Scalable Systems and Algorithms for Genomic Variant Analysis

by

Frank Austin Nothaft

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Anthony Joseph, Chair
Professor David Patterson, Co-chair
Professor Haiyan Huang

Fall 2017

The dissertation of Frank Austin Nothaft, titled Scalable Systems and Algorithms for Genomic Variant Analysis, is approved:

Chair _____ Date _____

Co-chair _____ Date _____

_____ Date _____

University of California, Berkeley

Scalable Systems and Algorithms for Genomic Variant Analysis

Abstract

Scalable Systems and Algorithms for Genomic Variant Analysis

by

Frank Austin Nothaft

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Anthony Joseph, Chair

Professor David Patterson, Co-chair

With the cost of sequencing a human genome dropping below $1,000, population-scale sequencing has become feasible. With projects that sequence more than 10,000 genomes becoming commonplace, there is a strong need for genome analysis tools that can scale across distributed computing resources while providing reduced analysis cost. Simultaneously, these tools must provide programming interfaces and deployment models that are easily usable by biologists.

In this dissertation, we describe the ADAM system for processing large genomic datasets using distributed computing. ADAM provides a decouped stack-based architecture that can accomodiate many data formats, deployment models, and data access patterns. Additionally, ADAM defines schemas that describe common genomic datatypes. ADAM's schemas and programming models enable the easy integration of disparate genomic datatypes and datasets into a single analysis.

To validate the ADAM architecture, we implemented an end-to-end variant calling pipeline using ADAM's APIs. To perform parallel alignment, we developed the Cannoli tool, which uses ADAM's APIs to automatically parallelize single node aligners. We then implemented GATK-style alignment refinement as part of ADAM. Finally, we implemented a biallelic genotyping model, and novel reassembly algorithms in the Avocado variant caller. This pipeline provides state-of-the-art SNV calling accuracy, along with high (97%) INDEL calling accuracy. To further validate this pipeline, we reanalyzed 270 samples from the Simons Genome Diversity Dataset.

Contents

Part I

Introduction and Principles

Chapter 1

Introduction

The rapid decrease in sequencing cost has made large scale sequencing tractible. The dramatic improvement in sequencing cost since the Human Genome Project has enabled a human whole genome sequence (WGS) to be generated for under $1,000 in wetlab costs [91]. This trend will continue for the forseeable future, as sequencing vendors like Illumina unveil new sequencers such as the NovaSeq that provide even higher throughput while also decreasing cost, and as radically new sequencing technologies like Oxford Nanopore come online [55]. The reduced cost of sequencing enables the use of genome sequencing in population health research projects and clinical practice. As a result, the total volume of sequencing data produced is expected to exceed that of YouTube by 2021 [121].

The massive scale of the sequencing data that is being acquired enables novel insight into biological phenomena. The Exome Aggregation project (ExAC, [67])–now gnomAD–provides an especially powerful demonstration: by sequencing more than 60,000 exomes, we have been able to better understand the impact of genomic variation on prion disease [83] and cardiovascular disease [135], and to better characterize the effect of structural variation [109]. However, this scale of data solves biological problems at the cost of techincal and logistical problems. Data storage and transfer has become a serious problem, and the focus of many researchers [43, 62] and standards organizations [99]. Not only is the volume of data large, but expensive processing is needed to analyze the data. Due to historical design decisions, much of this processing is currently restricted to single node architectures that assume POSIX storage APIs. As a result, it can take upwards of 100 hours to analyze the raw read data from a single genome.

We believe that distributed computing architectures are a good match for genomic data analysis. Horizontally scalable storage architectures can simultaneously provide increased data storage capacities, data access throughput, and reduced storage cost. Because most genomic analyses are centered on analyzing the genomic data at disparate genomic loci without coordination between loci, most genomic analysis tasks can be executed in parallel. Even more importantly, these analysis patterns cleanly map onto quasi-relational primitives that are powerful and can be executed in parallel. Finally, by building upon widely used open-source distributed processing architectures like Apache Spark [143] and Hadoop [8],

genomics can benefit from the engineering contributions that advance these large open source projects.

In this thesis, we introduce ADAM, an application programming interface (API) for processing genomic data using Apache Spark. ADAM is based around a novel stack-oriented architecture that uses schemas to define the narrow waist in the stack. On top of the schemas, we provide high-level APIs that allow computational biologists and bioinformaticians to manipulate collections of genomic data in a parallel fashion. The high level APIs extend Apache Spark's Resilient Distributed Dataset (RDD, see Zahaira et al. [143]) abstraction with genomics-specific functionality, and eliminates the low level "walker" pattern [79] that is common in genomics. At lower levels in the stack, we provide efficient implementations of the common genomics query models. By having clearly defined APIs between each level of the stack, we are able to exchange layers to optimize query performance for a given query, input data type, or cluster/cloud configuration.

Our work on ADAM has resulted in the broad ecosystem of projects depicted in Figure 1.1. We refer to the tools built on ADAM as the "Big Data Genomics" (BDG) project. In this dissertation, we will limit our focus to ADAM's architecture and APIs, and the tools and algorithms that form the core components of the BDG variant calling pipeline:

- Cannoli, which parallelizes single node genomic data processing tools. Cannoli is used in our pipeline for alignment.
- The ADAM read transformations, which correct for errors in the aligned reads.
- Avocado, a fully parallelized variant caller.

This pipeline is able to call variants on a high coverage (60×) whole genome in under one hour when running on commodity cloud computing resources. This represents a dramatic improvement in performance over the widely used Genome Analysis Toolkit (GATK, see DePristo et al. [34]), which needed over 100 hours to call variants on the same sample. These tools demonstrate how ADAM's APIs enable bioinformatics analyses to be written at a high level, while also allowing for the reuse of code from legacy bioinformatics tools.

In this dissertation, we begin by describing the deluge of genomic data and the tools that are used to process, manipulate, and store genomic data. In part two, we then describe the requirements for a distributed genomic data analysis framework and introduce ADAM's architecture. Part three describes how we built the Cannoli-ADAM-Avocado variant calling pipeline on top of ADAM's architecture, and the novel algorithms and architectural refinements that were needed. We then validate the accuracy of this pipeline in part four using ground truth datasets [149] and a large scale sequencing project [76]. We conclude by describing open problems and future directions for this research, as well as the impact of the Big Data Genomics/ADAM project.

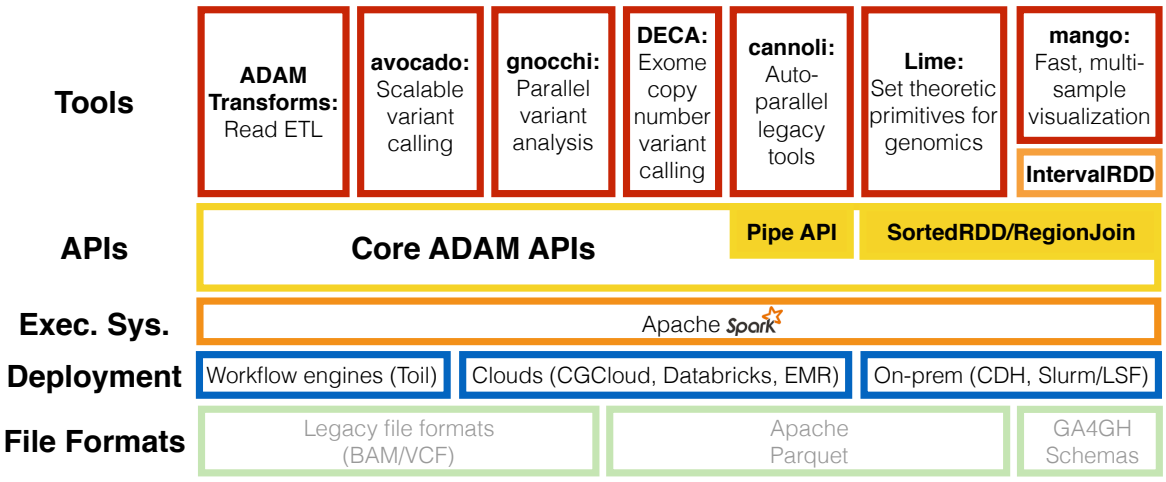

Figure 1.1: The Big Data Genomics ecosystem. Our work on ADAM has built a framework for scalable genomics using external projects like Apache Spark [143] and Apache Parquet [9]. On top of ADAM's core APIs, we have built a broad ecosystem of tools [129, 84, 75] demonstrating how Apache Spark can accelerate genomic data analysis. To enable the reproducible use of Apache Spark in scientific data analysis workloads, we have contributed to novel, cloud-native workflow systems [134].

1.1 Economic Trends and Population Scale Sequencing

The need for tools capable of processing large genomic datasets is precipitated by the rise of popultation scale sequencing. While the raw data from a single genome may be provide insight into the fitness of a single individual, genetic data is most meaningful when viewed in aggregate across large cohorts. For variants that do not have an obvious and severe pathogenic effect, our best lens for understanding their impact from genomic data is through statistical association testing. While genome wide association testing (GWAS) has yielded some successes, including prominent findings in neurogenetics [107], GWAS has faced several limitations. First, the association between genotype and phenotype is often weak, unless the variant under study is strongly pathogenic [114]. Additionally, few traits are truly Mendelian. For complex traits, which are driven by the combined effect of multiple genotypes [14], much heritability is explained through the complex interaction of variants that impact regulatory regions. Modeling the effect of non-coding changes is still an active area of work [136]. Additionally, some diseases decompose into disease subtypes when studied in aggregate. A strong example of this is acute myleoid leukemia (AML): genomic sequencing of germlines and tumors from a cohort of AML patients reveals that AML is composed of eight or more genetic subtypes [90]. This hinders classifying the impact of a single mutation, especially as

some gene mutations can be shared across disease subtypes.

These population scale sequencing projects have been enabled entirely by technical innovation. While it cost more than $100M to complete the data acquisition for the Human Genome Project, the cost of sequencing a single genome has dropped to under $1,000 [91]. This has been driven by continuous technical improvements by Illumina and competing sequencer vendors. With the continued improvement of nanopore sequencing [55], we may see an additional precipitous drop in sequencing costs, as nanopore sequencers have both lower capital and reagent costs relative to the Illumina sequencing platform, but are currently limited in throughput and accuracy.

At the intersection of these two trends, population-scale sequencing has arisen. The first population scale sequencing project was the 1,000 Genomes project [1], which collected a total data catalog of more than 75 terabytes (TB) of data. Since then, sequencing projects have pushed beyond petabyte-scale (PB), with the 3PB catalog collected by The Cancer Genome Atlas (TCGA, [137]) and the 300PB data catalog collected by the ExAC [67] project. While population-scale sequencing has largely been done in academic settings to date, these sequencing projects are beginning to move into industrial and medical settings. These include the collaboration between the United Kingdom's Biobank, GlaxoSmithKline, and Regeneron to sequence the 500,000 individuals in the UK Biobak [131], and the Geisenger MyCode collaboration with Regeneron [18].

1.2 The Case for Distributed Computing for Genomic Analysis

The rapid uptake of genome sequencing in academic, industrial, and clinical settings is driving the total number of human genomes sequenced to double approximately once every seven months [121]. This far outpaces Moore's law at its peak, and is a rate of increase approximately four times greater than current estimates of Moore's law, which peg the doubling of transistor counts to occur approximately every two years. As a result of this growth in the volume of sequencing data, legacy tools are struggling to handle genome-scale analyses across cohorts [110, 75]. We believe that distributed computing is a natural solution to these problems.

As asserted in the original GATK manuscript that proposed a single-node MapReduce architecture for genomic data processing [79], most genomic analysis tasks map naturally to a share-nothing computing architecture. Heavyweight genomic data analyses like alignment, variant calling, and association testing either typically work on unaligned data, or are implemented on a sorted stream traversing aligned data (the "walker" model). These patterns either lack data dependencies (unaligned reads, or analyses that look at a single genomic locus), or have well defined spatial communication patterns (process data overlapping a given locus). These computations can typically be paralellized with minimal communication. Additionally, many genomic analysis queries map directly onto relational primitives which are

implemented in existing distributed data analysis platforms [10]. An example of this is a genomic association test, which can be implemented as an aggregation query.

To take full advantage of distributed computing, we believe that we need to do a clean slate re-architecture of the genomic data processing infrastructure. In a typical genomic processing pipeline, the analysis tools are typically designed assuming a flattened stack running on a single node, or on a high performance computing (HPC)-style cluster with shared storage. These systems typically make strong assumptions about the cost of making random accesses into a POSIX file system, and present low level abstractions to users. While there have been several attempts to retrofit legacy tools onto distributed computing (CloudBurst [111] and Crossbow [64]), these approaches have typically used custom wrappers around Apache Hadoop Streaming and have been non-general. There have also been several attempts to retrofit genomics-specific file formats onto distributed query architectures (SegPig [112] and BioPig [93]), but these implementations provide either poor programming models or inefficient implementations. By doing a clean-slate rearchitecture, we can eliminate architectural problems and provide better user-facing query models with better performance.

1.3 Mapping Genomics onto Distributed Computing using ADAM

To address these problems, we propose ADAM, a comprehensive framework for processing genomic data using the Apache Spark framework for distributed computing. ADAM defines schemas for a full range of genomic datatypes, which provides a data-indepent query model. These schemas form the basis of a narrow waisted stack, which yields APIs that support both genomic query and metadata management. By extending ADAM onto Apache Spark SQL, these APIs can be used across multiple languages. This extends the power of distributed computing to bioinformatics users who are writing in the Python or R languages, as opposed to previous tools that were centered either around Java or the Pig scripting language [93, 112].

To demonstrate ADAM, we have built an end-to-end variant calling pipeline. This pipeline includes distributed implementations of alignment, read preprocessing, and variant calling. The pipeline can run end-to-end on a 60× coverage whole genome in under an hour, at a cost of <$15 on cloud computing. This pipeline provides results comparable to state-of-the-art for single nucleotide variant (SNV) calling, and high accuracy (97%) for insertion and deletion (INDEL) variant calling. Additionally, the alignment step in this pipeline is implemented on top of a general interface for parallelizing single-node genomics tools, which makes it possible to leverage distributed computing without reimplementing a tool or developing custom shims, unlike prior approaches [64, 111].

As a result, ADAM improves over conventional genomics tools by providing:

- Schemas which can support loading data from a large variety of formats.

- High level, quasi-relational APIs for manipulating genomic data in both single node and cluster environments.
- Parallel I/O across genomics file formats.
- A simple API for parallelizing single node genomic tools with a minimal amount of code.

In the rest of this dissertation, we explain the design goals behind ADAM. By reviewing the architecture and implementation of ADAM, we explain how these goals have shifted over time, informed by our development experiences. We then demonstrate the ADAM architecture through the Cannoli and Avocado tools, which implement the Big Data Genomics variant calling pipeline.

Chapter 2

Background and Related Work

While there are many ways to collect and then process genomic data, this dissertation focuses on the "genome resequencing" pipeline. In resequencing, we start with a known genome assembly, and identify the edits between an individual genome and the genome assembly for their species. In practice, a genome resequencing analysis pipeline will typically take short sequenced reads (100-300 base pairs, bp), align them to the reference genome, perform preprocessing on the reads to eliminate errors, and then probabilistically identify true variation from the reads. In this chapter, we will start by describing how the reads are sequenced (§2.1) and analyzed (§2.2). We will then dive deeper into the representations of genomic data (§2.2), and the architectures used to process this data (§2.2). Then, we will review the variant identification algorithms (§2.2). Finally, we will discuss the emergence of commodity distributed computing frameworks (§2.3) and how researchers have approached parallelizing genome resequencing pipelines (2.3)

2.1 Genome Sequencing Technologies

Since the Human Genome Project released the first assembly of the Human genome in 2001 [63], biochemical and algorithmic advancements have enabled the broad analysis of biological phenomena through sequencing. Although the full spectrum of sequencing-based analyses is beyond the scope of this manuscript, these assays rely on encoding a biological phenomena into DNA, which is then sequenced and analyzed statistically. In this section, we provide a brief introduction to the sequencing process before focusing on the algorithmic approaches used to determine the sequence variants in a single genome. We will focus on data generated using Illumina sequencers, as this sequencing modality is commonly used for genomic variant detection.

To run a sequencing assay, we start by preparing a sequencing library, which is then run through a sequencer. This stage creates genomic "reads", which include a string of bases (in the A, C, G, T alphabet used by DNA) along with estimates of the probability that a single base was read correctly. The library preparation stage converts the biological sample into

DNA fragments, which we can sequence. In the simplest case (sequencing a genome), we start by extract DNA from a collection of cells. We then slice the long DNA strands into shorter fragments, before selecting fragments of a certain length ("size selection"). The size of fragments collected depends on both the sequencing instrument that is being used, and the biological assay being conducted. Common variants on this process include exome sequencing (where we start from DNA, but select the regions of the genome that encode genes before fragmenting and size selecting reads), RNA-seq (where we start by converting single stranded RNA into DNA, which is then fragmented and size selected, see [86]). There are many biological assays that can be encoded as sequencing and a full review is beyond the scope of this manuscript; we refer readers to Soon et al [119] for a more comprehensive overview.

Many modern variant analysis pipelines use paired reads from Illumina sequencers. Depending on the specific sequencer model and chemistry, Illumina sequencers support read lengths ranging from 75 to 300 bases. All reads from an Illumina sequencer have the same length, as the length of the read is determined by the number of cycles that the sequencer is run. "Paired" means that we generate two reads from each DNA fragment; we read one read from each strand of the DNA, with the two reads coming from opposite ends of the DNA fragment. In a conventional sequencing libraries, the DNA fragments include bases that are not sequenced; these bases are typically referred to as the "insert", and the number of bases not sequenced ("insert size") are controlled through the size selection process. For example, if we wanted to prepare a paired sequencing library where the read length was 250 bases with an average insert size of 500 bases, we would select all fragments that were approximately 1,000 bases long (250 bases for the first read, approximately 500 bases between the first and second read, 250 bases for the second read). There are many variants on this process, including those that have negative insert sizes, and "mate pair" libraries that have very long insert sizes [77]. Additionally, library preparation varies tremendously between sequencing vendors. For long read sequencers such as the Pacific Biosciences sequencers [37] or the Oxford Nanopore sequencers [25], the libraries include long DNA fragments (¿5,000 bases) that generate a single, full length read.

Illumina sequencers generate reads through a sequencing-by-synthesis approach. In this process, flourescent dyes are attached to the DNA bases. The sequencer then takes an image of the dyes, which is then converted into the called bases. This process runs for a fixed number of cycles, which sets the length of the sequenced reads. To ensure that the bases from a single read show up in the same locations in the image betwen cycles, the ends of the reads are attached to knobs that protrude from glass plates. At the end of each cycle, the dyes are washed off of the end of the read, which exposes the next base in the read for a new round of dyes to attach to. The probability that a base was sequenced correctly is determined by looking at the color and intensity of the base on the captured image. Illumina platforms are susceptible to single base substition errors, which occur to a 0.5–2% of bases. This error rate is problematic for variant calling, as we expect a variant to occur at one in every one thousand bases.

Many of the analyses that use reads generated from Illumina sequencers are analyzed

with mapping-based approaches. In this class of approaches, we rely on the existence of a "reference genome" for an organization. This is a curated dataset that consists of the DNA sequences for all of the chromosomes in the genome of a species. For humans, the first reference genome was generated through the Human Genome Project [63]. New genome references are released every several years and include corrections to prior reference genomes and new assemblies for areas of the genome that are hard to sequence (typically caused by the genome being highly repetitive in a single area). The most recent release of the human genome (GRCh38, see [22]) was released at the end of 2014. A reference genome defines a two dimensional coordinate space, with one coordinate selecting the chromosome and the second coordinate defining the position on this chromosome.

2.2 Genomic Analysis Tools and Architectures

In a mapping-based approach, the reads are mapped to a location of the reference genome, and then locally aligned. Mappers query subsequences from a read against an index built from the reference genome. This process will identify the ranges in the genome that the read could plausibly align to. Once these ranges have been identified, the mapper will then locally align the read sequence against the genomic sequences from these ranges, using an edit calculation algorithm such as Smith-Waterman [118], Ukkonen's algorithm [130], or a pairwise sequence alignment Hidden Markov Model (HMM, [36]). Widely used mappers include BWA [73], which builds an index using the Burrows-Wheeler transform [16]; Bowtie [64], which builds an FM-index [42]; and SNAP [142], which builds a hash-based index. Several projects have applied hardware acceleration to alignment, including cloud-scale-bwamem [20, 19], Ahmed et al [5], and unpublished work out of Microsoft Azure [82].

Variant calling is one such mapping-based approach. To identify variants between the genomes of two individuals, we compute the difference of each individual against the reference genome, and then compute the transitive difference between the two individuals. To compute the variations between a single individual and the reference, we start by aligning the reads to the reference genome. From here, we can then look at each site in the genome to see if there are reads that support an sequence edit. This is typically done by applying a statistical model to the reads that looks at the base error probabilities attached to the reads that contain the reference sequence and the reads that contain the proposed sequence variant. Examples of the models used include the SAMTools mpileup variant caller [69], and the GATK UnifiedGenotyper [34]. However, the aligned reads frequently include errors that can lead to incorrect variant calls. To eliminate these errors, we rely on several preprocessing stages that are run between mapping and variant calling. In this paper, we will focus on three preprocessing stages: duplicate removal, local realignment, and base quality score recalibration. The variant calling pipeline has been targeted for hardware acceleration in the unpublished Edico DRAGEN processor [45].

The duplicate removal stage identifies DNA fragments that were duplicated during library preparation. If we are starting from a biological sample that contains very little DNA, we will

commonly use a polymerase chain reaction (PCR) during library preparation. This reaction will take our input DNA and replicate it, thereby increasing the amount of DNA that we can provide to the sequencer. However, as part of the PCR process, some fragments will be excessively replicated. This can lead to a single fragment being replicated 100 times. If this fragment contains a sequence variant or a sequence that is susceptible to being sequenced incorrectly, this can bias the genomic region where the read is located and lead to an incorrect variant being identified.

Local realignment is typically run after duplicate marking and addresses an issue inherent to the mapping process. Specifically, if there are larger sequence variants (e.g., multi-base insertions or deletions) into a read, the mapping process will commonly identify the correct genomic region that a read should map to, but will locally misalign the read relative to other reads that contain the same underlying sequence variant [34]. During local realignment, we start by identifying all possible insertion/deletion (INDEL) variants in our reads. For every region that contains an INDEL, we then look at the reads that map to that region. We identify the most common sequence variant in the set of reads and rewrite the local read alignments ensure that all reads that contain a single sequence variant are aligned with a consistent representation. This step is necessary because the algorithms used to compute the pairwise alignment of two sequences are fundamentally probabilistic [36, 118, 130], which can lead to inconsistent representations for equivalent sequence edits [71].

The final preprocessing stage is base quality recalibration. As mentioned earlier, when the reads are sequenced, the sequencer estimates the probability that a single base was sequenced in error from the color and intensity of the light emitted from the floursecent dye. In practice, sequencing errors correlate with various factors, including sequence context (the bases around the base that is being sequenced) and the stage when the base was sequenced (due to errors with the sequencing chemistry during that sequencing cycle). The base quality recalibration stage associated each base with error covariates, and then calculates the empirical error rate for the bases in that covariate by measuring the frequency with which bases in that covariate mismatch the reference genome. These error rates are then converted back into probabilites, which replace the probabilities attached to the reads.

We have chosen to focus on the read preprocessing algorithms used for variant calling for several reasons. First and foremost, variant calling is the most widely used analysis across contemporary genomics, and is a core part of large population-scale studies such as the 1,000 Genomes Project [2, 1], the Exome Aggregation Consortium [67], and The Cancer Genome Atlas [137]. Additionally, the read preprocessing stages are tremendously computationally expensive. For a single human genome sequenced with an average of 60 reads covering each genomic position, it takes over 160 hours to align the reads, preprocess the reads, and call variants, with approximately 110 of the hours spent preprocessing the reads. Finally, implementations of thes algorithms are available as part of the widely used Genome Analysis Toolkit [79, 34] and ADAM [78, 94] libraries.

Genomic Data Representations

Currently, genomic data are stored in a myriad of file formats that largely descend from formats that were developed during the 1,000 Genomes project [1]. Some of these formats are much older; many genomic feature file formats descend from the development of the University of California, Santa Cruz's (UCSC) Genome Browser [57], which was developed as part of the Human Genome Project [63], and informal specificiations for the FASTQ [26] and FASTA formats date back to at least the nineties, through their use in the PHRED [39] and FASTA/FASTP [101] tools.

The file formats developed during the 1,000 Genomes project stored high throughput sequencing data in tab separated value (TSV) files. These formats included the SEQUENCE ALIGNMENT/MAPPING (SAM) format [73], which represents genomic reads and the VARIANT CALL FORMAT (VCF) format [30], which was defined to store variants and genotypes. The 1000 Genomes project also made significant use of the TSV BROWSER EXTENSIBLE DATA (BED) format for storing genomic feature data. While the BED format had been introduced earlier, the introduction of BEDTOOLS [105] during the 1,000 Genomes Project drove the further use of the BED format. A plethora of textual file formats exist for storing genomic feature data, such as the NARROWPEAK format, a specialized variant of BED that is used by the MACS [145] tool; the INTERVALLIST format, which is used extensively by the GATK [34]; and the GENERAL FEATURE FORMAT (GFF), which is used extensively in sequence annotation projects like the Sequence Ontology [38].

Over time, some of these formats have been replaced by binary variants that provide improved compression and performance. SAM has been largely replaced in practice by the BINARY ALIGNMENT/MAPPING (BAM) format, and the binary VCF (BCF) has entered use for storing variant data. In practice, textual file formats are still broadly used for storing variant and feature data, but they are often compressed using a block-compressed codec, such as BGZF [70]. There has been significant research towards developing compressed storage formats for alignment data [62, 43]. The CRAM codec has achieved the broadest use, and uses reference-based compression to avoid storing read sequence that matches the reference genome. Additionally, CRAM can apply lossy compression schemes—such as base quality score binning—to achieve further compression.

Genomic Analysis Architectures

Although there exist myriad tools for analyzing genomic data, very few tools espouse a systematic architecture for traversing and processing genomic data. Instead, most tools are built around a UNIX-inspired philosophy that asserts that "a tool should do a single task well" [108], and simply traverse a stream of data. The most promiment example of a genomic analysis architecture is the quasi-map-reduce architecture employed by the legacy versions of the GATK [79]. This architecture uses an iterator-based model called a "walker" to traverse over data aligned to reference genome coordinates. The "map-reduce" nature of this API describes how chunks of genome aligned data can be parallelized, with a reduce operation

supported for summarizing tables of data across threads, as is used for combining Base Quality Score Recalibration (BQSR) tables. While this API could conceptually be used in a distributed setting, the GATK has historically only run as a multithreaded application on a single node. Instead, multi-node execution was provided through the Queue [34] workflow manager.

While the UNIX-like design philosophy espoused by many bioinformatics tools allows for the creation of tools with well defined boundaries, this seems to be fundamentally at odds with the reality of complex genomics workflows, where many tools must be cascaded one-after-the-other. As a result, genomics has embraced workflow management as an alternate paradigm, where tools are composed into an abstract workflow, which is then executed by the management system. A popular early system was the Galaxy [50] tool, which provided a graphical user interface for defining workflows and tool invocations. Recently, a set of novel workflow management systems have been developed, such as Toil [134], NextFlow [35], Rabix [56], Cromwell [124], and Cuneiform [15]. These systems exploit many-task parallelism, and are well suited to analyses where a cohort of many samples should be analyzed independently by sample. These systems differ in their approach to expressing workflows. Several efforts to standardize workflow descriptions have emerged, with the most prominent community being around the Common Workflow Language [28]. Toil is implemented as a Python library that allows for workflows to be natively defined in Python, and can also run workflows written in CWL, or in Cromwell's WDL dialect. Rabix executes CWL. NextFlow and Cuneiform both take a clean slate approach to implementing a workflow language, using dataflow and functional approaches to describe workflows.

Variant Calling Approaches

The accuracy of insertion and deletion (INDEL) variant discovery has been improved by the development of variant callers that couple local reassembly with haplotype-based statistical models to recover INDELs that were locally misaligned [6]. Now, several prominent variant callers such as the Genome Analysis Toolkit's (GATK) HaplotypeCaller [34], Scalpel [89], and Platypus [106]. Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls [12], this accuracy comes at the cost of end-to-end runtime [122]. Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled [13] or by improving the performance of the core algorithms used in local reassembly [106].

The performance issues seen in haplotype reassembly approaches derives from the high asymptotic complexity of reassembly algorithms. Although specific implementations may vary slightly, a typical local reassembler performs the following steps:

1. A de Bruijn graph is constructed from the reads aligned to a region of the reference genome,
2. All valid paths (*haplotypes*) between the start and end of the graph are enumerated,

3. Each read is realigned to each haplotype, typically using a pair Hidden Markov Model (HMM, see Durbin et al [36]),

4. A statistical model uses the read↔haplotype alignments to choose the haplotype pair that most likely represents the variants hypothesized to exist in the region,

5. The alignments of the reads to the chosen haplotype pair are used to generate statistics that are then used for genotyping.

In this paper, we focus on improving the algorithmic efficiency steps one through three of the local reassembly problem. We do not focus algorithmically on accelerating stages four and five, as there is wide variation in the algorithms used in stages four and five. However, we do provide an parallel implementation of a widely used statistical model for genotyping [69]. Stage one (graph creation) has approximately $\mathcal{O}(rl_r)$ time complexity, and stage two (graph elaboration) has $\mathcal{O}(h \max(l_h))$ time complexity. The asymptotic time cost bound of local reassembly comes from stage three, where cost is $\mathcal{O}(hrl_r \max(l_h))$, where h is the number of haplotypes tested in this region[1], r is the number of reads aligned to this region, l_r is the read length[2], and $\min(l_h)$ is the length of the shortest haplotype that we are evaluating. This complexity comes from realigning r reads to h haplotypes, where realignment has complexity $\mathcal{O}(l_r l_h)$.

In this paper, we introduce the indexed de Bruijn graph and demonstrate how it can be used to reduce the asymptotic complexity of reassembly. An indexed de Bruijn graph is identical to a traditional de Bruijn graph, with one modification: when we create the graph, we annotate each k-mer with the index position of that k-mer in the sequence it was observed in. This simple addition enables the use of the indexed de Bruijn graph for $\Omega(n)$ local sequence alignment with canonical edit representations for most edits. This structure can be used for both sequence alignment and assembly, and achieves a more efficient approach for variant discovery via local reassembly. To further improve the efficiency of this approach, we demonstrate in §7.1 how we can implement the canonicalization scheme that we demonstrate using indexed de Bruijn graphs without constructing a de Bruijn graph that contains both sequences.

Current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [71]. Although there are several approaches that do not make explicit use of reassembly, all realignment based variant callers use an algorithmic structure similar to the one described above. In non-assembly approaches like FreeBayes [44], stages one and two are replaced with a single step where the variants observed in the reads aligned to a given haplotyping region are filtered for quality and integrated directly into the reference haplotype in that region. In both approaches, local alignment errors (errors in alignment

[1]The number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see Depristo et al [34] and Garrison and Marth [44]) allow users to limit the number of haplotypes evaluated to improve performance.

[2]For simplicity, we assume constant read length. This is a reasonable assumption as many of the variant callers discussed target Illumina reads that have constant length.

within this region) are corrected by using a statistical model to identify the most likely location that the read could have come from, given the other reads seen in this area.

Although the model used for choosing the best haplotype pair to finalize realignments to varies between methods (e.g., the GATK's IndelRealigner uses a simple log-odds model [34], while methods like FreeBayes [44] and Platypus [106] make use of richer Bayesian models), these methods require an all-pairs alignment of reads to candidate haplotypes. This leads to the runtime complexity bound of $\mathcal{O}(hrl_r \min(l_h))$, as we must realign r reads to h haplotypes, where the cost of realigning one read to one haplotype is $\mathcal{O}(l_r \max(l_h))$, where l_r is the read length (assumed to be constant for Illumina sequencing data) and $\max(l_h)$ is the length of the longest haplotype. Typically, the data structures used for realignment ($\mathcal{O}(l_r \max(l_h))$ storage cost) can be reused. These methods typically retain *only* the best local realignment per read per haplotype, thus bounding storage cost at $\mathcal{O}(hr)$.

For non-reassembly based approaches, the cost of generating candidate haplotypes is $\mathcal{O}(r)$, as each read must be scanned for variants, using the pre-existing alignment. These variants are typically extracted from the CIGAR string, but may need to be normalized [71]. de Bruijn graph based reassembly methods have similar $\mathcal{O}(r)$ time complexity for building the de Bruijn graph as each read must be sequentially broken into k-mers, but these methods have a different storage cost. Specifically, storage cost for a de Bruijn graph is similar to $\mathcal{O}(k(l_{\text{ref}} + l_{\text{variants}} + l_{\text{errors}}))$, where l_{ref} is the length of the reference haplotype in this region, l_{variants} is the length of true variant sequence in this region, l_{errors} is the length of erroneous sequence in this region, and k is the k-mer size. In practice, we can approximate both errors and variants as being random, which gives $\mathcal{O}(kl_{\text{ref}})$ storage complexity. From this graph, we must enumerate the haplotypes present in the graph. Starting from the first k-mer in the reference sequence for this region, we perform a depth-first search to identify all paths to the last k-mer in the reference sequence. Assuming that the graph is acyclic (a common restriction for local assembly), we can bound the best case cost of this search at $\Omega(h \min l_h)$.

The number of haplotypes evaluated, h, is an important contributor to the algorithmic complexity of reassembly pipelines, as it sets the storage and time complexity of the realignment scoring phase, the time complexity of the haplotype enumeration phase, and is related to the storage complexity of the de Bruijn graph. The best study of the complexity of assembly techniques was done by Kingsford et al. [58], but is focused on *de novo* assembly and pays special attention to resolving repeat structure. In the local realignment case, the number of haplotypes identified is determined by the number of putative variants seen. We can naïvely model this cost with (2.1), where f_v is the frequency with which variants occur, ϵ is the rate at which bases are sequenced erroneously, and c is the coverage (read depth) of the region.

$$h \sim f_v l_{\text{ref}} + \epsilon l_{\text{ref}} c \tag{2.1}$$

This model is naïve, as the coverage depth and rate of variation varies across sequenced datasets, especially for targeted sequencing runs [40]. Additionally, while the ϵ term mod-

els the total number of sequence errors, this is not completely correlated with the number of *unique* sequencing errors, as sequencing errors are correlated with sequence context [34]. Many current tools allow users to limit the total number of evaluated haplotypes, or apply strategies to minimize the number of haplotypes considered, such as filtering observed variants that are likely to be sequencing errors [44], restricting realignment to INDELs (IndelRealigner, [34]), or by trimming paths from the assembly graph. Additionally, in a de Bruijn graph, errors in the first k or last k bases of a read will manifest as spurs and will not contribute paths through the graph. We provide (2.1) solely as a motivating approximation, and hope to study these characteristics in more detail in future work.

2.3 Distributed Computing Platforms

Google described the use of large clusters of commodity computers in their MapReduce system [31, 32] in 2004. Since then, there has been a surge of activity focusing on the development of distributed data analysis tools. In the open source world, this has spawned the Apache Hadoop project [8], which started as a open source reimplementation of Google's MapReduce. Hadoop led to the development of scripting languages like Pig [96], query systems like Hive [128], and resource management frameworks like YARN [132] and Mesos [53]. While traditional map-reduce platforms were well suited to extract, transform, load (ETL) pipelines that made a single pass over a large dataset, they were a poor fit to "advanced analytics" applications—like machine learning, or graph processing—that made several passes over a dataset. This was due to their reliance on the output of every computational phase being written to disk to ensure fault tolerance. A new set of distributed data processing tools were designed to address this problem by storing data in memory, and relying on different models for fault resilience. These systems include Spark [144, 143] and Flink [17]. Additionally, a set of highly efficient query engines came out, such as Cloudera Impala [59] and Spark SQL [10].

Distributed Genomic Analysis Tools

Genomics tools that leverage commodity distributed computing have typically taken one of two approaches: either they wrap a single-node tool so that it can be parallelized using a distributed computing framework, or they define a distributed query model for a single area/tool of focus. Beyond these two approaches, some tools have been built on distributed computing technologies from the HPC ecosystem. Additionally, cloud-friendly workflow management systems have entered broad usage.

There have been three waves of development focused on integrating single-node tools with distributed computing platforms. The first wave of development used Apache Hadoop Streaming as a simple mechanism for parallelising tools that had well defined chunking patterns. Examples of this approach include the CloudBurst aligner [111], which parallelized the RMAP aligner [117], and CrossBow [65], which integrates the Bowtie [66]

aligner with the SoapSNP [74] variant caller. The second wave of approaches built more fully featured applications on top of the Apache Hadoop framework that did not just rely on the streaming APIs. These applications include the SEAL [103] aligner, which extracted the BWA [73] aligner into a Python library which was executed on the PyDoop [68] bindings for Hadoop; BigBWA [3], which parallelizes BWA [73] using the Java Native Interface (JNI) on top of Apache Hadoop; and Halvade [33], which parallelized the complex dataflow in the GATK [34] using Apache Hadoop. The third wave of wrappers has been built around Apache Spark and includes SparkBWA [4], a successor to BigBWA [3], CloudScale-BWA MEM [19], which parallelizes BWA through the JNI, with the ability to support FPGA acceleration; and SparkGA [87], which uses a similar approach as Halvade to parallelize the GATK, but is implemented on Spark.

Several tools have implemented genomic analyses directly on top of distributed analysis tools from the Apache Hadoop and Spark ecosystem. Many of these tools build on top of the Hadoop-BAM library [92], which provides Hadoop-compatible parallel I/O libraries. The first generation of tools built query models for accessing genomic data through the Pig [96] scripting language. This was implemented in two separate tools: BioPig [93] and SeqPig [112]. Additionally, the OpenCB project has built Hadoop-based tools for manipulating genomic data via the hpg-bigdata project [97]. Recent work has moved on to Apache Spark. Beyond ADAM and the Big Data Genomics ecosystem, Spark has been used in the SparkSeq [139] and VariantSpark [98] tools. SparkSeq is geared towards RNA-seq analysis, and has been paired with SparkBwa [4] to build the Falco [141] single-cell RNA-seq pipeline which runs end-to-end on Apache Spark. VariantSpark includes novel methods for statistically analyzing genotype data on Spark, including an efficient implementation of random forests for wide-but-flat genomic data. There is increasing adoption of Apache Spark in genomics, with two large unpublished projects coming out of the Broad Institute. The first is the fourth edition of the GATK [126], which is reimplemented on Spark. The second project is Hail [125], which is a reimplementation of the PLINK population genomics [104] toolkit on Spark.

We do not extensively discuss non-resequencing pipelines for de novo genome assembly in this dissertation, but genome assembly has different access patterns that are more amenable to HPC-styled distributed implementations. Specifically, since de novo assembly operates on highly connected graphs, efficiently mapping de novo assembly to a graph-parallel framework like GraphX [51] is difficult. The ABySS assembler [116] uses the Message Passing Interface (MPI) to parallelize genome assembly across an HPC cluster. A new and exciting avenue of work is using HPC systems that support a parallel global address space (PGAS) and Remote Direct Memory Access (RDMA) to achieve extremely fine grained parallelism [47, 46, 48, 49].

Part II

Architecture and Infrastructure

Chapter 3

Design Principles for Scalable Genomics

When we started designing ADAM in 2013, Apache Spark was still in early development, and few organizations were actively working with massive genomics data sets. At the time, we believed that the major pain points in working with large scale genomics datasets centered around low-level APIs that made it difficult to represent complex genomic data manipulations and the use of file formats that were difficult to access in parallel and that had imprecise specifications. This led to the initial goals for the ADAM project:

- Provide clean APIs for writing large scale genomic data analyses
- Raise abstraction by centering data manipulation around schemas instead of file formats
- Allow these APIs to be exposed across commonly used languages
- Efficiently execute non-reference oriented query patterns

To achieve these goals, we designed a decoupled, stack-oriented architecture that was centered around schemas that provided a logical view over the genomic data that was being manipulated. This architecture was implemented on top of Apache Spark's RDD APIs, and provided the user with a distributed collection of genomic data which were encoded in Apache Avro [7]. This allowed for queries to be described at a high level through Spark's RDD APIs, which would execute the queries rapidly by running parallel scans over the data. Over time, our goals grew in scope to include:

- Support coordinate-space joins with genomic data
- Support exploratory data analysis on genomic datasets
- Allow people to reuse their existing genomic analysis tools on Spark with minimal modifications

Because of ADAM's decoupled architecture, we were able to easily enhance ADAM to support these query patterns. By refactoring how ADAM tracked data partitioning, the coordinate-space joins (§4.3) and the PIPE API for supporting legacy genomics tools (§4.3) were added to ADAM's core APIs. The MANGO project enhanced ADAM's ability to run interactive queries against genomic data by improving support for pushing down ranged predicates to disk [129] and by adding a spatial- and temporal- locality-aware in-memory caching layer [84]. These modifications replaced ADAM's default query and data access layers with layer implementations better suited to the query patterns at hand.

In this section, we will revisit the pain points we asserted, and describe how our understanding of these pain points changed over time. From these pain points, we then reify a set of functional requirements for a distributed data analysis platform for manipulating genomic data. We then introduce ADAM's stack architecture, and explain how it addresses these needs.

3.1 Pain Points with Single Node Genomics Tools

Most current genomic pipelines are built entirely out of single-node tools, and often single threaded tools. We believe that the barriers to making use of distributed tools are caused by the computational patterns used when building traditional single node tools. As the size and scope of genomic data continues to increase, single node analyses will become inconvenient or impractical to run.

Expressiveness of APIs

Traditionally, APIs for manipulating genomic data have been very low level. Typically, tools follow the "walker" pattern, which provides a sorted iterator over the genome. The user then implements any traversal that they need. This approach is undesirable for two reasons:

1. Due to the very low level nature of the API, programmers must implement their own complex transforms, such as grouping together reads that start at a single genomic position. This can lead to errors in user code.

2. A natural consequence of the first point is that low level APIs obscure the actual query pattern that is being implemented. For example, a duplicate marker typically groups together all reads that aligned to a single genomic locus. This pattern is clear when duplicate marking is written as a high level algorithm, but is unclear from a low level implementation of duplicate marking on a sorted iterator.

These two issues translate into two obvious consequences. First, a low level API increases the complexity of implementing a query, and thus necessitates increased developer effort. This introduces more locations where queries can be implemented incorrectly. We identified two concrete examples of this when working on the read preprocessing pipeline in ADAM.

Specifically, we identified that the Picard duplicate marker and the GATK base quality recalibrator incorrectly process reads that come from sequenced fragments whose insert size violates undocumented invariants that are internal to each tool.

The second obvious consequence is that monolithic queries are difficult to automatically optimize. To examine this consequence, we can look again at the duplicate marking kernel. If we are calling variants in a whole exome sequencing (WES) dataset, we would run a query pattern with several steps:

1. Align reads
2. Sort reads
3. Mark duplicates by grouping by alignment position
4. Filter out reads mapped outside of the exome
5. Call variants by aggregating statistics at each position covered by an aligned read

A query planner that is aware of the structure of the genome could apply several optimizations:

- Since the duplicate marker groups by position, sorting and duplicate marking can be combined into a single phase. This optimization can also be applied to variant calling.
- Since we will filter out reads mapped outside of the exome, we can push this predicate up to after alignment.

In the absence of the ability to optimize the variant calling query plan end-to-end, most genomics tools achieve performance benefits by enforcing sort order or grouping invariants. For example, the Picard [127] and Sambamba [123] duplicate markers require read inputs to be coordinate sorted, while the SAMBLASTER [41] duplicate marker requires the read data to be queryname grouped. These invariants are necessary for efficiency but come at the cost of increased complexity when integrating multiple tools together into a monolithic pipeline.

The combination of these two issues yields a final problem: it is difficult for a domain scientist to parallelize these queries. To parallelize single node tools across a large cluster, existing tools typically use a "scatter-gather" pattern (see discussion of many-task workflow patterns in Zheng et al. [146, 147]) that chunks a genomic dataset into many small parts (contiguous ranges of the genome) that are then processed independently. This presents several problems:

- Due to bias caused by repeated sequence in the genome, we cannot achieve optimal load balance with a partitioner that naïvely partitions the genome into uniformly sized ranges [21].

- This query pattern is restricted to storage systems that support efficient ranged access into files, and may not be efficient to implement on cloud-based shared-nothing stores [134].
- This approach makes it difficult to implement queries that need to run an all-reduce over the data. Examples of this include the base quality score recalibration kernel (see §6.1), or genome-wide machine learning methods [85].

Support for Parallel I/O

While most of the file formats used for genomics do not preclude parallel I/O, their structure makes parallel I/O difficult to implement in an efficient manner. This is caused by two primary factors:

1. The files are often expensive to split, in the absence of an index.
2. Due to the emphasis on chaining tools into streams or transformations upon a single file, all output must be serialized.

To demonstrate the source of costs for performing parallel reads, let us look at Hadoop-BAM [92], a popular library used for loading genomics data into Apache Hadoop or Spark. To split a BAM file, Hadoop-BAM implements Hadoop's InputFormat class. When a file is opened for read, Hadoop provides Hadoop-BAM with approximate ranges that the file should be split into. However, a BAM file cannot be arbitrarily split; we must find the first valid BAM record after the start of this file range. To do this, Hadoop-BAM must scan through the file, looking for a sequence of bytes that indicates the start of a record. Currently, this is implemented sequentially per split in a file. For a typical block size of 128MB, a BAM from a high coverage whole genome sequencing run will have 500–1500 splits. The cost of computing the splits can be very high, especially if the data is stored in a remote file store, as is common on cloud computing vendors. To eliminate this issue, Hadoop-BAM supports a proprietary index that stores valid split start positions. Recently, support was added that uses the linear BAM index to validate split start positions. An additional issue is that Hadoop-BAM does not always pick valid positions to begin reading. To address this issue, the Spark-BAM [140] project has begun rewriting parts of Hadoop-BAM to add more stringent record validation tests. These more stringent tests reduce the number of false positive record start positions to 0.

A general exception is files compressed with the GZIP codec, which is not splittable. GZIP is used in bioinformatics because of its high compression rate, which is useful for storing textual read data, such as FASTQ files. To allow splitting, many tools use the BGZF codec [70], which is splittable. BGZF incurs a small performance and compression overhead relative to the raw GZIP codec. Similar to the example with BAM files given above, BGZF also has high overhead for splitting without an index.

As mentioned above, another general issue is that genomics tools often assume that data is being processed as I/O streams, or that there is a single file that corresponds to data from a single sample. The architectural implication of this trend is that I/O must be serialized, which leads to a typical I/O subsystem loading data in through a single thread, which delegates the data to multiple decompression threads. The data are then decompressed, processed, and compressed again, at which point the writes to a disk or to a stream are serialized. This creates contention at both ends of the tool. The impact of this can easily be seen in a highly efficient tool, like the Salmon RNA-seq quantification engine [100]. Due to I/O contention, Salmon is unable to scale beyond 32 cores.

3.2 Goals for a Scalable Genomics Library

From the pain points we described above, we can assemble a set of goals for a clean slate genomic analysis platform. In this section, we describe both the original goals that we adopted when building ADAM, as well as the goals that evolved for the ADAM ecosystem over time. From our original set of goals, we designed the stack architecture that we introduce in the next section (§3.3). As we will see, the stack architecture made it easier for us accomodate the goals that evolved during the course of the project.

Original Goals for ADAM

In the original ADAM technical report [78], we introduced an architecture that presented fairly minimalistic wrappers around the Apache Spark APIs, but that was built out of technical components that were optimized for batch processing over large genomic datasets. This original architecture addressed the goals described in this section, and paved the way for supporting the goals introduced in the next section.

Our overarching goal was to abstract away from APIs that derived directly from the genomic file formats, towards higher level APIs for manipulating genomic data. In doing this, we had several specific goals:

1. Minimize the amount of code needed to implement a query.
2. Eliminate the need for sort order invariants.
3. Make it more efficient to execute queries that only touched a subset of genomic data.
4. Make our APIs usable across multiple languages.
5. Make it possible to easily query data using other parallel analysis tools than Apache Spark.

Because of the stack smashing described in §3.1, genomics libraries like htslib (the base for samtools [73]) and htsjdk (the base for Picard [127] and the GATK [79]) provide

APIs that iterate directly across a file. As a comparison, Apache Spark is built on top of the RDD API, which describes a collection of records which is parallelized across nodes in a cluster. Because the abstractions provided by the htslib and htsjdk systems were guided by the file formats themselves, idioms from data traversal (iterating across a collection) and I/O (closing a stream) became intermingled. Additionally, the behavior of a row in a collection was influenced by the I/O process. This is because all extant genomics file formats are row oriented, which means that the only way to improve the performance of a query that does not access all of the fields in a record was to lazily parse the fields as they were accessed.

Instead of bleeding abstractions from the I/O layer through the stack, we decided to introduce schemas that represented the major genomic datatypes. This enforces a strict separation between the I/O layer and the end-user API. This naturally supports the many file formats that can describe a given class of genomic data, since we can provide a view between the genomic file format and the schemas. Additionally, since a schema is fundamentally a logical representation of a record, our schemas need not be language-specific, and should be reusable across a large set of languages.

To eliminate the need for sort order invariants, we proposed a two pronged approach. First, since we were building on a system that enabled parallel I/O, we would be able to achieve high query performance by running full scans over the dataset in parallel. Additionally, by providing APIs for filtering by row when reading from the file system, or for selecting the specific columns that we were interested in parsing, we could minimize the amount of data read from disk. To further accelerate these queries, we used the Apache Parquet [9] columnar storage system, which enabled cheap column projections and predicates that could be pushed down into the storage system. Since Parquet was gaining broader adoption across the analytics ecosystem, storing data in Parquet meant that it could be accessed and manipulated with tools such as Apache Hive [128] and Cloudera Impala [59].

Goals That Evolved Over Time

Over the course of building out the ADAM project and the surrounding Big Data Genomics ecosystem, we added several design goals. These included:

1. Supporting interactive latency for exploratory queries and data visualization.
2. Being able to optimize queries to take advantage of presorted data.
3. Allowing legacy tools to be reused and parallelized.

There were several reasons that we added these design goals. We had originally intended ADAM to mainly support batch analyses, and deferred support for interactive analysis to query engines like Impala. However, we felt that there was a good opportunity to reshape interactive genomic data analysis by enabling exploratory data analysis using ADAM's APIs. To do this, we needed a way to lower the latency of typical Spark queries. We did this in

the Mango project [129, 84], which introduced better support for using primary indices on genomic position, as well as an efficient in-memory caching layer. These implementations replaced the default implementations of two levels of our stack.

As noted in the previous paragraph, interactive query was one reason we wanted to take advantage of data being sorted at query time. Additionally, as we built out ADAM's read preprocessing transformations (see Chapter 6), we realized that many of these queries depended on joining genomic data against other overlapping data (such as joining aligned reads against variants during BQSR's masking phase), or aggregating at a single genomic locus (as in duplicate marking). In our SIGMOD article [94], we introduced the region join to ADAM. This join provided functionality similar to BEDTools [105], and could be implemented using both a broadcast and a sort-merge strategy. While our goal was to eliminate the need for sort-order invariants, we saw that there was a good opportunity to accelerate these join and aggregate patterns by eliminating shuffles whenever a dataset was already sorted. We describe the extensions we made to ADAM to support these optimizations in §4.3.

Finally, while we feel that ADAM's APIs provide a significant improvement over traditional genomic query models, we realized over time that it was an unrealistic goal to supplant these APIs due to their widespread usage. Additionally, the experiences of our coworkers during the SNAP project [142] led us to realize that Spark's APIs were not a good fit to all genomic data analysis problems. Specifically, the large indices used during short read alignment are difficult to manage efficiently in the Java Virtual Machine's (JVM) managed memory model. Additionally, the prior work which used Hadoop or Spark to manually wrap and parallelize legacy tools [111, 65, 3, 4, 20, 19] led us to believe that there was interest in having a general API for parallelizing genomics tools. This led to the introduction of ADAM's pipe API, and the Cannoli tool, which is described in Chapter 5.

3.3 A Stack Architecture for Scientific Data Processing

The processing patterns being applied to scientific data shift widely as the data itself ages. Because of this change, we want to design a scientific data processing system that is flexible enough to accommodate our different use cases. At the same time, we want to ensure that the components in the system are well isolated so that we avoid bleeding functionality across the stack. If we bleed functionality across layers in the stack, we make it more difficult to adapt our stack to different applications. Additionally, as we discuss in §**??**, improper separation of concerns can actually lead to errors in our application.

These concerns are very similar to the factors that led to the development of the Open Systems Interconnection (OSI) model and Internet Protocol (IP) stack for networking services [148]. The networking stack models were designed to allow the mixing and matching of different protocols, all of which existed at different functional levels. The success of the networking stack model can largely be attributed to the "narrow waist" of the stack, which

simplified the integration of a new protocol or technology by ensuring that the protocol only needed to implement a single interface to be compatible with the rest of the stack.

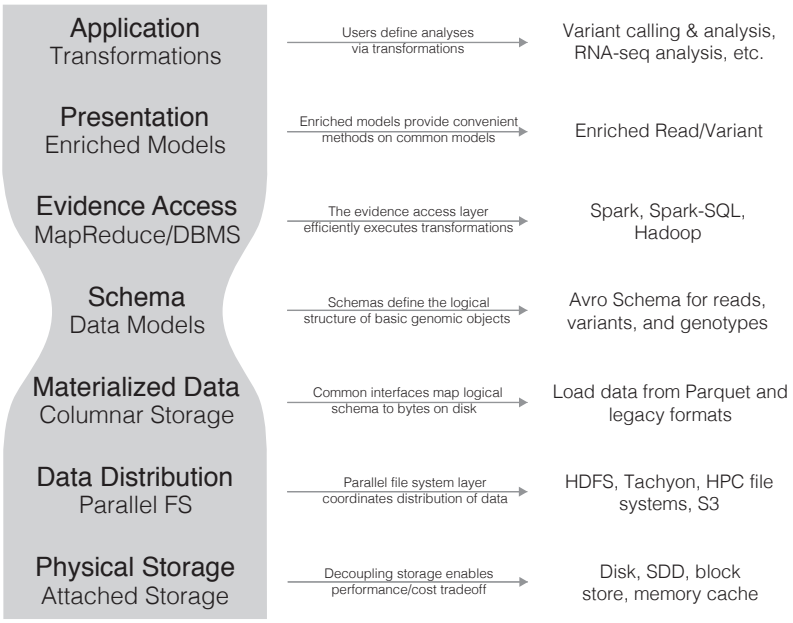

Figure 3.1: A stack model for scientific computing

Unlike conventional scientific systems that leverage custom data formats like BAM or SAM [73], or CRAM [43], we believe that the use of an explicit schema for data interchange is critical. In our stack model shown in Figure 3.1, the schema becomes the "narrow waist" of the stack. Most importantly, placing the schema as the narrow waist enforces a strict separation between data storage/access and data processing. Additionally, this enables literate programming techniques which can clarify the data model and access patterns. The seven layers of our stack model are decomposed as follows, and are numbered in ascending order from bottom to top:

1. **Physical Storage:** This layer coordinates data writes to physical media.
2. **Data Distribution:** This layer manages access, replication, and distribution of the files that have been written to storage media.
3. **Materialized Data:** This layer encodes the patterns for how data is encoded and stored. This layer determines I/O bandwidth and compression.
4. **Data Schema:** This layer specifies the representation of data, and forms the narrow waist of the stack that separates access from execution.

5. **Evidence Access:** This layer provides us with primitives for processing data, and allows us to transform data into different views and traversals.

6. **Presentation:** This layer enhances the data schema with convenience methods for performing common tasks and accessing common derived fields from a single element.

7. **Application:** At this level, we can use our evidence access and presentation layers to compose the algorithms to perform our desired analysis.

A well defined software stack has several other significant advantages. By limiting application interactions with layers lower than the presentation layer, application developers are given a clear and consistent view of the data they are processing, and this view of the data is independent of whether the data is local or distributed across a cluster or cloud. By separating the API from the data access layer, we improve flexibility. With careful design in the data format and data access layers, we can seamlessly support conventional whole file access patterns, while also allowing easy access to small slices of files. By treating the compute substrate and storage as separate layers, we also drastically increase the portability of the APIs that we implement.

As we discuss in more detail in §**??**, current scientific systems bleed functionality between stack layers. An exemplar is the SAM/BAM and CRAM formats, which expect data to be sorted by genomic coordinate. This order modifies the layout of data on disk (level 3, Materialized Data) and constrains how applications traverse datasets (level 5, Evidence Access). Beyond constraining applications, this leads to bugs in applications that are difficult to detect.[1] These views of evidence should be implemented at the evidence access layer instead of in the layout of data on disk. This split enforces independence of anything below the schema.

The idea of decomposing scientific applications into a stack model is not new; Bafna et al [11] made a similar suggestion in 2013. We borrow some vocabulary from Bafna et al, but our approach is differentiated in several critical ways:

- Bafna et al consider the stack model specifically in the context of data management systems for genomics; as a result, they bake current technologies and design patterns into the stack. In our opinion, a stack design should serve to abstract layers from methodologies/implementations. If not, future technology trends may obsolete a layer of the stack and render the stack irrelevant.

- Bafna et al define a binary data format as the narrow waist in their stack, instead of a schema. While these two seem interchangeable, they are not in practice. A schema is a higher level of abstraction that encourages the use of literate programming techniques and allows for data serialization techniques to be changed as long as the same schema is still provided.

[1] The current best-practice implementations of the BQSR and Duplicate Marking algorithms both fail when processing certain corner-case alignments. These errors are caused because of the requirement to traverse reads in sorted order.

- Notably, Bafna et al use this stack model to motivate GQL [60]. While a query system should provide a way to process and transform data, Bafna et al instead move this system down to the data materialization layer. We feel that this inverts the semantics that a user of the system would prefer and makes the system less general.

Deep stacks like the OSI stack [148] are generally simplified for practical use. Conceptually, the stack we propose is no exception. In practice, we combine layers one and two, and layers five and six. There are several reasons for these mergers. First, in HADOOP-based systems, the system does not have practical visibility below layer two, thus there is no reason to split layers one and two except as a philosophical exercise. Layers five and six are commingled because some of the enriched presentation objects are used to implement functionality in the evidence access layer. This normally happens when a key is needed, such as when repartitioning the dataset, or when reducing or grouping values.

Chapter 4

The ADAM Architecture

ADAM's architecture was introduced as a response to the challenges processing the growing volume of genomic sequencing data in a reasonable timeframe [110, 120]. While the per-run latency of current genomic pipelines such as the GATK could be improved by manually partitioning the input dataset and distributing work, native support for distributed computing is not provided. As a stopgap solution, projects like CLOUDBURST [111] and CROSSBOW [64] have ported individual analytics tools to run on top of HADOOP. While this approach has served well for proofs of concept, this approach provides poor abstractions for application developers. These poor abstractions make it difficult for bioinformatics developers to create novel distributed genomic analyses, and does little to attack sources of inefficiency or incorrectness in distributed genomics pipelines.

ADAM's architecture reconsiders how we build software for processing genomic data by eliminating the monolithic architectures that are driven by the underlying flat file formats used in genomics. These architectures impose significant restrictions, including:

- These implementations are locked to a single node processing model. Even the GATK's "map-reduce" styled WALKER API [79] is limited to natively support processing on a single node. While these jobs can be manually partitioned and run in a distributed setting, manual partitioning can lead to imbalance in work distribution and makes it difficult to run algorithms that require aggregating data across all partitions, and lacks the fault tolerance provided by modern distributed systems such as APACHE HADOOP or SPARK [143].

- Most of these implementations *assume* invariants about the sorted order of records on disk. This "stack smashing" (specifically, the layout of data is used to "accelerate" a processing stage) can lead to bugs when data does not cleanly map to the assumed sort order. Additionally, since these sort order invariants are rarely explicit and vary from tool to tool, pipelines assembled from disparate tools can be brittle. We discuss this more in Chapter 6.

- Additionally, while these invariants are intended to improve performance, they do this at the cost of opacity. If we can express the query patterns that are accelerated by these invariants at a higher level, then we can achieve both a better programming environment and enable various query optimizations.

At the core of ADAM, users use the ADAMContext to load data as GenomicRDDs, which they can then manipulate. Figure 4.1 depicts the GenomicRDD class hierarchy. In this class hierarchy, we provide several classes that contain functionality that is applicable to all genomic datatypes, such as the coordinate-space primitives described in §4.3 and the pipe primitive described in §5.1, and the genomic metadata management described in §4.2.

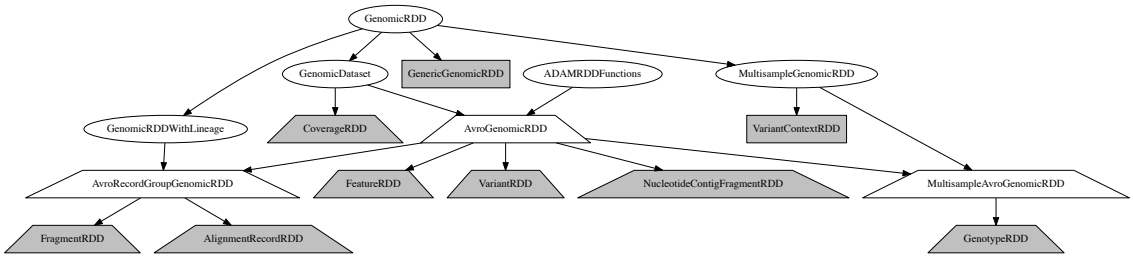

Figure 4.1: The GenomicRDD class hierarchy. Ovals are traits, trapezoids are abstract classes, rectangles are classes, and a bold border means that the abstract class is sealed. In Scala, sealing a class/trait restricts which classes can extend this class/mix-in this trait. The shaded classes are the types returned from the ADAMContext load methods and from the GenomicRDD methods.

While the ADAMContext API has existed since the beginning of the ADAM project, the GenomicRDD API is a fairly recent introduction, having been added in 2016, three years into the project. Despite its recent addition, the GenomicRDD API has been critical to achieving the vision behind ADAM:

- The GenomicRDD wraps the Apache Spark RDD [143] and Spark SQL DataFrame [10] APIs with genomics-specific and genomic-datatype specific functionality. This enables the use of both generic (RDD/DataFrame) and specialized APIs (the region join and pipe APIs) to process genomic data in a natively distributed manner.
- The rich GenomicRDD type hierarchy enables methodical management of genomics metadata. We support metadata management at a read group, sample, and computational lineage level.
- By building upon APIs from Spark SQL, the GenomicRDD API can be exposed in the Python and R languages, which enables the use of ADAM outside of the JVM.

In the rest of this chapter, we explain the design decisions behind the ADAM architecture, with an eye towards how the architecture has evolved over the four years of the project. We discuss the specific tradeoffs we needed to make in order to realize the stack architecture we had introduced, before diving into the schema design that we used to decouple ADAM from the genomics file formats. We review the genomics-specific query patterns supported in ADAM, and explain how we have broadened ADAM's APIs to support multiple languages.

4.1 Realizing A Decoupled Stack Architecture In ADAM

While a stack-based architecture provides many of the benefits we asserted in the previous section, these benefits can only be realized with careful API design. If the APIs are specified at too low of a level of abstraction, then the implementations of each layer will leak through, and the stack layers cannot be exchanged. If the APIs are specified at too high of a level of abstraction, then programmers who are implementing their applications on top of the APIs cannot meaningfully reason about the performance and semantics of the code that they are running.

In ADAM, we ultimately extend two important APIs. The ADAMContext is the entrypoint to loading all data, while the GenomicRDD API provides an abstract wrapper for genomic datasets. We specialize the GenomicRDD API across the various genomic datatypes. Once data has been loaded using the ADAMContext, users largely interact with the data by transforming the data enclosed in a GenomicRDD, which is described by our schemas (see §4.2). These core APIs contribute to realizing the stack vision in the following ways:

- The physical storage and data distribution layers (1 and 2) are largely deferred to Apache Spark and Hadoop. ADAM interacts with these layers through their APIs. There are a few places where ADAM specializes for a given file system; the primary example is when concatenating output files together to create a large unified output file. The Hadoop Distributed File System (HDFS, see Shvachko et al. [115]) and the Amazon Web Services Simple Storage System (AWS S3) provide APIs for concatenating sharded files, which allows the file merging process to be accelerated.

- The materialized data and schema layers (3 and 4) couple together in a critical way. The schemas provide a logical description of a given genomic datatype, and the materialized data layer provides a set of views between the ADAM schemas and legacy genomics file formats. This view is applied by the ADAMContext when loading data, and by the GenomicRDD when saving data. This decomposition is critically important: a common early misunderstanding of ADAM was that our schemas were introducing a new file format for genomic data. Rather, by supporting views between common

formats to our schemas, we eliminate the need to know which file formats the data was loaded from.

- The evidence access layer (5) is largely implemented in the GenomicRDD, but can also be specialized in downstream applications for their specific application/query pattern. At a basic level, ADAM uses Apache Spark [143] and Spark SQL [10] for evidence access. Without any optimizations, queries are implemented as scans over the full dataset. However, we apply several optimizations for coordinate-based queries, as described in §4.3. Additionally, since ADAM builds off of Spark's RDD API, any tools that optimize at the RDD level can be used to enrich the evidence access layer [129, 84].

- The presentation layer (6) provides datatype (e.g., read/feature/variant) specific methods on top of the GenomicRDD API. This includes many of the operations introduced in Chapter 6.

- The application layer (7) is where the user builds their code, and this is built by using the ADAMContext to load in a GenomicRDD, which is then transformed and saved.

While many of these layers have been unchanged since we introduced our stack model in our original technical report [78] and our SIGMOD manuscript [94], the evidence access and presentation layers (5 and 6) have changed. While the ADAMContext API has existed in ADAM since the beginning of the project, the GenomicRDD API was only introduced in 2016, three years into the project. The introduction of the GenomicRDD API was symbolic of larger changes, which included:

- The evidence access layer originally assumed that other query systems like Cloudera Impala [59] would be able to interoperate with the ADAM schemas to load data. While this is still true, we have deemphasized this view. Specifically, once data has been loaded using the ADAM schemas and saved into Apache Parquet, systems like Impala can interoperate with the data through the ADAM schemas. However, systems like Impala cannot interoperate with the lower levels of ADAM's stack, such as ADAM's views to and from legacy genomics file formats.

- Because the Hadoop ecosystem optimizes for full scans, we originally assumed that all queries were executed as full scans over the dataset, and these queries were executed without assuming any knowledge of the layout of the data. One of the advantages of introducing the GenomicRDD API is that we were better able to track the layout of data, and to specialize queries given a known layout. We describe one such mechanism for accelerating queries in the genomic coordinate space in §sec:query-patterns.

- The presentation layer (6) was originally implemented by enhancing the schemas. Specifically, the AlignmentRecord schema (see §4.2) was accompanied by a RichAlignmentRecord class, which provided convenience methods on top of the schema. Similar "enriched" classes existed for the Variant schema as well. While these enriched classes have not been eliminated from ADAM, we have eliminated the public API exposure of these classes. While these classes were useful for implementing some of the algorithms in the ADAM core, they ultimately proved difficult to understand the performance characteristics of. They leveraged the Scala language's compile-time implicit conversion functionality [95], which allows for the compile-time inclusion of a method that can satisfy a given type signature. Additionally, the enriched classes typically augmented the raw schemas by lazily computing expensive values, which could then be reused across calculations. Unfortunately, the way these patterns intersected made it very difficult to reason about when a state had been calculated, and thus what the performance of a call to an enriched class would be. Additionally, the implicit conversion pattern could not be supported across languages, and was out of the scope of experise of the average bioinformatics programmer. Instead, we surfaced transformations at the dataset level, which is closer to the level of abstraction expected by our average user.

While the introduction of the GenomicRDD class has enabled the changes to our stack model that were described above, it was driven by other factors. The ADAMContext would originally return unwrapped Spark RDDs. The introduction of the GenomicRDD class was driven largely by the need for better management of genomic metadata, as described in §4.2.

4.2 Schema Design for Representing Genomic Data

A common criticism of bioinformatics as a field surrounds the proliferation of file formats. Short read data alone is stored in four common formats: FASTQ [27], SAM [73], BAM, and CRAM [43]. While these formats all represent different layouts of data on disk, they tend to be logically harmonious. Due to this logical congruency of the different formats, we chose to build ADAM on top of a logical schema, instead of a binary format on disk. While we do use Apache Parquet [9] to materialize data on disk, the Apache Avro [7] schema is used as a narrow waist in the system, that enables "legacy" formats to be processed identically to data stored in Parquet with modest performance degradation.

We made several high level choices when designing the schemas used in ADAM [78]. Over time, we have gone backwards on some of these design decisions. First, the schemas were originally fully denormalized. Our rationale for thie decision was that this would reduce the cost of metadata access while simplifying metadata distribution. We believed we would be able to gain these benefits without greatly increasing the cost of memory access because our backing store (Parquet) made use of run length and dictionary encoding, which allows

for a single object to be allocated for highly repetitive elements on read. As we discuss in §4.2, this decision wound up being costly, and we reversed this decision as the ADAM schemas progressed. Another key design choice was to require that all fields in the schema are nullable; by enforcing this requirement, we enable arbitrary user specified projections. Arbitrary projections can be used to accelerate common sequence quality control algorithms such as Flagstat [78, 94]. We have still maintained this requirement.

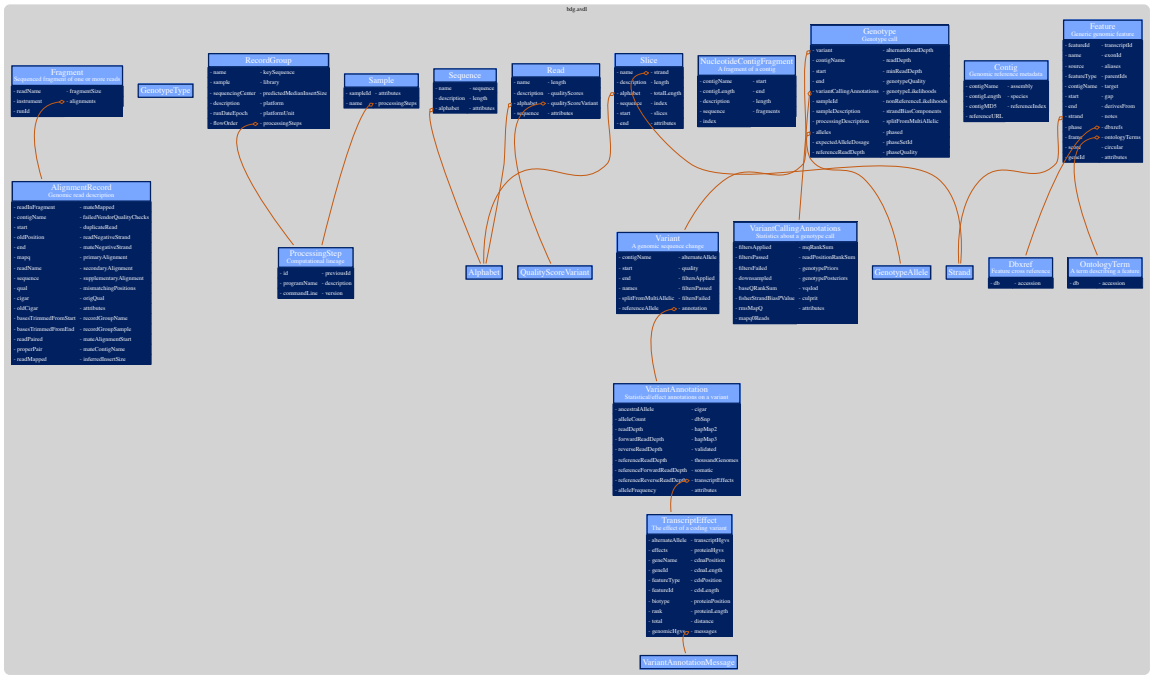

Figure 4.2: A UML diagram showing the dependencies and structure of the ADAM schemas. ADAM's read schema is fairly flat, which contrasts with the other schemas which employ nesting. The variant and feature schemas are deeply nested. This allows both rich annotation and the definition of the complex feature hierarchies necessary for describing genomic annotations [38].

We have reproduced the schemas used to describe reads, features, variants, and genotypes below. Figure 4.2 is a UML diagram depicting how the schemas connect. ADAM also contains schemas for describing assembled contigs, but we have not included them in this section. We discuss the current schemas, as well as their evolution over the course of the ADAM project. While ADAM's read schemas closely represent the SAM specification, the variation and feature representations deviate significantly from the current state of representing genomic variation and features.

Read Schemas

Our read schema maps closely to the logical layout of data presented by SAM and BAM. Unlike the SAM format, we split the flags out from a single field into many fields. This makes it much simpler to extract state from a record. Additionally, we promote several commonly used fields from SAM attributes to fields. These fields include the original quality, position, and CIGAR fields, which are set during realignment (see §6.2) and base quality recalibration (see §6.1).

Listing 4.1: ADAM read schema

```
record AlignmentRecord {

  union { int, null } readInFragment = 0;

  union { null, string } contigName = null;
  union { null, long } start = null;
  union { null, long } oldPosition = null;
  union { null, long } end = null;

  union { null, int } mapq = null;

  union { null, string } readName = null;

  union { null, string } sequence = null;
  union { null, string } qual = null;

  union { null, string } cigar = null;
  union { null, string } oldCigar = null;

  union { int, null } basesTrimmedFromStart = 0;
  union { int, null } basesTrimmedFromEnd = 0;

  union { boolean, null } readPaired = false;
  union { boolean, null } properPair = false;
  union { boolean, null } readMapped = false;
  union { boolean, null } mateMapped = false;
  union { boolean, null } failedVendorQualityChecks = false;
  union { boolean, null } duplicateRead = false;

  union { boolean, null } readNegativeStrand = false;
  union { boolean, null } mateNegativeStrand = false;
  union { boolean, null } primaryAlignment = false;
  union { boolean, null } secondaryAlignment = false;
  union { boolean, null } supplementaryAlignment = false;
```

```
  union { null, string } mismatchingPositions = null;
  union { null, string } origQual = null;

  union { null, string } attributes = null;

  union { null, string } recordGroupName = null;
  union { null, string } recordGroupSample = null;

  union { null, long } mateAlignmentStart = null;
  union { null, string } mateContigName = null;

  union { null, long } inferredInsertSize = null;
}
```

Additionally, we support a schema that groups together all of the reads from a single sequenced fragment. This fragment enables a traversal over read data that is similar to SAM's read-name grouping.

Listing 4.2: ADAM fragment schema

```
record Fragment {

  union { null, string } readName = null;

  union { null, string } instrument = null;
  union { null, string } runId = null;

  union { null, int } fragmentSize = null;

  array<AlignmentRecord> alignments = [];
}
```

We feel that the fragment schema is preferable to the queryname sort order and query grouping invariants that SAM allows one to specify. By using this schema, we can reduce the cost of duplicate marking by 50% (see §8.1). Additionally, this representation provides a useful alternative to the interleaved FASTQ format for streaming reads into an aligner. We demonstrate this usage in Chapter 5.

Variation Schemas

The variant and genotype schemas present a larger departure from the representation used by the Variant Call Format (VCF). The most noticeable difference is that we have migrated away from VCF's variant oriented representation to a matrix representation. Instead of the variant record serving to group together genotypes, the variant record is embedded within

the genotype. Thus, a record represents the genotype assigned to a sample, as opposed to a VCF row, where all individuals are collected together. The second major modification is to assume a biallelic representation. In a biallelic representation, we describe the genotype of a sample at a position or interval as the composition of a reference allele and a single alternate allele. If multiple alternate alleles segregate at the site (e.g., there are two known SNPs in a population at this site), we create multiple biallelic variants for the site. This differs from VCF, which allows multiallelic records. By limiting ourselves to a biallelic representation, we are able to clarify the meaning of many of the variant calling annotations. If a site contains a multiallelic variant (e.g., in VCF parlance this could be a $1/2$ genotype), we split the variant into two or more biallelic records. The sufficient statistics for each allele should then be computed under a reference model similar to the model used in genome VCFs. If the sample does contain a multiallelic variant at the given site, this multiallelic variant is represented by referencing to another record via the OtherAlt enumeration. A similar biallelic model is also used by the Hail project [125].

Listing 4.3: ADAM core variant and genotype schemas

```
record Variant {

  union { null, string } contigName = null;
  union { null, long } start = null;
  union { null, long } end = null;

  array<string> names = [];

  union { boolean, null } splitFromMultiAllelic = false;

  union { null, string } referenceAllele = null;
  union { null, string } alternateAllele = null;

  union { null, double } quality = null;

  union { null, boolean } filtersApplied = null;
  union { null, boolean } filtersPassed = null;
  array<string> filtersFailed = [];

  union { null, VariantAnnotation } annotation = null;
}

enum GenotypeAllele {
  REF,
  ALT,
  OTHER_ALT,
  NO_CALL
```

```
}

record Genotype {

  union { null, Variant } variant = null;
  union { null, string } contigName = null;
  union { null, long } start = null;
  union { null, long } end = null;

  union { null, VariantCallingAnnotations } variantCallingAnnotations = null;

  union { null, string } sampleId = null;
  union { null, string } sampleDescription = null;
  union { null, string } processingDescription = null;

  array<GenotypeAllele> alleles = [];

  union { null, float } expectedAlleleDosage = null;
  union { null, int } referenceReadDepth = null;
  union { null, int } alternateReadDepth = null;
  union { null, int } readDepth = null;
  union { null, int } minReadDepth = null;

  union { null, int } genotypeQuality = null;
  array<double> genotypeLikelihoods = [];
  array<double> nonReferenceLikelihoods = [];

  array<int> strandBiasComponents = [];

  union { boolean, null } splitFromMultiAllelic = false;

  union { boolean, null } phased = false;
  union { null, int } phaseSetId = null;
  union { null, int } phaseQuality = null;
}
```

The variant and genotype schemas are the schemas that have evolved the most over the course of the ADAM project. When we wrote the original ADAM technical report [78], the variant and genotype schemas were much narrower and closely mirrored the VCF specification. Since the VCF format contains many attributes that can be used to annotate both a genotype (via a FORMAT field) or a variant (via an INFO field), the original schemas largely contained overlapping field definitions. We refactored the two schemas to move to the biallelic-only variant model, and later, expanded the variant schema to include a nested structural variant description. This structural variant schema was removed during a large

refactor after the ADAM SIGMOD paper that improved support for variant annotations. We have also played with flattening the variant schema out of the genotype schema for performance reasons. During our work on Mango citetu16, morrow17, we realized that nesting the variant field decreased the performance of range queries by approximately an order of magnitude. In the variant annotation refactor, we added the schemas below.

Listing 4.4: ADAM variant annotation schemas

```
record TranscriptEffect {
  union { null, string } alternateAllele = null;

  array<string> effects = [];

  union { null, string } geneName = null;
  union { null, string } geneId = null;
  union { null, string } featureType = null;
  union { null, string } featureId = null;
  union { null, string } biotype = null;

  union { null, int } rank = null;
  union { null, int } total = null;

  union { null, string } genomicHgvs = null;
  union { null, string } transcriptHgvs = null;
  union { null, string } proteinHgvs = null;

  union { null, int } cdnaPosition = null;
  union { null, int } cdnaLength = null;

  union { null, int } cdsPosition = null;
  union { null, int } cdsLength = null;

  union { null, int } proteinPosition = null;
  union { null, int } proteinLength = null;

  union { null, int } distance = null;

  array<VariantAnnotationMessage> messages = [];
}

record VariantAnnotation {

  union { null, string } ancestralAllele = null;

  union { null, int } alleleCount = null;
```

```
  union { null, int } readDepth = null;
  union { null, int } forwardReadDepth = null;
  union { null, int } reverseReadDepth = null;
  union { null, int } referenceReadDepth = null;
  union { null, int } referenceForwardReadDepth = null;
  union { null, int } referenceReverseReadDepth = null;

  union { null, float } alleleFrequency = null;

  union { null, string } cigar = null;

  union { null, boolean } dbSnp = null;
  union { null, boolean } hapMap2 = null;
  union { null, boolean } hapMap3 = null;
  union { null, boolean } validated = null;
  union { null, boolean } thousandGenomes = null;

  union { boolean, null } somatic = false;

  array<TranscriptEffect> transcriptEffects = [];

  map<string> attributes = {};
}
```

These schemas provide a faithful representation of the VCF/ANN specification, which adds a formal and rich variant annotation specification to VCF [23]. This specification is used by both the Ensembl Variant Effect Predictor (VEP, see [81]) and SnpEff [24]. We have a separate schema for storing annotations on top of a genotype call; we refer to this object as a "variant calling annotation" instead of a "genotype annotation," since the annotations are specific to the variant calling process and the reads observed at the site, as opposed to the called genotype.

Listing 4.5: ADAM genotype annotation schema

```
record VariantCallingAnnotations {

  union { null, boolean } filtersApplied = null;
  union { null, boolean } filtersPassed = null;
  array<string> filtersFailed = [];

  union { null, boolean } downsampled = null;

  union { null, float } baseQRankSum = null;
  union { null, float } fisherStrandBiasPValue = null;
```

```
  union { null, float } rmsMapQ = null;
  union { null, int } mapq0Reads = null;
  union { null, float } mqRankSum = null;
  union { null, float } readPositionRankSum = null;

  array<float> genotypePriors = [];
  array<float> genotypePosteriors = [];

  union { null, float } vqslod = null;
  union { null, string } culprit = null;

  map<string> attributes = {};
}
```

The variant calling annotations object includes annotations that we feel are useful during the variant calling process, but that should probably be omitted from a final callset that is used for annotation and statistical testing. Many of these variant annotations are useful for hard filtering, as described in §7.2.

Feature Schemas

ADAM's feature schema provides an abstracted view of a genomic feature that can support most of the various genomic feature file formats, including NarrowPeak, BED, GTF/GFF2, GFF3, and IntervalList. We have full support for nested features, which are commonly used for describing genome annotations. Instead of nesting the features recursively inside of the feature record, we leverage the database cross reference schema, which is derived from the GTF/GFF2/GFF3 file format.

Listing 4.6: ADAM's feature schemas

```
enum Strand {
  FORWARD,
  REVERSE,
  INDEPENDENT,
  UNKNOWN
}

record Dbxref {
  union { null, string } db = null;
  union { null, string } accession = null;
}

record OntologyTerm {
  union { null, string } db = null;
  union { null, string } accession = null;
```

```
}

record Feature {

  union { null, string } featureId = null;
  union { null, string } name = null;

  union { null, string } source = null;

  union { null, string } featureType = null;

  union { null, string } contigName = null;
  union { null, long } start = null;
  union { null, long } end = null;

  union { null, Strand } strand = null;

  union { null, int } phase = null;
  union { null, int } frame = null;

  union { null, double } score = null;

  union { null, string } geneId = null;
  union { null, string } transcriptId = null;
  union { null, string } exonId = null;
  array<string> aliases = [];
  array<string> parentIds = [];

  union { null, string } target = null;

  union { null, string } gap = null;

  union { null, string } derivesFrom = null;

  array<string> notes = [];

  array<Dbxref> dbxrefs = [];

  array<OntologyTerm> ontologyTerms = [];

  union { null, boolean } circular = null;

  map<string> attributes = {};
}
```

ADAM originally contained an additional GenomeRDD subclass called the GeneRDD. This class would take the nested features contained in a FeatureRDD that described an annotated genome, and would perform all the joins and aggregations necessary to build out the fully nested annotation structure. Ultimately, we removed this class because the performance was poor, and most queries that would run on the nested structure could be refactored to run on the flattened feature hierarchy. This was possible due to the denormalized schema, which included the transcript and gene IDs in each feature record.

Managing Genomic Metadata

A major evolution of the ADAM schemas relates to how we manage and store metadata. In our original schemas, the metdata was denormalized across the record. For example, all of the metadata from the sequencing run and prior processing steps were packed into record group metadata fields, as opposed to being stored in a file header. Our rationale was that this metadata describes the processing lineage of the sample and is expected to have limited cardinality across all records, and thus would compresses extremely well.

This metadata is string heavy, and benefitted strongly from column-oriented decompression on disk. However, proper deserialization from disk proved to be difficult. Although the information consumes less than 5% of space on disk, a poor deserializer implementation may replicate a string per field per record, which greatly increased the amount of memory allocated and the garbage collection (GC) load. With this metadata included in each record, a dataset that was 200GB on disk would balloon into more than 5TB in memory. Additionally, implementing the deserializer conflicted with Apache Spark's serialization architecture. Spark assumes that deserializers have limited state, and write to a stream that does not support seeks. While this is a reasonable assertion for row-oriented serialization techinques, this made it extremely difficult to implement a column-oriented serialization. Prior to eliminating the denormalized metadata from our schemas, this meant that we would have very efficient memory utilization immediately after reading from disk, as Apache Parquet would only allocate a single string per replicated element. However, we would then see the memory explosion after the first shuffle in our query.

This memory explosion led to a major refactor of the ADAM schemas and the introduction of the GenomicRDD hierarchy. Originally, since the metadata was consolidated in each record, the ADAMContext load methods would return RDDs containing our Apache Avro [7] schema objects. The methods that made up the presentation layer (6) of our stack would be added at compile-time using Scala's implicit methods. When we refactored the schemas, we introduced the GenomicRDD classes as a way to track metadata alongside the Apache Spark RDD, moved the presentation layer methods into the implementations of the GenomicRDD classes, and eliminated the implicit conversions. Long term, we believe that the ideal mechanism for storing this metadata is a database designed for storing metadata, such as the Ground store [52]. We discuss the longterm metadata managment needs in §10.4.

Evaluating Compression Techniques

These representations achieve high compression versus the legacy formats. We provide a detailed breakdown of compression in §4.2. ADAM data stored in Parquet achieves an approximately 25% reduction in file size over compressed BAM for read data, and a 66% reduction over GZIPped VCF for variant data.

4.3 Query Patterns for Genomic Data Analysis

There are a wide array of experimental techniques and platforms in genome informatics, but many of these methods produce datapoints that are tied to locations in the genome through the use of genomic coordinates. Each cell contains a copy of the genome with one molecule per chromosome. Each molecule is a collection of DNA polymers coated with (and wrapped around) proteins and packed into the nucleus in a complex 3-dimensional shape. In practice, computational biologists abstract this complexity by storing a single long string that represents the nucleotides of the chromosome. We can then connect a datapoint or observation to the genome by associating the data with the chromosome name and a point or interval on a 1-dimensional space.

A platform for scientific data processing in genomics needs to understand these 1-dimensional coordinate systems because these become the basis on which data processing is parallelized. For example, when calling variants from sequencing data, the sequence data that is localized to a single genomic region (or "locus") can be processed independently from the data localized to a different region, as long as the regions are far enough apart.

Beyond parallelization, many of the core algorithms and methods for data aggregation in genomics are phrased in terms of geometric primitives on 1-D intervals and points where we compute distance, overlap, and containment. An algorithm for calculating quality control metrics may try to calculate "coverage," a count of how many reads overlap each base in the genome. A method for filtering and annotating potential variants might assess the validity of a variant using the quality characteristics of all reads that overlap the putative variant.

To support these algorithms, we provide a "region" or "spatial" join primitive. The algorithm used is described in algorithm 1 and takes as input two sets (RDDs, see Zaharia et al [143]) of ReferenceRegions, a data structure that represents intervals along the 1-D genomics coordinate space. It produces the set of all overlapping ReferenceRegion pairs. The *hulls* variable contains the set of convex hulls and is broadcasted to all compute nodes during the join.

To find the maximal set of non-overlapping regions, we must find the convex hull of all regions emitted. We present a distributed algorithm for finding convex hulls in Appendix 6.2. The distributed convex hull computation problem is important because it is used both for computing regions for partitioning during a region join and for performing INDEL realignment.

Algorithm 1 Partition And Join Regions via Broadcast

$left \leftarrow$ input dataset; left side of join
$right \leftarrow$ input dataset; right side of join
$regions \leftarrow left$.map($data \Rightarrow$generateRegion($data$))
$regions \leftarrow regions$.groupBy($region \Rightarrow region.name$)
$hulls \leftarrow regions$.findConvexHull()
$hulls$.broadcast()
$keyLeft \leftarrow left$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
$keyRight \leftarrow right$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
$joined \leftarrow keyLeft$.join($keyRight$)
$truePositives \leftarrow joined$.filter($r1, r2 \Rightarrow r1$.overlaps($r2$))
return *truePositives*

While the join described above is a broadcast join, a region join can also be implemented via a straightforward shuffle-based approach, which is described in Algorithm 2. The PARTITIONJOINFN function maintains two iterators (one each from both the left and right collections), along with a buffer. This buffer is used to track all key-value pairs from the right collection iterator that *could* match to a future key-value pair from the left collection iterator. We prune this buffer every time that we advance the left collection iterator. For simplicity, the description of Algorithm 2 ignores the complexity of processing keys that cross partition boundaries. In our implementation, we replicate keys that cross partition boundaries into both partitions.

Algorithm 2 Partition And Join Regions via Shuffle

$left \leftarrow$ input dataset; left side of join
$right \leftarrow$ input dataset; right side of join
$partitions \leftarrow left$.getPartitions()
$left \leftarrow left$.repartitionAndSort($partitions$)
$right \leftarrow right$.repartitionAndSort($partitions$)
$joined \leftarrow left$.zipPartitions($right$, PARTITIONJOINFN)
return *joined*

These joins serve as a core that we can use to build other abstractions with. For example, self-region joins and multi-region joins are common in genomics, and can be easily implemented using the above implementations. We are currently working to implement further parallel spatial functions such as sliding windows, using techniques similar to the shuffle-based join. We are working to characterize the performance differences between the two join strategies described above. In the future, we hope to enable the use of the region join in a SQL based system such as SPARK SQL [10].

4.4 Supporting Multi-Language Processing in ADAM

One of the original goals in using Apache Avro was to allow the ADAM schemas to be used across more languages than just Scala and Java, as Avro has language bindings for commonly used languages like C/C++/C#, Python, and JavaScript. However, this wound up not being a fruitful exercise. While Avro supported these languages, Apache Parquet did not, and since we used Avro to manage in-memory serialization but Parquet to manage on-disk serialization, users could not load our Parquet files from disk in non-JVM languages. Additionally, even if we had support for reading Parquet files from these other languages, it would've been prohibitive to port all of the logic in our stack (mostly at the materialized data and presentation layers) over to other languages.

Instead, we followed the approach used by Apache Spark's Python and R bindings [133] after Spark SQL was introduced [10]. Here, we created lightweight wrappers for the ADAMContext and GenomicRDD in the Python and R languages. These wrappers then called the JVM implementations of these classes through Py4j [29] or SparkR's interoperability layer [133]. Instead of exposing the underlying RDD, we exposed the parallel dataset through Spark SQL's DataFrame API. We chose this for several reasons:

- The DataFrame API is a close match to the programming models commonly used in Python and R [10], such as the Pandas [80] and Dyplr [138] libraries.
- A major performance pitfall in PySpark and SparkR was serializer performance. Specifically, to use PySpark, we would have needed to transform our binary Avro records into textual records that PySpark would pickle for use in Python.
- Additionally, the Spark SQL query engine [10] allows Python and R users to write query plans that can then be optimized and executed using optimized JVM implementations where possible. This is a contrast to Spark's RDD API, where both records and queries are black boxes.

While building on Spark SQL enabled cross-language support, it was not a straightforward process. Specifically, Spark SQL leveraged several Scala language features that were incompatible with our Avro schema representation. To address this, we built automation that translates between Spark SQL's desired representation and Avro at compile time.

Part III

Algorithms and Tools

Chapter 5

Automatic Parallelization of Legacy Tools with Cannoli

- Will never be able to eliminate single node tools
  - Most/all current widely used tools are designed for a single node
  - Certain tools (e.g., aligners) pay a steep performance penalty moving out of C/C++
  - Diversity of genomic workflows is too large (>100 sequencing assays)
- Several pipelines rely on manually chunking up tools to run in parallel:
  - GATK Queue
  - SpeedSeq/FreeBayes Parallel
- Several tools have custom Hadoop Streaming/Spark wrappers:
  - BWA through Seal, CS-BWAMEM, BWASpark
  - RNA-Rail
  - CrossBow, Cloudburst
- We should be able to automate this process:
  - Many genomics tools are built around a streaming paradigm
  - If we can provide automated chunking and process setup, then we are good to go
- Two part architecture:
  - pipe API in ADAM: auto-parallelization architecture
  - Tool wrappers in Cannoli

- Is it a truly general approach?
  - No.
  - Tools that need an all-reduce aren't a good match (e.g., Kallisto/Sailfish/Salmon, CNV callers).
  - However, this approach works well for the majority of tools.

5.1 Accomodating Single-node Tools in ADAM With the pipe API

- The PIPE API provides a simple API that autoparallelizes a command across a cluster that supports Apache Spark:
  - User specifies command, files to copy locally, environment settings
  - ADAM infers partitioning from attached sequences
  - Piped formats are specified at compile-time, optionally at runtime
- Implementation:
  - For data aligned to a reference genome, uses fixed size chunking, built on region join infrastructure (§4.3); user can specify flank size
  - Once data is partitioned, we open up a subprocess, and connect to stdin/out of this process.
  - Data is formatted using In/Out formatters:
    * Converts data from ADAM schemas into stream in legacy format, and vice versa
    * ADAM supports a broad range of codecs, including...

5.2 Packaging Parallelized Single-node Tools in Cannoli

- Approach:
  - Delegate to PIPE API as much as possible
  - Transparently support both local and "remote" reference files
  - Support native executables, but default to Docker for convenience of packaging
- Tools supported:

- Aligners:
  - BWA
  - Bowtie2
  - SNAP
- Variant callers:
  - FreeBayes
  - SAMTools mpileup
- Annotation tools:
  - SNPEff
  - BEDTools

Chapter 6

Scalable Alignment Preprocessing with ADAM

In ADAM, we have implemented the three most-commonly used pre-processing stages from the GATK pipeline [34]. In this section, we describe the stages that we have implemented, and the techniques we have used to improve performance and accuracy when running on a distributed system. These pre-processing stages include:

1. **Duplicate Removal:** During the process of preparing DNA for sequencing, reads are duplicated by errors during the sample preparation and polymerase chain reaction stages. Detection of duplicate reads requires matching all reads by their position and orientation after read alignment. Reads with identical position and orientation are assumed to be duplicates. When a group of duplicate reads is found, each read is scored, and all but the highest quality read are marked as duplicates.

   We have validated our duplicate removal code against Picard [127], which is used by the GATK for Marking Duplicates. Our implementation is fully concordant with the Picard/GATK duplicate removal engine, except we are able to perform duplicate marking for chimeric read pairs.[1] Specifically, because Picard's traversal engine is restricted to processing linearly sorted alignments, Picard mishandles these alignments. Since our engine is not constrained by the underlying layout of data on disk, we are able to properly handle chimeric read pairs.

2. **Local Realignment:** In local realignment, we correct areas where variant alleles cause reads to be locally misaligned from the reference genome.[2] In this algorithm, we first identify regions as targets for realignment. In the GATK, this identification is done by traversing sorted read alignments. In our implementation, we fold over partitions where we generate targets, and then we merge the tree of targets. This process allows us to eliminate the data shuffle needed to achieve the sorted ordering. As part of this

[1] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [72].
[2] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al [34].

fold, we must compute the convex hull of overlapping regions in parallel. We discuss this in more detail later in this section.

After we have generated the targets, we associate reads to the overlapping target, if one exists. After associating reads to realignment targets, we run a heuristic realignment algorithm that works by minimizing the quality-score weighted number of bases that mismatch against the reference.

3. **Base Quality Score Recalibration (BQSR):** During the sequencing process, systemic errors occur that lead to the incorrect assignment of base quality scores. In this step, we label each base that we have sequenced with an *error covariate*. For each covariate, we count the total number of bases that we saw, as well as the total number of bases within the covariate that do not match the reference genome. From this data, we apply a correction by estimating the error probability for each set of covariates under a beta-binomial model with uniform prior.

   We have validated the concordance of our BQSR implementation against the GATK. Across both tools, only 5000 of the ∼180B bases ($< 0.0001\%$) in the high-coverage NA12878 genome dataset differ. After investigating this discrepancy, we have determined that this is due to an error in the GATK, where paired-end reads are mishandled if the two reads in the pair overlap.

In the rest of this section, we discuss the high level implementations of these algorithms.

6.1 BQSR Implementation

Base quality score recalibration seeks to identify and correct correlated errors in base quality score estimates. At a high level, this is done by associating sequenced bases with possible error covariates, and estimating the true error rate of this covariate. Once the true error rate of all covariates has been estimated, we then apply the corrected covariate.

Our system is generic and places no limitation on the number or type of covariates that can be applied. A covariate describes a parameter space where variation in the covariate parameter may be correlated with a sequencing error. We provide two common covariates that map to common sequencing errors [88]:

- *CycleCovariate*: This covariate expresses which cycle the base was sequenced in. Read errors are known to occur most frequently at the start or end of reads.
- *DinucCovariate*: This covariate covers biases due to the sequence context surrounding a site. The two-mer ending at the sequenced base is used as the covariate parameter value.

To generate the covariate observation table, we aggregate together the number of observed and error bases per covariate. Algorithms 3 and 4 demonstrate this process.

Algorithm 3 Emit Observed Covariates

```
read ← the read to observe
covariates ← covariates to use for recalibration
sites ← sites of known variation
observations ← ∅
for base ∈ read do
  covariate ← identifyCovariate(base)
  if isUnknownSNP(base, sites) then
    observation ← Observation(1, 1)
  else
    observation ← Observation(1, 0)
  end if
  observations.append((covariate, observation))
end for
return observations
```

Algorithm 4 Create Covariate Table

```
reads ← input dataset
covariates ← covariates to use for recalibration
sites ← known variant sites
sites.broadcast()
observations ← reads.map(read ⇒ emitObservations(read, covariates, sites))
table ← observations.aggregate(CovariateTable(), mergeCovariates)
return table
```

In Algorithm 3, the OBSERVATION class stores the number of bases seen and the number of errors seen. For example, OBSERVATION(1, 1) creates an OBSERVATION object that has seen one base, which was an erroneous base.

Once we have computed the observations that correspond to each covariate, we estimate the observed base quality using equation (6.1). This represents a Bayesian model of the mismatch probability with Binomial likelihood and a Beta(1, 1) prior.

$$\mathbf{E}(P_{err}|cov) = \frac{\#\text{ERRORS}(cov) + 1}{\#\text{OBSERVATIONS}(cov) + 2} \tag{6.1}$$

After these probabilities are estimated, we go back across the input read dataset and reconstruct the quality scores of the read by using the covariate assigned to the read to look into the covariate table.

6.2 Indel Realignment Implementation

Although global alignment will frequently succeed at aligning reads to the proper region of the genome, the local alignment of the read may be incorrect. Specifically, the error models used by aligners may penalize local alignments containing INDELs more than a local alignment that converts the alignment to a series of mismatches. To correct for this, we perform local realignment of the reads against consensus sequences in a three step process. In the first step, we identify candidate sites that have evidence of an insertion or deletion. We then compute the convex hull of these candidate sites, to determine the windows we need to realign over. After these regions are identified, we generate candidate haplotype sequences, and realign reads to minimize the overall quantity of mismatches in the region.

Realignment Target Identification

To identify target regions for realignment, we simply map across all the reads. If a read contains INDEL evidence, we then emit a region corresponding to the region covered by that read.

Convex-Hull Finding

Once we have identified the target realignment regions, we must then find the maximal convex hulls across the set of regions. For a set R of regions, we define a maximal convex hull as the largest region $\hat{r}$ that satisfies the following properties:

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{6.2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{6.3}$$

$$\hat{R} \subset R \tag{6.4}$$

In our problem, we seek to find all of the maximal convex hulls, given a set of regions. For genomics, the convexity constraint described by equation (6.2) is trivial to check: specifically, the genome is assembled out of reference contigs[3] that define disparate 1-D coordinate spaces. If two regions exist on different contigs, they are known not to overlap. If two regions are on a single contig, we simply check to see if they overlap on that contig's 1-D coordinate plane.

Given this realization, we can define Algorithm 5, which is a data parallel algorithm for finding the maximal convex hulls that describe a genomic dataset.

The generateTarget function projects each datapoint into a Red-Black tree that contains a single region. The performance of the fold depends on the efficiency of the merge function. We achieve efficient merges with the tail-call recursive mergeTargetSets function that is described in Algorithm 6.

[3] *Contig* is short for *contiguous sequence*. In alignment based pipelines, reference contigs are used to describe the sequence of each chromosome.

Algorithm 5 Find Convex Hulls in Parallel

```
data ← input dataset
regions ← data.map(data ⇒generateTarget(data))
regions ← regions.sort()
hulls ← regions.fold(r1, r2 ⇒ mergeTargetSets(r1, r2))
return hulls
```

Algorithm 6 Merge Hull Sets

```
first ← first target set to merge
second ← second target set to merge
Require: first and second are sorted
if first = ∅ ∧ second = ∅ then
  return ∅
else if first = ∅ then
  return second
else if second = ∅ then
  return first
else
  if last(first) ∩ head(second) = ∅ then
    return first + second
  else
    mergeItem ← (last(first) ∪ head(second))
    mergeSet ← allButLast(first) ∪mergeItem
    trimSecond ← allButFirst(second)
    return mergeTargetSets(mergeSet, trimSecond)
  end if
end if
```

The set returned by this function is used as an index for mapping reads directly to realignment targets.

Candidate Generation and Realignment

Once we have generated the target set, we map across all the reads and check to see if the read overlaps a realignment target. We then group together all reads that map to a given realignment target; reads that don't map to a target are randomly assigned to a "null" target. We do not attempt realignment for reads mapped to null targets.

To process non-null targets, we must first generate candidate haplotypes to realign against. We support several processes for generating these consensus sequences:

- *Use known INDELs*: Here, we use known variants that were provided by the user to

generate consensus sequences. These are typically derived from a source of common variants such as dbSNP [113].

- *Generate consensuses from reads*: In this process, we take all INDELs that are contained in the alignment of a read in this target region.
- *Generate consensuses using Smith-Waterman*: With this method, we take all reads that were aligned in the region and perform an exact Smith-Waterman alignment [118] against the reference in this site. We then take the INDELs that were observed in these realignments as possible consensuses.

From these consensuses, we generate new haplotypes by inserting the INDEL consensus into the reference sequence of the region. Per haplotype, we then take each read and compute the quality score weighted Hamming edit distance of the read placed at each site in the consensus sequence. We then take the minimum quality score weighted edit versus the consensus sequence and the reference genome. We aggregate these scores together for all reads against this consensus sequence. Given a consensus sequence c, a reference sequence R, and a set of reads $\mathbf{r}$, we calculate this score using equation (6.5).

$$q_{i,j} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j \in \{0, \ldots, l_c - l_{r_i}\} \tag{6.5}$$

$$q_{i,R} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j = \text{pos}(r_i|R) \tag{6.6}$$

$$q_i = \min(q_{i,R}, \min_{j \in \{0,\ldots,l_c - l_{r_i}\}} q_{i,j}) \tag{6.7}$$

$$q_c = \sum_{r_i \in \mathbf{r}} q_i \tag{6.8}$$

In (6.5), $s(i)$ denotes the base at position i of sequence s, and l_s denotes the length of sequence s. We pick the consensus sequence that minimizes the q_c value. If the chosen consensus has a log-odds ratio (LOD) that is greater than 5.0 with respect to the reference, we realign the reads. This is done by recomputing the CIGAR and MDTag for each new alignment. Realigned reads have their mapping quality score increased by 10 in the Phred scale.

6.3 Duplicate Marking Implementation

Reads may be duplicated during sequencing, either due to clonal duplication via PCR before sequencing, or due to optical duplication while on the sequencer. To identify duplicated reads, we apply a heuristic algorithm that looks at read fragments that have a consistent

mapping signature. First, we bucket together reads that are from the same sequenced fragment by grouping reads together on the basis of read name and record group. Per read bucket, we then identify the 5' mapping positions of the primarily aligned reads. We mark as duplicates all read pairs that have the same pair alignment locations, and all unpaired reads that map to the same sites. Only the highest scoring read/read pair is kept, where the score is the sum of all quality scores in the read that are greater than 15.

Chapter 7

Rapid Variant Calling with Avocado

To use Avocado to call variants, we run two applications, each of which has several sub-stages:

1. **INDEL Reassembly:** Here, we clean up all reads that are aligned near INDEL variants. We do this as a two step process:
   a) We make a pass over all reads, using our indexed de Bruijn algorithm to extract INDEL variants. These INDEL variants are collected on a single node, and used as inputs to the next stage.
   b) Optionally, we run ADAM's [78, 94] INDEL realigner, using the discovered INDELs from stage one as "known INDELs" to realign to. This improves variant calling accuracy over solely using the indexed de Bruijn algorithm.
2. **Variant Calling:** In this phase, we discover all SNVs and INDELs, score them using the reads, and emit either called variants or genotype likelihoods in genome VCF (gVCF) format. This runs as a four step process:
   a) We extract all variants from the aligned reads by parsing the alignments.
   b) Using these variants, we compute all read/variant overlaps, and compute the likelihood that each read represents a given variant that it overlaps. In gVCF mode, we also calculate the likelihood of the reference allele at all locations covered by a read.
   c) We merge all of the per-read likelihoods per variant. This gives us final genotype likelihoods per each variant.
   d) Finally, we apply a standard set of hard filters to each variant.

All of these stages are implemented as a parallel application that runs on top of Apache Spark [144, 143], using the ADAM library [78, 94].

7.1 INDEL Reassembly

As opposed to traditional realignment based approaches, we canonicalize INDELs in the reads by looking for bubbles flanked by read vs. reference sequence matches. In a colored de Bruijn graph, a bubble refers to a location where the graph diverges between two samples. In §7.1, we demonstrate how we can use the reconvergence of the de Bruijn graph in the flanking sequence around a bubble to define provably canonical alignments of the bubble between two sequences. For a colored de Bruijn graph containing reads and the reference genome, this allows us to canonically express INDEL variants in the reads against the reference. In §7.1, we then show how this approach can be implemented efficiently without building a de Bruijn graph per read, or even adding each read to a de Bruijn graph. Once we have extracted a canonical set of INDELs, we realign the reads to each INDEL sequence using ADAM's INDEL realigner, in known INDELs mode. For a full description of the INDEL realignment process, see §6.2.

Preliminaries

Our method relies on an *indexed de Bruijn* graph, which is a slight extension of the colored de Bruijn graph [54]. Specifically, each k-mer in an indexed de Bruijn graph knows which sequence position (index) it came from in its underlying read/sequence. To construct an indexed de Bruijn graph, we start with the traditional formulation of a *de Brujin* graph for sequence assembly:

Definition 1 (de Bruijn Graph)**.** *A de Bruijn graph describes the observed transitions between adjacent k-mers in a sequence. Each k-mer s represents a k-length string, with a $k-1$ length prefix given by prefix(s) and a length 1 suffix given by suffix(s). We place a directed edge ($\rightarrow$) from k-mer s_1 to k-mer s_2 if prefix$(s_1)^{\{1,k-2\}}$ + suffix(s_1) = prefix(s_2).*

Now, suppose we have n sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Let us assert that for each k-mer $s \in \mathcal{S}_i$, then the output of function $\text{index}_i(s)$ is defined. This function provides us with the integer position of s in sequence $\mathcal{S}_i$. Further, given two k-mers $s_1, s_2 \in \mathcal{S}_i$, we can define a distance function $\text{distance}_i(s_1, s_2) = |\text{index}_i(s_1) - \text{index}_i(s_2)|$. To create an indexed de Bruijn graph, we simply annotate each k-mer s with the $\text{index}_i(s)$ value for all $\mathcal{S}_i, i \in \{1, \ldots, n\}$ where $s \in \mathcal{S}_i$. This index value is trivial to log when creating the original de Bruijn graph from the provided sequences.

Let us require that all sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are not repetitive, which implies that the resulting de Bruijn graph is acyclic. If we select any two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that share at least two k-mers s_1 and s_2 with common ordering ($s_1 \rightarrow \cdots \rightarrow s_2$ in both $\mathcal{S}_i$ and $\mathcal{S}_j$), the indexed de Bruijn graph G provides several guarantees:

1. If two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ share at least two k-mers s_1 and s_2, we can provably find the maximum edit distance d of the subsequences in $\mathcal{S}_i$ and $\mathcal{S}_j$, and bound the cost of

finding this edit distance at $\mathcal{O}(nd)$,[1]

2. For many of the above subsequence pairs, we can bound the cost at $\mathcal{O}(n)$, *and* provide canonical representations for the necessary edits,

3. $\mathcal{O}(n^2)$ complexity is restricted to aligning the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that exist *before* s_1 or *after* s_2.

Let us focus on cases 1 and 2, where we are looking at the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that are between s_1 and s_2. A trivial case arises when both $\mathcal{S}_i$ and $\mathcal{S}_j$ contain an identical path between s_1 and s_2 (i.e., $s_1 \to s_n \to \cdots \to s_{n+m} \to s_2$ and $s_{n+k} \in \mathcal{S}_i \wedge s_{n+k} \in \mathcal{S}_j \forall k \in \{0, \ldots, m\}$). Here, the subsequences are clearly identical. This determination can be made trivially by walking from vertex s_1 to vertex s_2 with $\mathcal{O}(m)$ cost.

However, three distinct cases can arise whenever $\mathcal{S}_i$ and $\mathcal{S}_j$ diverge between s_1 and s_2. For simplicity, let us assume that both paths are independent (see Definition 2). These three cases correspond to there being either a canonical substitution edit, a canonical INDEL edit, or a non-canonical (but known distance) edit between $\mathcal{S}_i$ and $\mathcal{S}_j$.

Definition 2 (Path Independence)**.** *Given a non-repetitive de Bruijn graph G constructed from $\mathcal{S}_i$ and $\mathcal{S}_j$, we say that G contains independent paths between s_1 and s_2 if we can construct two subsets $\mathcal{S}'_i \subset \mathcal{S}_i, \mathcal{S}'_j \subset \mathcal{S}_j$ of k-mers where $s_{i+n} \in \mathcal{S}'_i \forall n \in \{0, \ldots, m_i\}, s_{i+n-1} \to s_{i+n} \forall n \in \{1, \ldots, m_i\}$, $s_{j+n} \in \mathcal{S}'_j \forall n \in \{0, \ldots, m_j\}, s_{j+n-1} \to s_{j+n} \forall n \in \{1, \ldots, m_j\}$, and $s_1 \to s_i, s_j; s_{i+m_i}, s_{j+m_j} \to s_2$ and $\mathcal{S}'_i \bigcap \mathcal{S}'_j = \emptyset$, where $m_i = \text{distance}_{\mathcal{S}_i}(s_1, s_2)$, and $m_j = \text{distance}_{\mathcal{S}_j}(s_1, s_2)$. This implies that the sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ are different between s_1, s_2,*

We have a canonical substitution edit if $m_i = m_j = k$, where k is the k-mer size. Here, we can prove that the edit between $\mathcal{S}_i$ and $\mathcal{S}_j$ between s_1, s_2 is a single base substitution k letters after index(s_1):

Proof regarding Canonical Substitution. Suppose we have two non-repetitive sequences, $\mathcal{S}'_i$ and $\mathcal{S}'_j$, each of length $2k+1$. Let us construct a de Bruijn graph G, with k-mer length k. If each sequence begins with k-mer s_1 and ends with k-mer s_2, then that implies that the first and last k letters of $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are identical. If both subsequences had the same character at position k, this would imply that both sequences were identical and therefore the two paths between s_1, s_2 would not be independent (Definition 2). If the two letters are different *and* the subsequences are non-repetitive, each character is responsible for k previously unseen k-mers. This is the only possible explanation for the two independent k length paths between s_1 and s_2. □

To visualize the graph corresponding to a substitution, take the two example sequences CCACTGT and CCAATGT. These two sequences differ by a C $\leftrightarrow$ A edit at position three. With k-mer length $k = 3$, this corresponds to the graph in Figure 7.1.

[1]Here, $n = \max(\text{distance}_{\mathcal{S}_i}(s_1, s_2), \text{distance}_{\mathcal{S}_j}(s_1, s_2))$.

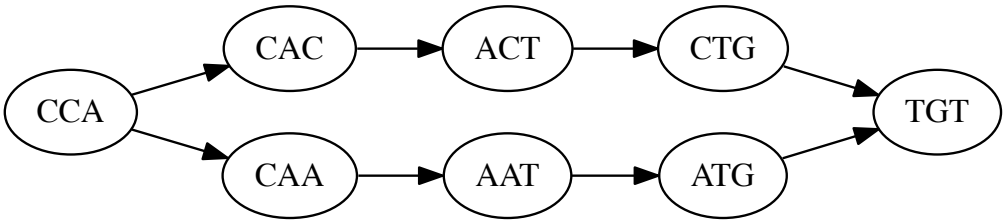

Figure 7.1: Subgraph Corresponding To a Single Nucleotide Edit

If $m_i = k-1, m_j \geq k$ or vice versa, we have a canonical INDEL edit (for convenience, we assume that $\mathcal{S}'_i$ contains the $k-1$ length path). Here, we can prove that there is a $m_j - m_i$ length insertion[2] in $\mathcal{S}'_j$ relative to $\mathcal{S}'_i$, $k-1$ letters *after* index(s_1):

Lemma 1 (Distance between k length subsequences)**.** Indexed de Bruijn *graphs naturally provide a distance metric for k length substrings. Let us construct an indexed de Bruijn graph G with k-mers of length k from a non-repetitive sequence $\mathcal{S}$. For any two k-mers $s_a, s_b \in \mathcal{S}, s_a \neq s_b$, the distance$_{\mathcal{S}}(s_a, s_b)$ metric is equal to $l_p + 1$, where l_p is the length of the path (in k-mers) between s_a and s_b. Thus, k-mers with overlap of $k-1$ have an edge directly between each other ($l_p = 0$) and a distance metric of 1. Conversely, two k-mers that are adjacent but not overlapping in $\mathcal{S}$ have a distance metric of k, which implies $l_p = k-1$.*

Proof regarding Canonical INDELs. We are given a graph G which is constructed from two non-repetitive sequences $\mathcal{S}'_i$ and $\mathcal{S}'_j$, where the only two k-mers in both $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are s_1 and s_2 and both sequences provide independent paths between s_1 and s_2. By Lemma 1, if the path from $s_1 \to \cdots \to s_2 \in \mathcal{S}'_i$ has length $k-1$, then $\mathcal{S}'_i$ is a string of length $2k$ that is formed by concatenating s_1, s_2. Now, let us suppose that the path from $s_1 \to \cdots \to s_2 \in \mathcal{S}'_j$ has length $k+l-1$. The first l k-mers after s_1 will introduce a l length subsequence $\mathcal{L} \subset \mathcal{S}'_j, \mathcal{L} \not\subset \mathcal{S}'_i$, and then the remaining $k-1$ k-mers in the path provide a transition from $\mathcal{L}$ to s_2. Therefore, $\mathcal{S}'_j$ has length of $2k+l$, and is constructed by concatenating $s_1, \mathcal{L}, s_2$. This provides a canonical placement for the inserted sequence $\mathcal{L}$ in $\mathcal{S}'_j$ between s_1 and s_2. □

To visualize the graph corresponding to a canonical INDEL, take the two example sequences CACTGT and CACCATGT. Here, we have a CA insertion after position two. With k-mer length $k = 3$, this corresponds to the graph in Figure 7.2.

Where we have a canonical allele, the cost of computing the edit is set by the need to walk the graph linearly from s_1 to s_2, and is therefore $\mathcal{O}(n)$. However, in practice, we will see differences that cannot be described as one of the earlier two canonical approaches. First, let us generalize from the two above proofs: if we have two independent paths between s_1, s_2 in the de Bruijn graph G that was constructed from $\mathcal{S}_i, \mathcal{S}_j$, we can describe $\mathcal{S}_i$ as a sequence created by concatenating $s_1, \mathcal{L}_i, s_2$.[3] The canonical edits merely result from special cases:

[2]This is equivalently an $m_j - m_i$ length deletion in $\mathcal{S}'_i$ relative to $\mathcal{S}'_j$.

[3]This property holds true for $\mathcal{S}_j$ as well.

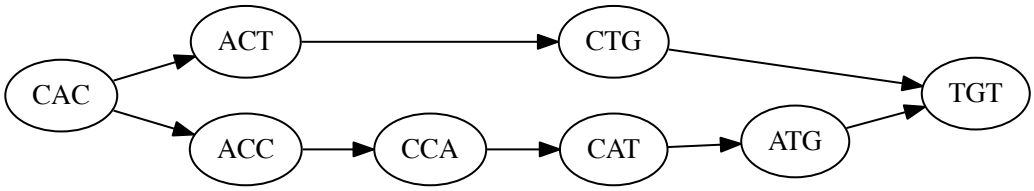

Figure 7.2: Subgraph Corresponding To a Canonical INDEL Edit

- In a canonical substitution edit, $l_{\mathcal{L}_i} = l_{\mathcal{L}_j} = 1$.
- In a canonical INDEL edit, $l_{\mathcal{L}_i} = 0, l_{\mathcal{L}_j} \geq 1$.

Conceptually, a non-canonical edit occurs when two edits occur within k positions of each other. In this case, we can trivially fall back on a $O(nm)$ local alignment algorithm (e.g., a pairwise HMM or Smith-Waterman, see Durbin et al [36] or Smith and Waterman [118]), *but* we only need to locally realign $\mathcal{L}_i$ against $\mathcal{L}_j$, which reduces the size of the realignment problem. However, we can further limit this bound by limiting the maximum number of INDEL edits to $d = |l_{\mathcal{L}_i} - l_{\mathcal{L}_j}|$. This allows us to use an alignment algorithm that limits the number of INDEL edits (e.g., Ukkonen's algorithm [130]). By this, we can achieve $O(n(d+1))$ cost. Alternatively, we can decide to not further canonicalize the site, and to express it as a combined insertion and deletion. For simplicity and performance, we use this approach in Avocado.

Implementation

As alluded to earlier in this section, we can use this indexed de Bruijn concept to canonicalize INDEL variants without needing to first build a de Bruijn graph. The insight behind this observation is simple: any section of a read alignment that is an exact sequence match with length greater than our k-mer length maps to a section of the indexed de Bruijn graph where the read and reference paths have converged. As such, we can use these segments that are perfect sequence matches to anchor the bubbles containing variants (areas where the read and reference paths through the graph diverge) without first building a graph. We can perform this process simply by parsing the CIGAR string (and MD tags) for each read [73]. We do this by:

- Iterating over each operator in the CIGAR string. We coalesce the operators into a structure that we call an "alignment block":
  - If the operator is a sequence match (CIGAR =, or CIGAR M with MD tag indicating an exact sequence match) that is longer than our k-mer length, we can create an alignment block that indicates a convergence in the indexed de Bruijn block (a sequence match block).

  - If the sequence match operator is adjacent to an operator that indicates that the read diverges from the reference (insertion, deletion, or sequence mismatch), we then take k bases from the start/end of the matching sequence and append/prepend the k bases to the divergent sequence. We then create an alignment block that indicates that the read and reference diverge, along with the two diverging sequences, flanked by k bases of matching sequence on each side. We call these blocks realignment blocks.

- We then loop over each alignment block. Since the sequence match blocks are exact sequence matches, they do not need any further processing and can be directly emitted as a CIGAR = operator. If the block is a realignment block, we then apply the observations from §7.1. Again, we can apply our approaches without building de Bruijn graphs for the bubble. Specifically, both of the canonical placement rules that we formulate in §7.1 indicate that the variant in a bubble can be recovered by trimming any matching flanking sequence. We begin by trimming the matching sequences from the reference and read, starting from the right, followed by the left. We then emit a CIGAR insertion, deletion, or sequence mismatch (X) operator for this block, along with a match operator if either side of the flanking sequence was longer than k.

This process is very efficient, as it can be done wholly with standard string operators in a single loop over the read. To avoid the cost of looking up the reference sequence from a reference genome, we require that all reads are tagged with the SAM MD tag. This allows us to reconstruct the reference sequence for a bubble from the read sequence and CIGAR.

One problem with this method is that it can be misled by sequencing errors that are proximal to a true variant. As can be seen in §**??**, solely using our indexed de Bruijn algorithm to clean up INDEL alignments leads to lower accuracy than the state-of-the-art toolkit. However, if the INDEL variant in a read that is discovered is a true variant, it is a good candidate to be used as an input to a local realignment scheme. To implement this approach, we used our indexed de Bruijn algorithm to canonicalize INDEL variants, and then we used our variant discovery algorithm (see §7.2) with filtration disabled to collect all canonical INDELs. We then fed these INDELs and our input reads into ADAM's INDEL realignment engine [78, 94]. This tool is based on the algorithms used in the GATK's INDEL realigner [34], and calculates the quality-score weighted Hamming edit distance between a set of reads, a consensus sequence (a haplotype containing a potential INDEL variant), and the reference sequence. If the sum weighted edit distance between the reads and the consensus sequence represents a suffient improvement over the sum weighted edit distance between the reads and the reference genome, the read alignments are moved to their lowest weighted edit distance position relative to the consensus sequence. A detailed description of this algorithm can be found in §6.2. As seen in §**??**, coupling local realignment with our INDEL canonicalization scheme improves SNP calling accuracy to comparable with the state-of-the-art, while improving INDEL calling accuracy by 2–5%.

7.2 Genotyping

Avocado performs genotyping as a several stage process where variants are discovered from the input reads and filtered, joined back against the input reads, and then scored. We use a biallelic likelihood model to score variants [69], and run all stages in parallel. Our approach does not rely on the input reads being sorted, and as such, is not unduly impacted by variations in coverage across the genome. This point is critical in a parallel approach, as coverage can vary dramatically across the genome [102]. If the input reads must be sorted, this can lead to large work imbalances between nodes in a distributed system, which negatively impacts strong scaling. An alternative approach is to use previously known data about genome coverage to statically partition tasks into balanced chunks [21]. Unlike the static partitioning approach used by SpeedSeq that discards regions with very high coverage, this allows us to call variants in regions with very high coverage. However, as is also noted in the SpeedSeq paper, variant calls in these regions are likely to be caused by artifacts in the reference genome that confound mapping and thus are uninformative or spurious, and are hard filtered by our pipeline (see §7.2).

Variant Discovery and Overlapping

To identify a set of variants to score, we scan over all of the input reads, and generate a set of variants per read where each variant is tagged with the mean quality score of all bases in the read that were in this variant. We then use Apache Spark's reduceByKey functionality to compute the number of times each variant was observed with high quality. We do this to discard sequence variants that were observed in a read that represent a sequencing error, and not a true variant. In our evaluation, we set the quality needed to consider a variant observation as high quality to Phred 18 (equivalent to a error probability of less than 0.016), and we require that a variant is seen in at least 3 reads.

To score the discovered variants, we use an "overlap join" primitive to find all of the variants that a single read overlaps. An overlap join is a relational join where the row equality function is defined as whether two objects overlap in the genomic coordinate space [94]. This primitive can be implemented in a distributed system as both a broadcast join (the smaller of the two datasets is sent to every node in the cluster), or as a sort-merge join, where the dataset is sorted. Our implementation uses a broadcast strategy, as the set of variants to score is typically small and this approach eliminates the work imbalance problem introduced earlier.

Our broadcast overlap join implementation starts by sorting the candidate variants by genomic locus. We collect the variants to the leader node, and then broadcast a sorted array of variants to each node in the cluster. To find all of the variants that overlap a single read, we run a binary search across the sorted array of variants. We prefer this strategy to building an indexed datastructure (such as an interval tree, see Kozanitis and Patterson [61]) because sorting can be efficiently parallelized across the Apache Spark cluster, while building an indexed structure would typically need to be done sequentially on a single node. Additionally,

a flat array of sorted variants is simpler to serialize and broadcast across the cluster than an indexed structure. When we query into the sorted array using binary search, the binary search algorithm will give us a variant that is overlapped by the read. Since we actually want to run a combined join-and-group query, we then search outwards from this first hit to identify all of the variants that overlap the read alignment.

One of the reasons that we filter out variant sites that are not supported by many high quality reads is an engineering limitation currently in Avocado. As we decrease the stringency of the filters and allow more variants to be detected, we increase the amount of variants that we need to broadcast between nodes. This causes the size of data that we must serialize to grow beyond the size of the maximum individual item that we can serialize (limited to 2GB due to the Java Virtual Machine, which is used by Apache Spark). We are working to eliminate this limitation. There are several possible strategies. A simple strategy would be to reduce the amount of data written to the serialization buffer by compressing the data before streaming it into the serialization buffer. However, our sorted array currently stores the genomic coordinate of a variant separately from the variant itself, which causes a minor amount of data duplication in memory. By eliminating this data duplication, we should be able to eliminate this engineering constraint.

Genotyping Model

Once we have joined our reads against our variants, we score each read using the biallelic genotyping model proposed by Li [69]. For each variant, we check to see if the variant allele is present in the read at the appropriate position in the alignment. If the variant is present, we treat the read as positive evidence supporting the variant. If the read contains the reference allele at that site, we treat the read as evidence supporting the reference. If the read neither matches the variant allele nor the reference, we do not use the read to calculate the genotype likelihoods, but we do use the read to compute statistics (e.g., for calculating depth, strand bias, etc.) about the genotyped site. We calculate the genotype likelihood for the genotype in log space, using Equation (7.1). Equation (7.1) is not our contribution and is reproduced from Li [69], but in log space.

$$\log \mathcal{L}(g) = -mk \sum_{i=0} j l_r(g, m-g, \epsilon_i) \sum_{i=j+1}^{k} l_r(m-g, g, \epsilon_i) \tag{7.1}$$

$$l_r(c_r, c_a, \epsilon) = \text{logsum}(\log c_r + \log \epsilon, \log c_a + \text{logm1}(\log \epsilon)) \tag{7.2}$$

In Equation (7.1), g is the genotype state (number of reference alleles), m is the copy number at the site, k is the total number of reads, j is the number of reads that match the reference genome, and ϵ is the error probability of a single read base, as given by the harmonic mean of the read's base quality, and the read's mapping quality, if present. The logsum function adds two numbers that are in log space, while logm1 computes the additive inverse of a number in log space. These functions can be implemented efficiently while

preserving numerical stability [36]. By doing this whole calculation in log space, we can eliminate issues caused by floating-point underflow. Additionally, since ϵ is derived from Phred scaled quantities and is thus already in log space (base ten), while g and $m - g$ are constants that can be pre-converted to log space. For all sites, we also compute a reference model that can be used in joint genotyping in a gVCF approach. Additionally, we support a gVCF mode where all sites are scored, even if they are not covered by a putative variant.

We compute the likelihoods for each read in parallel. This function maps over all of the reads, and emits a set of records describing each observation. In addition to storing the likelihood vector per read/variant pair, this record contains data necessary to compute several genotype annotations that are used for variant filtration (such as strand bias observations, mapping quality, etc., see §7.2). We use Apache Spark's reduceByKey function to merge all of the observations for a given locus. Once we have merged all of the observations for a given site, we call the genotype state by taking the genotype state with the highest likelihood. In single sample mode, we assume no prior probability. We support a joint variant calling mode that computes reference allele frequency for use in a binomial prior probability distribution.

Variant Filtration

Once we have called variants, we pass the calls through a hard filtering engine. First, unless we are in gVCF mode, we discard all homozygous reference calls and low quality genotype calls (default threshold is Phred 30). Additionally, we provide several hard filters that retain the genotype call, but mark the call as filtered. These include:

1. Quality by depth: the Phred scaled genotype quality divided by the depth at the site. Default value is 2.0 for heterozygous variants, 1.0 for homozygous variants. The value can be set separately for INDELs and SNPs.

2. Root-mean-square mapping quality: Default value is 30.0 for SNPs. By default, this filter is disabled for INDELs.

3. Depth: We filter out genotype calls below a minimum depth, or above a maximum depth. By default, the minimum depth is 10, and maximum depth is 200. This value can be set separately for INDELs and SNPs.

Currently, we do not support filtering variant sites in joint genotyping mode. However, we will add this functionality soon.

Part IV

Evaluation

Chapter 8

Benchmarking the ADAM Stack

8.1 Benchmarking Duplicate Markers

Chapter 9

The Simons Genome Diversity Dataset Recompute

Part V

Conclusion and Future Work

Chapter 10

Future Work

- What were our overarching the goals?
  - Make genomic analysis easier to scale
  - Improve variant calling accuracy
  - Make genomic data analysis faster
- Several future directions:
  - Further improving query performance in ADAM
  - Extending variant calling algorithms in Avocado
  - Hardware acceleration for genomics

10.1 Further Query Optimization in ADAM

- Preserving sorted knowledge in ADAM
  - Needed to improve performance of region joins
  - Currently implemented, but refactoring to extend to legacy genomic datatypes
  - Store sorted state as metadata in the GenomicRDD.
- Extending what transformations can be optimized by partitioning:
  - Currently just accelerate joins
  - Aggregation queries naturally fall out of this as well
- Exposing other primitives as well:
  - Convex hull merge

10.2 Extensions to Avocado

- Deep learning for variant calling:
  - Current model for INDEL discovery is intolerant of read errors
  - Doesn't model error distribution over technologies
  - Google DeepVariant uses deep learning but in an unglamorous way
  - Would the Avocado INDEL canonicalization algorithm be a better fit?
- Somatic variant calling:
  - Avocado currently only models a sample with known ploidy
  - Can we accomodiate pooled or somatic variant calling?
  - WIP towards porting over MuTect

10.3 Hardware Acceleration for Genomic Data Processing

- With improved I/O, single thread performance remains an issue
  - As seen with Cannoli, hard to outperform C++ from Java
  - However, marginal benefit to jumping to C++ is unclear
  - Can we go directly to hardware?
- WIP on Genie library
- How can we make this even more accessible?
  - SQL as target language is a great opportunity
  - Integrate Genie with Apache Arrow
  - Allow generation of FPGA from Spark SQL plan
- How can we easily accomodate UDFs?

10.4 Improved Metadata Management for Genomics

10.5 Efficient Consensus Methods

10.6 Improving Debugging Capabilities for Distributed Systems

Chapter 11

Conclusion

Bibliography

[1] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

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

[3] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo. BigBWA: Approaching the Burrows–Wheeler aligner to big data technologies. *Bioinformatics*, 31(24):4003–4005, 2015.

[4] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo. SparkBWA: Speeding up the alignment of high-throughput DNA sequencing data. *PloS one*, 11(5):e0155461, 2016.

[5] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars. Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm. In *Proceedings of the International Conference on Computer-Aided Deign (ICCAD)*, 2015.

[6] C. A. Albers, G. Lunter, D. G. MacArthur, G. McVean, W. H. Ouwehand, and R. Durbin. Dindel: Accurate indel calls from short-read data. *Genome research*, 21(6):961–973, 2011.

[7] Apache. Avro. `http://avro.apache.org`.

[8] Apache. Hadoop. `http://hadoop.apache.org`.

[9] Apache. Parquet. `http://parquet.apache.org`.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015.

[11] V. Bafna, A. Deutsch, A. Heiberg, C. Kozanitis, L. Ohno-Machado, and G. Varghese. Abstractions for genomics. *Communications of the ACM*, 56(1):83–93, 2013.

[12] R. Bao, L. Huang, J. Andrade, W. Tan, W. A. Kibbe, H. Jiang, and G. Feng. Review of current methods, applications, and data management for the bioinformatics analysis of whole exome sequencing. *Cancer informatics*, 13(Suppl 2):67, 2014.

[13] A. Bloniarz, A. Talwalkar, J. Terhorst, M. I. Jordan, D. Patterson, B. Yu, and Y. S. Song. Changepoint analysis for efficient variant calling. In *Research in Computational Molecular Biology (RECOMB '14)*, pages 20–34. Springer, 2014.

[14] E. A. Boyle, Y. I. Li, and J. K. Pritchard. An expanded view of complex traits: From polygenic to omnigenic. *Cell*, 169(7):1177–1186, 2017.

[15] J. Brandt, M. Bux, and U. Leser. Cuneiform: A functional language for large scale scientific data analysis. In *Proceedings of the Workshops of the EDBT/ICDT*, 2015.

[16] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[18] D. J. Carey, S. N. Fetterolf, F. D. Davis, W. A. Faucett, H. L. Kirchner, U. Mirshahi, M. F. Murray, D. T. Smelser, G. S. Gerhard, and D. H. Ledbetter. The Geisinger MyCode community health initiative: an electronic health record-linked biobank for precision medicine research. *Genetics in medicine: official journal of the American College of Medical Genetics*, 18(9):906, 2016.

[19] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei. When Apache Spark meets FPGAs: A case study for next-generation DNA sequencing acceleration. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2016.

[20] Y.-T. Chen, J. Cong, J. Lei, and P. Wei. A Novel High-Throughput Acceleration Engine for Read Alignment. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCMM)*, May 2015.

[21] C. Chiang, R. M. Layer, G. G. Faust, M. R. Lindberg, D. B. Rose, E. P. Garrison, G. T. Marth, A. R. Quinlan, and I. M. Hall. SpeedSeq: Ultra-fast personal genome analysis and interpretation. *Nature methods*, 12(10):966–968, 2015.

[22] D. M. Church, V. A. Schneider, K. M. Steinberg, M. C. Schatz, A. R. Quinlan, C.-S. Chin, P. A. Kitts, B. Aken, G. T. Marth, M. M. Hoffman, et al. Extending reference assembly models. *Genome biology*, 16(1):13, 2015.

[23] P. Cingolani, F. Cunningham, W. McLaren, and K. Wang. Variant annotations in VCF format. `http://snpeff.sourceforge.net/VCFannotationformat_v1.0.pdf`.

[24] P. Cingolani, A. Platts, L. L. Wang, M. Coon, T. Nguyen, L. Wang, S. J. Land, X. Lu, and D. M. Ruden. SnpEff: A program for annotating and predicting the effects of single nucleotide polymorphisms. *Fly*, 6(2):80–92, 2012.

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

[26] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2009.

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

[28] Common Workflow Language. Common workflow language specifications. `http://www.commonwl.org/v1.0/`.

[29] B. Dagenais. Py4j: A bridge between Python and Java. `https://www.py4j.org/`.

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

[31] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI '04)*. ACM, 2004.

[32] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[33] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier. Halvade: Scalable sequence analysis with MapReduce. *Bioinformatics*, 31(15):2482–2488, 2015.

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

[35] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[36] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ Press, 1998.

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

[38] K. Eilbeck, S. E. Lewis, C. J. Mungall, M. Yandell, L. Stein, R. Durbin, and M. Ashburner. The Sequence Ontology: A tool for the unification of genome annotations. *Genome biology*, 6(5):R44, 2005.

[39] B. Ewing, L. Hillier, M. C. Wendl, and P. Green. Base-calling of automated sequencer traces using Phred. *Genome research*, 8(3):175–185, 1998.

[40] H. Fang, Y. Wu, G. Narzisi, J. A. O'Rawe, L. T. J. Barrón, J. Rosenbaum, M. Ronemus, I. Iossifov, M. C. Schatz, and G. J. Lyon. Reducing INDEL calling errors in whole genome and exome sequencing data. *Genome Med*, 6:89, 2014.

[41] G. G. Faust and I. M. Hall. SAMBLASTER: fast duplicate marking and structural variant read extraction. *Bioinformatics*, page btu314, 2014.

[42] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000.

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

[44] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012.

[45] E. Genome. Dragen bio-it processor. `http://www.edicogenome.com/dragen_bioit_platform/`. Accessed 3/19/2017.

[46] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick. HipMer: An extreme-scale de novo genome assembler. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–11. IEEE, 2015.

[47] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de Bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 437–448. IEEE Press, 2014.

[48] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. merAligner: A fully parallel sequence aligner. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 561–570. IEEE, 2015.

[49] E. Georganas, S. Hofmeyr, R. Egan, A. Buluc, L. Oliker, D. Rokhsar, and K. Yelick. Extreme-scale de novo genome assembly. *arXiv preprint arXiv:1705.11147*, 2017.

[50] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[51] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[52] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, et al. Ground: A data context service. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '17)*, 2017.

[53] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[54] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

[55] M. Jain, S. Koren, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, S. Malla, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *bioRxiv*, page 128835, 2017.

[56] G. Kaushik, S. Ivkovic, J. Simonovic, N. Tijanic, B. Davis-Dusenbery, and D. Kural. Graph theory approaches for optimizing biomedical data analysis using reproducible workflows. *bioRxiv*, page 074708, 2016.

[57] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome research*, 12(6):996–1006, 2002.

[58] C. Kingsford, M. C. Schatz, and M. Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC bioinformatics*, 11(1):21, 2010.

[59] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russel, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '15)*, 2015.

[60] C. Kozanitis, A. Heiberg, G. Varghese, and V. Bafna. Using Genome Query Language to uncover genetic variation. *Bioinformatics*, 30(1):1–8, 2014.

[61] C. Kozanitis and D. A. Patterson. GenAp: A distributed SQL interface for genomic data. *BMC bioinformatics*, 17(1):63, 2016.

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

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

[64] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134, 2009.

[65] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome biology*, 10(11):R134, 2009.

[66] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

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

[68] S. Leo and G. Zanetti. Pydoop: A Python MapReduce and HDFS API for Hadoop. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 819–825. ACM, 2010.

[69] H. Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27(21):2987–2993, 2011.

[70] H. Li. Tabix: fast retrieval of sequence features from generic tab-delimited files. *Bioinformatics*, 27(5):718–719, 2011.

[71] H. Li. Towards better understanding of artifacts in variant calling from high-coverage samples. *arXiv preprint arXiv:1404.0929*, 2014.

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

[73] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[74] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, K. Kristiansen, and J. Wang. SNP detection for massively parallel whole-genome resequencing. *Genome research*, 19(6):1124–1132, 2009.

[75] M. D. Linderman, D. Chia, F. Wallace, and F. A. Nothaft. DECA: Scalable XHMM exome copy-number variant calling with ADAM and Apache Spark. *bioRxiv*, 2017.

[76] S. Mallick, H. Li, M. Lipson, I. Mathieson, M. Gymrek, F. Racimo, M. Zhao, N. Chennagiri, S. Nordenfelt, A. Tandon, et al. The Simons Genome Diversity Project: 300 genomes from 142 diverse populations. *Nature*, 538(7624):201–206, 2016.

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

[78] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. ADAM: Genomics formats and processing patterns for cloud scale computing. Technical report, UCB/EECS-2013-207, EECS Department, University of California, Berkeley, 2013.

[79] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[80] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. SciPy Austin, TX, 2010.

[81] W. McLaren, B. Pritchard, D. Rios, Y. Chen, P. Flicek, and F. Cunningham. Deriving the consequences of genomic variants with the ensembl api and snp effect predictor. *Bioinformatics*, 26(16):2069–2070, 2010.

[82] Microsoft. Microsoft computing method makes key aspect of genomic sequencing seven times faster. `https://blogs.microsoft.com/next/2016/10/18/microsoft-computing-method-makes-key-aspect-genomic-sequencing-seven-times-faster/` Accessed 3/19/2017.

[83] E. V. Minikel, S. M. Vallabh, M. Lek, K. Estrada, K. E. Samocha, J. F. Sathirapongsasuti, C. Y. McLean, J. Y. Tung, P. Linda, P. Gambetti, et al. Quantifying prion disease penetrance using large population control cohorts. *Science translational medicine*, 8(322):322ra9–322ra9, 2016.

[84] A. Morrow. Distributed visualization for genomic analysis. Master's thesis, EECS Department, University of California, Berkeley, May 2017.

[85] A. Morrow, V. Shankar, D. Petersohn, A. Joseph, B. Recht, and N. Yosef. Convolutional kitchen sinks for transcription factor binding site prediction. *arXiv preprint arXiv:1706.00125*, 2017.

[86] A. Mortazavi, B. A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature methods*, 5(7):621–628, 2008.

[87] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hofstee, and Z. Al-Ars. SparkGA: A Spark framework for cost effective, fast and accurate DNA analysis at scale. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 148–157. ACM, 2017.

[88] K. Nakamura, T. Oshima, T. Morimoto, S. Ikeda, H. Yoshikawa, Y. Shiwa, S. Ishikawa, M. C. Linak, A. Hirai, H. Takahashi, et al. Sequence-specific error profile of Illumina sequencers. *Nucleic acids research*, page gkr344, 2011.

[89] G. Narzisi, J. A. O'Rawe, I. Iossifov, H. Fang, Y.-h. Lee, Z. Wang, Y. Wu, G. J. Lyon, M. Wigler, and M. C. Schatz. Accurate de novo and transmitted indel detection in exome-capture data using microassembly. *Nature methods*, 11(10):1033–1036, 2014.

[90] C. G. A. R. Network et al. Genomic and epigenomic landscapes of adult de novo acute myeloid leukemia. *N Engl j Med*, 2013(368):2059–2074, 2013.

[91] NHGRI. DNA sequencing costs. `http://www.genome.gov/sequencingcosts/`.

[92] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, 2012.

[93] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013.

[94] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the International Conference on Management of Data (SIGMOD '15)*. ACM, 2015.

[95] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[96] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[97] OpenCB. Hpg bigdata. `https://github.com/opencb/hpg-bigdata`.

[98] A. R. OBrien, N. F. Saunders, Y. Guo, F. A. Buske, R. J. Scott, and D. C. Bauer. VariantSpark: Population scale clustering of genotype information. *BMC genomics*, 16(1):1052, 2015.

[99] B. Paten, M. Diekhans, B. J. Druker, S. Friend, J. Guinney, N. Gassner, M. Guttman, W. J. Kent, P. Mantey, A. A. Margolin, et al. The NIH BD2K center for Big Data in Translational Genomics. *Journal of the American Medical Informatics Association (JAMIA)*, 22(6):1143–1147, 2015.

[100] R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*, 14(4):417–419, 2017.

[101] W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in enzymology*, 183:63–98, 1990.

[102] R. Pinard, A. de Winter, G. J. Sarkis, M. B. Gerstein, K. R. Tartaro, R. N. Plant, M. Egholm, J. M. Rothberg, and J. H. Leamon. Assessment of whole genome amplification-induced bias through high-throughput, massively parallel whole genome sequencing. *BMC Genomics*, 7(1):216, 2006.

[103] L. Pireddu, S. Leo, and G. Zanetti. SEAL: A distributed short read mapping and duplicate removal tool. *Bioinformatics*, 27(15):2159–2160, 2011.

[104] S. Purcell, B. Neale, K. Todd-Brown, L. Thomas, M. A. Ferreira, D. Bender, J. Maller, P. Sklar, P. I. De Bakker, M. J. Daly, et al. PLINK: A tool set for whole-genome association and population-based linkage analyses. *The American Journal of Human Genetics*, 81(3):559–575, 2007.

[105] A. R. Quinlan and I. M. Hall. BEDTools: A flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2010.

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

[107] S. Ripke, B. M. Neale, A. Corvin, J. T. Walters, K.-H. Farh, P. A. Holmans, P. Lee, B. Bulik-Sullivan, D. A. Collier, H. Huang, et al. Biological insights from 108 schizophrenia-associated genetic loci. *Nature*, 511(7510):421, 2014.

[108] N. Rodríguez-Ezpeleta, M. Hackenberg, and A. M. Aransay. *Bioinformatics for high throughput sequencing*. Springer Science & Business Media, 2011.

[109] D. M. Ruderfer, T. Hamamsy, M. Lek, K. J. Karczewski, D. Kavanagh, K. E. Samocha, M. J. Daly, D. G. MacArthur, M. Fromer, S. M. Purcell, et al. Patterns of genic intolerance of rare copy number variation in 59,898 human exomes. *Nature genetics*, 48(10):ng–3638, 2016.

[110] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics*, 11(9):647–657, 2010.

[111] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[112] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[113] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin. dbSNP: the NCBI database of genetic variation. *Nucleic acids research*, 29(1):308–311, 2001.

[114] H. Shi, G. Kichaev, and B. Pasaniuc. Contrasting the genetic architecture of 30 complex traits from summary association data. *The American Journal of Human Genetics*, 99(1):139–153, 2016.

[115] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST '10)*, pages 1–10. IEEE, 2010.

[116] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. ABySS: A parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[117] A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC bioinformatics*, 9(1):128, 2008.

[118] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[119] W. W. Soon, M. Hariharan, and M. P. Snyder. High-throughput sequencing for biology and medicine. *Molecular systems biology*, 9(1):640, 2013.

[120] L. D. Stein et al. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[121] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[122] A. Talwalkar, J. Liptrap, J. Newcomb, C. Hartl, J. Terhorst, K. Curtis, M. Bresler, Y. S. Song, M. I. Jordan, and D. Patterson. SMASH: A benchmarking toolkit for human genome variant calling. *Bioinformatics*, page btu345, 2014.

[123] A. Tarasov, A. J. Vilella, E. Cuppen, I. J. Nijman, and P. Prins. Sambamba: fast processing of NGS alignment formats. *Bioinformatics*, 2015.

[124] The Broad Institue of Harvard and MIT. Cromwell. `https://github.com/broadinstitute/cromwell`.

[125] The Broad Institute of Harvard and MIT. Hail: Scalable genomic data analysis. `https://github.com/hail-is/hail`.

[126] The Broad Institute of Harvard and MIT. Official code repository for gatk versions 4 and up. `https://github.com/broadinstitute/gatk`.

[127] The Broad Institute of Harvard and MIT. Picard. `http://broadinstitute.github.io/picard/`, 2014.

[128] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[129] E. Tu. Interactive exploration on large genomic datasets. Master's thesis, EECS Department, University of California, Berkeley, May 2016.

[130] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[131] United Kingdom Biobank. GSK/Regeneron initiative to develop better treatments, more quickly. `http://www.ukbiobank.ac.uk/2017/03/gsk-regeneron-initiative-to-develop-better-treatments-more-quickly/`, March 2017.

[132] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the Symposium on Cloud Computing (SoCC '13)*, page 5. ACM, 2013.

[133] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, et al. Sparkr: Scaling R programs with Spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1099–1104. ACM, 2016.

[134] J. Vivian, A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314, 2017.

[135] R. Walsh, K. L. Thomson, J. S. Ware, B. H. Funke, J. Woodley, K. J. McGuire, F. Mazzarotto, E. Blair, A. Seller, J. C. Taylor, et al. Reassessment of mendelian gene pathogenicity using 7,855 cardiomyopathy cases and 60,706 reference samples. *Genetics in Medicine*, 19(2):192–203, 2016.

[136] S. Weingarten-Gabbay and E. Segal. The grammar of transcriptional regulation. *Human genetics*, 133(6):701–711, 2014.

[137] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, Cancer Genome Atlas Research Network, et al. The Cancer Genome Atlas pan-cancer analysis project. *Nature Genetics*, 45(10):1113–1120, 2013.

[138] H. Wickham and R. Francois. dplyr: A grammar of data manipulation. *R package version 0.4*, 1:20, 2015.

[139] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652–2653, 2014.

[140] R. Williams. Spark-bam. `https://github.com/hammerlab/spark-bam`.

[141] A. Yang, M. Troup, P. Lin, and J. W. Ho. Falco: A quick and flexible single-cell RNA-seq processing framework on the cloud. *Bioinformatics*, 33(5):767–769, 2016.

[142] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with SNAP. *arXiv preprint arXiv:1111.5572*, 2011.

[143] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association, 2012.

[144] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud '10)*, page 10, 2010.

[145] Y. Zhang, T. Liu, C. A. Meyer, J. Eeckhoute, D. S. Johnson, B. E. Bernstein, C. Nusbaum, R. M. Myers, M. Brown, W. Li, et al. Model-based analysis of ChIP-Seq. *Genome biology*, 9(9):R137, 2008.

[146] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 918–927. IEEE, 2015.

[147] Z. Zhang, K. Barbary, F. A. Nothaft, E. R. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Kira: Processing astronomy imagery using big data technology. *IEEE Transactions on Big Data*, 2016.

[148] H. Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[149] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit. Integrating human sequence data sets provides a resource of benchmark SNP and INDEL genotype calls. *Nature*, 201:4, 2015.