process each sample individually with a sequence of individual tools, that may be described via a higher-level workflow construct (such as in Figure 2.26, or using a framework like Butler, as described in Chapters 3, 4). There are, however, a number of factors that leave room for improvement in this model. These improvements lie along a set of dimensions that we describe briefly in the Introduction via a utility function $U_i = C_i + T_i + A_i$ for sample $i \in [1, N_s]$ that needs to be optimized, and that we describe in more detail here.

We use the following definitions throughout the text:

Table 5.1: Rheos common definitions

Symbol	Description
N_p	Number of people
$P = \{p_i : i \in [1, N_p]\}$	Set of individuals under study
N_s	Number of samples
$S = \{s_i : i \in [1, N_s]\}$	Set of sequenced DNA samples. Each individual can have one or more samples.
A_i	Accuracy score of analysis for sample i (precise definition of Accuracy TBD)
$C_i = c_{g_i} + c_{s_i} + c_{a_i} + c_{r_i}$	Cost score of data generation, storage, analysis, and retrieval respectively
T_i	Time score to process sample i
$U_i = C_i + T_i + A_i$	A utility function for individual i that penalizes high cost, high processing time, and low accuracy
$U = \sum_{i=1}^{N_s} U_i$	Overall utility of processing N_s samples through Rheos.

5.2 Data Streaming Architecture

The overall technical architecture of the Rheos system is set up as a Service Oriented Architecture (SOA)[170] which is an information system architecture paradigm where the overall problem that the system is trying to solve is broken down into a collection of loosely-coupled components called services. Each service has a well defined interface of inputs that it accepts and outputs that it produces. Services can be combined and orchestrated together to produce the overall desired output for the system. A key distinguishing feature of this architectural approach is that each service can be individually optimized to fulfill its contract most efficiently helping break down some of the performance limitations brought about by the necessity to simultaneously tackle competing constraints in more monolithic information system designs. Additionally, within a services framework, the dependencies between separate services can be negotiated not only in terms of service interfaces, but also in terms Service Level Agreements which constitute Quality of Service promises made by one service to its dependents[83]. Because it is unlikely that a service designer will be able to accurately foresee all of the demands that will be placed on a service during its lifetime the SLAs provide a valuable feedback framework through which the service can be evaluated as it operates in production, as well as serving as a basis for negotiating evolving requirements between dependent services.

While general web services can support any data processing paradigm, in the Rheos framework we adopt a data streaming approach[12]. In this approach we assume that the input to any service is a randomly ordered sequence of messages $M = m_1, m_2, ...$ where each message represents a fact about the underlying domain that the service reasons over, as well as some metadata, including an identifier, and a variety of timestamps of interest. The content of each message may provide a

datum, such as the measurement of a quantity of interest, or signal that a particular event has taken place. It is in general assumed that the data stream is infinite in size, that messages may arrive out of order, and that any message that is placed in the stream is observed at most once, and may, in fact, never be observed. Messages are typically not sent directly from one service to another, instead the transfer of messages is mediated by a queuing system using a publish-subscribe[50] model. Under this model each queue acts as a *topic*. Message producers can publish data to the topic, and message consumers subscribe to receive messages from the topic. A message is consumed from the queue only after all of the subscribed consumers have seen it. End users retrieve information from the system via a set of User Interfaces that support both push (notifications) and pull (querying) models of data retrieval. A more detailed description of the architectural aspects of the system follows:

5.2.1 Service-Oriented Data Streaming Model

A data stream $M_{s,d} = m_1, m_2, ...$ is a sequence of datagrams transmitted over the network with the following properties:

- The stream has a source s and a destination d.
- A message m in the stream is a tuple of the form $(header, payload)$, where:
  - *header* is a tuple of the form $(id, \dots)$ that holds at minimum a unique identifier id for messages, and may hold additional metadata.
  - *payload* is an arbitrary data structure that holds the informational content of the message.
- $|M| = \infty$ by assumption.
- Messages may not arrive at destination d in the same order that they were sent from source s.
- If $t_{i,s}$ is the time message m_i leaves the source s and $t_{i,d}$ is the time of arrival at destination d, then $\sup_i\{t_{i,d} - t_{i,s}\} = \infty$, i.e. a given sent message may never arrive at its destination.

A service $S = \{o_i\}$ is a collection of operations o_i that act on one or more input data streams $\{M_i\}$ to produce one or more transformed output data streams $\{M_j\}$. Specifically:

An operation O is a tuple of the form:

$$O = (i, o, p, f) \tag{5.1}$$

where:

- $i = \{M_j, j \in [0, K]\}$ is a set of $K \geq 0$ input data streams.

- $o = \{M'_j, j \in [0, L]\}$ is a set of $L \geq 0$ output data streams.
- $f : M^K \mapsto M'^L$ is a transformation function that produces messages m' in the output streams based on messages m observed in the input streams.
- $p = \{p_i\}$ is a set of potentially optional query parameters.

There are several distinct categories of operations that a service can perform on a set of input streams. We describe these here:

Windowing Function - Service S observes a sliding window, which is a sample of size n of messages from stream M_i and computes a summary statistic (see Figure 5.1) over the sample which is meant to be an approximation of the corresponding population parameter.

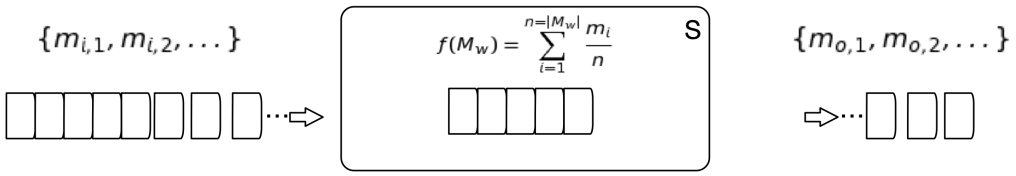


Figure 5.1: Service S computes a summary statistic over a window of messages from stream M

Decorator Function - Service S observes messages m_i and applies a function that augments (decorates) each message with additional attributes (see Figure 5.2) producing augmented messages m_o as output.

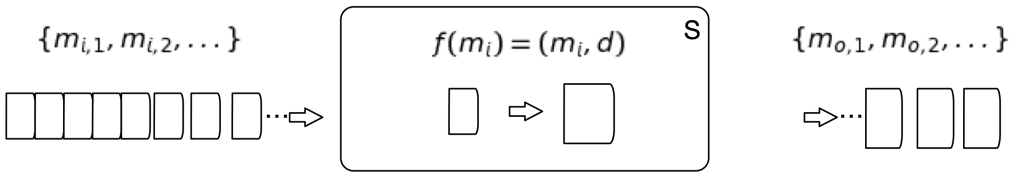


Figure 5.2: Service S augments messages from M with an additional set of attributes.

Filter Function - Service S observes messages m_i and applies a function $f : M \mapsto \{True, False\}$ that evaluates to a boolean value (see Figure 5.2). Only messages that map to $True$ are emitted as output.

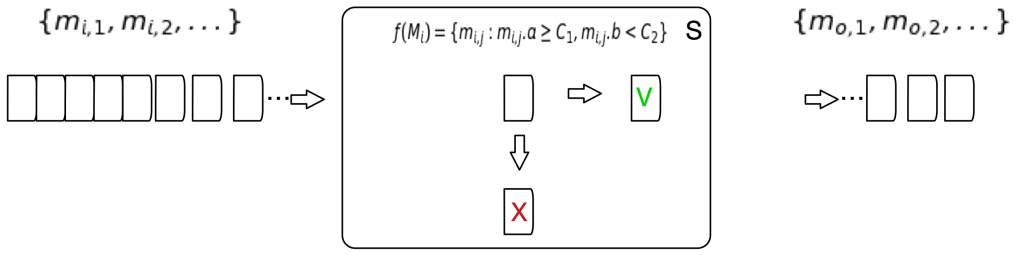


Figure 5.3: Service S filters messages from input stream M and only allows through those that pass the filtering condition.

Aggregator Function - Service S observes messages from N different streams $\{M_j : j \in [1, N]\}$ and applies a function $f : M_i^N \mapsto M_o$ that aggregates messages from these streams to produce its output (see Figure 5.4). Because aggregation happens over groups of messages that may not all arrive at the same time the service S requires a mechanism for keeping local state so that it can accumulate messages that have already arrived while waiting for those that are necessary to compute f yet have not been observed. The statefulness requirement of this type of service places an extra level of complexity (related to state-management and request routing) as well as inherent scalability limitations compared to stateless services[144].

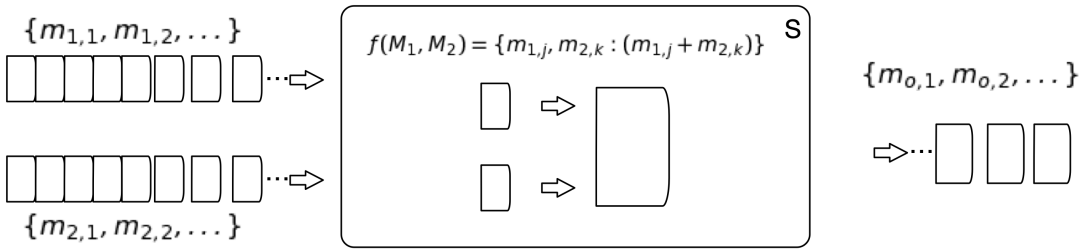


Figure 5.4: Service S integrates messages from multiple input streams M_i to produce an aggregated output stream M_o via f.

Local State Aggregator Function - Service S observes an input stream M_i which it integrates with a local (non-stream) queryable data store (see Figure 5.5). Messages m_i are integrated with query results q_i to produce an output stream M_i. This type of service also requires management of state and scalability concerns similar to the Aggregator service, especially when the local data store is itself distributed.

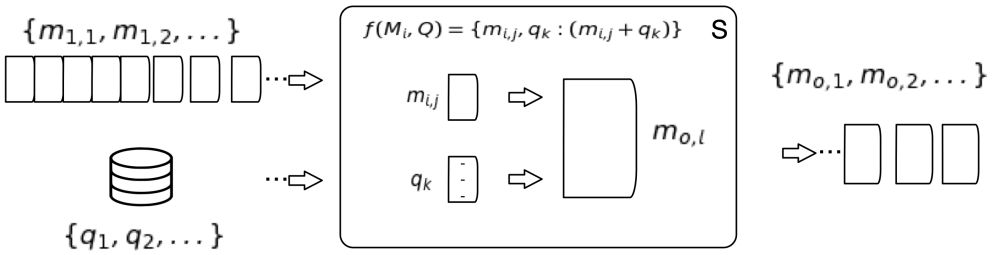


Figure 5.5: Service S aggregates m_i with query results q_i obtained from a local data store.

Persistence Function - Service S observes messages m_i and is responsible for persisting them to a data store where their contents can later be queried (see Figure 5.6). Although persistence of data to, and subsequent querying of data from, a store, such as a database, are comparatively more expensive operations than immediate reasoning over a live data stream, such mechanisms are necessary for situations where data may need to be accessed multiple times, or where data may need to be retained for audit purposes.

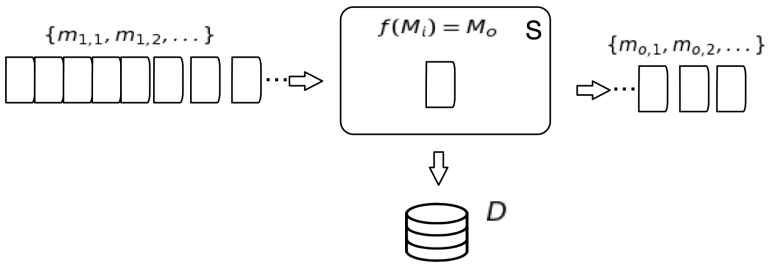


Figure 5.6: Service S processes messages m_i into persistent storage. The output stream M_o contains persistence confirmation and error events.

Query Function - Service S observes a stream of queries Q_i. The queries are fulfilled against a data store D and the results emitted via the output stream M_o.

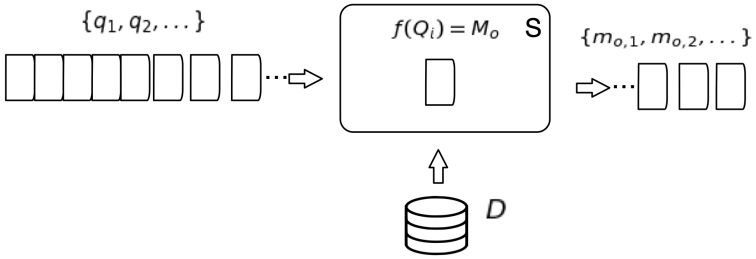


Figure 5.7: Service S filters messages from input stream M and only allows through those that pass the filtering condition.

The basic operations above can be combined to produce arbitrarily complex logic

on data streams.

One of the key advantages of a service-oriented approach is that, because services are typically constantly executing, it naturally lends itself to an examination of the system's runtime characteristics. This applies to both service-internal characteristics that are related to each operation a service performs, as well as to external characteristics that relate to the contracts a service establishes with its dependencies. We consider both of these.

For each given operation $o_i \in S$ it is instrumental to understand the resource requirements of the operation on typical inputs and limiting factors that affect the efficiency with which the operation can be performed by S. Of particular interest are the per-operation profiles of:

- CPU utilization
- RAM
- Secondary storage
- Network utilization

If o_i is a long-running operation that takes multiple seconds to complete on average, a detailed distribution in time of each metric above may be necessary. If the operation can be completed at a sub-second rate then summary statistics (min, max, mean, median, inter-quartile range, 90th, and 99th percentiles) may be sufficient. This level of understanding is necessary in order to make sure that the service can adequately deal with the incoming message stream while the messages are first loaded into memory, since subsequent retrieval from secondary storage is several orders of magnitude slower and may cause further delays in processing. If o_i is stateless, i.e. it does not require the storage and retrieval of any local state that depends on the content of each arriving message $m_j \in M_i$, then the service S can be scaled "horizontally"[187] with respect to o_i. Given that the performance-limiting condition of o_i is known (CPU, memory, etc.), the ability of S to efficiently deal with fluctuations in the rate of incoming messages M_i can be successfully achieved simply by adding and removing servers that execute S (see Figure 5.8), which can be done automatically[125]. If o_i is stateful and requires access to databases, or predictable request routing via sessions, then horizontal scalability may not be possible and thus, detailed understanding of the performance profile and performance-limiting conditions of o_i is even more important as vertical scaling of services is more expensive and challenging to accomplish, and may increase system complexity by necessitating data partitioning, for example[187].

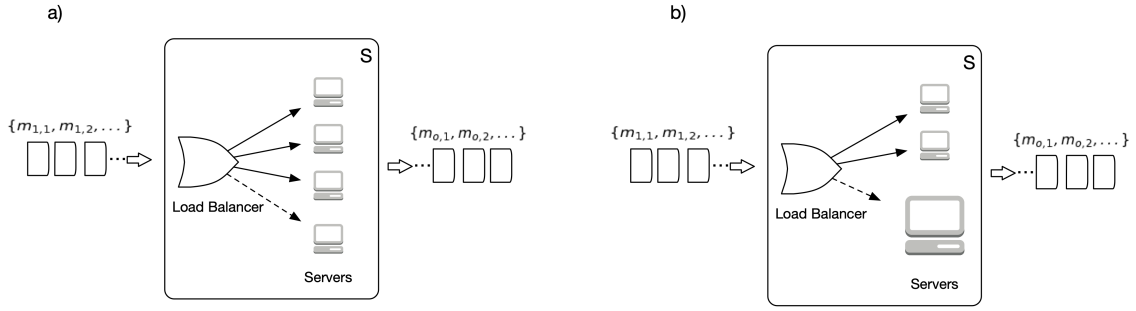


Figure 5.8: a) In horizontal scaling new servers are added and removed (dashed arrow) behind a load balancer as the rate of data stream M_i fluctuates. b) In vertical scaling more powerful servers need to be launched (dashed arrow) to replace smaller servers (with potential service outage) when the rate of M_i increases beyond capacity.

Assume service S implements operation o supported by n physical servers $V = \{v_j : j \in [1, n]\}$ by consuming a stream of incoming messages M_i. For a suitable time increment t, let $r_{M_i} = |M_i|/t$ be the incoming message arrival rate, and $r_{M_{o,j}} = |M_{o,j}|/t$ be the processing rate for server v_j. If $r_{M_i} > \sum_{j=1}^{n} r_{M_{o,j}}$, then S will not be able to adequately process all of the incoming messages from M_i and messages will either be lost or need to be backlogged while more servers are added to S to deal with the incoming message rate. Since commissioning new servers takes considerable time and the timing and magnitude of increases in r_{M_i} may be unpredictable, serious information loss may result if measures are not put in place to mitigate the message rate fluctuations.

A queue is the mechanism that we put in place to address this concern (see Figure 5.9). A queue Q is a message buffering system which consists of a set of n "topics" $P = \{p_i : i \in [1, n]\}$, where each topic is a tuple of the form $p_i = (D_p, B_p, C_p)$. Here $D_p = \{d_i : i \in [1, k]\}$ is a set of k data producers that put messages into Q, B_p is a message buffer of max capacity N_{max} dictated by underlying server hardware characteristics, containing a sequence of messages $\{m_t, m_{t-1}, m_{t-2}, ..., m_1\}$ that are accessible in a Fist In First Out (FIFO) manner, and $C_p = \{c_i : i \in [1, l]\}$ is a set of l consumers that are interested in observing messages from p_i.

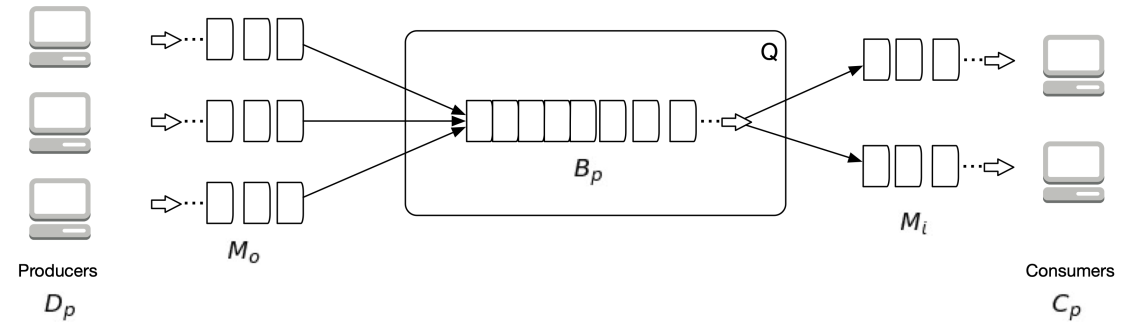


Figure 5.9: A
queue Q establishes a message buffer B_p between a set of message producers D_p and consumers C_p, for a given topic p.

Messages arrive into a particular topic of Q from all producers D_p, and are marked safe for deletion only when all of the subscribed consumers C_p have observed a particular message. Thus, for topic p_i, the incoming message rate is $R_i = \sum_{j=1}^{k} r_{M_{o,j}}$ i.e. the sum of the message processing rates for all of the producers for this topic. The queue message processing rate $R_o = \min_{i \in [1,l]} \{r_{M_{o,i}}\}$ is the slowest message processing rate among all consumers. Assuming $R_i > R_o$ and that there are N messages presently in B_p, there remains $t = \frac{N_{max} - N}{R_i - R_o}$ time before queue overflow occurs. The situation should then be remedied by allocating additional hardware to Q or those services S_i whose consumers are slowest, until the condition $R_o \geq R_i$ can be reliably maintained. If the queue does reach its maximum capacity overflow measures need to be put in place. Depending on the data stream in question data loss may or may not be acceptable. If data loss is acceptable then overflow messages can be simply discarded. If data loss is not acceptable then producers must block waiting for additional queue capacity to become available. This not only degrades performance locally, but can have a drastic effect on the entire system if the effects are allowed to percolate trough the complex distributed system. As message rates evolve through time with system load, the scheme above sets up a framework for flow control and hardware allocation within the architecture.

When designing a service-oriented system the interfaces of operations provided by the service are of utmost importance as they define the capabilities that the service offers to its clients. Of secondary, but also significant, importance is the set of Service-Level Agreements (SLAs)[200] that a service advertises. These SLAs are a set of commitments that a service makes to its clients that describe the operational characteristics of the service, such as:

Availability - Guarantees related to the service uptime, maintenance outages, disaster recovery, etc.

Throughput - The number of requests serviced per unit time.

Latency - The delay between a request being sent and a response being received.

Abandonment Rate - Proportion of requests that are never answered.

Error Rate - Proportion of well-formed requests that result in an error.

Based on the SLAs that are advertised by a given service, the services that depend on it can make assumptions about expected runtime behavior, and take action when expectations are not met. Furthermore, when requirements evolve and features are added to or removed from a service, the impact on the advertised SLAs helps communicate the full effect of the changes. Lastly, the costs of operating a service are more clearly understood through the SLA framework, where improvements to a particular SLA metric, such as Transactions-Per-Minute (TPM) can be transparently traced to a corresponding increase in operational costs.

The set of services $\{S\}$ that communicate over data streams $\{M_{s,d}\}$, mediated by a set of queues $\{Q\}$ with a set of established SLAs $\{L_s\}$ together form the overall framework of Rheos that is used to tackle the challenges of large-scale genomic data processing in a manner the enables active tradeoffs between the competing constraints of cost, time, and accuracy.

5.3 Domain-specific Problems

Having laid out the general data-streaming service-oriented architecture of Rheos in the previous section we now turn to a discussion of the set of actual domain-specific problems that need to be solved within the data-streaming paradigm in order to enable the comprehensive genomic characterization of large cohorts of samples within Rheos, as we have set out to do. We make use of the flow of data types from the most raw to the most refined (see Figure 5.10) to illustrate the challenges that need to be solved during transformation of the input data between each successive stage, first in summary form, and then in full detail, below.

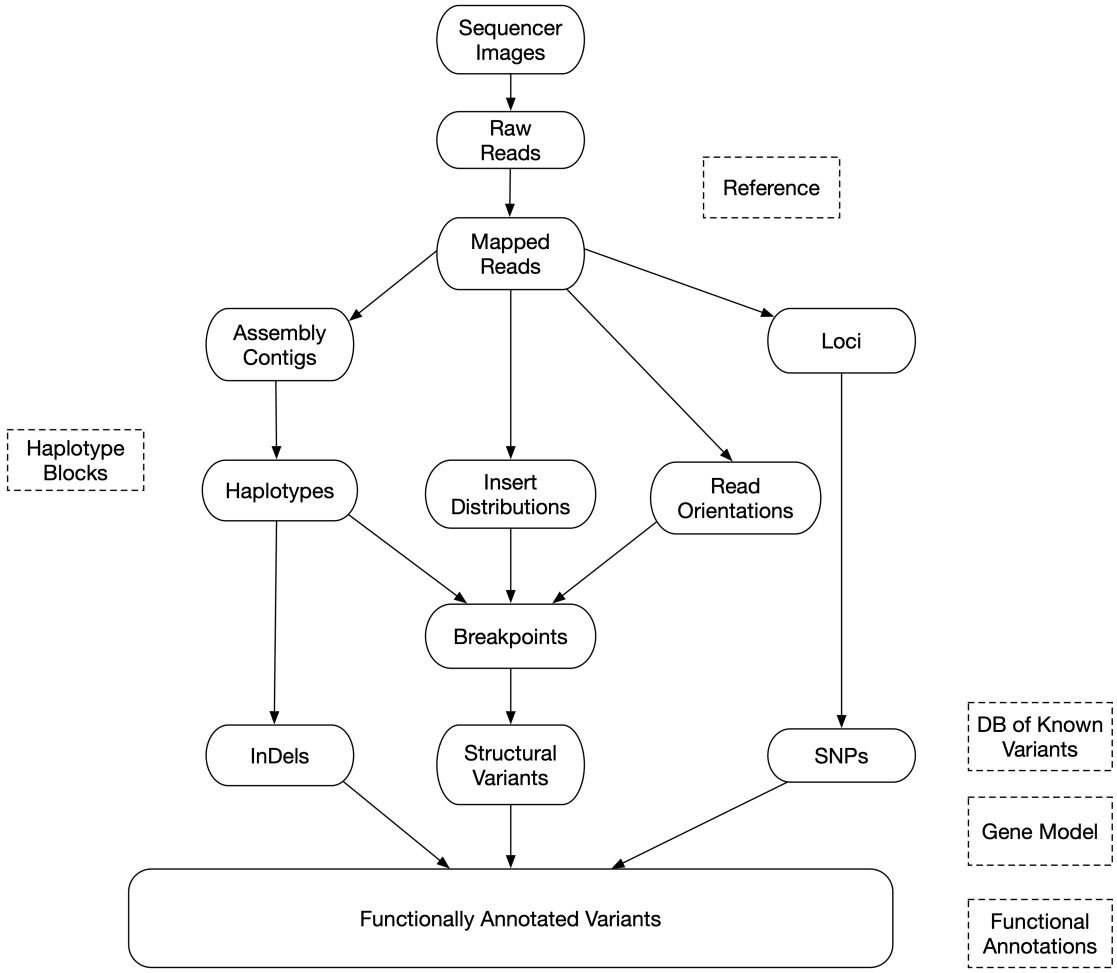
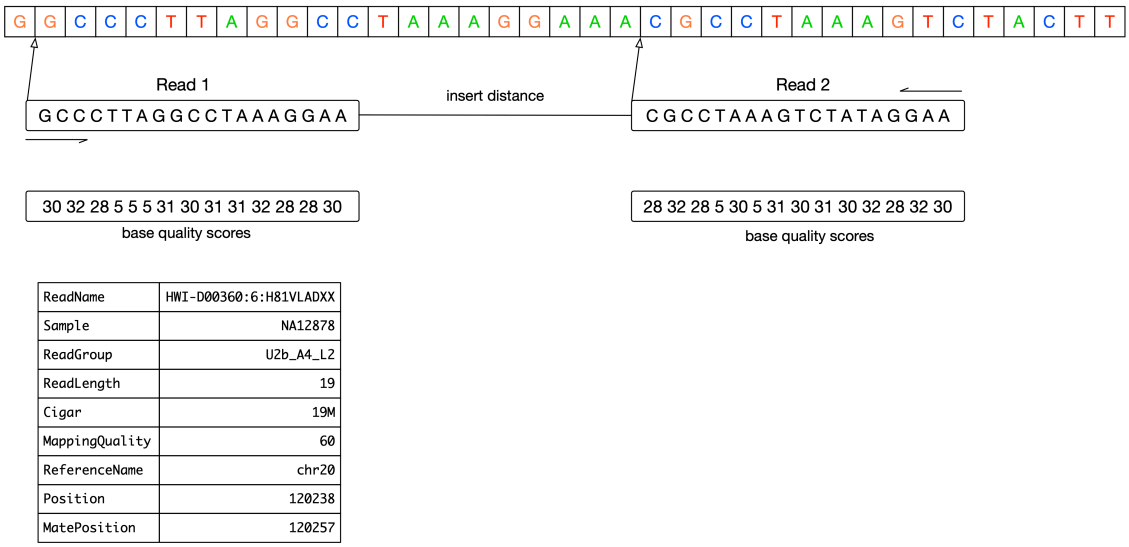


Figure 5.10: The conceptual flow of data types within Rheos from the most raw - Sequencer Images, to the most refined - a set of Functionally Annotated Variants.

The most raw data type that is produced from a sequencing experiment is the set of raw image files generated by the sequencer. Although, conceptually, processing of the raw images could also be accomplished within Rheos, it is presently outside of the scope of this work. Instead, we assume the most basic data type to be raw sequencing reads, as found in a FASTQ[25] file.



ReadName	HWI-D00360:6:H81VLADXX
Sample	NA12878
ReadGroup	U2b_A4_L2
ReadLength	19
Cigar	19M
MappingQuality	60
ReferenceName	chr20
Position	120238
MatePosition	120257

Figure 5.11: A read-pair that is aligned to the reference.

Each read is a tuple of the form:

$$r = (s\_id, r\_id, b, q, f_p) \tag{5.2}$$

where:

- s_id - is the sample ID, which is unique among all samples.
- r_id - is the read ID, which is unique among all reads for that sample.
- $b = \{b_1, b_2, ..., b_n\}$ - is the sequence of DNA bases, where $b_i \in \{A, C, G, T, N\}$.
- $q = \{q_1, q_2, ..., q_n\}$ - is the set of PHRED-scaled base quality scores corresponding to the probability that the base has been called incorrectly. See discussion on FASTQ format in Section 2.2.1 for details.
- $f_p \in \{True, False\}$ - is a boolean flag indicating whether this read is the first read in a pair.

5.3.1 Read QC Metrics

Section 2.2.3 of the Background chapter discusses various metrics of interest that are based on observations of read data and the tools that are used to collect them.

Here we describe how to collect the most typical metrics in the streaming paradigm of Rheos. As before, there are per-read metrics such as Base Quality Distribution, and Adapter Sequence Presence, as well as per-sample metrics such as Average GC Content, Insert Distribution, Read Length Distribution, and others. The utility of these metrics is to be able to set up filters for low quality data as well as input for downstream variant-calling models (see Section 2.2.6 for example).

Assume that we are observing a stream $M_{raw} = \{m_i : m_i = (header, payload)\}$ of read messages where the payload is a read r as defined above. Under the assumptions of Section 5.2 we know that the number of elements in the stream is unbounded. We are able to straightforwardly calculate incremental estimates for metrics such as mean, variance, max, and min, but require more sophisticated structures for computing estimates of rank statistics such as median and other quantiles to maintain operations in bounded space. We use the following update rules for min, max, mean, and variance[197] calculations:

$$min_k(M) = \begin{cases} m_k, & \text{if } m_k < min_{k-1}(M) \\ min_{k-1}(M), & \text{otherwise} \end{cases} \tag{5.3}$$

$$max_k(M) = \begin{cases} m_k, & \text{if } m_k > max_{k-1}(M) \\ max_{k-1}(M), & \text{otherwise} \end{cases} \tag{5.4}$$

$$\mu_k(M) = \mu_{k-1}(M) + \frac{m_k - \mu_{k-1}}{k} \tag{5.5}$$

$$\sigma_k^2(M) = \frac{\sigma_{k-1}^2(M) + (m_k - \mu_{k-1}(M))(m_k - \mu_k(M))}{k-1} \tag{5.6}$$

In order to set up the mechanisms to answer quantile queries the following definitions are used[63]:

- Given a set S of size n, and a quantile $\phi \in [0, 1]$, return $v \in S$ whose rank in sorted S is ϕn.
- An ϵ-approximate ϕ-quantile is a value v whose rank $r*(v) \in [n(\phi-\epsilon), n(\phi+\epsilon)]$.
- A quantile summary is $Q = q_1, q_2, ..., q_l : q_1 \leq q_2 \leq \cdots \leq q_l, q_i \in S, i \in [1, l]$ where each q_i has rank at least $rmin_Q(q_i)$ and at most $rmax_Q(q_i)$ in S, and $rmax_Q(q_1) \leq \epsilon|S|$, and $rmin_Q(q_l) \geq (1-\epsilon)|S|$.
- A quantile summary $Q(\epsilon)$ is ϵ-approximate if it can be used to answer any quantile query with ϵ-accuracy.

We use two approaches for computing quantile summaries, one due to Greenwald and Khanna[69] is able to compute the quantile summary using $O(log(\epsilon n)/\epsilon)$

space, and the other by Shrivastava et al.[172] computes the quantile summary in $O(log(M)/\epsilon)$ when the values are integers in range $[1, M]$. Both algorithms work for a scenario where one node sees all of the data in a stream, but can also be generalized to topologies where the stream is observed by multiple nodes in parallel.

We provide several examples of QC queries of interest that are specified on a data stream:

Average Base Quality - As an assessment of the individual quality of each read we are interested in the average base quality so that we can filter out reads that are of low quality as a whole. We use a Decorator Function construct from Section 5.2.1.

Table 5.2: Definition of q_{av} which computes average base quality for a read

Inputs	$M_{raw} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p)$ as in 5.2.
Operation	$q_{av} = \frac{\sum_{i \in [1,\|r\|]} r.q_i}{\|r\|}$
Outputs	$M_{out} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p, q_{av})$

Base Quality Distribution - The distribution of base quality scores per base position of a read and per sample are of interest to investigate the presence of systemic biases in base quality scores as a function of the position within the read.

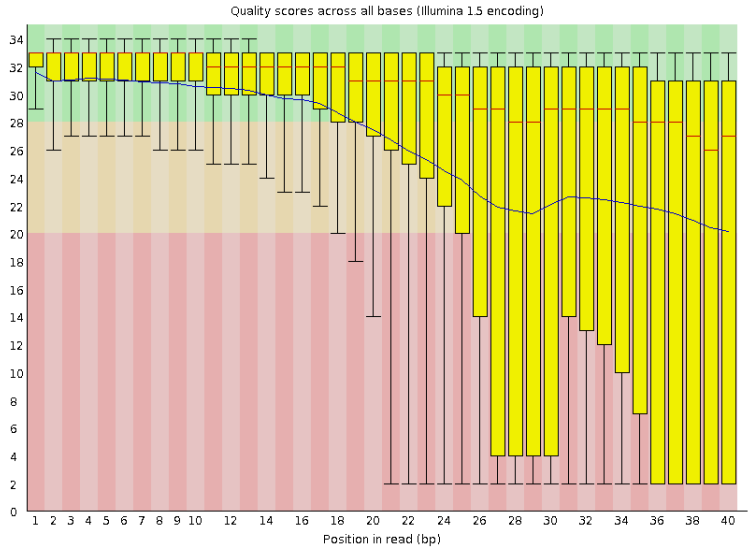


Figure 5.12: Distribution of base qualities per read position, from https://www.bioinformatics.babraham.ac.uk/projects/fastqc. Large quality drop-off can be see towards the end of the read.

Because PHRED-scaled quality scores are integers that fall in a fixed range $q \in [0, 96]$ building quantile summaries using the q-gram[172] approach is the most space-efficient. Because base quality scores need to be aggregated over many reads and tracked for many samples, a service that implements this functionality needs to keep local state, and the operation to update the quantile summaries based on incoming reads follows the Local State Aggregator pattern from Section 5.2.1.

Table 5.3: Definition of *updateQuantileSummaries*()

Inputs	$M_{raw} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p)$ as in 5.2.
Operation	$updateQuantileSummaries(r)$
Outputs	$M_{out} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (s\_id, Q_{bqd})$.

Since the local state required for storing the quantile summaries may not fit in memory and may need to be persisted to disk, updating the summaries may be too expensive to do for every single read that is observed in a read input stream. Instead, reads may be buffered into a set of reservoirs, triggering an update of the quantile summaries when the reservoir is full. The contents of the reservoir would then be purged and an updated set of quantile summaries $Q_s, bqg = \{q_i : i \in [1, max\_bases]\}$, where each q_i is a quantile summary corresponding to the Base Quality Distribution at a particular read position, issued to the output stream (see Algorithm 4).

Algorithm 4: Updating quantile summaries for Base Quality Distribution.

```
Function updateBQDQuantileSummaries(r) begin
    reservoir ←getReservoir(r.s_id)
    reservoir.addNewRead(r)
    if reservoir.isFull then
        summaries ←getQuantileSummaries(r.s_id)
        for read in reservoir do
            for index, read.q in read do
                updateQGram(summaries[index], read.q)          // per [172]
        purgeReservoir(reservoir)
        outputQuantileSummary(r.s_id, summaries)
```

Insert Size Distribution - The insert size distribution is an important metric because it is not only indicative of the overall quality of a sample's data, but it is also used by structural variant calling to find read-pairs that map abnormally far apart (indicating a deletion), or abnormally close together (indicating an insertion).

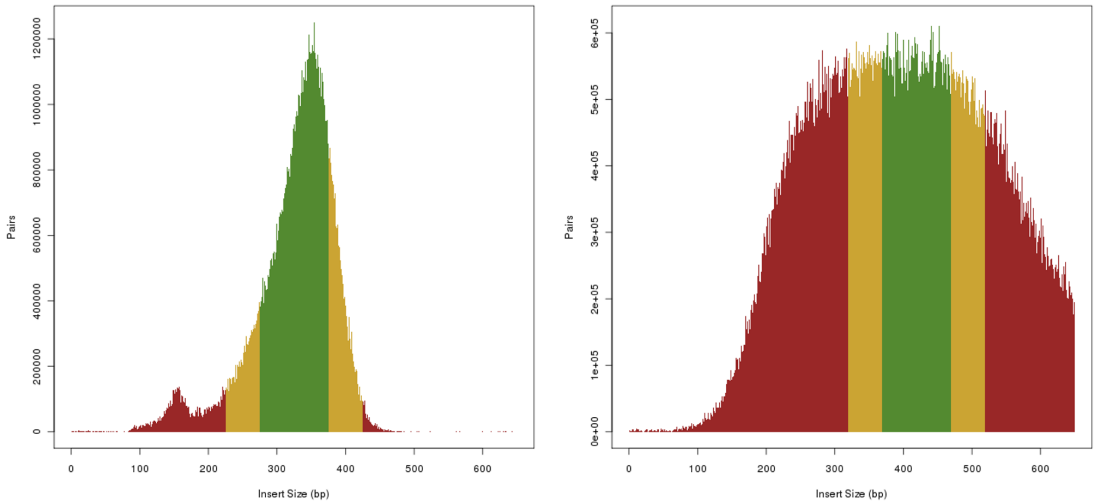


Figure 5.13: Distribution of insert sizes from two ICGC pancreatic cancer patients DO35138 and DO22154.[178]

Calculating this metric requires a stream of read-pairs, where both reads have been successfully mapped to the reference genome. Given a mapped read-pair (r_1, r_2) where each read has a beginning coordinate $r.pos$ and an end coordinate $r.end$, the insert size is $l = r_2.end - r_1.pos$. We are interested in the mean, variance and quantiles of the insert size distribution. Because the insert length can be any size the quantile summary method of Greenwald and Khanna[69] is most appropriate for the quantiles. Read pairs are observed on the input stream and buffered in per-sample reservoirs. When a reservoir is full the read pairs are used to update and output an appropriate insert size distribution mean, variance, and quantile summary.

Table 5.4: Definition of $updateInsertSizeDistribution()$

Inputs	$M_{pair} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$, and $r = (s\_id, r\_id, b, q, f_p)$ as in 5.2.
Operation	$updateInsertSizeDistribution(s\_id, r_1, r_2)$
Outputs	$M_{out} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (s\_id, \mu_{isd}, \sigma^2_{isd}, Q_{isd})$.

Other QC metrics, such as those measuring GC Content Distribution, Read Length Distribution, etc. can be collected analogously.

5.3.2 Alignment

Section 2.2.2 of Chapter 2 provides an overview of the existing approaches in the extremely important and computationally intensive genome alignment stage of the overall NGS processing pipeline. In the overall data flow diagram (Figure 5.10), alignment is primarily responsible for transforming Raw Reads into Mapped Reads, but is also used in QC (insert size distribution, sample contamination) as well as in

Algorithm 5: Updating metrics for Insert Size Distribution.

```
Function updateInsertSizeDistribution(s_id, r1, r2) begin
    pairReservoir ←getReservoir(s_id)
    pairReservoir.addNewReadPair(r1, r2)
    if pairReservoir.isFull then
        summary ←getQuantileSummary(s_id)
        mu ←getMu(s_id)
        sigmaSq ←getSigmaSq(s_id)
        for (r1, r2) in pairReservoir do
            insertSize ←r2.end − r1.pos
            updateQuantileSummary(summary, insertSize)        /* per [69] */

            newMu ←updateMu(mu, insertSize)                   /* using Eq. 5.5 */

            newSigmaSq ←updateSigmaSQ(sigmaSq, mu, newMu, insertSize)
             /* using Eq. 5.6 */

        purgeReservoir(pairReservoir)
        outputInsertSizeDistribution(s_id, newMu, newSigmaSq, summary)
```

the construction and evaluation of local haplotypes for variant calling. In this section we describe the adaptation of already established read mapping best practices to the stream and services based domain of Rheos. Because of the generally independent nature of individual read observations (except for read pairs), this read mapping problem is highly amenable to a stream-based approach. We begin by enumerating and describing the types of alignment tasks that the Rheos framework needs to be able to accomplish and follow up by describing how these tasks will be performed within Rheos.

Single read alignment to reference - Given a representation of the human reference genome we are interested in finding a coordinate relative to this reference where the given read best matches. If we are not able to find a high quality mapping the read should be flagged as unmapped.

Read pair alignment to reference - Given a pair of reads and an estimate of fragment size, attempt to find a high quality consistent mapping for both reads in a window around the expected fragment size.

Single read alignment to multiple references - Given a read and a database of several genome references from multiple species determine if the read has a high quality mapping in any of the references. This can be used for assessing sample contamination.

Candidate haplotype alignment to reference - Given a candidate haplotype i.e. a contiguous (potentially long) sequence locally assembled from a set of reads, align the sequence to the reference genome to identify locations of potential variation.

Single read alignment to list of alternative haplotypes - Given a read and a list of alternative haplotypes for a region align the read to all of the haplotypes in the list to determine which haplotype is best supported by the reads.

Single read split-alignment to reference - Given a read that does not align well to a contiguous region of the reference, search for an alignment where individual pieces of the read might align to separate and possibly distant locations on the reference, suggesting that the read spans a region of structural variation.

Single read alignment to reference Assume we are observing a data stream of raw unmapped short (<500 bp) reads $M_{raw} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p)$ as in 5.2. Using an existing human reference genome, such as *GRCh38*, we would like to align each read in $m.payload$ to produce two new output streams M_{aln} and M_{unaln}, depending on alignment results. M_{aln}, will contain messages from M_{raw} with additional attributes related to the alignment, as described in Section 2.2.1, replicating the information contained in a SAM alignment record (see Table 2.3). M_{unaln} shall contain reads from M_{raw} that failed to align to a reference sequence, with an additional flag $unmapped = true$.

For M_{aln} the following values are computed:

rname - name of the reference contig read is aligned to.

pos - 1-based position offset of the left end of the read alignment to the contig specified in qname.

mapq - Phred-scaled mapping quality representing probability that the read is misaligned.

cigar - CIGAR string (as in SAM specification Section 2.2.1).

flags - a tuple of flags as specified in the FLAG field of a SAM record.

Given that BWT and FM-index based algorithms[95],[105] have currently shown the best balanced performance characteristics on both simulated and real data (see Figures 2.16, 2.15) and the fact that these algorithms are amenable to straightforward parallelization, as in [96] for example, we adopt this approach as well in Rheos. Using this approach we will be able to locate k locations for potential matches of a seed d_i of read sequence $r.q$ in $\mathcal{O}(|r.q.d_i| + k)$ time using a data structure that takes $\mathcal{O}(|R|)$ space for a reference genome R. The seeds will then be extended using a version of Smith-Waterman[175] dynamic programming based alignment with affine gap penalites that can be performed in $\mathcal{O}(|R||r.q|)$ time and $\mathcal{O}(|r.q|)$ space based on [139] and [54]. The overall processing pipeline closely follows Figure 2.14 i.e. given a read r:

- Using Algorithm 3 find a set E of SMEMs of $r.q$[105] using the FMD index formulation described in Section 2.2.2 in relation to BWA.
- Organize SMEMs e_i in E into co-linear chains of the form $C_i = \{(e_j, e_{j+1}, p)\}$, where e_j and e_{j+1} are neighbouring SMEMs in the chain and p is the mapping

coordinate of e_j. Here co-linearity means that the SMEMs are in the same order and orientation on the query read and the reference genome, and a chain of maximal length is selected at each genomic location for the following processing step.

- For each chain, complete the read alignment between chain seeds using a vectorized SIMD-enabled implementation of Smith-Waterman local alignment, as in [54].
- Output alignment with the highest alignment score (based on number of mismatched bases and number of secondary alignments) if it's above a minimum quality threshold.

Because of the FMD index formulation used in finding seeds, both the read and its complement are considered at the same time. *rname* and *pos* values are straightforwardly obtained from the coordinate and name of the leftmost position of the winning alignment. The *cigar* string (see Figure 5.14) is produced directly as a concatenation of the winning dynamic programming paths, and the SMEM seeds which are exact matches. *mapq* converts alignment score to Phred-scale, and $flags$ encode a set of boolean values as per the SAM spec. This provides all of the information necessary for the output mapping.

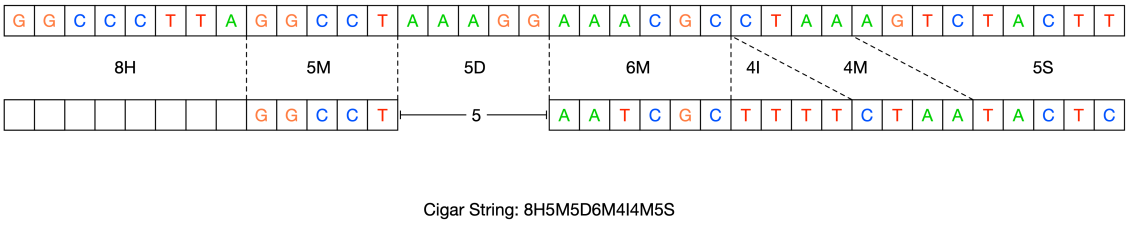


Figure 5.14: Example of an alignment CIGAR string.

If a co-linear set of seeds is not available, or the alignment score fails to reach the predetermined minimal score threshold then the read is emitted into the unmapped read stream with an appropriate flag set.

Table 5.5: Definition of $mapToReference()$

Inputs	$M_{raw} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p)$ as in 5.2.
Operation	$mapToReference(r, ref\_id)$
Outputs	$M_{aln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, ref\_id, r\_id, b, q, f_p, rname, pos, mapq, cigar, flags)$
	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, r\_id, b, q, f_p, unmapped = true)$

The reference pointed to by ref_id when $mapToReference()$ is invoked consists of the typical data structures required by the FM Index i.e. the reference BWT, suffix

array, occurrence array, as in [56], but encoding both the forward and reverse complement of the reference sequence as in [105] to produce the FMD index. Because the reference sequence is static, updated less frequently than once a year, the requisite data structures can be computed once offline and stored in secondary storage. During service initialization they are loaded in RAM and kept memory-resident for the duration of the operation of the service. Because, once computed, the reference index is read-only it can be accessed in a thread-safe manner by multiple concurrent threads without the need for explicit concurrency management. Because of the embarrassingly-parallel nature of read alignment this operation can be scaled up as necessary simply by adding servers, provided additional computational resources exist, and network capacity is not exhausted.

Read pair alignment to reference Since paired-end sequencing produces two reads that represent the opposite ends of a single molecule of approximately known size (known as insert size) it is possible to use knowledge about the insert size distribution along with corresponding pairs of reads to improve the quality of mapping for these reads, and even rescue mappings for reads that do not have a high quality unique mapping themselves but are anchored by a high quality mate. The mapping operation in this context proceeds similarly to the already described single read mapping but requires an input stream of read pairs along with a stream of sample QC metrics, including the empirical insert size distribution (as described in Section 5.3.1).

Depending on the result of the mapping operation for each read in a pair the output stream will contain pairs of reads of type
$r_{aln} = (s\_id, r\_id, b, q, f_p, rname, pos, mapq, cigar, flags)$
or
$r_{unaln} = (s\_id, r\_id, b, q, f_p, unmapped = true)$.

Table 5.6: Definition of $mapPairToReference()$

Inputs	$M_{pair} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$, and $r_i = (s\_id, r\_id, b, q, f_p)$ as in 5.2. $M_{qc} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (s\_id, \mu_{isd}, \sigma^2_{isd}, Q_{isd})$
Operations	$mapPairToReference(r_1, r_2, ref\_id)$ $updateInsertSizeDistribution(s\_id, \mu_{isd}, \sigma^2_{isd})$
Outputs	$M_{aln} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$ and each read is of type r_{aln} $M_{unaln} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$ and each read is of type r_{aln} or r_{unaln} depending on the outcome of mapping.

In order to perform the $mapPairToReference()$ operation the service needs to have information about the insert size distribution for the sample the read pairs

are originating from. It is notified with updated information about the insert size distribution by the *updateInsertSizeDistribution*() operation which is subscribed to the appropriate message stream from the Read QC Service. This information is stored locally for each sample and may be cached. Assuming that the insert size distribution is approximately gaussian with mean μ_{isd} and variance σ^2_{isd}, which is the case for well-behaved samples, a scoring metric can be constructed that favours paired alignments that fall close to the expected insert size. This metric helps select among non-uniquely mapping reads. For instance, BWA-MEM uses the following metric:

$$S_{ij} = S_i + S_j - \min -a \log_4 P(d_{ij}), U \tag{5.7}$$

Here S_i and S_j are the alignment scores for the individual reads in the pair obtained by single-end mapping via $mapToReference(r, ref\_id)$. d_ij is the insert size implied by the mapping, $P(d_{i,j})$ is the probability of observing an insert size larger than d under the assumption $D \sim \mathcal{N}(\mu_{isd}, \sigma^2_{isd})$, a is a matching score, and U is a thresholding constant. Thus, given a set of possible mapping locations for each read in a pair, the joint mapping that maximizes the pairing metric is chosen.

$mapPairToReference()$ outputs read pairs to two output streams. If both reads are mapped successfully then they are output to the M_{aln} stream. If at least one of two reads in a pair does not have a high quality mapping assigned through the paired mapping process, the pair is emitted through the M_{unaln} stream.

Pairing information is vital to downstream variant calling because it can both help with fragment assembly, when reads are properly paired and mapped, and signal the location of potential structural variants when reads are not properly paired and mapped within expected distances.

Single read alignment to multiple references When high average base quality reads fail to map to the reference genome this can be the result of sequencing errors, genomic rearrangements or sample contamination. It's important to be able to detect contamination both at the individual read level, to discard such reads from the analysis, lest they lead to spurious variant calls, as well as at the sample level, where samples with an overabundance of contaminated reads may need to be discarded completely due to low confidence in the resulting call set. Contamination may occur in a variety of ways including sample-swap between tumour and normal DNA samples, between individuals, or accidental introduction of DNA from other species including other mammals, plants, bacteria, and viruses.

Because a relatively large collection of species' reference genomes has already been built up, one relatively simple way of detecting cross-species contamination is to classify reads by how well they align to a database of known references, if they fail to align to the human reference with a high enough quality. This can be accomplished with a general-purpose aligner like Bowtie or BWA, as well as with some purpose built approaches like Kraken[201] or Centrifuge[90]. While general purpose aligners appear to offer similar specificity and sensitivity to the purpose built tools, they are

an order of magnitude slower and require approximately five times more RAM for storing the reference database[90]. Kraken breaks down the references into k-mers and uses k-mer counting to build a database of k-mer abundances and a taxonomic tree of species that can be searched for classification purposes, while Centrifuge uses the same FM index approach taken by Bowtie and BWA, but merges genomes from similar species into a graph structure and applies exact k-mer search when classifying reads from a sample (which is why it's faster than general purpose aligners since it avoids the expensive dynamic programming step).

Since the FM index implementation is necessitated by other alignment use cases it makes sense to adopt this approach for contamination analysis as well.

Table 5.7: Definition of $mapReadToReferenceDB()$

Inputs	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, r\_id, b, q, f_p, unmapped = true)$
Operations	$mapReadToReferenceDB(r)$
Outputs	$M_{aln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, ref\_id, r\_id, b, q, f_p, rname, pos, mapq, cigar, flags, contamination = true)$
	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, r\_id, b, q, f_p, unmapped = true, contamination = false)$

The key to performing the $mapReadToReferenceDB()$ is the construction of the reference DB itself. Assuming there are N genome references $G = \{g_i : i \in [1, N]\}$, create a joined reference $T = g_1\$_1g_2\$_2...\$_{N-1}g_N$ by concatenating all of the references interspersed with a set of sentinel characters $\{\$_i : i \in [1, N-1]\}$ that don't occur in G, are lexicographically smaller than any character in G, and having $\$_i < \$_j$ iff $i < j$. Record the index d of the occurrence of each $\$_i$ in T, such that $T[d_i] = \$_i$. Then construct and search the FM Index of T as usual. Given that a read r maps to some location l in T, $|\{d_i : d_i < l\}|$ identifies the rank of reference genome g_j in T that r belongs to, thus identifying the reference of origin. The alignment process itself may be tuned to only perform exact matching as in Centrifuge, or carry out dynamic programming alignment as well, depending on cost and performance requirements. Reads in the output stream that are successfully aligned to a reference that is part of the database of contaminants are emitted with the corresponding reference ID and the contamination flag set.

Candidate haplotype alignment to reference Many of the modern leading variant callers[42, 64, 156] rely on local assembly of alternative haplotypes to detect variants with high accuracy. The haplotypes that are produced may be up to hundreds of kilobases long. These are aligned to the reference genome to reveal the location of potential variants that are implied by each alternative haplotype. The variants are subsequently evaluated in the context of read evidence to select the variants that are best supported by the data. This brings about the problem of aligning

long sequences, potentially with many mismatches, to a reference genome. Additionally, long read technologies such as PacBio SMRT[155] and Oxford Nanopore Technologies[119] routinely generate reads that are many kilobases long and may enjoy increased use in sequencing projects due to their ability to resolve structural variation.

BWT and FM Index seed-and-extend approaches[104, 95] that have been successful for shorter read lengths have proven to be exceedingly slow or crash altogether for long reads, because of the extra backtracking required by the dynamic programming step in the presence of read sequence divergence from the reference, whereas a k-mer hash table chaining approach based on minimizers[157] was shown to be highly accurate and performant[108]. To take advantage of the improved alignment performance on long reads (30-70 times faster than BWA-MEM) we adopt Minimap's approach when aligning locally assembled alternative haplotypes to the reference sequence.

Table 5.8: Definition of $mapLongReadToReference()$

Inputs	$M_{asm} = \{m_i : m_i = (header, payload)\}$ where $m.payload = r = (s\_id, r\_id, b, q, f_p)$.
Operation	$mapLongReadToReference(r, ref\_id)$
Outputs	$M_{aln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, ref\_id, r\_id, b, q, f_p, rname, pos, mapq, cigar, flags)$

As described in 2.2.2 and [108] the reference index in this case consists of a hashmap of minimizers (see Figure 2.17) of the reference genome where the key is the minimizer sequence and the value is a list of locations of that sequence. As with BWT based indexes, the structure is static over time, can be pre-generated, stored on disk, and loaded into memory during operation. When a read arrives on the input stream M_{asm} the $mapLongReadToReference()$ operation breaks the read into a set of minimizers and searches these against the reference hash table. Sets of co-linear exact matches are formed into chains. Sequences that are between matches in a chain are locally aligned with dynamic programming.

Single read alignment to list of alternative haplotypes NEED TO THINK ABOUT HOW THIS SHOULD WORK

Single read split-alignment to reference When a high average base quality read fails to align to the reference with high mapping quality one of the reasons may be that the read covers a DNA double-strand breakpoint in the sample, at the site of a structural variant that is longer than the size of the read. Such a read, if it were to be aligned to the sequence of the sample would map normally, but when mapped to the reference sequence, one part of the read maps to one location in te genome, and another part maps to a different location (see Figure 5.15). Such

reads are called split-reads and they form an important source of signal for precise detection of structural variant breakpoints[153, 100].

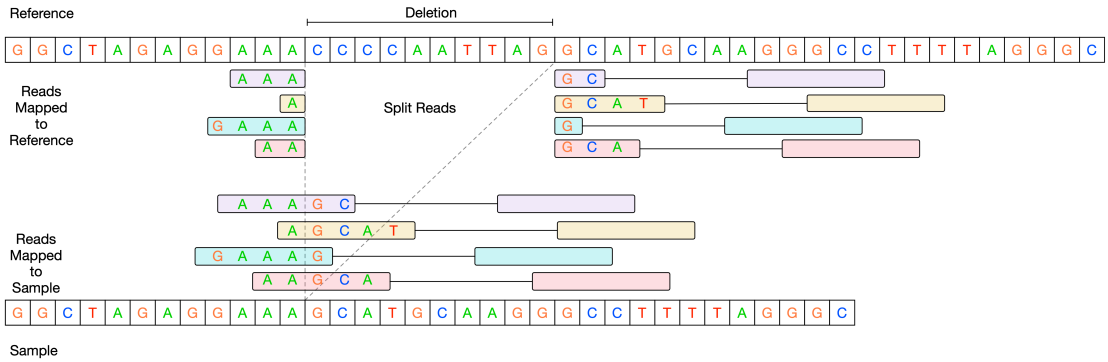


Figure 5.15: Split-reads at the site of a deletion breakpoint.

Although the techniques used for single read mapping, described above, can be used for split-read alignment there are several issues to consider that make the creation of a separate $mapSplitReadToReference()$ operation desirable.

Since a split-read consists essentially of two or more separate pieces of varying lengths that may map to distant locations in the genome and have different orientations, determining where to "split" the read affects the quality of the subsequent mapping, as improper splits will generate many mismatches. $mapToReference()$ is intended to be optimized for speed on well-behaved reads which constitute most of the data that is seen for a given sample, but as a result make a number of assumptions, such as a single k-mer size and co-linearity of matches, that are invalid or sub-optimal for split-read alignment. Furthermore, the data representation for an aligned read in $mapToReference()$ of a single sequence b of bases, q of base qualities, and *cigar* string representing the alignment is not well suited to representing an alignment consisting of multiple, possibly distant, pieces of possibly different orientations, as this is not readily representable in CIGAR.

In current software that relies on writing SAM files as a medium for alignments, split-reads are represented as multiple records with the same ID, where one record is arbitrarily chosen to be primary (or representative) and others are marked supplementary (via an 0x800 FLAG value). Each record represents a separate piece of the split read where the unmatched portion is masked out via soft-clipping in the CIGAR string or is hard-clipped (see Section 2.2.1 for details on the SAM format). The details of the split-read alignment are captured in each read's optional SA tag in the form of a semi-colon separated list of strings that describe the supplementary alignments. In Figure 5.16, for example, there is a split read pictured that was obtained from a sample on chromosome 20 of the NA12878 individual in the 1000 Genomes Project, aligned by BWA-MEM, where one portion of the read aligns on the positive(+) strand, while another portion of the read aligns on the negative(-) strand some distance away. The yellow area in the figure describes the first part of the read which is labeled primary. The read is 250 basepairs long and the CIGAR string is 68S81M101S, indicating 68 soft-clipped bases, followed by an 81 basepair match to the reference, and followed by another 101 soft-clipped bases. The 101

basepair segment has low base quality, so it likely consists primarily of sequencing errors, but the 81 basepair segment has base qualities around 30, which is relatively high. The SA tag reads - SA:Z:20,42589216,-,179S71M,60,0;, indicating that there is a secondary alignment mapped to chromosome 20, starting at position 42589216, on the negative strand, with CIGAR string 179S71M (179 soft-clipped bases followed by a 71 basepair match to the reference). This is indeed the case and the corresponding read is pictured in Figure 5.16 shaded purple. Investigating the SAM record for this mapping reveals that the actual record has CIGAR string 179H71M, i.e. the bases at the beginning are actually hard-clipped, not soft-clipped (soft-clipped bases are excluded from the alignment but the sequence is still preserved in the file record, whereas hard-clipped bases are excluded from alignment and the sequence is not recorded in the file). Such a split-read might imply the presence of deletion proximal to an inversion in the sample with respect to the reference sequence, although the evidence from the read should be investigated in light of the evidence from all of the other reads that overlap this locus to make an accurate determination of the presence of a potential variant in that location.

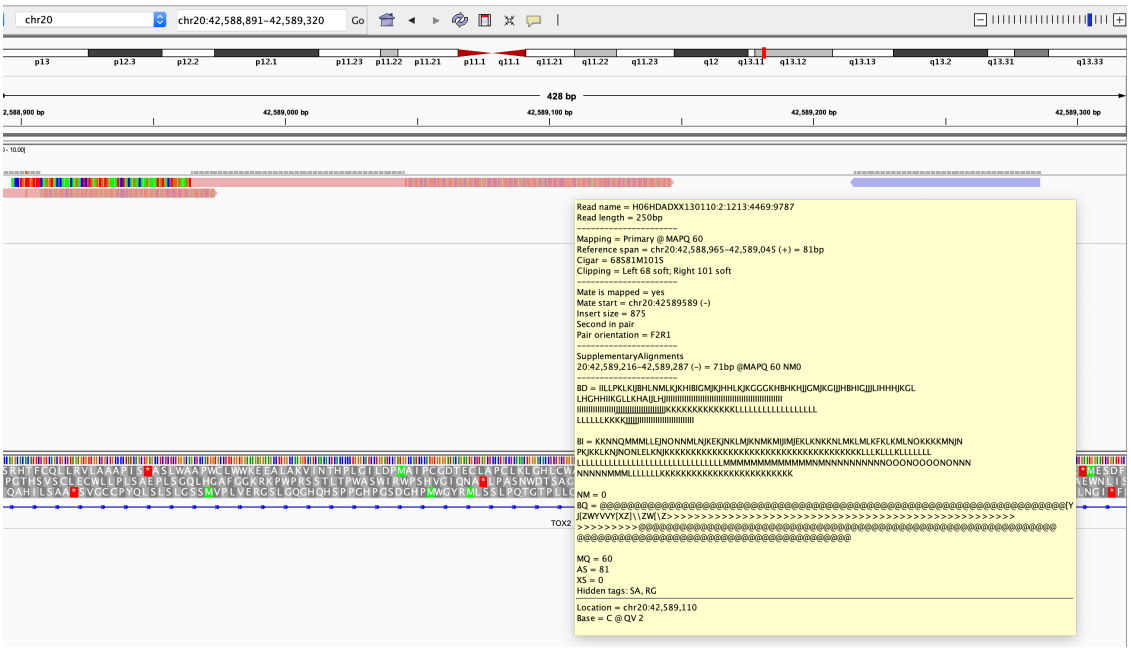


Figure 5.16: IGV viewer rendering of a split-read on chromosome 20 from NA12878 sample.

There are several issues with this scheme in addition to the apparent disagreement between duplicated information in the two records. The duplication is wasteful of space, and prone to errors as demonstrated above. There is no apparent ordering of the segments of the read that map to different locations because the next or previous segment are not specified for a given record, instead all secondary records point back to the primary. There is no well defined schema for the optional SA tag other than specifying a record separator. Thus, formats and contents of the encoding may vary between tools and even from record to record, without the ability to validate correctness of the encoding. Additionally, because the records in the SAM file are typically either in random order or coordinate-sorted order, the records for a split-

read are not easily locatable and in the worst-case the entire file may need to be scanned in order to get the full split-read alignment for a single read.

As Rheos does not rely on traditional genomics file formats for encoding, we chose a split-read representation that alleviates the issues above and encodes the entirety of a split-read alignment in a single record. Define:

$$r_{split} = (s\_id, ref\_id, r\_id, b, q, f_p, aln) \tag{5.8}$$

Here $s\_id, ref\_id, r\_id, b, q, f_p$ are as in Equation 5.2 defining a raw read. $aln = (a_1, a_2, ..., a_n)$ - an ordered list of alignment segments, where each $a_i = (i, rname, pos, offset, len, mapq, cigar, strand, flags)$, and:

i - ordinal number of a_i in aln.

rname - name of reference contig to which a_i maps.

pos - position on $rname$ where a_i maps.

offset - offset on r_split where a_i begins.

len - length of a_i.

mapq - mapping quality.

cigar - CIGAR of the alignment.

strand - alignment strand, 0 for positive, 1 for negative. Strand disagreement between r_{split} and a_i indicates sequence inversion.

flags - various flags as in SAM format.

Table 5.9: Definition of $mapSplitReadToReference()$

Inputs	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, r\_id, b, q, f_p, unmapped = true)$.
Operation	$mapSplitReadToReference(r, ref\_id)$
Outputs	$M_{split} = \{m_i : m_i = (header, r)\}$ where r is as defined in Equation 5.8.
	$M_{unaln} = \{m_i : m_i = (header, r)\}$ where $r = (s\_id, r\_id, b, q, f_p, unmapped = true, split\_align = false)$

The split-read alignment operates on reads that emerge unmapped from the regular alignment stage via $mapToReference()$. Alignment proceeds using the same general FM Index based framework as $mapToReference()$ but here each read is broken down into a series of progressively smaller k-mers. Each k-mer is aligned

individually, and the alignment that maximizes aggregate alignment score across all k-mer sizes and candidate alignments is chosen.

The alignment operations that have been considered in this section present a set of distinct challenges on the basis of the gamut of query sequence and reference database size combinations. Each scenario presented admits optimization with respect to its specific parameters. Existing aligners typically attempt to solve all or most of these problems with a single approach, and thus suffer from inability to fully optimize each specific use case. Our treatment, that separates the different alignment scenarios into distinct operations, with well delineated optimization criteria, and even into potentially separate services that may run on separate physical machines, allows us to fully optimize each operation without negatively impacting others.

5.3.3 Simple Germline SNP Calling

SNPs are the most abundant and widely studied type of genetic variant[28]. Section 2.2.4 provides an overview of existing methods for germline SNP calling. For a human diploid sample when calling variants on one of the autosomes (chromosomes 1-22), germline SNP calling comes down to selecting between three alternative models for each genomic locus based on the set of reads that are observed - homozygous reference, heterozygous, or homozygous variant (see Figure 5.17).

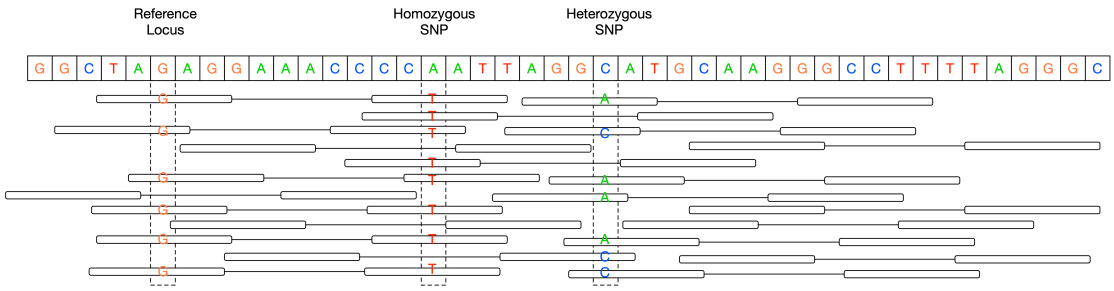
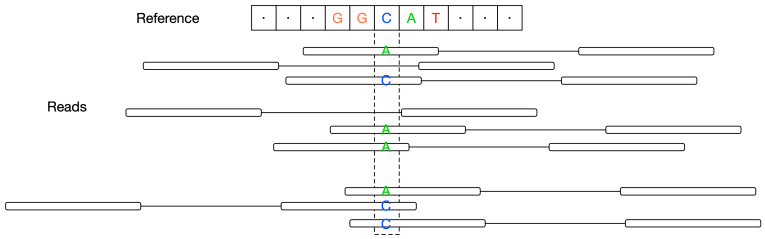


Figure 5.17: Examples of genomic loci that match the reference sequence, have a heterozygous SNP, and a homozygous SNP.

Although the most accurate currently available variant callers model the region around a SNP via local assembly of haplotypes[42, 156, 64], a simpler approach assumes that every locus is independent and can still achieve high accuracy. For instance, samtools[103], which makes the independence assumption, has recently been compared to several local assembly based methods in [116] and was shown to have a slightly higher false-negative rate than platypus, freebayes, and Haplotype Caller, but also a slightly lower false-positive (FPPM = false positives per million bases) rate (see Figure 2.25). Additionally, in [156] Table 1, samtools has the highest sensitivity, but also the highest false discovery rate (FDR) on SNPs. Thus, the simpler independent locus model should be sufficient for calling germline SNPs and is of interest in the context of the data streaming approach of Rheos.

The random order in which reads appear in a Rheos data stream means that a new approach is required to successfully implement germline SNP calling on this data. Specifically, all of the existing tools for variant calling assume that all of the reads for a given sample have been observed, mapped to the reference, sorted by increasing reference coordinate, and stored in a SAM or BAM file. Variant calling (see Section 2.2.4) proceeds by traversing the data in a coordinate-ordered fashion, loading all of the reads that overlap a given locus into memory, into a structure called a read pileup (see Figure 5.18), and evaluating a set of alternative models for each locus, typically in a Bayesian framework, selecting either the maximum-likelihood estimate (MLE) or the maximum a posteriori (MAP) estimate as the called genotype.



Locus

A
C

Chromosome	2
Position	120238
Genotype	0/1
Genotype Quality	30
Genotype Likelihoods	-323.03,-99.29,-802.53
Ref	C
Alt	A
Read Depth	7
Ref Count	3
Alt Count	4
Sum of Ref Qual	92
Sum of Alt Qual	120

Figure 5.18: A read pileup over a locus with a heterozygous SNP.

The Bayesian framework that is typically adopted follows this form (see also Section 2.2.4):

$$P(G|D) = \frac{P(G)P(D|G)}{\sum_i P(G_i)P(D|G_i)} \tag{5.9}$$

Here some prior distribution $P(G)$ of genotypes at a locus is assumed. This can be an uninformative prior (for instance one that assigns equal probability to all possible genotypes), or it can be a prior based on a population genetics model, or empirically obtained distribution of genotypes in a given population. The genotype likelihood $P(D|G)$ under assumptions of independent observations (reads) and independent errors, factors into a product of individual observation likelihoods i.e. $P(D|G) = \prod_j P(D_j|G)$. Each observation $P(D_j|G)$ is further subject to a sequencing error probability ϵ_j, derived from the recorded base quality for the read over the locus being considered. Thus, $P(D_j|G) = 1 - \epsilon_j$ when $D_j = G$ and $P(D_j|G) = \epsilon_j$, otherwise. Genotype likelihoods are then computed as in Equations 2.7 for samtools,

and similarly Equations 2.13, 2.18, 2.15 for freebayes, Equation 2.20 for GATK, and Equation 2.21 for platypus. A site is called variant if the probability of having at least one non-reference allele at the locus exceeds the probability of having 0 non-reference alleles. A genotype is assigned if the ratio of the genotype with the highest likelihood to the genotype with the second highest likelihood exceeds some pre-determined constant, i.e., assuming genotype $g \in [0, 1, 2]$ denotes the number of reference alleles at the locus, and, for example, given $\mathcal{L}(g_2) > \mathcal{L}(g_1) > \mathcal{L}(g_0)$, assign the genotype $g = 2$ if $\frac{\mathcal{L}(g_2)}{\mathcal{L}(g_1)} > c$, where $c > 1$ is some user-set threshold. The calling proceeds locus-by-locus in coordinate-ordered fashion, and the results are written to a file in VCF format (see 2.2.1) where each row represents a variant.

The process above is efficient in that each locus is looked at only once in order to be able to make a call, but it is inefficient in that all of the data needs to have been observed, mapped, and sorted in order to even begin calling. The streaming approach that we adopt in Rheos allows us to begin calling immediately as we start seeing data, in return for needing to re-evaluate a locus multiple times as new data arrives. Because each read is immediately integrated into the model once it is observed, the state of our model of the genome is at all times consistent with all of the read evidence that has been observed to date. Each locus then simply provides an estimate of the underlying genotype based on the data that is available at that time. Initially, these estimates may be inaccurate, but accuracy improves as more and more data arrives and the estimates are refined in an iterative fashion.

We are thus interested in an iterative update rule that, given a current set of genotype likelihood estimates at a locus, that incorporate evidence from n reads that have already been observed, when the read $n + 1$ arrives, is able to produce an updated set of genotype likelihoods that is consistent with evidence from all $n + 1$ reads, as well as any other auxiliary structures necessary for producing a variant call record. To do this we adopt a sequential Bayesian framework, where the posterior distribution of genotype likelihoods at step n becomes the prior (albeit unnormalized) for step $n + 1$.

Let:

Symbol	Description
R	The reference sequence.
S	The sample sequence.
$D = \{d_i : i \in [1, n]\}$	The set of n observations seen to date.
d_i	i'th read observation.
g_i	Genotype of sample S at locus i. $g \in [0, 1, 2]$ counts the number of reference alleles at locus i
$P_n(G_i)$	Probability of genotype $G = g$ at locus i, after seeing n observations d that overlap locus i.
$\mathcal{L}(d_n \mid G_i)$	Genotype likelihood function at locus i for the n'th observation.

We take $P_0(G_i)$ to be the prior probability of G at i. After observing the first data point d_1 which has base quality (probability of error) ϵ_1, we calculate:

$$\mathcal{L}(d_1|G=0) = 1-\epsilon_1$$
$$\mathcal{L}(d_1|G=1) = \frac{1}{2}$$
$$\mathcal{L}(d_1|G=2) = \epsilon_1$$

if d_1 is different from reference, and:

$$\mathcal{L}(d_1|G=0) = \epsilon_1$$
$$\mathcal{L}(d_1|G=1) = \frac{1}{2}$$
$$\mathcal{L}(d_1|G=2) = 1-\epsilon_1$$

if d_1 matches the reference. Consequently, by Bayes' Rule:

$$\mathcal{L}(G=0|d_1) \propto \mathcal{L}(d_1|G=0) * P_0(G=0)$$
$$\mathcal{L}(G=1|d_1) \propto \mathcal{L}(d_1|G=1) * P_0(G=1)$$
$$\mathcal{L}(G=2|d_1) \propto \mathcal{L}(d_1|G=2) * P_0(G=2)$$

Now, assuming that we have observed n data points and calculated $\mathcal{L}(G|D_n)$ (likelihood of genotype given n data points D_n). We obtain a new data point d_{n+1} and calculate:

$$\mathcal{L}(d_{n+1}|G=0) = 1-\epsilon_1$$
$$\mathcal{L}(d_{n+1}|G=1) = \frac{1}{2}$$
$$\mathcal{L}(d_{n+1}|G=2) = \epsilon_1$$

if d_{n+1} is different from reference, and:

$$\mathcal{L}(d_{n+1}|G=0) = \epsilon_1$$
$$\mathcal{L}(d_{n+1}|G=1) = \frac{1}{2}$$
$$\mathcal{L}(d_{n+1}|G=2) = 1-\epsilon_1$$

if d_{n+1} matches the reference. Using this likelihood and the posterior from step $n+$ as the prior for step $n+1$, again by Bayes' Rule, we obtain:

$$\begin{aligned}\mathcal{L}(G=0|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=0) * \mathcal{L}(G=0|D_n)\\ \mathcal{L}(G=1|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=1) * \mathcal{L}(G=1|D_n)\\ \mathcal{L}(G=2|D_{n+1}) &\propto \mathcal{L}(d_{n+1}|G=2) * \mathcal{L}(G=2|D_n)\end{aligned} \tag{5.10}$$

Equation 5.10 provides the general update rule for incorporating new evidence about a given locus i having already made and recorded n observations, upon making observation $n+1$. This iterative approach is mathematically equivalent to the batch update approaches such as [103], that collect all of the data about a locus before evaluating genotype likelihoods. The posteriors obtained by equation 5.10 can be used directly to evaluate relative likelihoods of the possible genotypes, and to assign genotypes via a likelihood ratio, for example, but in order to obtain a proper posterior probability mass function over the genotypes they need to be normalized, to obtain:

$$P(G=g|D) = \frac{\mathcal{L}(G=g|D)}{\sum_{g' \in G} \mathcal{L}(G=g'|D)} \tag{5.11}$$

This equation can be used to obtain a probability estimate of any given genotype, having observed any given amount of data about a specific locus.

Armed with the update rule for integrating read evidence we are ready to define a data stream operation of Local State Aggregator type that is responsible for processing a stream of aligned read pairs, updating its local state with the genetic variation evidence contained in the reads, and emitting a stream of updated genomic loci (see Fig. 5.18) that can be used by downstream services for variant calling.

Table 5.10: Definition of $updateLociFromRead()$

Inputs	$M_{aln} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$ and each read is of type r_{aln}.
Operation	$updateLociFromRead(r, ref, current_loci)$
Outputs	$M_{loci} = \{m_i : m_i = (header, L)\}$ where $L = \{l : l = (s\_id, ref\_id, contig, pos, alt, gl_0, gl_1, gl_2, dp, ro, qr, ao, qa)\}$

A Locus l for SNP calling represents a single genomic location for a single sample and a particular reference genome. For each Locus we keep track of the following fields:

s_id - Sample ID.

ref_id - Reference Genome ID.

contig - Name of the reference contig.

pos - Reference-relative position of the Locus on the contig.

alt - The alternative allele at this locus (if any alternative alleles have been observed).

gl_0 - Likelihood of homozygous alternative genotype at this locus, based on the data seen so far.

gl_1 - Likelihood of heterozygous genotype at this locus, based on the data seen so far.

gl_2 - Likelihood of homozygous reference genotype at this locus, based on the data seen so far.

dp - Read depth - number of reads observed at this locus so far.

ro - Number of reference observations at this locus.

qr - Sum of base qualities of reference observations at this locus.

ao - Number of alternative observations at this locus.

qa - Sum of base qualities of alternative observations at this locus.

The Locus Processing Service which is responsible for implementing $updateLociFromRead()$ maintains a local data store of Locus objects, ideally in memory. When a new mapped read arrives via the input stream, the service walks along the read and for each position on the read compares it to the respective position on the reference sequence. It retrieves the Locus corresponding to this position and, depending on whether the read matches, or is different from the reference sequence, updates the Locus genotype likelihoods (and other fields) according to the update rule of Equation 5.10. Those Locus objects that have been updated, and have alternative allele observations (i.e. potential variants) are added to a list of objects that should be emitted by the service as output of $updateLociFromRead()$. Since a read alignment consists of not only matches and mismatches with respect to the reference sequence, but also encodes insertions and deletions as an array of CIGAR elements (see 2.2.2), and the alignment may occur on the forward or the reverse strand of DNA (requiring reverse complementing) special care needs to be taken during read processing to account for all of the possible cases of simultaneous read and reference traversal. Read processing proceeds as in the following algorithms:

Algorithm 6 is responsible for locating the genomic coordinates of those parts of the alignment that are flagged as matches (a match may still have differences from the reference). This is accomplished by traversing the CIGAR elements of the alignment and updating the coordinate offset of the read, the reference, or both, depending on the type of CIGAR element that is encountered. When a CIGAR element corresponding to a match is encountered it is processed by Algorithm 7, which is responsible for updating the set of loci that the matching region overlaps with the evidence for (or against) variation that exists in the read. In Algorithm 7, the DNA strand to which the read maps determines whether the read sequence

Algorithm 6: Process read and reference in tandem to find matching CIGAR elements.

```
Function processReadPair(read_pair, ref) begin
    for read in read_pair do
        strand ← read.strand
        cigar_els ← read.cigar_els
        ref_offset ← 0
        read_idx ← read.query_start
                        /* Alignment does not necessarily start at index 0. */

        for el in cigar_els do
            if el.type = MATCH or el.type = DEL then
                if el.type = MATCH then
                    updated_loci
                      ←handleMatch(read, read_idx, el.length, ref_offest, ref)
                ref_offset ← ref_offest + el.length * strand
                     /* both matches and deletions consume reference sequence.
                  Reads that align on the negative strand consume reference
                  backwards. */

            if el.type! = DEL and el.type! = SOFT_CLIP then
                read_idx = read_idx + el.length      /* All elements but deletions
                  and soft-clips consume read sequence. */
```

needs to be reverse complemented and whether the match offsetting proceeds from the beginning or the end of the reference sequence. After offsets are computed the actual matching always proceeds moving forward (in the direction of increasing coordinates) along the reference sequence. At each location the reference and read sequences are compared and the underlying Locus model is updated based on the update rule of Equations 5.10. Those loci that have a non-zero count of alternative allele observations are added to a list of loci that potentially harbour variants and will be emitted from the Locus Processing Service as a stream M_{loci} for further downstream processing by the variant caller.

The Variant Calling Service - is responsible for actually making SNP variant calls based on a set of user-defined calling criteria, such as a region of interest, variant quality, genotype likelihood, number of supporting observations, etc. The service translates between a set of Locus models maintained by Rheos and an external data format (namely VCF). This service does not operate on a stream of data, but instead acts as a query service that can be invoked by the user ad hoc with specified parameters to answer queries of interest.

For each locus inside the queried region the Variant Calling Service evaluates the set of genotype likelihoods at that locus and calculates normalized genotype posterior probabilities using Equation 5.11. The variant quality is defined to be the Phred-scale transformed probability that a site is homozygous reference, i.e. sites that, based on the read evidence, have a very low probability of being reference,

Algorithm 7: Process a portion of the read that maps to reference as a match, updating all loci that it overlaps.

```
Function HANDLEMATCH(read, read_idx, match_length, ref_offest, ref) begin
    seq ← read.seq[read_idx : read_idx + match_len]
    qual ← read.qual[read_idx : read_idx + match_len]
    if read.strand = 1 then
        ref_start ← read.position + ref_offest − 1
    else
        /* When read is on negative strand sequence is reverse complemented
        and matching starts from end of reference. */
        seq ←REVERSECOMPLEMENT(seq)
        ref_start ← read.end + ref_offest − match_length − 1
    read_ref ← ref[ref_start : ref_start + match_length]
    for pos_idx in seq do
        cur_ref ← ref_start + pos_idx
        cur_locus ←ALL_LOCI.GETLOCUS(cur_ref)
        p_error ←BASEQUALITYTOPROBABILITYOFERROR(qual[pos_idx])
        cur_locus.dp ← cur_locus.dp + 1
        cur_locus.gl_het ← cur_locus.get_het ∗ 0.5
        if seq[pos_idx]! = ref[pos_idx] then
            cur_locus.alt ← seq[pos_idx]
            cur_locus.gl_ref ← cur_locus.gl_ref ∗ p_error
            cur_locus.gl_hom ← cur_locus.gl_ref ∗ (1 − p_error)
            cur_locus.ao ← cur_locus.ao + 1
            cur_locus.qa ← cur_locus.qa ∗ qual[pos_idx]
        else
            cur_locus.gl_ref ← cur_locus.gl_ref ∗ (1 − p_error
            cur_locus.gl_hom ← cur_locus.gl_ref ∗ p_error
            cur_locus.ao ← cur_locus.ro + 1
            cur_locus.qa ← cur_locus.qr ∗ qual[pos_idx]
```

Table 5.11: Definition of $callVariants()$

Operation	$callVariants(region, calling\_threshold)$
Outputs	$outputVCF$

under the Phred-scale transformation end up with a high variant quality (see Equation 2.11). Variants that exceed *calling_threshold* will be emitted as records to the output VCF file. It is possible to envision more sophisticated versions of the Variant Calling Service that provide extended querying and filtering capabilities.

5.3.4 Germline Structural Variant Calling

As Rheos aims to provide comprehensive sample characterization it is not enough to only be able to call single nucleotide polymorphisms, the ability to call other variants, such as indels and structural variants is also required. Here we examine an approach for calling a specific class of structural variants, namely deletions,

using a stream of mapped read pairs made available by the Read Mapping Service's $mapPairToReference()$ operation.

In general, structural variant callers typically make use of three types of evidence for the presence of structural variation - read depth, discordantly mapped reads, and split-reads (see Section 2.2.6). Here we utilize discordantly mapped reads i.e. those read pairs that align farther away from each other than expected in order to assess the presence of sequence deletions in the sample with respect to the reference.

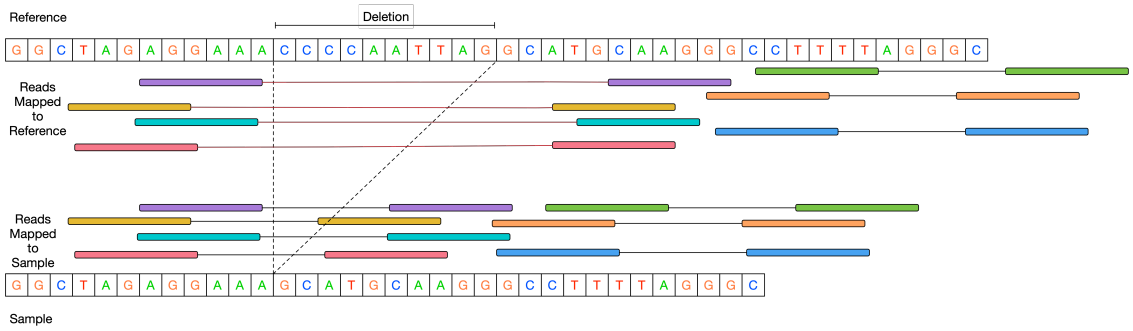


Figure 5.19: Effect on read-pair mapping distance for reads overlapping a deletion.

In Figure 5.19 the sample being sequenced has a deletion with respect to the genome reference, namely the sequence "CCAAATTAG" is deleted. When sequencing with read pairs, the term "insert size" refers to the size of a DNA fragment that is being sequenced, less the size of the sequence adapters on both ends(see 5.20). Thus, assuming that both reads in a pair have been mapped, and have mapped in the proper orientations (5' towards 3' on both strands). If the Read Mapping Service emits the read pair as (r_1, r_2) where r_i are as previously defined and $r_1.pos$, $r_1.end$, $r_2.pos$, $r_2.end$, and assuming without loss of generality that r_1 is the read that maps on the positive strand, then the insert size $l_{r_1,r_2} = r_2.end - r_1.pos$.

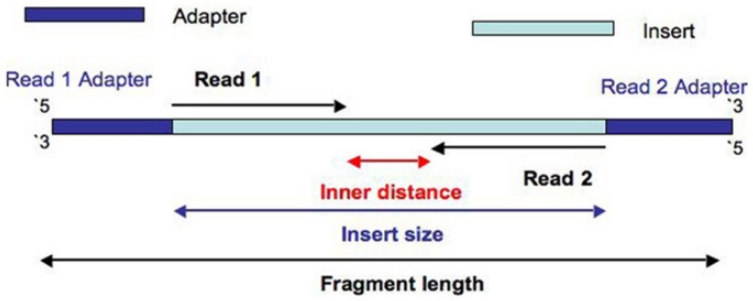


Figure 5.20: Read-pair insert size is the size of the DNA molecule being sequenced minus the length of adapters on both sides[184].

As can be seen from Figure 5.19, for those reads that span the site of the deletion the inferred insert size is larger than the actual insert size (approximately by the size of the deletion), since the read are mapped farther apart on the reference than they are on the sample. Such read pairs are said to be discordantly mapped. Since the insert size distribution for a given sample is relatively predictable (see Figure 5.13),

it is possible to detect the presence of deletions in a sample by identifying clusters of read pairs that have an inferred insert size larger than some predefined threshold. Such an approach is already used by several structural variant callers[153, 100] and we adopt it in Rheos with several key innovations.

Detection proceeds in two stages performed by two different services. The Insert Size Filtering Service is responsible for listening to the general stream of read pairs and filtering out all of the pairs that don't meet the requisite criteria. The Insert Size Clustering Service is responsible for consuming a stream of discordantly mapped read pairs, clustering the reads that are located close to each other, and outputting those clusters that have sufficient evidence for being considered sites of deletions.

For the Insert Size Filtering Service we define the following operation:

Table 5.12: Definition of $filterDiscordantPairs()$

Inputs	$M_{aln} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$ and each read is of type r_{aln}
Operation	$filterDiscordantPairs(r_1, r_2, min\_mapq, mad\_threshold, min\_sample\_size)$
Outputs	$M_{disc} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$ and each read is of type r_{aln}

Here, the $filterDiscordantPairs()$ operation subscribes to the stream of mapped read pairs that is produced by the Read Mapping Service. The goal is to produce a stream of read pairs that are suitable for clustering. The following conditions must be satisfied in order for a pair to be considered suitable:

Both reads in a pair must be mapped - the Read Mapping Service actually emits some read pairs where only one of the reads is mapped into the mapped reads stream.

Both reads in must be mapped to the same contig - Read pairs where individual reads map to different contigs may be indicative of genomic translocations (another type of structural variant), but are not suitable for the detection of deletions, since different contigs are actually different physical molecules.

Both reads must have mapping quality $\geq$ min_mapq - A mapping quality of at least 30 (the default value) on the Phred scale, for instance, implies a probability of no more than 10^{-3} that a read is mapped to the wrong location on the reference.

Insert size must be > MAD * mad_threshold - the Median Absolute Deviation is taken as a measure of the central tendency of the distribution of insert sizes that is robust against extreme outliers, which can be quite common. Typical fragment size of the DNA that is being sequenced is in the hundreds of nucleotides long, whereas certain inferred insert sizes can reach millions of bases due to structural variants or spurious mappings. Thus, it is desirable to have a metric similar to variance that will not be sensitive to these extreme

values. Given a set of insert sizes $L = \{l_i : l_i = r_2.end - r_1.pos\}$, define $MAD(L) = median(\mid l_i - median(L) \mid)$. The cutoff of mad_threshold*MAD places a lower bound on the size of deletions that can be detected with this method. A value of 5 is arbitrarily chosen as the default for *mad_threshold*.

Since obtaining an accurate estimate of median insert size and MAD requires observing a sample of read pairs, the Insert Size Filtering Service does not emit any reads at first, but accumulates observations in a local data store while collecting enough information to obtain reliable estimates of these metrics. *min_sample_size* (10^5 by default) read pairs are collected before filtering begins. Once the metrics are obtained, collected reads are filtered first and released, before processing any of the remaining reads on the stream.

Read pairs that satisfy all of the conditions above are emitted in an output stream for consumption by the Insert Size Clustering Service. The main points of control for the user are the *min_mapq* and *mad_threshold* parameters. Decreasing *min_mapq* increases sensitivity by allowing more reads through while simultaneously decreasing specificity. Likewise, decreasing *mad_threshold* increases sensitivity and allows detection of smaller size deletions,at the expense of allowing "less surprising" insert sizes through the filter. Choosing a higher *min_sample_size* provides more accurate estimates of median insert size and MAD at the expense of a higher delay before producing filtered output, and higher peak memory requirements for the Insert Size Filtering Service. Ideal parameter values may be project dependent.

The Insert Size Clustering Service is responsible for consuming the stream of discordantly mapped reads and figuring out locations of read clusters in genomic space that may be the site of potential deletions. Those read clusters that have sufficient evidence are then emitted as variant calls. Because discordantly mapping read pairs are fairly rare (out of 6705731 mapped read pairs on Chromosome 20 of sample NA12878 analyzed as part of this work, only 2663 or 0.04% successfully passed through the Insert Size Filtering Service with $min\_mapq = 30$ and $mad\_threshold = 5$), and the addition of a single new read pair can only potentially affect a small number of variants, it does not make sense to re-evaluate the clustering model for each new pair that comes in. Instead, the Insert Size Clustering Service accumulates discordantly mapped read pairs from the input stream and will process them as part of an ad hoc query or a periodically scheduled execution, producing an updated call set as a result. We define the service's operations in Table 5.13.

Here $addReadPair()$ is an operation that reads the stream of discordantly mapped reads and stores each read pair in the service's data store. The data store is an in-memory data structure that consists of an array for general read processing and an Interval Tree for performing queries for finding those reads that support a particular putative variant. The Interval Tree is a type of balanced binary search tree (described in [34], for example) that allows efficient querying of which intervals overlap a particular query interval. The tree can be constructed in $\mathcal{O}(n \log n)$ time, $\mathcal{O}(n)$ space, and answers queries in $\mathcal{O}(\log n + m)$ time, where n is the total number of

Table 5.13: Operations of Insert Size Clustering Service

Input	$M_{disc} = \{m_i : m_i = (header, payload)\}$ where $m.payload = (r_1, r_2)$, and $r_i = (s\_id, r\_id, b, q, f_p)$ as in 5.2.
Operations	$addReadPair(r_1, r_2)$ $callDeletions(region, insert\_size\_threshold, min\_read\_support)$
Output	$M_{del} = \{m_i : m_i = (header, payload)\}$ where $m.payload = V = \{v_i : v_i = (s\_id, contig, pos, length, vqual, R)\}$ is the set of called variants, and $R = \{r_i : r_i = (r_1, r_2)\}$ is the set of read pairs supporting the variant call.

intervals stored and m is the number of intervals being returned by the query (see Figure 5.21). Insertions into the tree after construction take $\mathcal{O}(\log n)$ time.

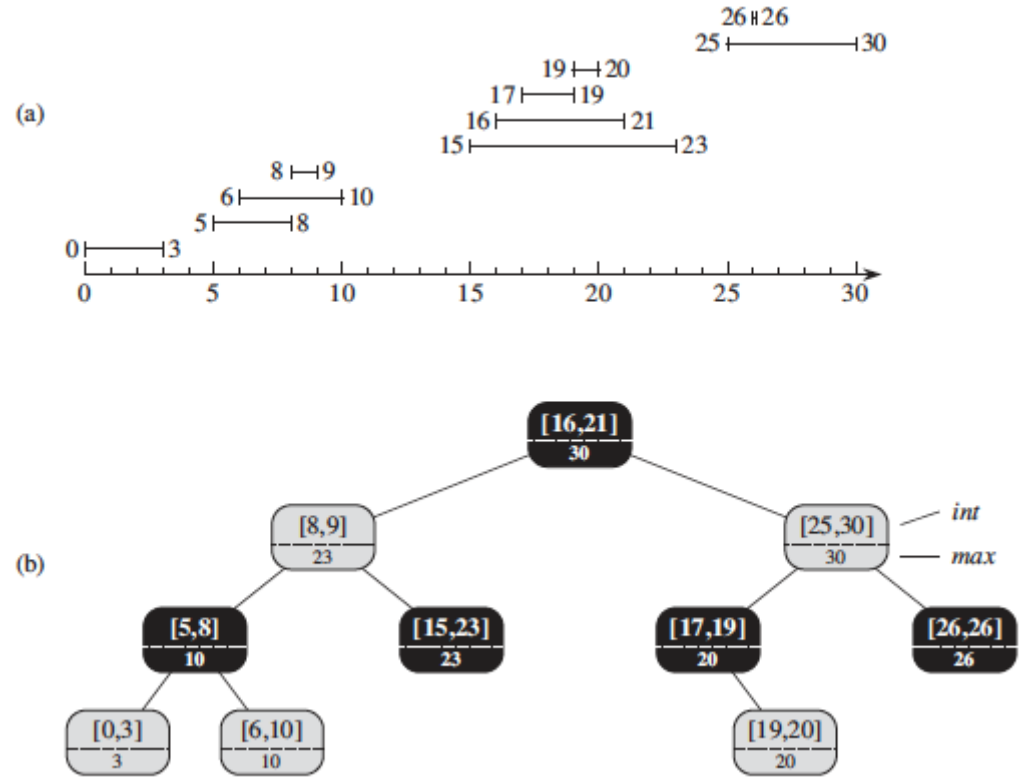


Figure 5.21: a) Intervals arranged on the number line. b)An Interval Tree constructed from the same intervals. Each node contains an interval and the maximum value of any interval in the subtree rooted at that node.[34].

Invocation of *callDeletions*() uses the data accumulated by *addReadPair*() to perform the actual deletion calling. The calling proceeds as follows:

1. For each read pair determine the center point of the interval spanned by the pair.

2. Using pair centers perform Kernel Density Estimation[161, 147] to estimate the distribution of the location of pair centers.
3. Using the estimated density detect regions with local maxima (clusters) that become locations for putative deletions.
4. Using the sites of putative deletions query the Interval Tree for a list of reads that overlaps each site.
5. For each site and set of read overlaps call the deletion boundary to be the maximal interval that is spanned by the inner distance (see Fig. 5.20) of all overlapping read pairs.
6. Based on the width of the deletion remove all overlapping reads whose inferred insert size is greater than $deletion\_width + insert\_size\_threshold$.
7. Output a deletion if the number of supporting reads after filtering above is $> min\_read\_support$.

The key approach used in this method for the clustering of read pair interval centers is Kernel Density Estimation. This approach has been used as a highly successful and efficient one-dimensional clustering method in a variety of settings[75, 93, 35], but has not been applied in the context of genomic variant calling to date.

If we have a one-dimensional data set of iid observations $(x_1, x_2, ..., x_n)$ from some unknown distribution and we wish to estimate the pdf f we can do so using:

$$\widehat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

Here K is a non-negative real-valued integrable windowing function, and h is a smoothing parameter called bandwidth. A variety of kernels are possible. In this method we use the gaussian kernel as a way to attenuate the influence of each read pair center with distance from that center.

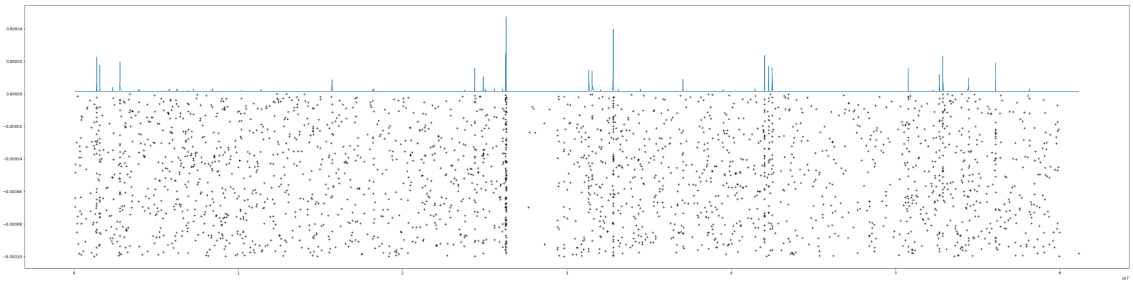


Figure 5.22: Centers of discordantly mapped read pairs on Chromosome 20 of NA12878 (plotted as points below x-axis) along with the KDE estimate plotted above x-axis.

In Figure 5.22 we demonstrate the results of applying KDE with a gaussian kernel and a bandwidth of 100 on the 2663 discordantly mapped read pairs from Chromosome 20 of the NA12878 sample, sequenced by the Genome In A Bottle[212] consortium and processed by Rheos. The x-axis specifies integer genomic coordinates from

0 to 6.3×10^7 and the y-axis is real-valued. Each point below the x-axis represents the coordinates of a centre for a single read-pair. The y-axis values for each point are randomly generated from a small interval in order to obtain visual point separation on the graph. The graph above the x-axis shows the estimated density function based on the 2663 points, with peaks indicating the position of clusters, and hence, the location of putative deleted regions. The location of peaks is obtained via a peak finding algorithm that returns the maximum value in an n-neighbourhood of points.

Once the location of the center points of putative deletions is known it is important to determine deletion size and evaluate the amount of support that exists for each deletion call. To achieve this *callDeletions*() makes use of the Interval Tree data structure that has been constructed from all of the read pairs. The Interval Tree is queried for all read pairs that overlap the position of a given deletion center. This query is answered in $\mathcal{O}(\log n + m)$ time.

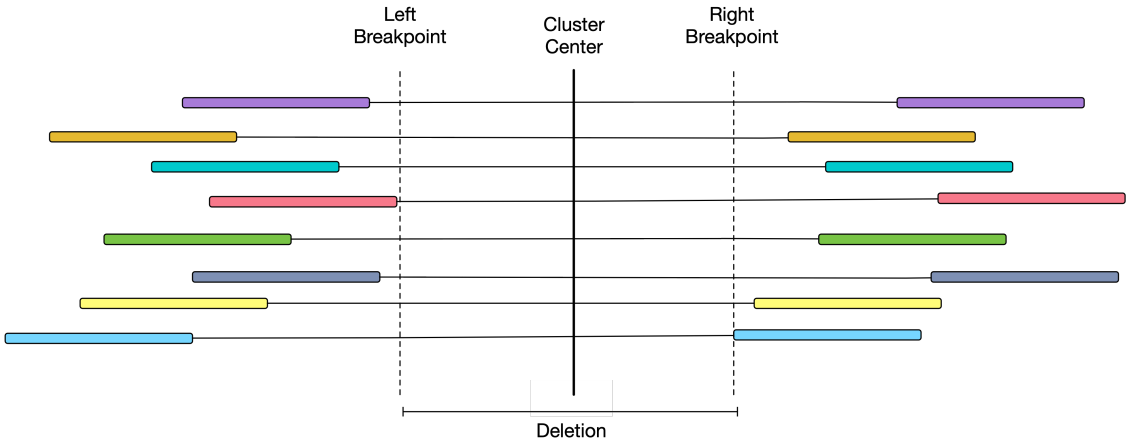


Figure 5.23: A cluster of reads spanning a germline deletion.

Given a set of read pairs $R = \{(r_1, r_2) : r = (r.pos, r.end)\}$ spanning the site of the deletion we are interested in determining the extent (position, and size) of the deletion. We take the deletion site (see Figure 5.23) to be an interval $[\max_{r_{1,i} \in R}(r_{1,i}.end), \min_{r_{2,i} \in R}(r_{2,i}.pos)]$, based on the following argument. If the deletion is homozygous, then no read in the sample should map into the deletion sequence, thus the deletion should be entirely contained in the inner distance between reads spanning the deletion. If the deletion is heterozygous, then the reads that are discordantly mapped come from the haplotype with the deleted allele, and should thus, also not have any sequence map into the deletion, again containing the deletion entirely in the inner distance between reads. Thus, the deletion boundaries are in the interval between the largest end coordinate for a read to the left of the deletion center, and the smallest start coordinate to the right of the deletion center.

Given the location and width of a putative deletion it is necessary to determine whether the deletion has sufficient support in the reads to be considered real, or may be dismissed as spurious. Many filtering strategies are possible, but the simplest relies on counting the number of read pairs that support a deletion and only retaining

those that exceed a minimum threshold of *min_read_support* supplied by the user. To reduce the number of false positive results we would like to remove remove from the list of reads supporting a deletion call those that are likely spurious. Since, low quality reads are already filtered by the Insert Size Filtering Service one of the few remaining sources of false positive signal are reads for which the inferred insert size is much larger than the actual deletion call that they span. These reads are either part of another, much larger, deletion, in which case they are accounted for in that deletion call, or they are spuriously mapped, as there is no other plausible reason for their large insert size. We thus, use a thresholding cutoff *insert_size_threshold*, supplied by the user, such that read pairs with $insert\_size > deletion\_width + insert\_size\_threshold$ are removed from the list of reads supporting a particular call. Afterwards, deletions that have read support exceeding the *min_read_support* threshold are selected as called variants.

The variant quality for the deletion is taken to be $vqual = \sum_{r \in R} r.mapq$ i.e. the Phred-scaled probability that all of the reads that support the deletion have been mis-mapped. All of the variants that have been called are emitted in the Insert Size Clustering Service's output stream in the format $M_{del} = \{m_i : m_i = (header, payload)\}$ where $m.payload = V = \{v_i : v_i = (s\_id, contig, pos, length, vqual, R)\}$, as described in Table 5.13, where the header includes the query details that are being answered.

5.4 Rheos implementation

In order to prove the viability of the concepts behind Rheos we have built and tested a limited implementation of Rheos. This implementation provides a data streaming architecture of services and is able to perform genome alignment, followed by online germline SNP and deletion calling, as described in Section 5.3. The source code is available on github at - https://github.com/llevar/rheos. In this section we describe the details of the technical implementation, including the key approaches taken for the establishment of data streaming, service organization, scalable deployment, and performance optimization. Even though the implementation is fully functional and has been used to analyze real data, it is worth noting that due to the extremely broad scope of the overall Rheos framework, and the limited resources available, the implementation focuses on a narrow set of use cases and is meant to be treated as a proof of concept rather than a production system.

As in the implementation of Butler (see Chapters 3, 4) we make a concentrated effort to rely on established Open Source software frameworks where possible to ensure that components of Rheos are robust and scalable, require minimal maintenance, and can be easily deployed on a variety of platforms. Some of the key frameworks used by Rheos are Apache Kafka, Google Kubernetes, Docker, and Prometheus.

The implementation is focused on the following key services:

Read Mapping Service - Reads a FASTQ file and turns it into a set of streams

for mapped and unmapped reads and read-pairs.

Locus Processor Service - Reads a stream of mapped read pairs and incorporates variant evidence from the reads into a model of genomic Loci. Emits a stream of updated Loci.

Locus Saver Service - Reads a stream of Loci and stores them in a distributed data store.

Variant Calling Service - Reads Loci from a data store and performs germline SNP calling. Emits a VCF file.

Insert Size Filtering Service - Reads a stream of mapped read pairs and emits a stream of filtered high quality read pairs that are discordantly mapped and can be used for calling deletions.

Insert Size Clustering Service - Reads a stream of discordantly mapped read pairs and calls germline deletions via KDE clustering. Emits a stream of deletion variants.

We first focus on the general architecture of a Rheos service and describe how services communicate with each other, then describe the deployment and operation of the Rheos system as a whole, and finally describe the individual services to comprise the Rheos implementation.

5.4.1 Rheos Service

A service in Rheos is a continuously running program with a well defined interface (implementing one or more operations described in Section 5.2.1), and a set of well understood operating characteristics. Even though a service author is completely free to use different technologies for different services (as long as the interface is respected), we have chosen to implement the initial set of Rheos services in Python. Since Rheos services are typically streaming services, their operations are most often invoked automatically when new data appears on the stream.

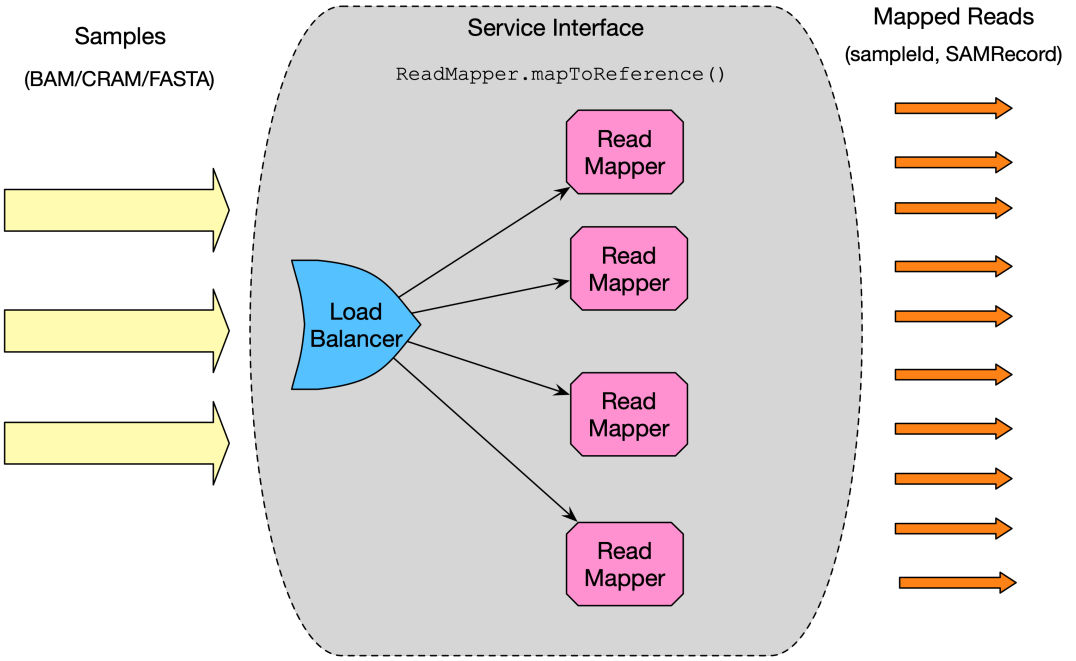


Figure 5.24: Several instances of the Read Mapping Service are managed behind a Load Balancer.

Figure 5.24 demonstrates several key concepts behind Rheos services using the example of a Read Mapping Service:

- The service transforms the granularity of the data from sample-level on the input to read-level on the output.
- The service implements a streaming operation of type Decorator, it decorates the incoming read data with additional information, that of the read coordinates, CIGAR string, mapping quality, etc.
- The service provides a uniform interface to its clients, that of *ReadService.mapToReference()*. Machines that implement this interface are indistinguishable from each other.
- Clients of the service talk to the service interface through the Load Balancer. The DNS entry for the service returns the IP of the Load Balancer which is then responsible for routing requests to one of the service instances. This allows one to control the scalability of the service, seamlessly adding and removing instances of the Read Mapper depending on the rate of the data that is coming into the system, and the ability of downstream services to process it.

When services are deployed it is desirable to be able to easily deploy them to a variety of different environments and achieve a level of resource and environment isolation, so that other programs running on the same machine can only minimally

impact the installation and running of the service. To accomplish these goals, all of the services in Rheos follow a containerized approach using Docker[134].

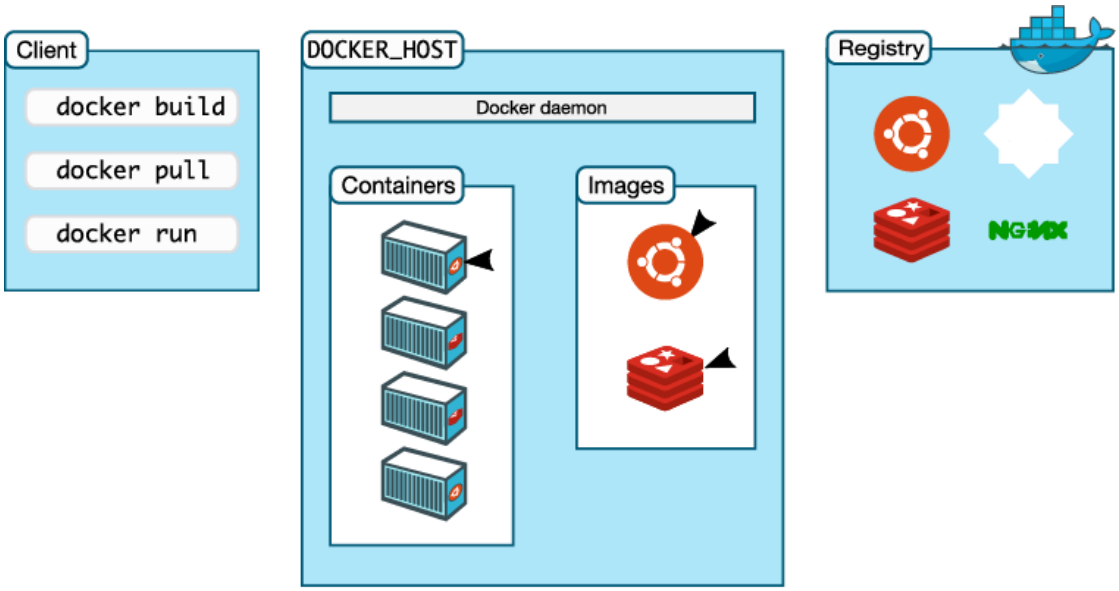


Figure 5.25: Docker system architecture (https://docs.docker.com/engine/docker-overview/).

Docker is an Open Source framework that isolates the execution of a program from the host machine and other programs by encapsulating it in a lightweight Linux container. The container is a separate instance of an operating system and a program running inside it is not aware of its enclosing environment. The Docker architecture consists of three main components (see Figure 5.25). A machine that can run Docker containers (the Docker Host) needs to be running the Docker Daemon. A container is a runtime instance of a Docker environment, but the blueprint from which all containers are created is called a Docker Image. The user starts with a base image that only has an operating system in it. The user can then augment the image by installing additional software packages and custom programs. This can be done interactively inside a shell inside a running container, or declaratively via a Dockerfile. When the user is finished preparing their image they can use the Docker Client to build it and upload it to an external Docker Registry. This is an external registry where various Docker images are hosted. When Docker Daemon needs to create a container it pulls down the requisite image from a registry to the local machine, and then instantiates the container. Containers can be started and stopped, created and deleted easily, and provide a great deal of abstraction and simplification when it comes to running many different applications, with potentially conflicting dependencies on a server. They also provide a seamless migration path from the local development machine, through the testing environment,and into production. All Rheos services are implemented to be encapsulated inside Docker containers.

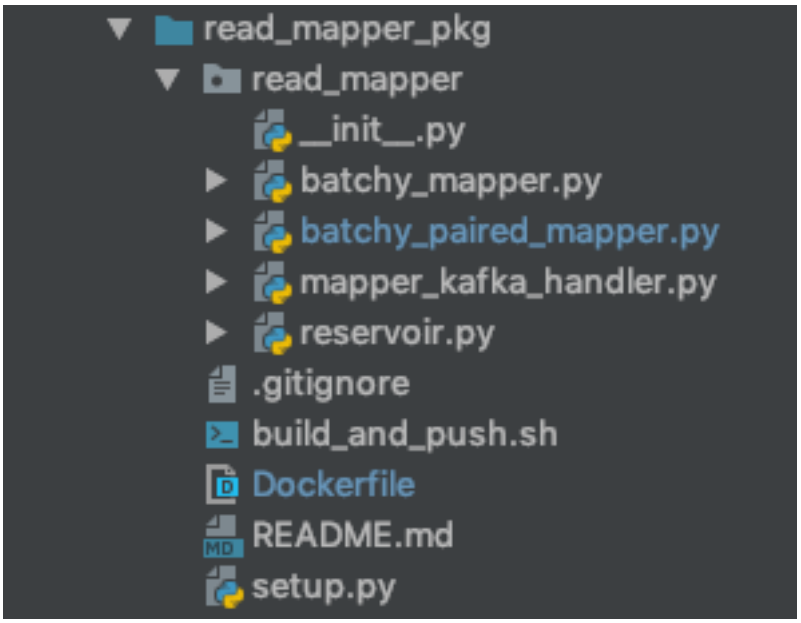


Figure 5.26: Rheos service file structure.

The layout of a Rheos service file structure is standard (see Fig. 5.26). One or more Python files implement the service functionality in the form of a Python module. The file *setup.py* provides a list of dependencies that are needed for the functionality of the service. The *Dockerfile* specifies how to build a Docker image of the service (See Listing 1).

Listing 1: Example Dockerfile for setting up a service.

```
1 FROM python:3.7.2-alpine3.8
2 RUN apk add --no-cache build-base python3-dev zlib-dev cython make bzip2-dev
  ↪ xz-dev libcurl
3 WORKDIR /app
4 COPY . /app/
5 RUN python setup.py install
6 RUN mkdir /app/log && touch /app/log/mapper.log
7 ENTRYPOINT ["python", "read_mapper/batchy_mapper.py"]
```

The Dockerfile uses a base image of Alpine Linux with python 3.7.2 preinstalled. Several addon linux packages are installed. The service module code is copied to the `/app` directory. `setup.py` is run to install all of the service's dependencies, and the program entrypoint for the Docker container is set to point to the service executable. The file `build_and_push.sh` is a shell script that is responsible for building a Docker image using the supplied Dockerfile and pushing this image to a Docker registry (hub.docker.com). Any deployment of the service can now pull down this image from the registry and instantiate new containers from it.
As there is a good deal of functionality that is common between several Rheos services there is a separate Python module called `rheos-common` that houses both utility classes and commonly used models (such as that for genomic loci) in its

submodules. This module is hosted on Pypi and can be installed from anywhere using standard Python installation techniques such as *pip*.
The wire format for messages exchanged between services is currently schema-less and relies on standard Python serialization mechanisms (namely *pickle*), and *json*. Although *json* is completely generic, *pickle* is proprietary to Python and thus represents a weakness of the current implementation. A generic wire format with a defined schema will be a future improvement.
Serialized messages are streamed between services via distributed queues.

5.4.2 Queueing

For message queueing Rheos relies on an Open Source framework called Apache Kafka[92]. Kafka queues provide a distributed fault-tolerant implementation of the publish/subscribe mechanism, and streaming.

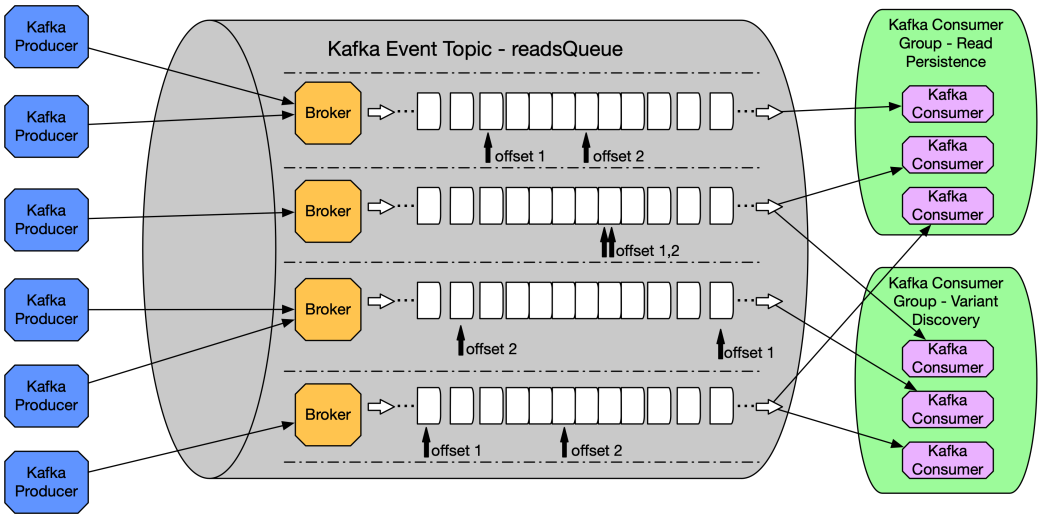


Figure 5.27: Example of a Kafka queue with multiple data producers and consumers.

At the heart of Kafka lies an Event Topic (see Figure 5.27). The topic is a distributed transaction log. When messages are put in the queue, they are never deleted by a consumer. Instead, consumers keep an offset into the log that tells them where they currently are in the processing of the data. Offsets can be moved freely so that a consumer may process the same message multiple times if needed. Data is deleted from the queue by virtue of a retention policy that removes data that is older than some specified time period.
The queue is distributed between several machines by using partitions. A Kafka partition is a function that specifies how many logical and physical partitions of the data exist. Data may be partitioned onto several machines based on data size considerations, or it may be partitioned based on a logical grouping. In the former case, an automated partitioning function is employed that decides which partition to send new messages to in round-robin fashion. In the latter case a logical partitioning function needs to be supplied by the user. Rheos mainly uses the latter approach to partition data by genomic coordinate (see Section 5.4.5 for details.)
Data producers put messages into the queue by communicating with a Broker. A Broker is a Kafka service that manages one or more queues on a particular machine.

Multiple Brokers may be put in a cluster for load balancing. A data producer establishes a connection with a broker using a client library (`kafka-python` in the case of Rheos). The partitioning function determines which machine a particular message ends up on.
On the data consumer side, consumers are organized into Consumer Groups. Each Consumer Group is a set of consumers that are assumed to be a separate entity from the consumers in all the other consumer groups. A Consumer Group has its on separate set of offsets into every partition of a topic that it is subscribed to and these do not interefer with the offsets of other groups. Assignment of consumers within a group to partitions of a topic can be automated or manual. In the case of automated assignment Kafka will decide on its own how to pair a consumer with a partition. In the case of manual assignment the partitioning function is used to make the assignment. It is in general not advisable to have multiple consumers from the same group reading data from the same partition.
We implement a generic Kafka Handler in the `rheods-common` module. This handler allows Rheos services to initialize Kafka producers and consumers. serialize and deserialize data using pickle and json, and to store and retrieve messages using the topics that have been set up for inter-service communication.

5.4.3 Partitioning

Partitioning is important in Rheos not only because the data is quite big and thus needs to be parceled out to multiple machines, but also because data exhibits strong locality subsequent to the read mapping stage, where most of the data that is required to make decisions about the presence or absence of a genomic variant resides in some small genomic neighbourhood around the variant. It is thus natural for us to partition the read data (which is biggest in size) by genomic coordinate.

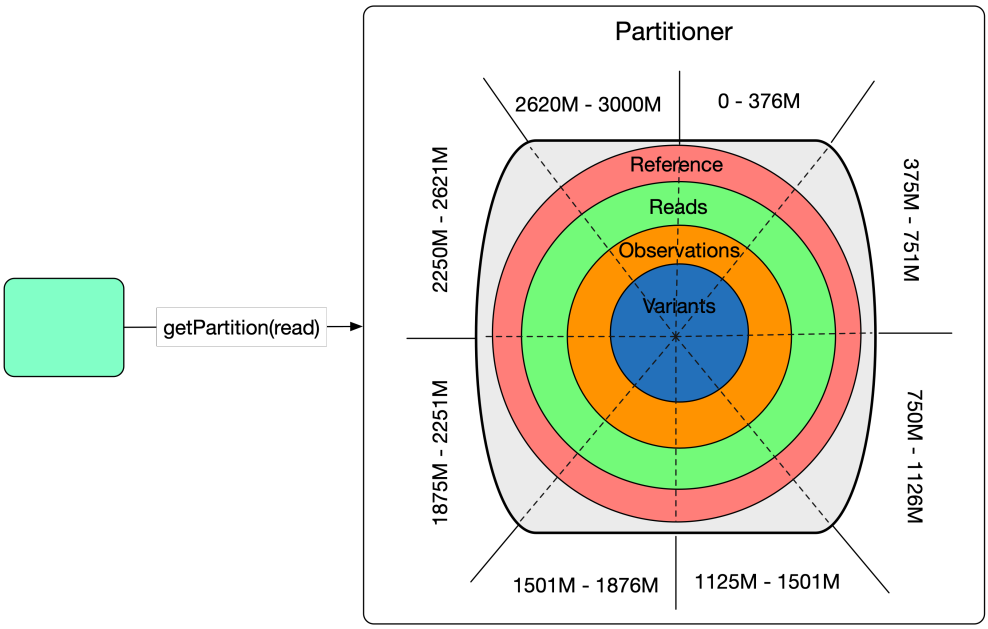


Figure 5.28: Rheos data is partitioned by genomic coordinate.

Partitioning can be challenging because different partitions end up on different ma-

chines, yet certain reads and structural variants end up spanning one or several partitions. We need to decide what to do at the partition boundaries, i.e. if a particular read ends up spanning the boundary, does it end up in one partition, the other, or both? If a read ends up in two partitions how does one avoid double counting the read in downstream processing? Likewise, if the reads supporting one breakpoint of a structural variant lie in one partition, and the reads supporting the other breakpoint lie in another partition, how can these be put together to form a single variant?
We take a simple approach in the initial Rheos implementation that will require further work when the system is elaborated. The following assumptions are made by the Partitioner:

- All partitions are of equal size, except at the end of contigs.
- Reads that span multiple partitions are assigned to all partitions that they map to (see Figure 5.29).
- A given read will always map to the same set of partitions.
- All reads that are unmapped end up in the ”unmapped” partition.
- All reads that map to special contigs (like decoy sequences) are mapped to a special ”other” partition.

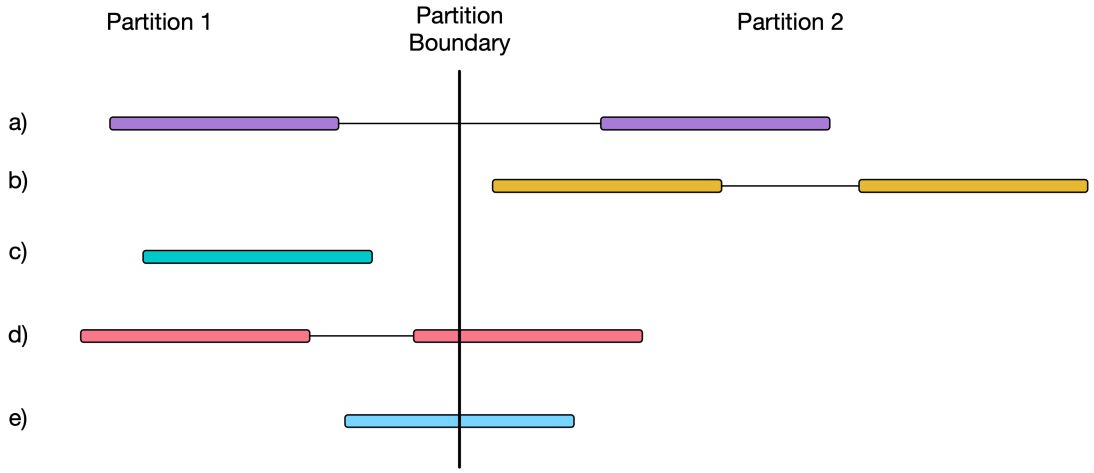


Figure 5.29: Reads that map near the partition boundary - a) The read pair maps to partitions 1 and 2, b) The read pair maps to partition 2 only, c) The single read maps to Partition 1 only, d) The read pair maps to partitions 1 and 2, e) The single read maps to partitions 1 and 2.

In general the partitioning problem can be stated as follows:
Given a reference genome $G = G_{main} \cup G_{other}$, where $G_{main} = \{g_i : g_i = (name_i, length), name_i \in \{1, .., 22, X, Y, MT\}\}$ and $G_{other} = \{g_i : g_i = (name_i, length), name_i \in Aux\_Contig\_Names\}$, we assign a set of indexed partitions $P = \{(g_i.name, start, end, i) : g_i \in G_{main}, i \in [1, |G_{main}|]\} \cup \{("other", |G_{main}| + 1), ("unmapped", |G_{main}| + 2\}$, where given a set of reads $R = \{r_i : r_i =$

$(contig, pos, end)\}$ (most often a read or a read pair), we desire to have a function $f : R \mapsto \mathcal{P}(P)$ (where $\mathcal{P}(P)$ is the power set of P) that maps those reads to a set of partitions from P.
We opt for partitions of fixed width *partition_width* (except at ends of contigs) and implement Algorithm 8, that, given G, produces P, and for a given interval or read pair returns a set of partitions they map to:

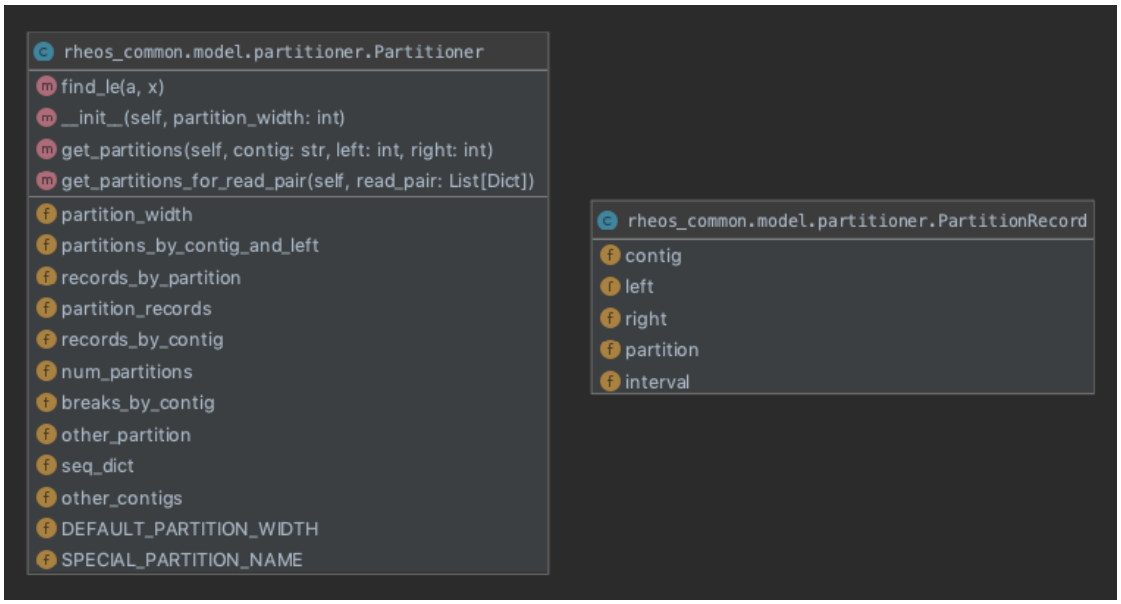


Figure 5.30: Rheos Partitioner class diagram.

Algorithm 8 is implemented in Rheos via the `Partitioner` and `PartitionRecord` classes of the `rheos-common` package. Namely, members of P are implemented as instances of `PartitionRecord`, function INITIALIZEPARTITIONER($g\_main, g\_other, partition\_width$) is implemented in `Partitioner.__init__()`, function GETPARTITIONSFORINTERVAL($contig, left, right$) is implemented in `Partitioner.get_partitions()` method, and function GETPARTITIONSFORREADPAIR($read\_pair$) is implemented in `Partitioner.get_partitions_for_read_pair()`.
In our tests we have been using a partition width of 2×10^7 bases, resulting in 148 partitions for read data. The Partitioner is used in the Read Mapping Service to determine which partition to write data to, as well as in the Locus Processor Service to determine which partition is assigned to which instance of the service.

5.4.4 Deployment

Deployment of large-scale distributed systems can be complex as each component can have its own requirements related to computational resources, environment, or the ability to scale. This heterogeneity of requirements demands a flexible deployment approach that can cater to the individual needs of the various components of the system. Additionally, there is a wide variety of computational infrastructures into which a user may wish to deploy the system, again with a distinct set of capabilities and limitations. With Butler (see Chapters 3, 4) we solve this problem by utilizing a separate framework (Terraform) for abstracting the specifics of cloud

Algorithm 8: Partitioning the reference genome by a fixed width and mapping reads to partition sets.

```
Function initializePartitioner(g_main, g_other, partition_width) begin
    partition_index ← 0
    for name, length in g_main do
        num_partitions ← ceiling(length/partition_width)
        for i ← 0 to num_partitions do
            left ← i × partition_width + 1
            right ← min((i + 1) × partition_width, length)
            new_record ← PartitionRecord(name, left, right, partition_index)
             // Create a member of P

            addBreaksByContig(name, left)      // Keep a hashmap of partition
            boundaries by contig

            addPartitionsByContigAndLeft(partition_index, name, left)  // Keep
            a hashmap of partition records by contig and left boundary.

            partition_index ← partition_index + 1
Function getPartitionsForInterval(contig, left, right) begin
    partitions_for_int ← Set()        // Partition records are a set, to avoid
     duplicates

    if contig ∉ getOtherContigNames() then
        breaks ← getSortedBreaksByContig()    // Breaks are sorted by left
         coordinate

        left_index ← binarySearch(breaks, left)
        left_partition ←
         getPartitionByContigAndLeft(contig, breaks[left_index − 1])
        addToSet(partitions_for_int, left_partition)
        right_index ← binarySearch(breaks, right)
        right_partition ←
         getPartitionByContigAndLeft(contig, breaks[right_index − 1])
        addToSet(partitions_for_int, right_partition)
    else
        addToSet(partitions_for_int, getOtherPartition)
Function getPartitionsForReadPair(read_pair) begin
    partitions_for_reads ← Set()
    for read in read_pair do
        if read.mapped is TRUE then
            left ← read.pos + 1
            right ← read.end
            contig ← read.contig
            addAllToSet(partitions_for_reads,getPartitionsForInterval(contig, left, right))

        else
            addToSet(partitions_for_reads, getUnmappedPartition)
```

provider's provisioning API, another separate framework (Saltstack) for providing a flexible configuration management facility that can configure software on a variety of platforms, and a separate suite of frameworks (InfluxDB, Grafana, Telegraf, Kapac-

itor) for operational management of the deployed components. We take a somewhat different approach with Rheos, one that based on the containerized nature of Rheos services, and one that relies on a single common execution environment being deployed on top of any cloud provider, or other types of computational infrastructure. The environment that we use is Google Kubernetes[15].

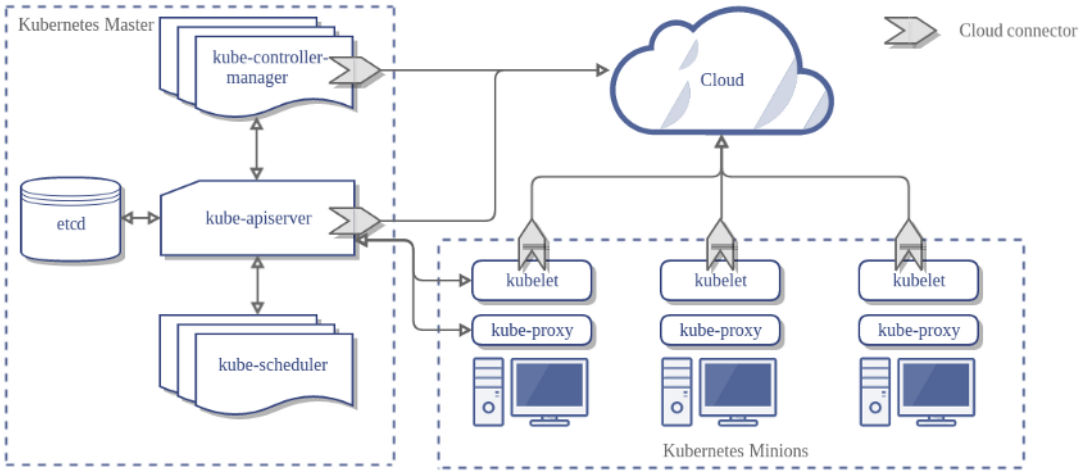


Figure 5.31: Kubernetes high level architecture (https://kubernetes.io/docs/concepts/architecture).

Kubernetes is a cluster resource management and container execution framework developed by Google. It can deploy on virtually any cloud as well as bare metal hardware (which makes it more flexible than the approach adopted in Butler). A Kubernetes cluster is a collection generic compute hosts (Nodes) each running the `Docker` and `kubelet` daemons. These nodes go into a generic pool of computational resources, although nodes with specialized capabilities, such as high RAM, or advanced GPUs can be held in separate pools. A master node is responsible for keeping track of the configuration and computational capabilities of the cluster by virtue of an etcd registry that it keeps. This node runs a special management daemon called `kubeadm` and schedules the deployment and execution of workloads on the cluster, all of which are encapsulated in Docker containers. See Figure 5.31 for the general high level architecture of Kubernetes. This architecture is scalable to over 2500 compute nodes per cluster[3].
In order to facilitate inter-container communication Kubernetes provides an overlay network that is responsible for managing address translation, network security, and local DNS by running a proxy agent `kube-proxy` on each node. This allows various components of the system to refer to each other by name rather than IP address, and enables role-based access to various network resources.
Kubernetes was built for deploying and operating services and provides a large number of high-level abstractions to facilitate the full service lifecycle. Kubernetes objects are managed through declarative documents in YAML format that describe the desired state of the system. The user interacts with the cluster by virtue of a CLI (`kubectl`) running on the master node, or via a GUI dashboard (see Figure 5.32).
At the lowest level of abstraction in Kubernetes hierarchy is a Pod. The Pod is a wrapper around a Docker container and describes some useful metadata for cate-

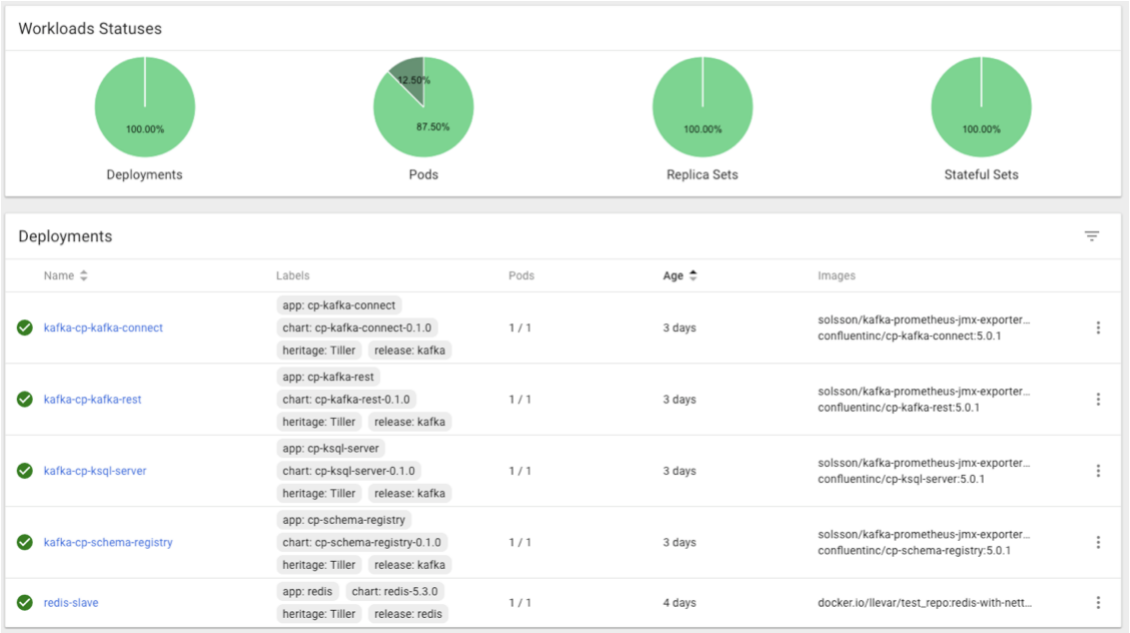


Figure 5.32: Kubernetes dashboard.

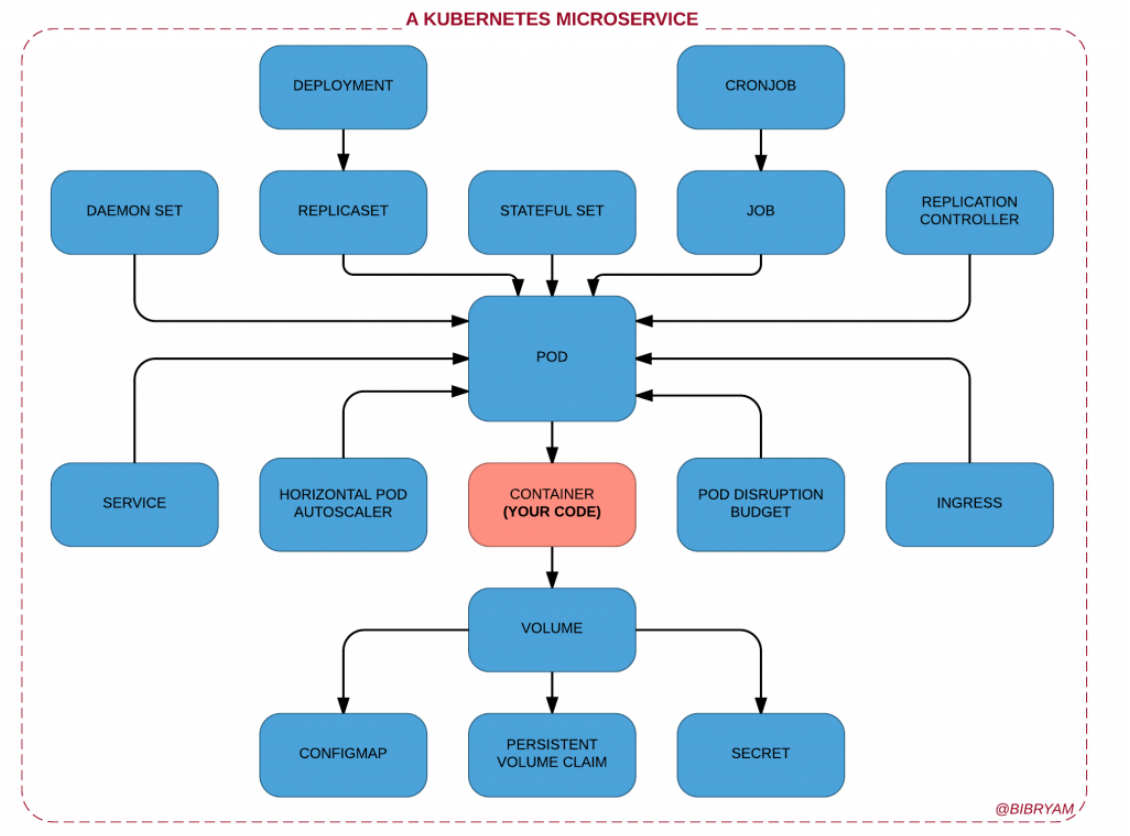


Figure 5.33: Kubernetes microservice components (from [4], by Bilgin Ibryam)

gorizing as well as describing the resource requirements of the underlying container that are used for scheduling the container onto nodes. These resources include CPU and memory requirements, as well as requests for various types of storage (called

a PersistentVolumeClaim). A Pod is scheduled for execution only when the cluster manager can find a suitable node that can meet all of the pod's resource requirements. If the Pod exceeds its stated resource requirements during its lifetime it will be terminated by the scheduler.
Since the goal of kuberenetes is to enable computation at scale, Pods are rarely run as single instances. Depending on the intended usage of the underlying container Pods can be organized in groups such as a DaemonSet, ReplicaSet, StatefulSet, Job, etc. These abstractions describe the runtime behaviour of groups of Pods and provide lifecycle management support for each. For instance, a ReplicaSet provides a group stateless interchangeable Pods where each Pod can identically service a request from a client, and Pods can be scaled horizontally. The configuration for a ReplicaSet may specify the number of such Pods that should be running at a given time and the scheduler will create and delete Pods as necessary to make sure that the requisite number of Pods is running. Other groups of Pods that do not scale horizontally, and require access to non-ephemeral secondary storage (such as those running a database) may use the StatefulSet to provide such services. A DaemonSet makes sure that every node in a cluster is running an instance of a particular Pod. A Service is a higher level abstraction that provides a logical grouping of pods and provides a number of capabilities such as DNS, load balancing, and security.
In Rheos we make use of Kubernetes for two purposes - one is to deploy and configure the various products that Rheos services rely on for their operation. These include:

Kafka - The queueing framework that all Rheos services use for communicating is deployed on Kubernetes as a set of fault tolerant Docker containers.

Redis - Is a scalable in-memory key-value data store that Rheos services use to store intermediate results.

PostgreSQL - Is an SQL database used for persisting variants to disk.

Prometheus - Is a monitoring and alerting framework used to maintain operational control of the running system.

Grafana - Is a metrics dashboarding software used for visualization of health metrics.

Elasticsearch - Is an indexing product used for harvesting and storing application logs.

Jenkings - Is an automated build and deployment tool used for creating software builds.

Kubernetes Dashboard - Is a UI used for managing Kubernetes clusters.

For each product we maintain a set of configuration files used for Kubernetes deployments and configuration of these tools (see Figure 5.34).
Additionally we use Kubernetes to configure and deploy the services of Rheos themselves, organized as fleets of Docker containers. This allows us to enjoy the flexibility of being able to deploy on virtually any hardware, the envirnment isolation provided by Docker, and the resource management and elastic scalability afforded by Kubernetes, which are all key concerns when operating a large-scale service-oriented system.

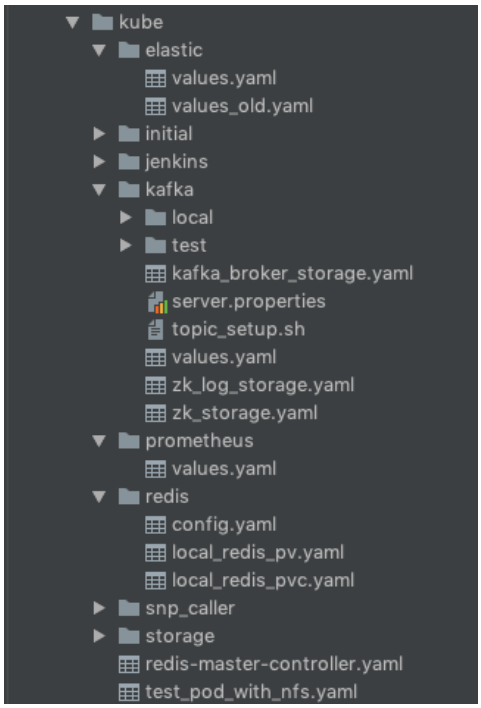


Figure 5.34: Rheos Kubernetes configurations file layout.

5.4.5 Services of Rheos

We build on top of the concepts of a general Rheos Service, Kafka topics, the Kubernetes framework and other products described earlier in this section to provide a cohesive implementation of a set of services that implement two major end-to-end use cases encountered in genomic analysis - germline SNP calling, and germline SV (deletion) calling. As Rheos matures these services will be extended to provide enhanced functionality and increased scalability required for production applications of the framework.

Read Mapping Service

The Read Mapping Service is the entrypoint of the initial Rheos implementation. Although future versions of Rheos will likely have a different entrypoint, namely a Read Streaming Service that will accept data from an external endpoint, such as a data repository or directly from the sequencer, we proceed directly to read mapping in this initial implementation as a matter of simplification.
This service aims to implement two read mapping operations specified in Section 5.3.2 - $mapToReference()$ and $mapPairToReference()$. It reads a user-specified FASTQ file (the sample), as well as a genome reference file from disk and emits four data streams $mappedReads$, $mappedReadPairs$, $unmappedReads$, and $unmappedReadPairs$. These operations allow the service to deal with both paired and unpaired short reads produced by Illumina sequencers. The partitioning strategy of Section implemented in the `rheos-common` package is used to determine which partition of their respective topics reads end up in. Instead of implementing our own read mapper from scratch we are making use of a python module called `mappy`, which is an interface to the popular read alignment tool minimap2[109], by Heng Li, which is based on the hashmap approach to alignment discussed in Section

2.2.2. The reason we chose to provide a wrapper rather than a full implementation is that minimap2 already has an API that supports a read-by-read alignment that can be adopted for a streaming use case. Additionally, minimap2 is known to be accurate, fast, and supports mapping of long reads, and other use cases, making it an attractive framework to work within for read alignment.
Although the Read Mapping Service can map and stream individual reads, this approach proves quite impractical. In our tests of performance, we observed that for a single read the cost of serializing the mapped read data to send it over the wire was nearly 40% of the overall processing time. To reduce the proportion of computational resources used for serialization/deserialization we implemented a read batching strategy based on the reservoir concept, that groups similar reads together and sends them over the wire as a group. A `PartitionedReservoirSet` that implements this functionality is shown in Figure 5.35.

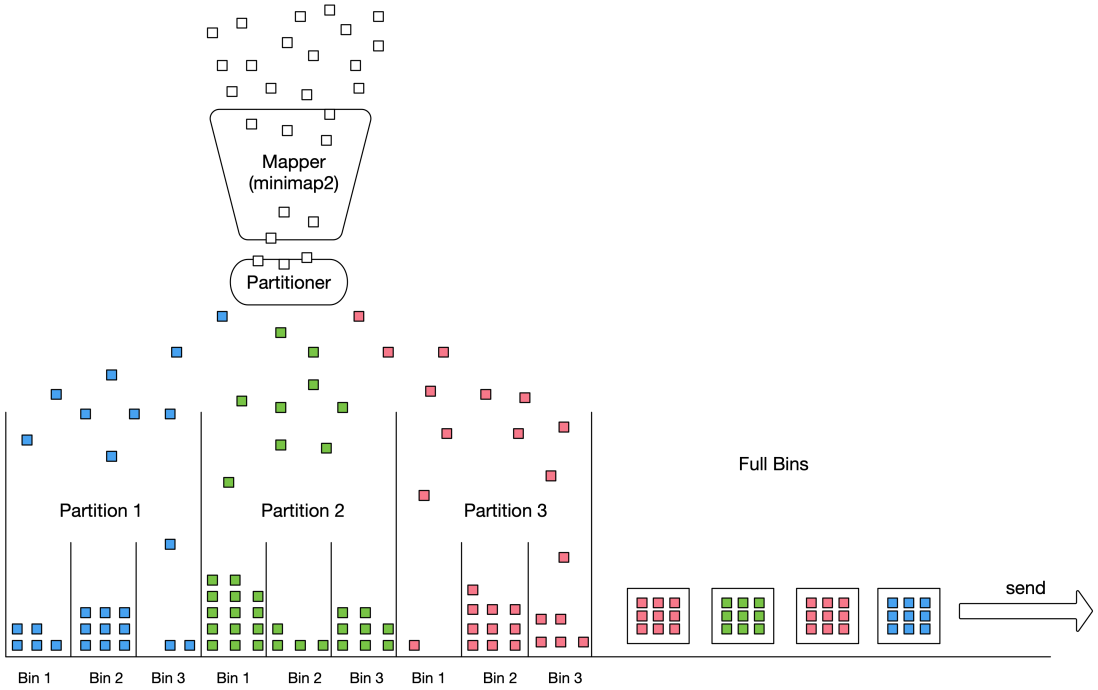


Figure 5.35: A `PartitionedReservoirSet` is used to group together similar reads before streaming them as a mini batch.

Reads are read one-by-one from a FASTQ file. Each read is attempted to be paired. If no pair exists for the read it is put into an accumulator (a type of hashmap). If a read completes a read pair, it is popped from the accumulator and the read pair is sent through to alignment by minimap2. Once the read pair is aligned it is put through the Partitioner to assign a set of PartitionRecords to it. Each PartitionRecord is further subdivided by genomic interval into a set of bins that act as reservoirs. The size of the bins is controlled by user supplied parameters where *num_bins* specifies how many bins are created per partition, and *bin_size* specifies capacity in read pairs. Each `Reservoir` accumulates read pairs that map within its genomic region, until reservoir capacity is reached. At that point all of the read pairs from that reservoir are serialized and emitted from the service as a single message. The reservoir is then emptied and is ready to accept new elements. When a whole sample is finished alignment all of the non-empty bins are serialized and emitted by

the service as individual messages. Figure 5.36 shows the classes and methods of the Read Mapping Service and the `PartitionedReservoirSet`.

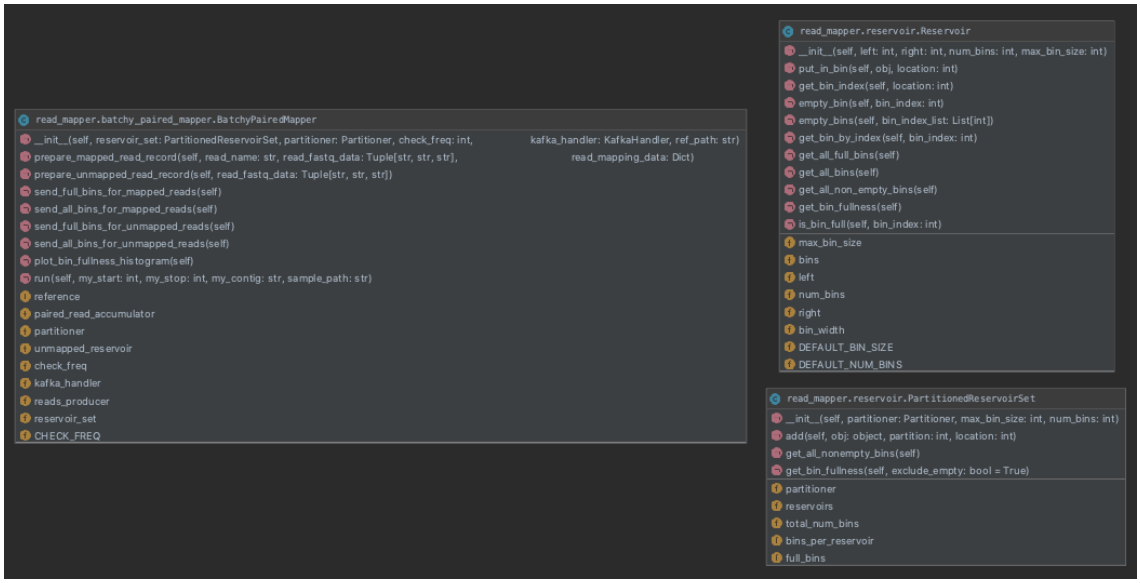


Figure 5.36: Classes of the `ReadMappingService`.

The disadvantages of the mini-batching approach are the decreased initial throughput (no data is emitted by the service before at least one of the reservoirs fills up), and increased memory footprint of the Read Mapping Service. The reservoir approach, however, reduces the proportion of the overall processing time due to serialization by over 90%. Additionally, there are benefits to processing the read data in small batches of closely-mapped reads by downstream services, because all of the loci that need to be updated by the reads can be fetched into memory and efficiently processed, and the since the reads map closely together, fewer total loci are updated by such a batch than by a random collection of reads. In general, since the usage of the `PartitionedReservoirSet` can be controlled by the user via service parameters, there is a good degree of flexibility when it comes to trading off these performance considerations.

Locus Processor Service

The Locus Processor Service is a Rheos service that listens to the stream of mapped read pairs produced by the Read Streaming Service and updates a set of genomic loci that the reads overlap with the variant evidence contained in the reads. It then emits a stream of updated loci that can be further processed by downstream services for variant calling. Figure 5.37 shows a schematic view of the Locus Processor Service. The Locus Processor Service is of Local State Aggregator type and implements the iterative Bayesian inference framework described in Section 5.3.3, namely it implements the stream operation *updateLociFromRead*(), Equations 5.10, and Algorithms 6, and 7. The service uses the same Paritioner to decide where to get data from as is used by the Read Mapping Service.
A single message received by the Locus Processor Service represents a full bin of mapped read pairs from a single partition of the *mapped_read_pairs* kafka topic. `BatchyPairedLocusProcessor.process_messages()` is the main processing loop of the service (see Figure 5.38). An in-memory collection of `Locus`

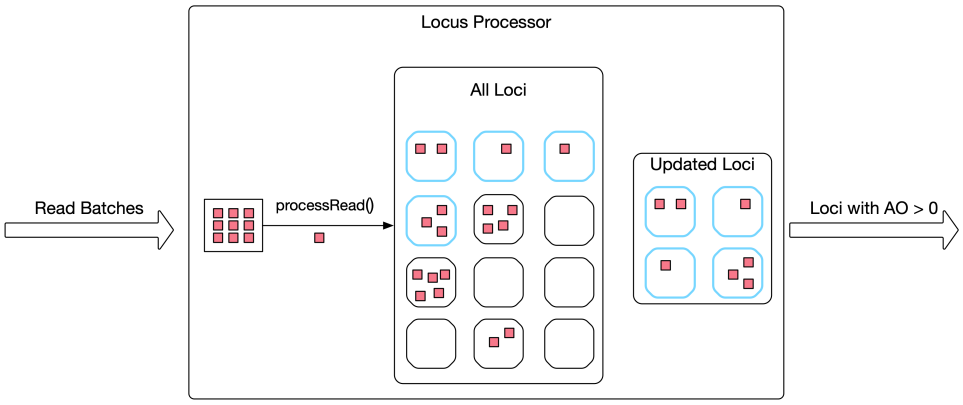


Figure 5.37: The Locus Processor Service uses read data to update a set of loci.

objects for the assigned parition, organized in a hashmap by genomic coordinate is the main holder of state for the service. For each received message, the `BatchyPairedLocusProcessor.process_reads()` method is responsible for processing all of the contained reads. It acts as the implementation of Algorithm 6.
Namely, for each read, the algorithm simultaneously walks the genome reference and the read CIGAR string looking for matches. Different CIGAR elements consume the reference, the read, or both. For instance, a deletion consumes the reference sequence, but not the read. It's the opposite case for an insertion, and a sequence match consumes both. Deletion and insertion CIGAR elements will in the future be used for indel variant calling, but at the moment they are simply passed over. When a match CIGAR element is found, the `BatchyPairedLocusProcessor.handle_match()` method is called.

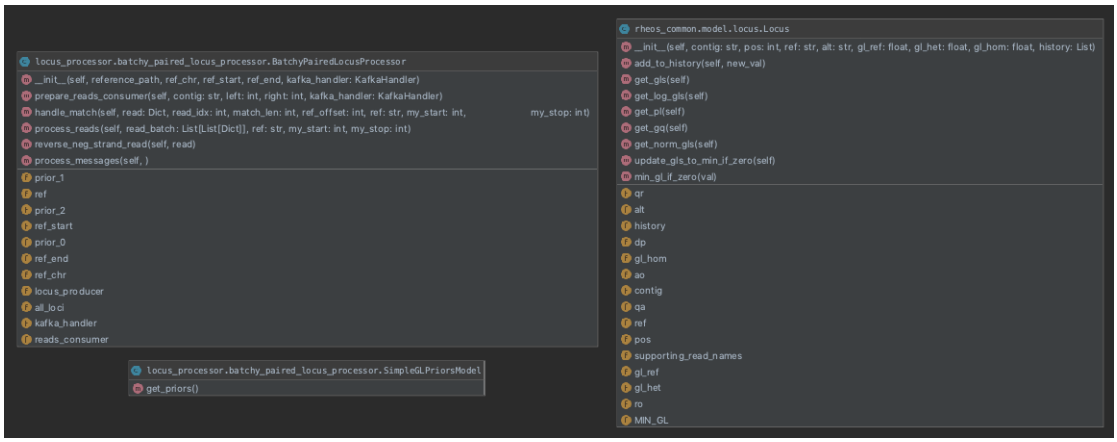


Figure 5.38: The Locus Processor Service classes and methods.

This method implements Algorithm 7. For each match the set of `Locus` objects overlapping the match are retrieved. The model of each `Locus` is updated using Equation 5.10, depending on whether that position in the read matches the reference. Reads that map to the negative strand are reverse complemented using `BatchyPairedLocusProcessor.reverese_neg_strand_read()` and matched from the end.
For each input message received, all `Locus` models that have been updated by reads in this message, and that potentially have variants (determined by `Locus.AO` > 0,

where AO is the count of alternative allele observations for that locus) are bundled together and emitted as a single message of updated loci from the Locus Processor Service.

Locus Saver Service

The Locus Saver Service is a fairly simple service that acts as an aggregating and persistence mechanism for `Locus` objects. Rheos uses an open source in-memory distributed key-value store called Redis[21]. Using this product allows Rheos to keep a large number of `Locus` objects in memmory for fast retrieval, while offering options for distributed access and optional disk persistence.

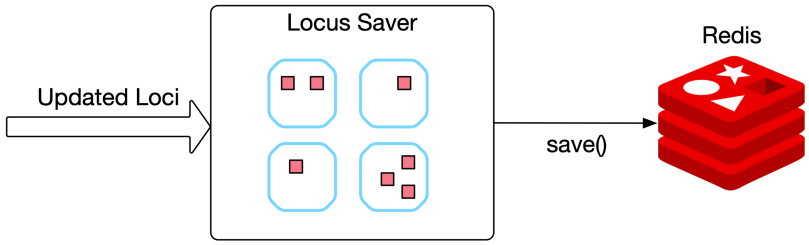


Figure 5.39: The Locus Saver Service data processing schematic.

The Locus Saver Service recieves messages from the *snp_loci_batchy* kafka topic, where each message contains a batch of updated `Locus` objects. The service writes all of the updated loci to Redis potentially overwriting any existing values.

Variant Calling Service

The Variant Calling Service is not a stream-based service, instead it provides a querying interface into Rheos via its $callVariants(region, calling\_threshold)$ operation, where the user provides a region of interest and variant quality threshold and the service creates a VCF file of all called variants in the region that meet the variant quality criteria.

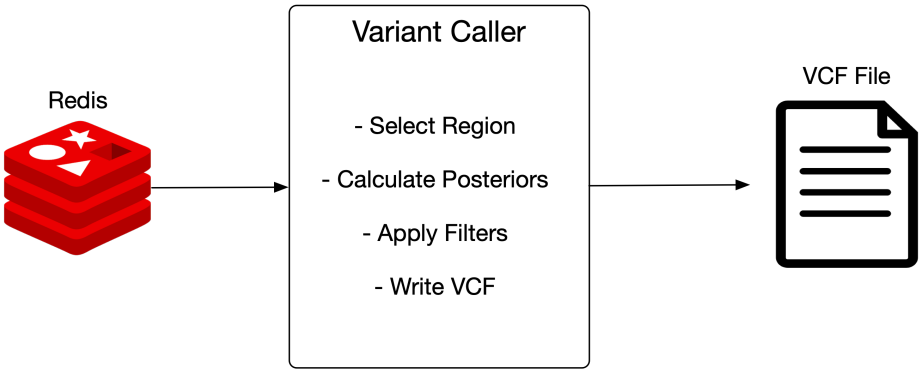


Figure 5.40: The Variant Calling Service data processing schematic.

This service communicates with the Redis locus store (see Figure 5.40) and retrieves all of the loci that match the query region. For each locus it calculates the posterior genotype probabilities via Equation 5.11. Variants the exceed the user provided

quality threshold will be written to the output VCF file. It is possible to apply other variant filtering criteria at this stage to improve calling stringency. As the service requires access to secondary storage it is deployed as a StatefulSet under Kubernetes.

Insert Size Filtering Service

The Insert Size Filtering Service is a streaming service of Filter type. It implements the $filterDiscordantPairs()$ operation in order to obtain a stream of read pairs that are fit for germline deletion calling as described in Section 5.3.4. The service initially only accumulates read pairs in order to estimate the insert size distribution. The default initial sample size for this purpose is 100000 read pairs, but this value is controlled by the user. Using this sample we estimate the median insert size, and subsequently the Median Absolute Deviation of the insert size which acts as our cutoff for discordantly mapped reads. Once the estimate is obtained we send the accumulated reads through the rest of the filtering process, followed by any other new read pairs that are observed in the data stream. Read pairs that pass filtering are sent to the *discordant_read_pairs* kafka topic.

Insert Size Clustering Service

The Insert Size Clustering service implements the $addReadPair()$ streaming operation and the $callDeletions()$ querying operation. $addReadPair()$ observes the *discordant_read_pairs* data stream and adds any messages that are received into the internal data structures used for deletion calling.

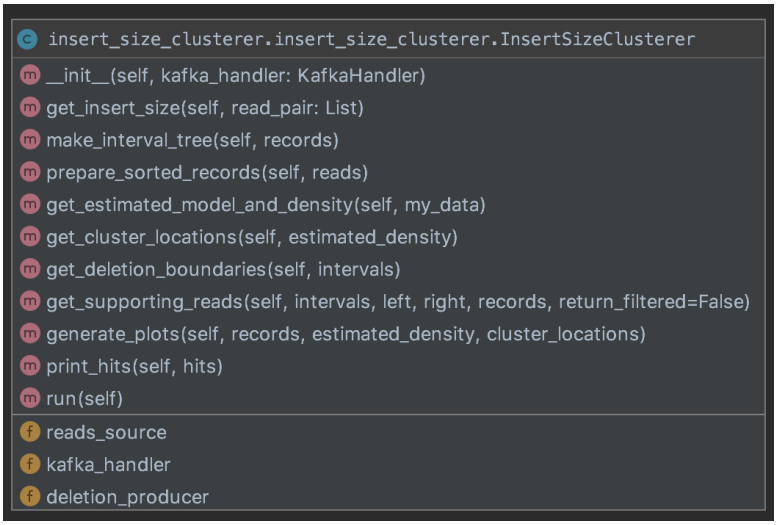


Figure 5.41: The Insert Size Clusterer class methods.

The reads are stored as both a List for general handling and clustering, as well as an Interval Tree (see Figure 5.21) - a type of balanced binary search tree useful for finding interval overlaps. We use an open source implementation of the Interval Tree from https://github.com/chaimleib/intervaltree.
When $callDeletions()$ is executed the accumulated reads are prepared for processing by finding their center and insert size. The read centers are then clustered using the `KernelDensity` object from the `sklearn.neighbors` Python package (see Figure

5.22). Peaks of the estimated distribution are found using the `argrelextrema()` function from the `scipy.signal` package. Given the location of a cluster center we find the set of read pairs that overlap the center via the Interval Tree. Given a pre-constructer Interval Tree t and a cluster center coordinate *center* we obtain the necessary set of intervals by simply calling $overlaps = t[rec.center]$.
For each set of overlap intervals the deletion size is set to be the maximum common overlap between the inner sizes of all such intervals (see Figure 5.23). All overlapping read pairs whose $insert\_size > deletion\_size + insert\_size\_threshold$ are removed from the list of reads that support the deletion, and if the number of remaining reads is $> min\_support\_threshold$ the deletion is called as real.

5.5 Experimental Validation

We focused on two primary use cases in validating the initial implementaion of the Rheos framework. The first is the ability of the framework to accurately perform germline SNP calling. The second is the ability of the framework to accurately perform germline deletion calling. Deployment of the framework onto cloud computing infrastructure was also tested. Detailed performance testing was not performed because of the limited scope of the project.

5.5.1 Deployment

The Rheos framework has been deployed onto the academic cloud computing infrastructure provided by EMBL/EBI Embassy Cloud, an OpenStack environment with 164 CPUs, 744 GB RAM, and 15 TB of block volume storage. Resources were split into 12 virtual machines with the following variety of characteristics:

Table 5.14: Rheos deployment infrastructure

Count	CPU	RAM	Ephemeral Disk
1	4	8GB	60GB
4	8	32GB	100GB
6	16	64GB	100GB
1	32	224GB	100GB

One VM with 4 CPU and 8 GB RAM was designated as the Kubernetes master node, with other small services like the Kubernetes Dashboard and Weave Networking Dashboard deployed to it as well. Other VMs were put into the pool of worker nodes with automated deployment scheduling of Docker containers provided by Kubernetes. A Weave overlay network was set up between all of the nodes in the Kubernetes cluster, creating a private subnet for inter-VM and inter-container communication but inaccessible from the outside.
The following products were set up to support the running of Rheos:

Kafka - A Kubernetes Kafka service was deployed with a redundancy of 3, deploying 3 borkers on 3 separate nodes, to ensure availability and scalability.

Zookeeper - A Kubernetes Zookeeper service was deployed with a redundancy of 3.

Redis - A Kubernetes Redis server was set up with 1 master and 1 replication slave.

Prometheus - Prometheus was set up for monitoring with 1 master, and an agent running on every node.

Grafana - Grafana was set up as a single container for monitoring dashboarding.

Elasticsearch - Elasticsearch was set up as a single server for collection of application logs.

Jenkins - Jenkins was set up as a single server for automated deployment and continuous integration of new builds in the testing environment.

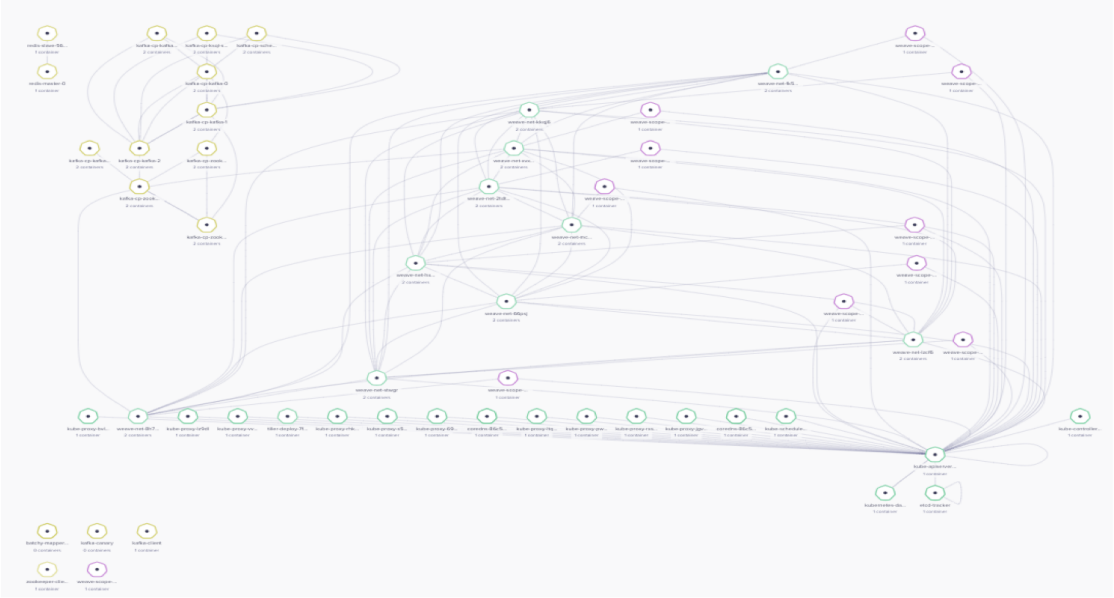


Figure 5.42: Docker containers that are part of the test Rheos deployment.

The services of Rheos were deployed into the Kubernetes environment in the following configuration:

Read Mapping Service - The service was deployed as a set of 2 Kubernetes Pods on separate nodes with access to a 500TB NFS share for reading sample data, and a local copy of the human reference genome, preprocessed for use by minimap2.

Locus Processor Service - The service was deployed as a set of 6 Kubernetes Pods, each with a local copy of the human reference genomes in FASTA format.

Locus Saver Service - The service was deployed as a set of 2 Kubernetes Pods on separate nodes.

Variant Caller Service - The service was deployed as a single Kubernetes Pod, with access to a 500TB NFS share for writing the VCF file.

Insert Size Filtering Service - The service was deployed as a single Kubernetes Pod.

Insert Size Clustering Service - The service was deployed as a single Kubernetes Pod.

The following Kafka topics were set up for management of in-flight data. A replication-factor of 1 was used.

Table 5.15: Rheos Kafka topics.

Name	Partitions
$mapped\_read_pairs\_batchy$	168
$unmapped\_read_pairs\_batchy$	2
$mapped\_read_pairs\_secondary\_batchy$	10
snp_loci_batchy	2
$discordant\_read\_pairs$	1
$deletions$	1

Figure 5.42 shows the set of Docker containers that were used in the Rheos Kubernetes deployment. In addition to the Kubernetes test environment, local testing of individual components was performed on a MacBook Pro 2018, with a 6-core 2.9 GHz Intel Core i9 processor, and 32 GB RAM.

5.5.2 Sample Selection and Preparation

To assess the ability of Rheos to perform germline SNP and SV calling we have selected a well characterized genomic sample from the Genome in a Bottle Consortium[212], namely the NA12878 sample that was first sequenced in the context of the 1000 Genomes Project, but has subsequently been sequenced by a variety of orthogonal sequencing technologies to very high levels of coverage and is very well studied. Choosing this sample allows us to work with data that is known to be of high quality and one that has a high quality set of published genomics variants to compare Rheos calls against.
We downloaded the sample RMNISTHS_30xdownsample.bam a 147GB file from the GIAB FTP repository at ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/NA12878/NIST_NA12878_HG001_HiSeq_300x/. This sample is a high coverage 300x sequencing run of the Illumina HiSeq 2500 instrument using a 2x148 bp paired end library. The original data has been mapped to the GRCh37d5 human reference genome using BWA-MEM, and subsequently downsampled to 30x average coverage.
We performed our experiments on Chromosome 20 of the sample, which is a DNA molecule that is 63025520 nucleotides long. We extracted raw unaligned reads for Chromosome 20 from the BAM file into a separate FASTQ file. The size of this FASTQ file is 4.3 GB and it contains 13424120 reads. All of the alignment information was discarded in the transformation from BAM to FASTQ.
The GRCh37 human reference genome was obtained from the 1000 Genomes Project FTP site at ftp://ftp.ncbi.nih.gov/1000genomes/ftp/technical/reference/phase2_reference_assembly_sequence/. The reference is in FASTA format and is 3GB in size. It contains the reference sequence for human chromosomes 1-22, X, Y, MT (mitochondrial) as well as a set of decoy sequences and unplaced

contigs. We use the entire reference genome because reads that were mapped to Chromosome 20 by BWA-MEM may be mapped elsewhere by the minimap2 aligner. We pre-process the reference for use by minimap2 to generate a minimizer index genome.mmi using default parameters.

5.5.3 Germline SNP Calling

For germline SNP calling we put together a pipeline using Rheos services that takes a FASTQ sample file, streams the reads, maps them to a reference genome, processes the variant evidence contained in the reads, save the resulting genomic loci in a data store, and performs variant calling, outputting a result VCF (see Figure 5.43). We then compare the results of the VCF output to the output of other well-established variant callers.

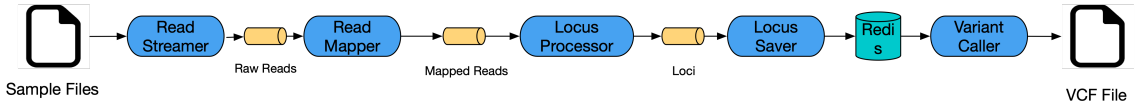


Figure 5.43: The Rheos germline SNP calling pipeline.

In our initial implementation we have actually combined the first two steps, so that the Read Mapping Service reads a FASTQ file directly from disk, rather than a stream, and then streams the already mapped reads for further processing. We use a single instance of the Read Mapping Service for this task invoking it with the following parameters:
file -i data/giab/RMNISTHS_30xdownsample_chr20.fastq
-r data/reference/genome.mmi
-g 20:1-63025520
-k kafka:9092
–bin_size=10000
–num_bins=100
–check_freq=1000
–log_level=DEBUG
–root_log_level=DEBUG
–logger_type=stream

These parameters indicate that we are processing the entire Chromosome 20 from a FASTQ file, that we would like to use 100 bins of capacity 10000 read pairs per partition in the `PartitionedReservoirSet`, and that we would like to check which bins are full once every 1000 mapped read pairs. Partition size was left at the default 20000000 bases. Using a single CPU and an allotment of 16GB RAM this process took 1 hour and 23 minutes to complete in full, processing 13424120 reads out of which 245052 reads were unpaired, sending 12482 bins for donwstream processing. Looking at the system resource usage during the execution (see Figure 5.44) we see that this process appears to be CPU bound, with full utilization over almost the entirety of the processing cycle. Memory usage grows initially as more and more bins accumulate mapped reads, and peaks at 15.5GB as the rate of bin filling is counteracted by sending off bins that are full. Due to the stateless nature of the process, the wall time of the mapping process can be sped up arbitrarily by splitting

the input file into chunks and parallelizing across CPUs.

Figure 5.44: System resource usage by Read Mapping Service during alignment of Chromosome 20 of NA12878 form GIAB.

We next investigate the operation of the Locus Processor Service by simultaneously executing 4 instances of the service, one for each partition of Chromosome 20, on the stream data produced by the Read Mapping Service. We invoke the Locus Processor Service with the following set of parameters:
service
-r data/reference/genome.fa
-g [20:1-20000000 | 20:20000001-40000000 | 20:40000001-60000000 | 20:60000001-63025520]
-k kafka:9092
–log_level=DEBUG
–root_log_level=DEBUG
–logger_type=stream

Here, each of the service instances has a separate partition and genomic region assigned via the -g parameter. This process completed in 29 minutes and 42 seconds running four instances simultaneously and took 0.452 seconds to process a single message on average, sending an average of 32908 updated loci for downstream processing per message. Consulting the operational metrics collected during the run (Figure 5.45) we see a fairly stable CPU and memory utilization profiles, however significant use of swap is observed (bottom right panel), indicating that if the amount of memory per process was increased, performance could be improved.

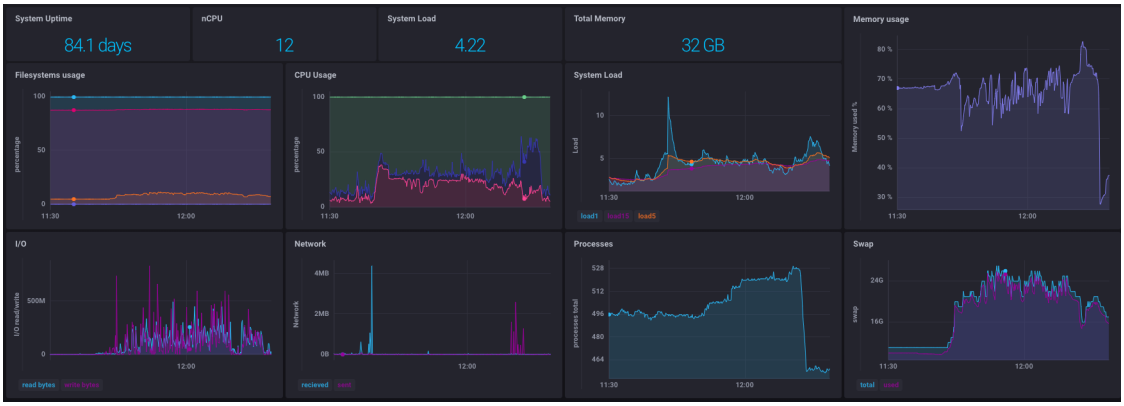


Figure 5.45: System resource usage by Locus Processor Service during processing of Chromosome 20 of NA12878 form GIAB.

We run a single instance of the Locus Saver Service to process locus updates with

the following parameters:
service
-k kafka:9092
-r redis-master -p 6379
–log_level=DEBUG
–root_log_level=DEBUG
–logger_type=stream
This is a fairly lightweight service and takes 18 minutes and 58 seconds to process all of the messages produced by the 4 instances of Locus Processor and save them to Redis. 961 messages in total are processed, with average processing time of 0.72 seconds per message. Figure 5.46 shows operational metrics collected from the service and the Redis server during the execution runtime. As can be seen, the execution fully utilizes the CPU of the service host (likely for serialization/deserialization of messages). Resources of the Redis server are only midly utilized indicating the ability to run several instances of the Locus Saver per instance of Redis. The bottom-left panel shows the impact of automated disk persistence of Redis in-memory data. This feature can be turned off to obtain higher throughput at the expense of lower availability, i.e. if Redis crashes data will need to be reprocessed.
The last component in the processing pipeline is the Variant Calling Service. We invoke this service with the following parameters:
file
-g 20:1-63025520
-r redis-master -p 6379
–calling_threshold=50
–max_vqual=5000
-t variant_caller_pkg/variant_caller/rheos_vcf_template.vcf
-o data/na12878_20_1_63025520_rheos_latest.vcf
–log_level=DEBUG
–root_log_level=DEBUG
–logger_type=stream
These parameters indicate that we are calling variants on the entirety of Chromosome 20, that we only include variants exceeding variant quality of 50 (i.e. 1 in 10000 variants is estimated to be a false positive), and with a variant quality ceiling of 5000 (if a variant has vqual >5000 it will be capped at 5000). Running a single instance of this service took 1 hour and 21 minutes to complete calling on the entire Chromosome 20 of the sample, evaluating 12754171 potential variant sites from Redis. Figure 5.47 shows steady CPU utilization and relatively low memory use by the service (top panels), and steady but low CPU utilization, but high memory consumption by Redis (bottom panels), as expected.

5.6 Conclusions

Rheos is great!

Figure 5.46: System resource usage by Locus Saver Service during processing of Chromosome 20 of NA12878 form GIAB.

Figure 5.47: System resource usage by Variant Caller Service during processing of Chromosome 20 of NA12878 form GIAB.

Chapter 6

Discussion and Conclusion

6.1 Validation and Conclusion

We have deployed Butler in a production setting at the EMBL/EBI's Embassy Cloud in a configuration that utilizes 1500 CPUs, 6 TB RAM, 1 PB of Isilon storage accessed over NFS, and 40 TB of block-storage. Furthermore, we have built a series of workflows that facilitate the large-scale cancer genomics analyses carried out by the Germline Working Group of the Pan Cancer Analysis of Whole Genomes project, including:

- Germline SNV discovery
- Germline SNV joint-genotyping
- Germline SV genotyping
- Variant Filtration
- Sample submission

Using these workflows wSe have carried out a number of analyses on a 725TB data set of 2834 cancer patients' DNA samples consuming a total of 546,552 CPU hours. Each analysis took no longer than two weeks to complete and utilized only 1.5% - 2.2% of the overall compute capacity for management overhead. On several occasions we were able detect large scale cluster instability and program crashes utilizing the Operational Management system and take corrective action with a minimal impact on overall cluster productivity.
Subsequent to the success of these analyses several research groups from the European Bioinformatics Institute, Ontario Institute for Cancer Research, Francis Crick Institute, and the Centre for Genomic Regulation have expressed their interest in utilizing Butler for their own large scale analyses in the cloud.
Based on the adherence of the Butler design and implementation to the stated set of requirements, and sustained successful production operation in a large scale deployment on a multitude of scientific analyses of significant scope and size, we conclude that the Butler framework is an effective tool for large scale scientific workflow management in the cloud.

6.2 Future Direction

Butler has been created to facilitate scientific analyses at scale and we have demonstrated that it is able to successfully perform at the level required for today's big data initiatives in the genomics domain. There are projects on the horizon, however, that are one to two orders of magnitude larger than the current biggest projects, these include the UK's 100,000 Genomes Project[130], and the US Precision Medicine Initiative[26] (with up to 1,000,000 genomes). This means that in order to not have to proportionately increase the timeline for theses projects the computational infrastructure will have to be scaled up instead. It is thus imperative for Butler's continued relevance to be able to ascertain the framework's performance level at 1 or 2 orders of magnitude larger than the current 1500 core empirically obtained result. The most immediate opportunity to do so will come up in 2017 when the EMBL/EBI's Embassy Cloud will be upgraded to 5000 CPU cores and Butler has been invited to take part in the stress-testing of the upgraded cloud.

It is important to grow the library of workflows that are readily available for the Butler system to make the framework more appealing to new users. The Technical Working Group of the PCAWG project is in the process of migrating all of the main computational pipelines that have been used in the project into Docker[134] containers. Although the workflows that have been developed for the Germline Working Group have not yet been ported to Docker, Airflow, the workflow system underlying Butler has support for running Docker containers. Thus, a key next step for growing the library of Butler workflows lies in the adaptation of the core PCAWG workflows to be able to easily run them on a Butler instance. This would allow Butler to offer a comprehensive set of next generation sequencing workflows that are used for cancer genomics analysis.

Deploying Butler to a larger variety of environments will confirm the multi-cloud purpose of the framework and allow for the development of a richer set of configuration and provisioning profiles, as necessitated by the differences between deployment environments. On the basis of the already completed analyses for the PCAWG Germline Working Group, the Butler framework has also been selected to help deliver the science demonstrator work packet of the European Open Science Cloud Pilot[51] initiative that is launching in 2017. Additionally, de.NBI - The German Network for Bioinformatics Infrastructure[81] which is working to establish a German academic cloud computing environment for bioinformatics research will be using Butler to deliver a number of new bioinformatics pipelines on its cloud in 2017.

Thus, over the course of the next 12 months the focus of Butler development will be on supporting improved scalability, developing a richer set of computational pipelines and operating in a number of new cloud computing environments. These steps should result in a more robust, feature rich, and useful tool.

Appendix A

Code Listings

Listing 2: Terraform configuration of a worker VM

```
provider "openstack" {
        user_name = "${var.user_name}"
        password = "${var.password}"
        tenant_name = "${var.tenant_name}"
        auth_url = "${var.auth_url}"
}

resource "openstack_compute_instance_v2" "worker" {
          image_id = "${var.image_id}"
        flavor_name = "s1.massive"
        security_groups = ["internal"]
        name = "${concat("worker-", count.index)}"
        network = {
                uuid = "${var.main_network_id}"
        }
        connection {
                user = "${var.user}"
                 key_file = "${var.key_file}"
                 bastion_key_file = "${var.bastion_key_file}"
                 bastion_host = "${var.bastion_host}"
                 bastion_user = "${var.bastion_user}"
                 agent = "true"

        }
        count = "175"
        key_pair = "${var.key_pair}"
        provisioner "remote-exec" {
                inline = [
                        "sudo mv /home/centos/saltstack.repo
                        ↪  /etc/yum.repos.d/saltstack.repo",
                        "sudo yum install salt-minion -y",
                        "sudo service salt-minion stop",
```

```
32                          "echo 'master: ${var.salt_master_ip}' | sudo tee  -a
                            ↪  /etc/salt/minion",
33                          "echo 'id: ${concat("worker-", count.index)}' | sudo tee
                            ↪  -a /etc/salt/minion",
34                          "echo 'roles: [worker, germline, consul-client]' | sudo
                            ↪  tee -a /etc/salt/grains",
35                          "sudo hostname ${concat("worker-", count.index)}",
36                          "sudo service salt-minion start"
37                  ]
38          }
39  }
```

Listing 3: Terraform configuration of a security group

```
1   resource "openstack_compute_secgroup_v2" "internal" {
2           name = "internal"
3           description = "Allows communication between instances"
4           #SSH
5           rule {
6                   from_port = 22
7                   to_port = 22
8                   ip_protocol = "tcp"
9                   self = "true"
10          }
11          #Saltstack
12          rule {
13                  from_port = 4505
14                  to_port = 4506
15                  ip_protocol = "tcp"
16                  self = "true"
17          }
18  }
```

Listing 4: Salt Pillar for specifying test data location.

```
1   test_data_sample_path: /shared/data/samples
2
3   test_data_base_url: http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/
4
5   test_samples:
6     NA12874:
7       -
8         - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam
9         - 88a7a346f0db1d3c14e0a300523d0243
10      -
```

```
      - NA12874.chrom11.ILLUMINA.bwa.CEU.low_coverage.20130415.bam.bai
      - e61c0668bbaacdea2c66833f9e312bbb
```

Listing 5: Using Salt Mine to look up a server's IP Address.

```
consul-client:
  service.running:
    - enable: True
    - watch:
      - file: /etc/opt/consul.d/*
{%- set servers = salt['mine.get']('roles:(consul-server|consul-bootstrap)',
↪  'network.ip_addrs', 'grain_pcre').values() %}
{%- set node_ip = salt['grains.get']('ip4_interfaces')['eth0'] %}
# Create a list of servers that can be used to join the cluster
{%- set join_server = [] %}
{%- for server in servers if server[0] != node_ip %}
{% do join_server.append(server[0]) %}
{%- endfor %}
join-cluster:
  cmd.run:
    - name: consul join {{ join_server[0] }}
    - watch:
      - service: consul-client
```

Listing 6: Using Top File to map States to Roles.

```
base:
  '*':
    - consul
    - dnsmasq
    - collectd
  'G@roles:monitoring-server':
    - influxdb
    - grafana
  'G@roles:job-queue':
    - rabbitmq
```

Listing 7: Collectd configuration for metrics collection.

```
# Read metrics about cpu usage
[[inputs.cpu]]
  ## Whether to report per-cpu stats or not
  percpu = true
  ## Whether to report total system cpu stats or not
```

```
  totalcpu = true
  ## If true, collect raw CPU time metrics.
  collect_cpu_time = false
  ## If true, compute and report the sum of all non-idle CPU states.
  report_active = false


# Read metrics about disk usage by mount point
[[inputs.disk]]
  ## By default, telegraf gather stats for all mountpoints.
  ## Setting mountpoints will restrict the stats to the specified mountpoints.
  # mount_points = ["/"]

  ## Ignore some mountpoints by filesystem type. For example (dev)tmpfs (usually
  ## present on /run, /var/run, /dev/shm or /dev).
  ignore_fs = ["tmpfs", "devtmpfs", "devfs"]


# Read metrics about disk IO by device
[[inputs.diskio]]
  ## By default, telegraf will gather stats for all devices including
  ## disk partitions.
  ## Setting devices will restrict the stats to the specified devices.
  # devices = ["sda", "sdb"]
  ## Uncomment the following line if you need disk serial numbers.
  # skip_serial_number = false
  #
  ## On systems which support it, device metadata can be added in the form of
  ## tags.
  ## Currently only Linux is supported via udev properties. You can view
  ## available properties for a device by running:
  ## 'udevadm info -q property -n /dev/sda'
  # device_tags = ["ID_FS_TYPE", "ID_FS_USAGE"]
  #
  ## Using the same metadata source as device_tags, you can also customize the
  ## name of the device via templates.
  ## The 'name_templates' parameter is a list of templates to try and apply to
  ## the device. The template may contain variables in the form of '$PROPERTY' or
  ## '${PROPERTY}'. The first template which does not contain any variables not
  ## present for the device is used as the device name tag.
  ## The typical use case is for LVM volumes, to get the VG/LV name instead of
  ## the near-meaningless DM-0 name.
  # name_templates = ["$ID_FS_LABEL","$DM_VG_NAME/$DM_LV_NAME"]


# Get kernel statistics from /proc/stat
[[inputs.kernel]]
  # no configuration
```

```
54
55
56 # Read metrics about memory usage
57 [[inputs.mem]]
58   # no configuration
59
60
61 # Get the number of processes and group them by status
62 [[inputs.processes]]
63   # no configuration
64
65
66 # Read metrics about swap memory usage
67 [[inputs.swap]]
68   # no configuration
69
70
71 # Read metrics about system load & uptime
72 [[inputs.system]]
73   # no configuration
```

Listing 8: Filebeat Prospector configuration.

```
1  [collectd]
2    enabled = true
3    bind-address = ":8096"
4    database = "metrics"
5    retention-policy = ""
6    batch-size = 5000
7    batch-pending = 10
8    batch-timeout = "10s"
9    read-buffer = 0
10   typesdb = "/usr/share/collectd/types.db"
```

Listing 9: TICKscript for alerting on CPU value.

```
1 // Parameters
2 var info = 70
3 var warn = 80
4 var crit = 90
5 var infoSig = 2.5
6 var warnSig = 3
7 var critSig = 3.5
8 var period = 10s
9 var every = 10s
```

```
10
11 // Dataframe
12 var data = stream
13   |from()
14     .database('metrics')
15     .retentionPolicy('default')
16     .measurement('cpu_value')
17     .where(lambda: "type" == 'percent' AND "type_instance" == 'idle')
18   |eval(lambda: 100 - "value")
19     .as('used')
20   |window()
21     .period(period)
22     .every(every)
23   |mean('used')
24     .as('stat')
25
26 // Thresholds
27 var alert = data
28   |eval(lambda: sigma("stat"))
29     .as('sigma')
30     .keep()
31   |alert()
32     .id('{{ index .Tags "host"}}/cpu_value')
33     .message('{{ .ID }}:{{ index .Fields "stat" }}')
34     .info(lambda: "stat" > info OR "sigma" > infoSig)
35     .warn(lambda: "stat" > warn OR "sigma" > warnSig)
36     .crit(lambda: "stat" > crit OR "sigma" > critSig)
37
38 // Alert
39 alert
40   .log('/tmp/cpu_alert_log_2.txt')
```

Listing 10: TICKscript for handling dead VMs.

```
1  {% raw %}
2  var db = 'telegraf'
3  var rp = 'autogen'
4  var measurement = 'system'
5  var groupBy = ['host']
6  var whereFilter = lambda: TRUE
7  var period = 30s
8  var name = 'Host Deadman'
9  var idVar = name + ':{{.Group}}'
10 var blah = '{{index .Tags "host"}}'
11 var message = 'The host {{index .Tags "host"}} is offline as of {{.Time}}.'
12 var messageN = 'The host {{index .Tags "host"}} is back online at {{.Time}}.'
13 var idTag = 'alertID'
```

```
14 var levelTag = 'level'
15 var messageField = 'message'
16 var durationField = 'duration'
17 var outputDB = 'chronograf'
18 var outputRP = 'autogen'
19 var outputMeasurement = 'alerts'
20 var triggerType = 'deadman'
21 var threshold = 0.0
22 var data = stream
23     |from()
24         .database(db)
25         .retentionPolicy(rp)
26         .measurement(measurement)
27         .groupBy(groupBy)
28         .where(whereFilter)
29
30 var trigger = data
31     |deadman(threshold, period)
32         .stateChangesOnly()
33         .message('{{ if eq .Level "CRITICAL" }}' + message + '{{else}}' +
           ↪  messageN + '{{end}}')
34         .id(idVar)
35         .idTag(idTag)
36         .levelTag(levelTag)
37         .messageField(messageField)
38         .durationField(durationField)
39         .slack()
40         .channel('#embassyalerts')
41 {% endraw %}
42         .exec('butler_healing_agent', 'relaunch-worker', '-t', '{{
           ↪  pillar['terraform_files'] }}', '-s', '{{ pillar['terraform_state']
           ↪  }}', '-v', '{{ pillar['terraform_vars'] }}', '-p', '{{
           ↪  pillar['terraform_provider'] }}')
43 {% raw %}
44 trigger
45     |eval(lambda: "emitted")
46         .as('value')
47         .keep('value', messageField, durationField)
48     |influxDBOut()
49         .create()
50         .database(outputDB)
51         .retentionPolicy(outputRP)
52         .measurement(outputMeasurement)
53         .tag('alertName', name)
54         .tag('triggerType', triggerType)
55
56 trigger
57     |httpOut('output')
```

```
{% endraw %}
```

Listing 11: Butler healing agent code for restarting the Airflow Scheduler.

```
def call_command(command, cwd=None):
  try:
      logging.debug("About to invoke command: " + command)
      my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
      logging.debug("Command output is: " + my_output)
      return my_output
  except CalledProcessError as e:
      logging.error("An error occurred! Command output is: " +
      ↪  e.output.decode("utf-8") )
      raise

def is_critical(level):
  return level == "CRITICAL"

def parse_alert_data():
  return json.loads(sys.stdin.read())

def get_host_name(alert_data):
  return alert_data["data"]["series"][0]["tags"]["host"]

def restart_service(host, service_name):
  call_command("pepper {} service.restart {}".format(host, service_name), None)

def parse_args():
  my_parser = argparse.ArgumentParser()

  sub_parsers = my_parser.add_subparsers()

  common_args_parser = argparse.ArgumentParser(
      add_help=False, conflict_handler='resolve')

  restart_airflow_scheduler_parser = sub_parsers.add_parser(
      "restart-airflow-scheduler", parents=[common_args_parser],
      ↪  conflict_handler='resolve')

  ↪  restart_airflow_scheduler_parser.set_defaults(func=restart_airflow_scheduler_command)

def restart_airflow_scheduler_command(args, alert_data):
  if is_critical(alert_data["level"]):
      restart_service("-G 'roles:tracker'", "airflow-scheduler")
```

Listing 12: Butler healing agent code for relaunching a failed VM.

```
def call_command(command, cwd=None):
  try:
      logging.debug("About to invoke command: " + command)
      my_output = check_output(command, shell=True, cwd=cwd, stderr=STDOUT)
      logging.debug("Command output is: " + my_output)
      return my_output
  except CalledProcessError as e:
      logging.error("An error occurred! Command output is: " +
      ↪  e.output.decode("utf-8") )
      raise

def is_critical(level):
  return level == "CRITICAL"

def parse_alert_data():
  return json.loads(sys.stdin.read())

def get_host_name(alert_data):
  return alert_data["data"]["series"][0]["tags"]["host"]

def restart_service(host, service_name):
  call_command("pepper {} service.restart {}".format(host, service_name), None)

def parse_args():
  my_parser = argparse.ArgumentParser()

  sub_parsers = my_parser.add_subparsers()

  common_args_parser = argparse.ArgumentParser(
    add_help=False, conflict_handler='resolve')

  relaunch_worker_parser = sub_parsers.add_parser(
      "relaunch-worker", parents=[common_args_parser],
      ↪  conflict_handler='resolve')
  relaunch_worker_parser.add_argument(
      "-t", "--terraform_location", help="Location of the terraform definition
      ↪  files.",
      dest="terraform_location", required=True)
  relaunch_worker_parser.add_argument(
      "-s", "--terraform_state_location", help="Location of the terraform state
      ↪  file.",
      dest="terraform_state_location", required=True)
  relaunch_worker_parser.add_argument(
      "-v", "--terraform_var_file_location", help="Location of the terraform vars
      ↪  file.",
      dest="terraform_var_file_location", required=True)
  relaunch_worker_parser.add_argument(
      "-p", "--terraform_provider", help="The terraform provider to use.",
```

```
44          choices = provider_list,
45          dest="terraform_provider", required=True)
46      relaunch_worker_parser.set_defaults(func=relaunch_worker_command)
47
48      def is_key_present(key_data, host_name):
49      parsed_key_data = json.loads(key_data)
50      return_data = parsed_key_data["return"][0]["data"]["return"]
51
52      if "minions" in return_data:
53          return_vals = return_data["minions"]
54          for val in return_vals:
55              if val == host_name:
56                  return True
57
58      return False
59
60  def locate_minon_key(host_name):
61      minion_connect_try = 1
62      while minion_connect_try <= MINION_CONNECT_MAX_RETRIES:
63          logging.info("Attempt #{} of {} to retrieve minion key for host {} from the
            ↪  master.".format(minion_connect_try, MINION_CONNECT_MAX_RETRIES,
            ↪  host_name))
64          key_data = call_command("pepper --client=wheel key.name_match
            ↪  match={}".format(host_name))
65          logging.debug("Retrieved key data: " + key_data)
66          if is_key_present(key_data, host_name):
67              return True
68          else:
69              logging.debug("Key data for host {} not found at time {}. Sleeping for
                ↪  {} seconds.".format(host_name, datetime.now(),
                ↪  MINION_CONNECT_SLEEP_PERIOD))
70              time.sleep(MINION_CONNECT_SLEEP_PERIOD)
71              minion_connect_try = minion_connect_try + 1
72
73      return False
74
75  def relaunch_worker_command(args, alert_data):
76      if is_critical(alert_data["level"]):
77          host_name = get_host_name(alert_data)
78
79
80          tf_location = args.terraform_location
81          tf_state_location = args.terraform_state_location
82          tf_var_file_location = args.terraform_var_file_location
83          tf_resource = provider_resource_lookup[args.terraform_provider]
84          worker_number = host_name.split("-")[1]
85
86          call_command("pepper --client=wheel key.delete match={}".format(host_name))
```

```
        call_command("terraform taint -lock=false -state={}
        ↪  {}.worker.{}".format(tf_state_location, tf_resource, worker_number),
        ↪  tf_location)
        call_command("terraform apply -lock=false -state={} --var-file {}
        ↪  -auto-approve".format(tf_state_location, tf_var_file_location),
        ↪  tf_location)

        locate_minon_key(host_name)

        call_command("pepper '*' mine.update")
        call_command("pepper {} state.apply dnsmasq".format(host_name))
        call_command("pepper {} state.apply consul".format(host_name))
        call_command("pepper {} state.highstate".format(host_name))
```

Listing 13: Consul service definition for PostgreSQL.

```
{
        "results_base_path": "/shared/data/results/discovery/",
        "results_local_path": "/tmp/discovery/",
        "freebayes": {
                "mode": "discovery",
                "flags": "--min-repeat-entropy 1
                ↪  --report-genotype-likelihood-max"
        }
}
```

Listing 14: Source code for the freebayes workflow.

```
from airflow import DAG
from airflow.operators import BashOperator, PythonOperator
from datetime import datetime, timedelta

import os
import logging
from subprocess import call

import tracker.model
from tracker.model.analysis_run import *
from tracker.util.workflow_common import *


def run_freebayes(**kwargs):

    config = get_config(kwargs)
    logger.debug("Config - {}".format(config))
```

```
18
19      sample = get_sample(kwargs)
20
21      contig_name = kwargs["contig_name"]
22      contig_whitelist = config.get("contig_whitelist")
23
24
25      if not contig_whitelist or contig_name in contig_whitelist:
26
27          sample_id = sample["sample_id"]
28          sample_location = sample["sample_location"]
29
30          result_path_prefix = config["results_local_path"] + "/" + sample_id
31
32          if (not os.path.isdir(result_path_prefix)):
33              logger.info(
34                  "Results directory {} not present,
                    ↪  creating.".format(result_path_prefix))
35              os.makedirs(result_path_prefix)
36
37          result_filename = "{}/{}_{}.vcf".format(
38              result_path_prefix, sample_id, contig_name)
39
40          freebayes_path = config["freebayes"]["path"]
41          freebayes_mode = config["freebayes"]["mode"]
42          freebayes_flags = config["freebayes"]["flags"]
43
44          reference_location = config["reference_location"]
45
46          if freebayes_flags == None:
47              freebayes_flags = ""
48
49          if freebayes_mode == "discovery":
50              freebayes_command = "{} -r {} -f {} {} {} > {}".\
51                  format(freebayes_path,
52                         contig_name,
53                         reference_location,
54                         freebayes_flags,
55                         sample_location,
56                         result_filename)
57          elif freebayes_mode == "regenotyping":
58              variants_location = config["variants_location"]
59
60              freebayes_command = "{} -r {} -f {} -@ {} {} {} > {}".\
61                  format(freebayes_path,
62                         contig_name,
63                         reference_location,
64                         variants_location[contig_name],
```

```
                        freebayes_flags,
                        sample_location,
                        result_filename)
        else:
             raise ValueError("Unknown or missing freebayes_mode -
             ↪  {}".format(freebayes_mode))

        call_command(freebayes_command, "freebayes")

        compressed_sample_filename = compress_sample(result_filename, config)
        generate_tabix(compressed_sample_filename, config)
        copy_result(compressed_sample_filename, sample_id, config)
    else:
        logger.info(
            "Contig {} is not in the contig whitelist,
            ↪  skipping.".format(contig_name))


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime.datetime(2020, 01, 01),
    'email': ['airflow@airflow.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG("freebayes", default_args=default_args,
          schedule_interval=None, concurrency=10000, max_active_runs=2000)


start_analysis_run_task = PythonOperator(
    task_id="start_analysis_run",
    python_callable=start_analysis_run,
    provide_context=True,
    dag=dag)


validate_sample_task = PythonOperator(
    task_id="validate_sample",
    python_callable=validate_sample,
    provide_context=True,
    dag=dag)

validate_sample_task.set_upstream(start_analysis_run_task)

```

```
complete_analysis_run_task = PythonOperator(
    task_id="complete_analysis_run",
    python_callable=complete_analysis_run,
    provide_context=True,
    dag=dag)

for contig_name in tracker.util.workflow_common.CONTIG_NAMES:
    freebayes_task = PythonOperator(
        task_id="freebayes_" + contig_name,
        python_callable=run_freebayes,
        op_kwargs={"contig_name": contig_name},
        provide_context=True,
        dag=dag)

    freebayes_task.set_upstream(validate_sample_task)

    complete_analysis_run_task.set_upstream(freebayes_task)
```

Listing 15: Saltstack state for workflow deployment.

```
current_runs = run_session.query(Configuration.config[("sample","
↪  sample_id")].astext).\
        join(AnalysisRun, AnalysisRun.config_id == Configuration.config_id).\
        join(Analysis, Analysis.analysis_id == AnalysisRun.analysis_id).\
        filter(and_(Analysis.analysis_id == analysis_id, AnalysisRun.run_status
        ↪  != tracker.model.analysis_run.RUN_STATUS_ERROR)).all()

available_samples = sample_session.query(PCAWGSample.index.label("index"),
↪  sample_id.label("sample_id"), sample_location.label("sample_location")).\
        join(SampleLocation, PCAWGSample.index == SampleLocation.donor_index).\
        filter(and_(sample_location != None, sample_id.notin_(current_runs))).\
        limit(num_runs).all()
```

Listing 16: Butler Analysis configuration for SNP genotyping.

```
{
        "variants_location": {
            "1": "/freebayes.chr_1.sites.snv_indel.annot.final.vcf.gz",
            "2": "/freebayes.chr_2.sites.snv_indel.annot.final.vcf.gz",
            "3": "/freebayes.chr_3.sites.snv_indel.annot.final.vcf.gz",
            "4": "/freebayes.chr_4.sites.snv_indel.annot.final.vcf.gz",
            "5": "/freebayes.chr_5.sites.snv_indel.annot.final.vcf.gz",
            "6": "/freebayes.chr_6.sites.snv_indel.annot.final.vcf.gz",
            "7": "/freebayes.chr_7.sites.snv_indel.annot.final.vcf.gz",
            "8": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
```

```
            "9": "/freebayes.chr_8.sites.snv_indel.annot.final.vcf.gz",
            "10": "/freebayes.chr_10.sites.snv_indel.annot.final.vcf.gz",
            "11": "/freebayes.chr_11.sites.snv_indel.annot.final.vcf.gz",
            "12": "/freebayes.chr_12.sites.snv_indel.annot.final.vcf.gz",
            "13": "/freebayes.chr_13.sites.snv_indel.annot.final.vcf.gz",
            "14": "/freebayes.chr_14.sites.snv_indel.annot.final.vcf.gz",
            "15": "/freebayes.chr_15.sites.snv_indel.annot.final.vcf.gz",
            "16": "/freebayes.chr_16.sites.snv_indel.annot.final.vcf.gz",
            "17": "/freebayes.chr_17.sites.snv_indel.annot.final.vcf.gz",
            "18": "/freebayes.chr_18.sites.snv_indel.annot.final.vcf.gz",
            "19": "/freebayes.chr_19.sites.snv_indel.annot.final.vcf.gz",
            "20": "/freebayes.chr_20.sites.snv_indel.annot.final.vcf.gz",
            "21": "/freebayes.chr_21.sites.snv_indel.annot.final.vcf.gz",
            "22": "/freebayes.chr_22.sites.snv_indel.annot.final.vcf.gz",
            "X": "/freebayes.chr_X.sites.snv_indel.annot.final.vcf.gz",
            "Y": "/freebayes.chr_Y.sites.snv_indel.annot.final.vcf.gz"
        },
        "results_base_path":
        ↪  "/shared/data/results/regenotype_freebayes_discovery/",
        "results_local_path": "/tmp/regenotype_freebayes_discovery/",
        "freebayes": {
                "mode": "regenotyping",
                "flags": "-l"
        }
}
```

Listing 17: Butler Workflow configuration for Data Submission.

```
{
        "gnos": {
                "ebi": {
                        "url": "https://gtrepo-ebi.annailabs.com"
                },
                "osdc_icgc": {
                        "url": "https://gtrepo-osdc-icgc.annailabs.com"
                },
                "osdc_tcga": {
                        "url": "https://gtrepo-osdc-tcga.annailabs.com"
                }
        },
        "rsync": {
                "flags": "-a -v --remove-source-files"
        }
}
```

Listing 18: Butler Analysis configuration for Data Submission.

```
{
        "gnos": {
                "ebi": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
                },
                "osdc_icgc": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_pcawg_gnos_icgc.pem"
                },
                "osdc_tcga": {
                        "key_location":
                        ↪  "/home/airflow/.ssh/sergei_bionimbus_gnos_may.pem"
                }
        },
        "metadata_template_location": "/opt/pcawg-germline/workflows/gtupload-wo
        ↪  rkflow/analysis_template.xml",
        "submission_base_path":
        ↪  "/shared/data/results/freebayes_discovery_gnos_submission/",
        "destination_repo_mapping": {
                "ICGC": "ebi",
                "TCGA": "osdc_tcga"
        }
}
```

Listing 19: Python code for the run_delly function which implements the functionality of the delly_genotype task inside the Butler Delly Workflow.

```
def run_delly(**kwargs):

    config = get_config(kwargs)
    sample = get_sample(kwargs)

    sample_id = sample["sample_id"]
    sample_location = sample["sample_location"]

    result_path_prefix = config["results_local_path"] + "/" + sample_id

    if (not os.path.isdir(result_path_prefix)):
        logger.info(
            "Results directory {} not present,
            ↪  creating.".format(result_path_prefix))
        os.makedirs(result_path_prefix)

    delly_path = config["delly"]["path"]
    reference_location = config["reference_location"]
```

```
    variants_location = config["variants_location"]
    variants_type = config["variants_type"]
    exclude_template_path = config["delly"]["exclude_template_path"]

    result_filename = "{}/{}_{}.bcf".format(
        result_path_prefix, sample_id, variants_type)

    log_filename = "{}/{}_{}.log".format(
        result_path_prefix, sample_id, variants_type)

    delly_command = "{} call -t {} -g {} -v {} -o {} -x {} {} > {}".\
        format(delly_path,
               variants_type,
               reference_location,
               variants_location,
               result_filename,
               exclude_template_path,
               sample_location,
               log_filename)

    call_command(delly_command, "delly")

    copy_result(result_filename, sample_id, config)
```

Listing 20: Butler Delly Workflow analysis configuration to genotype deletions.

```
{
        "variants_location": "/delly_deletion_sites/del.sites.bcf",
        "results_base_path":
        ↪  "/shared/data/results/delly_germline_deletions_14_07_2016/",
        "results_local_path": "/tmp/delly_germline_deletions/",
        "variants_type": "DEL"

}
```

Listing 21: Example of a VCF file (from https://samtools.github.io/hts-specs/VCFv4.3.pdf).

```
{
  ##fileformat=VCFv4.3
  ##fileDate=20090805
  ##source=myImputationProgramV3.1
  ##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
  ##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb
  ↪  2da,species="Homo
  ↪  sapiens",taxonomy=x>
```

```
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA00001 NA00002 NA00003
20 14370 rs6054257 G A 29 PASS NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ
↪  0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20 17330 . T A 3 q10 NS=3;DP=11;AF=0.017 GT:GQ:DP:HQ 0|0:49:3:58,50
↪  0|1:3:5:65,3 0/0:41:3
20 1110696 rs6040355 A G,T 67 PASS NS=2;DP=10;AF=0.333,0.667;AA=T;DB
↪  GT:GQ:DP:HQ 1|2:21:6:23,27 2|1:2:0:18,2 2/2:35:4
20 1230237 . T . 47 PASS NS=3;DP=13;AA=T GT:GQ:DP:HQ 0|0:54:7:56,60
↪  0|0:48:4:51,51 0/0:61:2
20 1234567 microsat1 GTC G,GTCT 50 PASS NS=3;DP=9;AA=G GT:GQ:DP 0/1:35:4
↪  0/2:17:2 1/1:40:3
```

Bibliography

[1] URL: http://www.langmead-lab.org/teaching-materials/.

[2] URL: https://software.broadinstitute.org/gatk/documentation.

[3] URL: http://www.langmead-lab.org/teaching-materials/.

[4] .

[5] Enis Afgan et al. "Galaxy CloudMan: delivering cloud compute clusters". In: *BMC bioinformatics* 11.12 (2010), p. 1.

[6] Charu C Aggarwal. *Data streams: models and algorithms*. Vol. 31. Springer Science & Business Media, 2007.

[7] Daniel Aird et al. "Analyzing and minimizing PCR amplification bias in Illumina sequencing libraries". In: *Genome biology* 12.2 (2011), R18.

[8] Tyler S Alioto et al. "A comprehensive assessment of somatic mutation detection in cancer using whole-genome sequencing". In: *Nature communications* 6 (2015), p. 10001.

[9] Can Alkan, Bradley P Coe, and Evan E Eichler. "Genome structural variation discovery and genotyping". In: *Nature Reviews Genetics* 12.5 (2011), p. 363.

[10] Carmen J Allegra et al. "American Society of Clinical Oncology provisional clinical opinion: testing for KRAS gene mutations in patients with metastatic colorectal carcinoma to predict response to anti–epidermal growth factor receptor monoclonal antibody therapy". In: *Journal of clinical oncology* 27.12 (2009), pp. 2091–2096.

[11] Simon Andrews et al. "FastQC: a quality control tool for high throughput sequence data". In: (2010).

[12] Brian Babcock et al. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, pp. 1–16.

[13] Ralph M Barnes. "Motion and time study." In: (1949).

[14] James O Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013.

[15] David Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[16] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. "Trimmomatic: a flexible trimmer for Illumina sequence data". In: *Bioinformatics* 30.15 (2014), pp. 2114–2120.

[17] Kym M Boycott et al. “Rare-disease genetics in the era of next-generation sequencing: discovery to translation”. In: *Nature Reviews Genetics* 14.10 (2013), pp. 681–691.

[18] Benoit G Bruneau. “The developmental genetics of congenital heart disease”. In: *Nature* 451.7181 (2008), pp. 943–948.

[19] Michael Burrows and David J Wheeler. “A block-sorting lossless data compression algorithm”. In: (1994).

[20] Rajkumar Buyya et al. “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[21] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[22] Xiaoyu Chen et al. “Manta: rapid detection of structural variants and indels for germline and cancer sequencing applications”. In: *Bioinformatics* 32.8 (2015), pp. 1220–1222.

[23] Kristian Cibulskis et al. “ContEst: estimating cross-contamination of human samples in next-generation sequencing data”. In: *Bioinformatics* 27.18 (2011), pp. 2601–2602.

[24] Kristian Cibulskis et al. “Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples”. In: *Nature biotechnology* 31.3 (2013), pp. 213–219.

[25] Peter JA Cock et al. “The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants”. In: *Nucleic acids research* 38.6 (2009), pp. 1767–1771.

[26] Francis S Collins and Harold Varmus. “A new initiative on precision medicine”. In: *New England Journal of Medicine* 372.9 (2015), pp. 793–795.

[27] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. “How to apply de Bruijn graphs to genome assembly”. In: *Nature biotechnology* 29.11 (2011), p. 987.

[28] 1000 Genomes Project Consortium et al. “A map of human genome variation from population-scale sequencing”. In: *Nature* 467.7319 (2010), pp. 1061–1073.

[29] ENCODE Project Consortium et al. “An integrated encyclopedia of DNA elements in the human genome”. In: *Nature* 489.7414 (2012), pp. 57–74.

[30] International HapMap Consortium et al. “The international HapMap project”. In: *Nature* 426.6968 (2003), p. 789.

[31] Wellcome Trust Case Control Consortium et al. “Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls”. In: *Nature* 447.7145 (2007), p. 661.

[32] Charles E Cook et al. “The european bioinformatics institute in 2016: Data growth and integration”. In: *Nucleic acids research* 44.D1 (2016), pp. D20–D26.

[33] Georgiana Copil et al. “Multi-level elasticity control of cloud services”. In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 429–436.

[34] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[35] Antonio Cuevas, Manuel Febrero, and Ricardo Fraiman. "Cluster analysis: a further approach based on density estimation". In: *Computational Statistics & Data Analysis* 36.4 (2001), pp. 441–459.

[36] Vasa Curcin and Moustafa Ghanem. "Scientific workflow systems-can one size fit all?" In: *2008 Cairo International Biomedical Engineering Conference*. IEEE. 2008, pp. 1–9.

[37] Mark J Daly et al. "High-resolution haplotype structure in the human genome". In: *Nature genetics* 29.2 (2001), p. 229.

[38] Petr Danecek et al. "The variant call format and VCFtools". In: *Bioinformatics* 27.15 (2011), pp. 2156–2158.

[39] Charles Darwin. *On the origin of species*. D. Appleton and Co., 1871. DOI: 10.5962/bhl.title.28875.

[40] Mayur Datar et al. "Maintaining stream statistics over sliding windows". In: *SIAM journal on computing* 31.6 (2002), pp. 1794–1813.

[41] J Davis II et al. *Overview of the Ptolemy project*. Tech. rep. ERL Technical Report UCB/ERL, 1999.

[42] Mark A DePristo et al. "A framework for variation discovery and genotyping using next-generation DNA sequencing data". In: *Nature genetics* 43.5 (2011), pp. 491–498.

[43] Wil MP van Der Aalst et al. "Workflow patterns". In: *Distributed and parallel databases* 14.1 (2003), pp. 5–51.

[44] Juliane C Dohm et al. "Substantial biases in ultra-short read data sets from high-throughput DNA sequencing". In: *Nucleic acids research* 36.16 (2008), e105–e105.

[45] Richard Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[46] Mark TW Ebbert et al. "Evaluating the necessity of PCR duplicate removal from next-generation sequencing data and a comparison of approaches". In: *BMC bioinformatics* 17.7 (2016), p. 239.

[47] Genomics England. "The 100,000 genomes project". In: *The* 100 (2016), pp. 1–2.

[48] Thomas Erl. *Service-oriented architecture (SOA): concepts, technology, and design*. 2005.

[49] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.

[50] Patrick Th Eugster et al. "The many faces of publish/subscribe". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[51] *European Open Science Cloud | Open Science - Research and Innovation - European Commission*. URL: http://ec.europa.eu/research/openscience/index.cfm?pg=open-science-cloud (visited on 10/31/2016).

[52] Kelly R Ewen et al. “Identification and analysis of error types in high-throughput genotyping”. In: *The American Journal of Human Genetics* 67.3 (2000), pp. 727–736.

[53] Warren J Ewens. “The sampling theory of selectively neutral alleles”. In: *Theoretical population biology* 3.1 (1972), pp. 87–112.

[54] Michael Farrar. “Striped Smith–Waterman speeds database searches six times over other SIMD implementations”. In: *Bioinformatics* 23.2 (2006), pp. 156–161.

[55] Gregory G Faust and Ira M Hall. “YAHA: fast and flexible long-read alignment with optimal breakpoint detection”. In: *Bioinformatics* 28.19 (2012), pp. 2417–2424.

[56] Paolo Ferragina and Giovanni Manzini. “Opportunistic data structures with applications”. In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE. 2000, pp. 390–398.

[57] Simon A Forbes et al. “COSMIC: exploring the world’s knowledge of somatic mutations in human cancer”. In: *Nucleic acids research* 43.D1 (2015), pp. D805–D811.

[58] G David Forney. “The viterbi algorithm”. In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278.

[59] Rosalind E Franklin and Raymond G Gosling. “Molecular configuration in sodium thymonucleate”. In: *Nature* 171 (1953), pp. 740–741.

[60] Markus Hsi-Yang Fritz et al. “Efficient storage of high throughput DNA sequencing data using reference-based compression”. In: *Genome research* 21.5 (2011), pp. 734–740.

[61] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. “Mining data streams: a review”. In: *ACM Sigmod Record* 34.2 (2005), pp. 18–26.

[62] Hector Garcia-Molina, Frank Germano, and Walter H Kohler. “Debugging a distributed computing system”. In: *IEEE Transactions on Software Engineering* 2 (1984), pp. 210–219.

[63] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016.

[64] Erik Garrison and Gabor Marth. “Haplotype-based variant detection from short-read sequencing”. In: *arXiv preprint arXiv:1207.3907* (2012).

[65] *Glossary V1 - EGIWiki*. URL: `https://wiki.egi.eu/wiki/Glossary_V1` (visited on 10/28/2016).

[66] Jeremy Goecks, Anton Nekrutenko, and James Taylor. “Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences”. In: *Genome biology* 11.8 (2010), p. 1.

[67] A Gordon and GJ Hannon. “Fastx-toolkit”. In: *FASTQ/A short-reads preprocessing tools (unpublished) http://hannonlab. cshl. edu/fastx_toolkit* 5 (2010).

[68] Christopher Greenman et al. “Patterns of somatic mutation in human cancer genomes”. In: *Nature* 446.7132 (2007), pp. 153–158.

[69] Michael Greenwald and Sanjeev Khanna. “Space-efficient online computation of quantile summaries”. In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 58–66.

[70] Mendel Gregor. “Versuche uber Pflanzen-Hybriden. Verhandlungen des naturforschenden Vereines in Brunn”. In: 4 (1865), pp. 3–47.

[71] Robert L Grossman et al. “An overview of the open science data cloud”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010, pp. 377–384.

[72] Daniel F Gudbjartsson et al. “Large-scale whole-genome sequencing of the Icelandic population”. In: *Nature genetics* 47.5 (2015), pp. 435–444.

[73] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[74] Douglas Hanahan and Robert A Weinberg. “Hallmarks of cancer: the next generation”. In: *cell* 144.5 (2011), pp. 646–674.

[75] John A Hartigan. “Clustering algorithms”. In: (1975).

[76] Michael J Heller. “DNA microarray technology: devices, systems, and applications”. In: *Annual review of biomedical engineering* 4.1 (2002), pp. 129–153.

[77] Robert L Henderson. “Job scheduling under the portable batch system”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995, pp. 279–294.

[78] Joel N Hirschhorn and Mark J Daly. “Genome-wide association studies for common diseases and complex traits”. In: *Nature Reviews Genetics* 6.2 (2005), p. 95.

[79] Alan Hodgkinson and Adam Eyre-Walker. “Human triallelic sites: evidence for a new mutational mechanism?” In: *Genetics* 184.1 (2010), pp. 233–241.

[80] David Hollingsworth. “The workflow reference model”. In: (1995).

[81] *Home*. URL: `https://www.denbi.de/` (visited on 10/31/2016).

[82] Eun Pyo Hong and Ji Wan Park. “Sample size and statistical power calculation in genetic association studies”. In: *Genomics & informatics* 10.2 (2012), pp. 117–122.

[83] David B Ingham, Fabio Panzieri, and Santosh K Shrivastava. “Constructing dependable Web services”. In: *Advances in Distributed Systems*. Springer, 2000, pp. 277–294.

[84] Vernon M Ingram et al. “Gene mutations in human haemoglobin: the chemical difference between normal and sickle cell haemoglobin”. In: *Nature* 180.4581 (1957), pp. 326–328.

[85] Zamin Iqbal et al. “De novo assembly and genotyping of variants using colored de Bruijn graphs”. In: *Nature genetics* 44.2 (2012), p. 226.

[86] Mark Jobling, Matthew Hurles, and Chris Tyler-Smith. *Human evolutionary genetics: origins, peoples & disease*. Garland Science, 2013.

[87] Peter A Jones and Stephen B Baylin. “The epigenomics of cancer”. In: *Cell* 128.4 (2007), pp. 683–692.

[88] Jocelyn Kaiser and Jennifer Couzin-Frankel. “Biden seeks clear course for his cancer moonshot”. In: *Science* 351.6271 (2016), pp. 325–326.

[89] Juha Kärkkäinen and Peter Sanders. “Simple linear work suffix array construction”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2003, pp. 943–955.

[90] Daehwan Kim et al. “Centrifuge: rapid and sensitive classification of metagenomic sequences”. In: *Genome research* (2016).

[91] Bartha Maria Knoppers and Ruth Chadwick. “Human genetic research: emerging trends in ethics”. In: *Nature Reviews Genetics* 6.1 (2005), pp. 75–79.

[92] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7.

[93] Hans-Peter Kriegel et al. “Density-based clustering”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1.3 (2011), pp. 231–240.

[94] Eric S Lander et al. “Initial sequencing and analysis of the human genome”. In: *Nature* 409.6822 (2001), pp. 860–921.

[95] Ben Langmead and Steven L Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature methods* 9.4 (2012), pp. 357–359.

[96] Ben Langmead et al. “Searching for SNPs with cloud computing”. In: *Genome biology* 10.11 (2009), R134.

[97] Ben Langmead et al. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. In: *Genome biology* 10.3 (2009), R25.

[98] Martin Lauss et al. “Monitoring of technical variation in quantitative high-throughput datasets”. In: *Cancer informatics* 12 (2013), CIN–S12862.

[99] Michael S Lawrence et al. “Discovery and saturation analysis of cancer genes across 21 tumour types”. In: *Nature* 505.7484 (2014), pp. 495–501.

[100] Ryan M Layer et al. “LUMPY: a probabilistic framework for structural variant discovery”. In: *Genome biology* 15.6 (2014), R84.

[101] Hane Lee et al. “Clinical exome sequencing for genetic identification of rare Mendelian disorders”. In: *Jama* 312.18 (2014), pp. 1880–1887.

[102] Monkol Lek et al. “Analysis of protein-coding genetic variation in 60,706 humans”. In: *Nature* 536.7616 (2016), pp. 285–291.

[103] Heng Li. “A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data”. In: *Bioinformatics* 27.21 (2011), pp. 2987–2993.

[104] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv preprint arXiv:1303.3997* (2013).

[105] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv preprint arXiv:1303.3997* (2013).

[106] Heng Li. “Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly”. In: *Bioinformatics* 28.14 (2012), pp. 1838–1844.

[107] Heng Li. “Mathematical Notes on SAMtools Algorithms”. In: (2010).

[108] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 1 (2018), p. 7.

[109] Heng Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 1 (2018), p. 7.

[110] Heng Li. “Tabix: fast retrieval of sequence features from generic TAB-delimited files”. In: *Bioinformatics* 27.5 (2011), pp. 718–719.

[111] Heng Li. “Towards better understanding of artifacts in variant calling from high-coverage samples”. In: *Bioinformatics* (2014), btu356.

[112] Heng Li and Richard Durbin. “Fast and accurate long-read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 26.5 (2010), pp. 589–595.

[113] Heng Li and Richard Durbin. “Fast and accurate short read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.

[114] Heng Li and Nils Homer. “A survey of sequence alignment algorithms for next-generation sequencing”. In: *Briefings in bioinformatics* 11.5 (2010), pp. 473–483.

[115] Heng Li, Jue Ruan, and Richard Durbin. “Mapping short DNA sequencing reads and calling variants using mapping quality scores”. In: *Genome research* (2008), gr–078212.

[116] Heng Li et al. “A synthetic-diploid benchmark for accurate variant-calling evaluation”. In: *Nature methods* 15.8 (2018), p. 595.

[117] Heng Li et al. “The sequence alignment/map format and SAMtools”. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[118] Ruiqiang Li et al. “SOAP: short oligonucleotide alignment program”. In: *Bioinformatics* 24.5 (2008), pp. 713–714.

[119] Hengyun Lu, Francesca Giordano, and Zemin Ning. “Oxford Nanopore MinION sequencing and genome assembly”. In: *Genomics, proteomics & bioinformatics* 14.5 (2016), pp. 265–279.

[120] Bertram Ludäscher et al. “Scientific workflow management and the Kepler system”. In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[121] Ramon Luengo-Fernandez et al. “Economic burden of cancer across the European Union: a population-based cost analysis”. In: *The lancet oncology* 14.12 (2013), pp. 1165–1174.

[122] David Malkin et al. “Germ line p53 mutations in a familial syndrome of breast cancer, sarcomas, and other neoplasms”. In: *Science* (1990), pp. 1233–1238.

[123] Udi Manber and Gene Myers. “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5 (1993), pp. 935–948.

[124] Teri A Manolio. “Genomewide association studies and assessment of the risk of disease”. In: *New England Journal of Medicine* 363.2 (2010), pp. 166–176.

[125] Ming Mao and Marty Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE. 2011, pp. 1–12.

[126] Elaine R Mardis. "Next-generation DNA sequencing methods". In: *Annu. Rev. Genomics Hum. Genet.* 9 (2008), pp. 387–402.

[127] Ronald Margolis et al. "The National Institutes of Health's Big Data to Knowledge (BD2K) initiative: capitalizing on biomedical big data". In: *Journal of the American Medical Informatics Association* 21.6 (2014), pp. 957–958.

[128] Gabor T Marth et al. "A general approach to single-nucleotide polymorphism discovery". In: *Nature genetics* 23.4 (1999), p. 452.

[129] Vivien Marx. "Biology: The big challenges of big data". In: *Nature* 498.7453 (2013), pp. 255–260.

[130] Vivien Marx. "The DNA of a nation". In: *Nature* 524.7566 (2015), pp. 503–505.

[131] Aaron McKenna et al. "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data". In: *Genome research* 20.9 (2010), pp. 1297–1303.

[132] William McLaren et al. "The ensembl variant effect predictor". In: *Genome biology* 17.1 (2016), p. 122.

[133] Peter Mell and Tim Grance. "The NIST definition of cloud computing". In: (2011).

[134] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[135] C Mohinudeen et al. "An Overview of Next-Generation Sequencing (NGS) Technologies to Study the Molecular Diversity of Genome". In: *Microbial Applications Vol. 1*. Springer, 2017, pp. 295–317.

[136] Fruzsina Molnár-Gábor et al. "Computing patient data in the cloud: practical and legal considerations for genetics and genomics research in Europe and internationally". In: *Genome Medicine* 9.1 (2017), p. 58.

[137] Paul Muir et al. "The real cost of sequencing: scaling computation to keep pace with data generation". In: *Genome biology* 17.1 (2016), p. 53.

[138] Shanmugavelayutham Muthukrishnan et al. "Data streams: Algorithms and applications". In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236.

[139] Eugene W Myers and Webb Miller. "Optimal alignments in linear space". In: *Bioinformatics* 4.1 (1988), pp. 11–17.

[140] Michael W Nachman. "Single nucleotide polymorphisms and recombination rate in humans". In: *TRENDS in Genetics* 17.9 (2001), pp. 481–485.

[141] Rasmus Nielsen et al. "Genotype and SNP calling from next-generation sequencing data". In: *Nature Reviews Genetics* 12.6 (2011), pp. 443–451.

[142] Bill Nitzberg and Virginia Lo. "Distributed shared memory: A survey of issues and algorithms". In: *Distributed Shared Memory-Concepts and Systems* (1991), pp. 42–50.

[143] Tom Oinn et al. "Taverna: a tool for the composition and enactment of bioinformatics workflows". In: *Bioinformatics* 20.17 (2004), pp. 3045–3054.

[144] David Oppenheimer and David A Patterson. "Architecture and dependability of large-scale internet services". In: *IEEE Internet Computing* 6.5 (2002), pp. 41–49.

[145] Stavros Papadopoulos et al. "The TileDB array data storage manager". In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 349–360.

[146] Mike P Papazoglou. "Service-oriented computing: Concepts, characteristics and directions". In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE. 2003, pp. 3–12.

[147] Emanuel Parzen. "On estimation of a probability density function and mode". In: *The annals of mathematical statistics* 33.3 (1962), pp. 1065–1076.

[148] Ravi K Patel and Mukesh Jain. "NGS QC Toolkit: a toolkit for quality control of next generation sequencing data". In: *PloS one* 7.2 (2012), e30619.

[149] Jaume Pellicer, Michael F Fay, and Ilia J Leitch. "The largest eukaryotic genome of them all?" In: *Botanical Journal of the Linnean Society* 164.1 (2010), pp. 10–15.

[150] James L Peterson. "Petri net theory and the modeling of systems". In: (1981).

[151] *Picard toolkit*. `http://broadinstitute.github.io/picard/`. 2018.

[152] Erin D Pleasance et al. "A comprehensive catalogue of somatic mutations from a human cancer genome". In: *Nature* 463.7278 (2010), p. 191.

[153] Tobias Rausch et al. "DELLY: structural variant discovery by integrated paired-end and split-read analysis". In: *Bioinformatics* 28.18 (2012), pp. i333–i339.

[154] Knut Reinert et al. "Alignment of next-generation sequencing reads". In: *Annual review of genomics and human genetics* 16 (2015), pp. 133–151.

[155] Anthony Rhoads and Kin Fai Au. "PacBio sequencing and its applications". In: *Genomics, proteomics & bioinformatics* 13.5 (2015), pp. 278–289.

[156] Andy Rimmer et al. "Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications". In: *Nature genetics* 46.8 (2014), pp. 912–918.

[157] Michael Roberts et al. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18 (2004), pp. 3363–3369.

[158] Nicola D Roberts et al. "A comparative analysis of algorithms for somatic SNV detection in cancer". In: *Bioinformatics* (2013), btt375.

[159] Richard J Roberts, Mauricio O Carneiro, and Michael C Schatz. "The advantages of SMRT sequencing". In: *Genome biology* 14.6 (2013), p. 405.

[160] Dan Robinson et al. "Integrative clinical genomics of advanced prostate cancer". In: *Cell* 161.5 (2015), pp. 1215–1228.

[161] Murray Rosenblatt. “Remarks on some nonparametric estimates of a density function”. In: *The Annals of Mathematical Statistics* (1956), pp. 832–837.

[162] Richard M Russell. “The CRAY-1 computer system”. In: *Communications of the ACM* 21.1 (1978), pp. 63–72.

[163] Fred Sanger and Alan R Coulson. “A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase”. In: *Journal of molecular biology* 94.3 (1975), 441IN19447–446IN20448.

[164] Frederick Sanger, Steven Nicklen, and Alan R Coulson. “DNA sequencing with chain-terminating inhibitors”. In: *Proceedings of the national academy of sciences* 74.12 (1977), pp. 5463–5467.

[165] Wataru Satake et al. “Genome-wide association study identifies common variants at four loci as genetic risk factors for Parkinson’s disease”. In: *Nature genetics* 41.12 (2009), p. 1303.

[166] Stephan C Schuster. “Next-generation sequencing transforms today’s biology”. In: *Nature* 200.8 (2007), pp. 16–18.

[167] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. “OpenStack: toward an open-source solution for cloud computing”. In: *International Journal of Computer Applications* 55.3 (2012).

[168] Dennis J Selkoe. “Amyloid β-protein and the genetics of Alzheimer’s disease”. In: *Journal of Biological Chemistry* 271.31 (1996), pp. 18295–18298.

[169] Robert Shapiro. “A technical comparison of XPDL, BPML and BPEL4WS”. In: *Cape Visions* (2002).

[170] Mary Shaw and David Garlan. *Software architecture*. Vol. 101. Prentice Hall Englewood Cliffs, 1996.

[171] Hui Shen et al. “Comprehensive characterization of human genome variation by high coverage whole-genome sequencing of forty four Caucasians”. In: *PLoS One* 8.4 (2013), e59494.

[172] Nisheeth Shrivastava et al. “Medians and beyond: new aggregation techniques for sensor networks”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, pp. 239–249.

[173] Jared T Simpson and Richard Durbin. “Efficient construction of an assembly string graph using the FM-index”. In: *Bioinformatics* 26.12 (2010), pp. i367–i373.

[174] Jared T Simpson and Richard Durbin. “Efficient de novo assembly of large genomes using compressed data structures”. In: *Genome research* 22.3 (2012), pp. 549–556.

[175] Temple F Smith and Michael S Waterman. “Comparison of biosequences”. In: *Advances in applied mathematics* 2.4 (1981), pp. 482–489.

[176] Lincoln D Stein et al. “Data analysis: create a cloud commons”. In: *Nature* 523 (2015), pp. 149–151.

[177] Zachary D Stephens et al. “Big data: astronomical or genomical”. In: *PLoS biology* 13.7 (2015), e1002195.

[178] Zachary D Stephens et al. “Simulating next-generation sequencing datasets from empirical mutation and sequencing models”. In: *PloS one* 11.11 (2016), e0167047.

[179] Michael R Stratton, Peter J Campbell, and P Andrew Futreal. “The cancer genome”. In: *Nature* 458.7239 (2009), pp. 719–724.

[180] Peter H Sudmant et al. “An integrated map of structural variation in 2,504 human genomes”. In: *Nature* 526.7571 (2015), pp. 75–81.

[181] Frederick Winslow Taylor. *Scientific management*. Routledge, 2004.

[182] *The Cost of Sequencing a Human Genome - National Human Genome Research Institute (NHGRI)*. URL: `https://www.genome.gov/sequencingcosts/` (visited on 11/04/2016).

[183] Lindsey A Torre et al. “Global cancer statistics, 2012”. In: *CA: a cancer journal for clinicians* 65.2 (2015), pp. 87–108.

[184] Frances Susan Turner. “Assessment of insert sizes and adapter content in fastq data from NexteraXT libraries”. In: *Frontiers in genetics* 5 (2014), p. 5.

[185] Esko Ukkonen. “On-line construction of suffix trees”. In: *Algorithmica* 14.3 (1995), pp. 249–260.

[186] Geraldine A Van der Auwera et al. “From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline”. In: *Current protocols in bioinformatics* (2013), pp. 11–10.

[187] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. “Dynamically scaling applications in the cloud”. In: *ACM SIGCOMM Computer Communication Review* 41.1 (2011), pp. 45–52.

[188] J Craig Venter et al. “The sequence of the human genome”. In: *science* 291.5507 (2001), pp. 1304–1351.

[189] J Craig Venter et al. “Shotgun sequencing of the human genome”. In: *Science* 280.5369 (1998), pp. 1540–1542.

[190] Kathleen A Vermeersch and Mark P Styczynski. “Applications of metabolomics in cancer research”. In: *Journal of carcinogenesis* 12 (2013).

[191] Karl V Voelkerding, Shale A Dames, and Jacob D Durtschi. “Next-generation sequencing: from basic research to diagnostics”. In: *Clinical chemistry* 55.4 (2009), pp. 641–658.

[192] Jeremiah A Wala et al. “SvABA: genome-wide detection of structural variants and indels by local assembly”. In: *Genome research* (2018).

[193] Zhong Wang, Mark Gerstein, and Michael Snyder. “RNA-Seq: a revolutionary tool for transcriptomics”. In: *Nature reviews genetics* 10.1 (2009), pp. 57–63.

[194] James D Watson, Francis HC Crick, et al. “Molecular structure of nucleic acids”. In: *Nature* 171.4356 (1953), pp. 737–738.

[195] John N Weinstein et al. “The cancer genome atlas pan-cancer analysis project”. In: *Nature genetics* 45.10 (2013), pp. 1113–1120.

[196] Jeffrey N Weitzel et al. “Genetics, genomics, and cancer risk assessment: state of the art and future directions in the era of personalized medicine”. In: *CA: a cancer journal for clinicians* 61.5 (2011), pp. 327–359.

[197] BP Welford. “Note on a method for calculating corrected sums of squares and products”. In: *Technometrics* 4.3 (1962), pp. 419–420.

[198] Danielle Welter et al. “The NHGRI GWAS Catalog, a curated resource of SNP-trait associations”. In: *Nucleic acids research* 42.D1 (2013), pp. D1001–D1006.

[199] Justin P Whalley et al. “Framework For Quality Assessment Of Whole Genome, Cancer Sequences”. In: *bioRxiv* (2017), p. 140921.

[200] Philipp Wieder et al. *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[201] Derrick E Wood and Steven L Salzberg. “Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome biology* 15.3 (2014), R46.

[202] K Robin Yabroff et al. “Economic burden of cancer in the United States: estimates, projections, and future research”. In: *Cancer Epidemiology Biomarkers & Prevention* 20.10 (2011), pp. 2006–2014.

[203] Yaping Yang et al. “Clinical whole-exome sequencing for the diagnosis of mendelian disorders”. In: *New England Journal of Medicine* 369.16 (2013), pp. 1502–1511.

[204] Kai Ye et al. “Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads”. In: *Bioinformatics* 25.21 (2009), pp. 2865–2871.

[205] Christina K Yung et al. “ICGC in the cloud”. In: *Cancer Research* 76.14 Supplement (2016), pp. 3605–3605.

[206] Shelemyahu Zacks. *The theory of statistical inference*. Vol. 34. Wiley New York, 1971.

[207] Matei Zaharia et al. “Faster and more accurate sequence alignment with SNAP”. In: *arXiv preprint arXiv:1111.5572* (2011).

[208] Daniel R Zerbino and Ewan Birney. “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”. In: *Genome research* 18.5 (2008), pp. 821–829.

[209] Min Zhao et al. “Computational tools for copy number variation (CNV) detection using next-generation sequencing data: features and perspectives”. In: *BMC bioinformatics* 14.11 (2013), S1.

[210] Qian Zhou et al. “QC-Chain: fast and holistic quality control method for next-generation sequencing data”. In: *PloS one* 8.4 (2013), e60234.

[211] Songnian Zhou. “Lsf: Load sharing in large heterogeneous distributed systems”. In: *I Workshop on Cluster Computing*. Vol. 136. 1992.

[212] Justin Zook et al. “Reproducible integration of multiple sequencing datasets to form high-confidence SNP, indel, and reference calls for five human genome reference materials”. In: *BioRxiv* (2018), p. 281006.