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 and analyzed for under $1,000 in total costs [100]. Costs will continue to decrease 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 [63]. 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 [133].

The massive scale of the sequencing data enables novel insight into biological phenomena. The Exome Aggregation project (ExAC, [73])—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 [92] and cardiovascular disease [149], and to better characterize the effect of structural variation [119]. 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 [51, 69] and standards organizations [108]. Not only is the volume of data large, but the processing cost to analyze the data is high. 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. Because the computational cost of processing genomic data is so high, working with large genomic datasets is often limited to large sequencing centers. As such, one of our goals is to democratize genomic data analysis by develop tools that make it easy and efficient to process large genomics datasets.

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 them, 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, horizontally-scalable distributed processing architectures like Apache Spark [158] and Hadoop [9], 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 targets bioinformaticians who need to implement custom queries across large genomic datasets, and who want to reduce the latency/increase the throughput of their queries by using cluster or cloud computing. 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. [158]) abstraction with genomics-specific functionality, and eliminates the low level "walker" pattern [87] that provides a sorted iterator over a genomic dataset, which 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.

Since 2013, our work on ADAM has resulted in the broad ecosystem of projects, which Figure 1.1 depicts. 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. [41]), 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 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

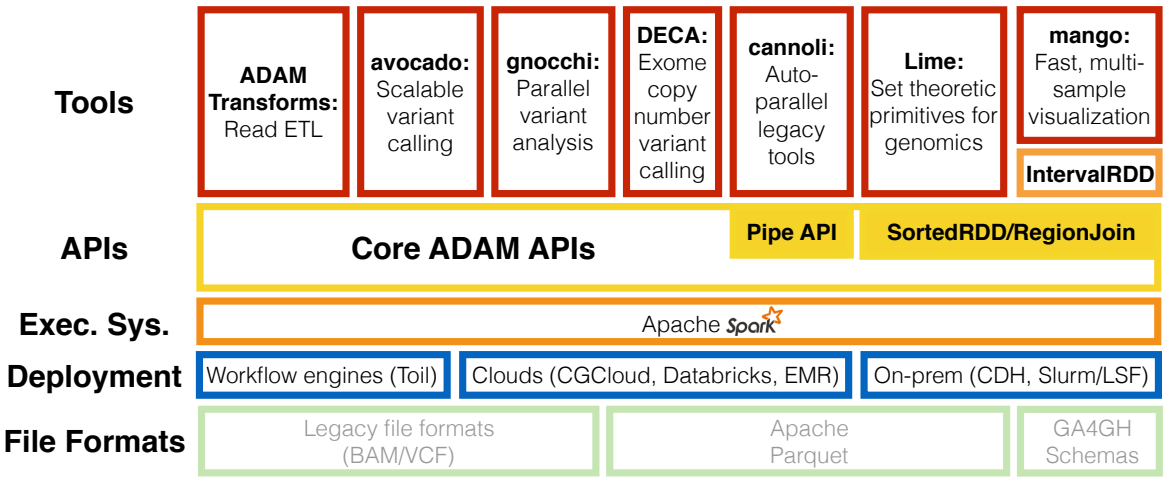

Figure 1.1: The Big Data Genomics ecosystem. Our work on ADAM has built a framework for scalable genomics using external projects like Apache Spark [158] and Apache Parquet [10]. On top of ADAM's core APIs, we have built a broad ecosystem of tools [82, 93, 141] demonstrating how Apache Spark can accelerate 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 [148].

ground truth datasets [165] and a large scale sequencing project [83]. We conclude by describing open problems and future directions for this research, as well as the impact of the Big Data Genomics/ADAM project.

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 [117], GWAS has faced several limitations. First, the association between genotype and phenotype is often weak, unless the variant under study is strongly pathogenic [127]. Additionally, few traits are truly Mendelian. For complex traits, which are driven by the combined effect of multiple genotypes [18], 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 [150]. 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 [22]. 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 [100]. This low cost has been driven by continuous technical improvements by Illumina and competing sequencer vendors. With the continued improvement of nanopore sequencing [63], 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 [2], 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, [151]) and the 300PB data catalog collected by the ExAC [73] 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 Biobank [144], and the Geisenger MyCode collaboration with Regeneron [24].

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 every seven months [133]. 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 [82, 121]. 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 [87], 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 that are implemented in existing distributed data analysis platforms [11]. 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 a clean slate re-architecture of the genomic data processing infrastructure. In a conventional 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 [122] and Crossbow [71]), 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 [123] and BioPig [102]), 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 developed 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. Support for processing genomic data with Spark SQL 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 [102, 123].

To demonstrate ADAM, we have built an end-to-end alignment and 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 generalized 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 [71, 122].

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

- Schemas which can support loading data from a large variety of formats, which improves programmer productivity by allowing queries against genomic datasets to be written in a format-independent manner.
- High level, quasi-relational APIs for manipulating genomic data in both single node and cluster environments, that abstract away low level details like sort-order invariants.
- Parallel I/O across genomics file formats, which enables efficient ad hoc query over large datasets.
- A simple API for parallelizing single node genomic tools with a minimal amount of code, which enables the reuse of common bioinformatics tools in a distributed computing architecture.

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 a 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 then dive deeper into the representations of genomic data (§2.2), and the architectures used to process this data (§2.2). Then, we review the variant identification algorithms (§2.2). Finally, we 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 [70], 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 extracting 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"). Depending on the sample collection methodology and the sequencing assay, we may replicate DNA sequences via a polymerase chain reaction (PCR). Errors during PCR can lead to duplication of specific fragments, which biases variant calling. 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 [95]). 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 [131] 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 library, 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 [84]. Additionally, library preparation varies tremendously between sequencing vendors. While we focus on short reads sequenced using Illumina platforms that are typically generated from fragments that are less than 1,000 bases long, long read sequencers such as the Pacific Biosciences sequencers [45] or the Oxford Nanopore sequencers [33], 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, as opposed to de novo analyses. In alignment-based methods, we rely on the existence of a "reference genome" for an organism. The reference genome 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 [70]. New genome references are released every few 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 [28]) 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.

Unlike alignment-based methods, de novo methods do not posit the existence of a reference genome. De novo methods are commonly used when a reference genome is not available (e.g., to assemble a genome that has not been previously assembled), or when performing an analysis that can be biased by the use of a reference genome, like structural variant discovery. These methods are outside of the scope of this dissertation.

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 [130], Ukkonen's algorithm [143], or a pairwise sequence alignment Hidden Markov Model (HMM, [43]). Widely used mappers include BWA [80], which builds an index using the Burrows-Wheeler transform [21]; Bowtie [72], which builds an FM-index [50]; and SNAP [157], which builds a hash-based index. Several projects have applied hardware acceleration to alignment, including Cloud-Scale-BWAMEM [25, 26], Ahmed et al [5], and unpublished work out of Microsoft Azure [91].

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. The test for read support 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 [75], and the GATK UnifiedGenotyper [41]. 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 dissertation, 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 [44].

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 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. PCR duplication 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) in 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 [41]. 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 to 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 [43, 130, 143], which can lead to inconsistent representations for equivalent sequence edits [77].

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 associates 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 [1, 2], the Exome Aggregation Consortium [73], and The Cancer Genome Atlas [151]. Additionally, the read preprocessing stages are 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 [87, 41] and ADAM [85, 103] 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 [2]. 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 [65], which was developed as part of the Human Genome Project [70]. Informal specificiations for the FASTQ [34] and FASTA formats date back to at least the 1990s, through their use in the phred [47] and FASTA/FASTP [110] 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 [80], which represents genomic reads, and the Variant Call Format (VCF) format [37], 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 [115] 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 [160] tool; the IntervalList format, which is used extensively by the GATK [41]; and the General Feature Format (GFF), which is used extensively in sequence annotation projects like the Sequence Ontology [46].

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 [76]. There has been significant research towards developing compressed storage formats for alignment data [69, 51]. 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" [118], and simply traverse a serial 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 [87]. 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 [41] workflow manager. This is resolved in the newest version of the GATK, which is implemented on Apache Spark.

While the UNIX-like design philosophy embraced 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 [58] 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 [148], NextFlow [42], Rabix [64], Cromwell [136], and Cuneiform [19]. 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 [35]. 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, haplotype-based models are used by several prominent variant callers such as the Genome Analysis Toolkit's (GATK) HaplotypeCaller [41], Scalpel [98], and Platypus [116]. Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls [15],

this accuracy comes at the cost of end-to-end runtime [134]. Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled [17] or by improving the performance of the core algorithms used in local reassembly [116].

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 [43]),
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 dissertation, we introduce algorithms that improve the algorithmic efficiency of 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 [75]. 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, r is the number of reads aligned to this region, l_r is the read length, 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 $\mathcal{O}(l_r l_h)$. Note that the number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see Depristo et al [41] and Garrison and Marth [53]) allow users to limit the number of haplotypes evaluated to improve performance. 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, unless the reads have been trimmed during quality control.

In this dissertation, 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 [77]. 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 [53], 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 *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 varies between methods (e.g., the GATK's IndelRealigner uses a simple log-odds model [41], while methods like FreeBayes [53] and Platypus [116] 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 [77]. 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 re-

alignment 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. [66], 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 [48]. Additionally, while the ϵ term models 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 [41]. 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 [53], restricting realignment to INDELs (IndelRealigner, [41]), 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

Dean and Ghemawat described the use of large clusters of commodity computers in their MapReduce system [38, 39] 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 [9], which started as a open source reimplementation of Google's MapReduce. Hadoop led to the development of scripting languages like Pig [105], query systems like Hive [140], and resource management frameworks like Apache YARN [145] and Apache Mesos [61]. While traditional map-reduce platforms are well suited to extract, transform, load (ETL) pipelines that made a single pass over a large dataset, they are a poor fit to "advanced analytics" applications—like machine learning, or graph processing—that made several passes over a dataset. This inefficiency 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 Apache Spark [159, 158] and Apache Flink [23]. Additionally, a set of highly efficient query engines came out, such as Cloudera Impala [67] and Spark SQL [11].

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 [122], which parallelized the RMAP aligner [129], and CrossBow [71], which integrates the Bowtie [72] aligner with the SoapSNP [81] 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 [112] aligner, which extracted the BWA [80] aligner into a Python library which was executed on the PyDoop [74] bindings for Hadoop; BigBWA [3], which parallelizes BWA [80] using the Java Native Interface (JNI) on top of Apache Hadoop; and Halvade [40], which parallelized the complex dataflow in the GATK [41] 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 [25], which parallelizes BWA through the JNI, with the ability to support FPGA acceleration; and SparkGA [96], 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 [101], which provides Hadoop-compatible parallel I/O libraries. The first generation of tools built query models for accessing genomic data through the Pig [105] scripting language. Support for Pig was implemented in two separate tools: BioPig [102] and SeqPig [123]. Additionally, the OpenCB project has built Hadoop-based tools for manipulating genomic data via the hpg-bigdata project [106]. Recent work has moved on to Apache Spark. Beyond ADAM and the Big Data Genomics ecosystem, Spark has been used in the SparkSeq [153] and VariantSpark [107] tools. SparkSeq is geared towards RNA-seq analysis, and has been paired with SparkBwa [4] to build the Falco [156] 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 [138], which is reimplemented on Spark. The second project is Hail [137], which is a reimplementation of the PLINK population genomics [113] toolkit on Spark.

We do not extensively discuss non-resequencing pipelines for de novo genome assem-

bly 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 [59] is difficult. The ABySS assembler [128] 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 [55, 54, 56, 57].

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 centered around schemas that provided a logical view over the genomic data being manipulated. This architecture was implemented on top of Apache Spark's Resilient Distributed Dataset (RDD) APIs [158], and provided the user with a distributed collection of genomic data which were encoded in Apache Avro [8] and 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 [141] and by adding a spatial- and temporal-locality-aware in-memory caching layer [93]. 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. Writing against low level APIs 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 are 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. A low level API introduces more locations where queries can be implemented incorrectly. We identified two concrete examples 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 [139] and Sambamba [135] duplicate markers require read inputs to be coordinate sorted, while the SAMBLASTER [49] 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, as discussed in Zheng et al. [161, 162]. This scatter-gather pattern chunks a genomic dataset into many small parts (contiguous ranges of the genome) that are then processed independently. This scatter-gather pattern presents several problems:

- Due to bias caused by repeated sequences in the genome, we cannot achieve optimal load balance with a partitioner that naïvely partitions the genome into uniformly sized ranges [27].

- 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 [148].
- This approach makes it difficult to implement queries that need to run an all-reduce over the data. Examples of an all-reduce include the base quality score recalibration kernel (see §6.1), or genome-wide machine learning methods [94].

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 limitation 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 [101], 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 [154] 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 un-splittable GZIP codec. GZIP is used in bioinformatics because of its high compression ratio, which is useful for storing textual read data, such as FASTQ files. To allow splitting, many tools use the BGZF codec [76], 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. Serialized I/O 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 [109]. 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 [85], 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. We had several specific goals that we felt we could achieve through building a higher-level API:

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 [80]) and htsjdk (the base for Picard [139] and the GATK [87]) 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 design choice 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 architecture enforces a strict separation between the I/O layer and the end-user API. A stack-based architecture can 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 [10] 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 [140] and Cloudera Impala [67].

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. Enabling 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. Thus, we needed a way to lower the latency of typical Spark queries to the interactive

(< 500ms) time range. The Mango project [141, 93] improved latency by introducing 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 queries were 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 [103], we introduced the region join to ADAM. This join provided functionality similar to BEDTools [115], 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 [157] 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, prior work which used Hadoop or Spark to manually wrap and parallelize legacy tools [122, 71, 3, 4, 26, 25] 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 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 Chapter 6, 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 [164]. 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.

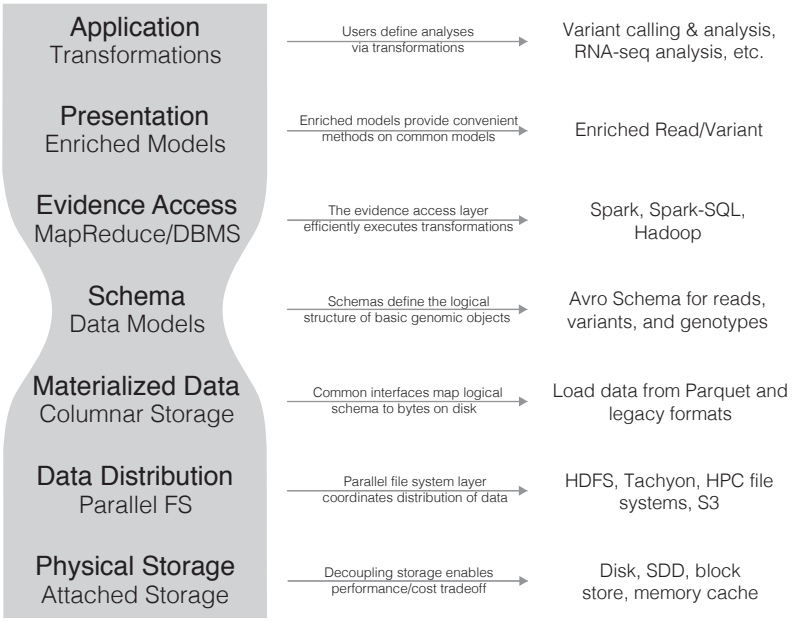

Figure 3.1: A stack model for scientific computing

Unlike conventional scientific systems that depend on custom data formats like BAM or SAM [80], or CRAM [51], we believe that the use of an explicit schema for data interchange is critical, as it allows bioinformaticians to program at a level of abstraction that hides the implementation details of the file format. 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 of implementation. 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 Chapter 4, 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. 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. [14] 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.

- Notably, Bafna, et al. use this stack model to motivate GQL [68]. 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 [164] 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 [121, 132]. 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 was not provided. As a stopgap solution, projects like Cloudburst [122] and Crossbow [71] 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 [87] 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 [158].
- 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.

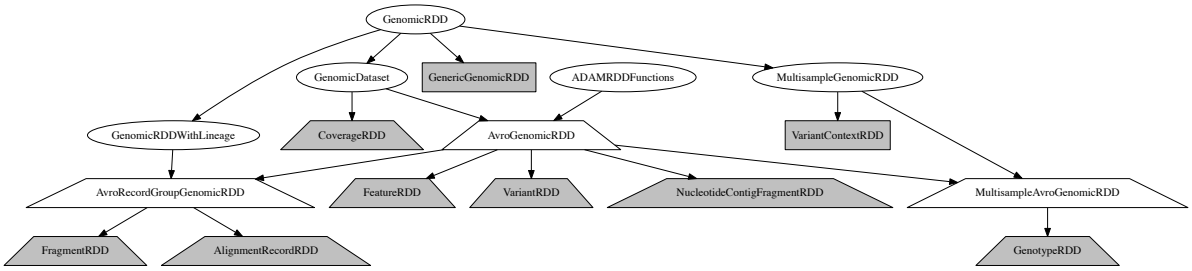

Figure 4.1: The GenomicRDD class hierarchy. Ovals are traits, trapezoids are abstract classes, rectangles are classes, and a bold border means that the abstract class is sealed. In Scala, sealing a class/trait restricts which classes can extend this class/mix-in this trait. The shaded classes are the 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 ADAM's vision:

- The GenomicRDD wraps the Apache Spark RDD [158] and Spark SQL DataFrame [11] APIs with genomics-specific and genomic-datatype specific functionality. Access to both APIs from a single class 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.
- 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 [158] and Spark SQL [11] 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 [141, 93].

- The presentation layer (6) provides datatype (e.g., read/feature/variant) specific methods on top of the GenomicRDD API. This layer 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 [85] and our SIGMOD manuscript [103], the evidence access and presentation layers (5 and 6) have changed. 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 [67] 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 §4.3.

- 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 [104], 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 [34], SAM [80], BAM, and CRAM [51]. 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 [10] to materialize data on disk, the Apache Avro [8] 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 [85]. Over time, we have gone backwards on some of these design decisions. First, the schemas were originally fully denormalized. Our rationale for this 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 [85, 103]. We have still maintained this requirement.

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 specifica-

tion, the variation and feature representations deviate significantly from the current state of representing genomic variation and features.

Read Schemas

Our read schema (§A.1) 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. Separating the flags 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).

Additionally, we support a schema that groups together all of the reads from a single sequenced fragment (§A.2). The fragment data structure enables a traversal over read data that is similar to SAM's read-name grouping. 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 (§A.3) 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. Restricting records to be biallelic 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 [137].

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 [85], the variant and genotype schemas were much narrower and closely mirrored the VCF spec-

ification. 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 [141, 93], 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 variant annotation schemas (§A.3).

These schemas provide a faithful representation of the VCF/ANN specification, which adds a formal and rich variant annotation specification to VCF [31]. This specification is used by both the Ensembl Variant Effect Predictor (VEP, see [89]) and SnpEff [32]. We have a separate schema for storing annotations on top of a genotype call (§A.3); 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. 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.

During the refactor that improved ADAM's support for variant annotations, we refactored the conversion class that provided a view from the VCF format to ADAM's genotype and variant records. In this refactor, we replaced the extant conversion code with a pure-functional converter. Our approach generated a collection of conversion functions. To convert between the ADAM and htsjdk representations of a variant/genotype, we would fold over this collection of functions. In turn, each function accesses the field they are converting, and mutates the new object that is being built during the conversion. The use of a pure-functional converter had several upsides:

- One of the difficulties in working with the VCF file format is that it is semi-structured and user extensible. Beyond the info and format attributes in the VCF specification, users can add any attributes they desire, restricted only by the attribute types in the VCF specification. As part of this update, we added the ability to generate attribute conversion functions from the VCF header.

- Our new converter has a single function per field that is converted. As a result, it is very easy to write highly directed functional tests to achieve a very high level of test coverage for the converter. Since the functions are all orthogonal and have no shared state, we can also guarantee that any two functions that independently pass their directed tests can be called in a chain correctly.

- A minor problem with our previous, monolithic converters was that they could not support projecting out a subset of fields, unlike data stored in Apache Parquet using

the ADAM schemas. While we would still need to read all of the data in a whole row, being able to limit the number of fields projected would reduce the amount of parsing we need to do, which can be quite costly for semi-structured text formats like VCF. The pure-functional converter naturally supports user-specified projections, as we can arbitrarily filter out any conversion functions that we do not want to execute.

In the long term, we hope to apply this pure-functional conversion approach to all of our data converters. We discuss this more in §10.1.

Feature Schemas

ADAM's feature schema (§A.4) 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.

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 metadata 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 [8] 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 [60].

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. 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 locus can be processed independently from the data localized to a different region.

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 the region join primitive. The algorithm used is described in Algorithm 1 and takes as input two RDDs of ReferenceRegions, a data structure that represents intervals along the 1-D genomics coordinate space. It produces the set of all overlapping ReferenceRegion pairs. This join can be implemented using either a broadcast or sort-merge approach. Algorithm 1 demonstrates the broadcast approach.

Our implementation of the broadcast region join has been refactored twice. The first implementation was described in our SIGMOD manuscript [103] and required the computation of the convex hulls of the left side of the dataset. Computing the convex hulls was done using an algorithm similar to the one described in §6.2. Additionally, this approach required both sides to be hash-partitioned. The first rewrite of the broadcast join was very similar

Algorithm 1 Join Regions via Broadcast

$left \leftarrow$ input dataset; left side of join
$right \leftarrow$ input dataset; right side of join
$sortedLeft \leftarrow left$.sort()
$collectedLeft \leftarrow sortedLeft$.collect()
$tree \leftarrow collectedLeft$.buildTree()
$broadcastTree \leftarrow tree$.broadcast()
$joined \leftarrow right$.map($r \rightarrow broadcastTree$.overlaps($r$))
return *joined*

to Algorithm 1. However, it used an interval tree [120] to store the left side of the join. We abandoned this approach because of the cost of serializing the interval tree. Instead, we introduced a new data structure, the interval array, which is a flattened representation of the interval tree. The interval array is an array whose elements are sorted by coordinate-order. To look up overlapping elements from an interval array, we begin by running binary search to find the first element that overlaps our search value. In the interval array, we also compute and store the length of the longest region in the array. We then search forward in the array until we have searched the distance of the longest element. Since this data structure is flat, it is inexpensive to (de)serialize. Binary search provides the same $\mathcal{O}(\log n)$ time complexity as searching into an interval tree. However, since we require an additional linear search, we can only guarantee the best case $\Theta(\log n)$ complexity for searching into our interval array.

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 ex-

ample, 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. Additionally, we are working to improve the performance of these joins by eliminating shuffles of data that is already sorted on disk. We discuss these avenues for future work in §10.1.

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 have 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 [147] after Spark SQL was introduced [11]. 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 [36] or SparkR's interoperability layer [147]. 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 [11], such as the Pandas [88] and Dyplr [152] 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 [11] 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.

Since translating between a Spark RDD and a Spark SQL DataFrame requires a (de)serialization pass, we introduced a scheme that we call "lazy binding" to minimize serialization overhead. In Figure 4.1, you may have noticed that most of the core GenomicRDD types were sealed abstract classes. In practice, they are all implemented using the lazy binding scheme. Figure 4.3 depicts the bound variants of the AlignmentRecordRDD type. In this scheme, we track whether the last transformation applied to the GenomicRDD was applied to the DataFrame or RDD representation of the genomic data. The next phase of computation will run on top of the bound output of the prior stage. Avoiding a conversion between the RDD and SQL DataFrame representations can provide a major performance win if running Spark SQL queries, as the Spark SQL query optimizer performs major optimizations to amortize the method call overhead of chained transformations [11]. A further optimization deals with loading data from Apache Parquet. While data that is being loaded from legacy file formats like BAM or VCF must be loaded as an RDD, Parquet can be loaded as either an RDD or DataFrame, and Spark SQL can make further optimizations if it knows that it is loading data from Parquet. To handle this, we added the "unbound" GenomicRDD. This datatype does not bind to the RDD or DataFrame representation of a Parquet table until a transformation is invoked. We believe that there are further opportunities to extend lazy binding to legacy genomic file formats, which we discuss in §10.1.

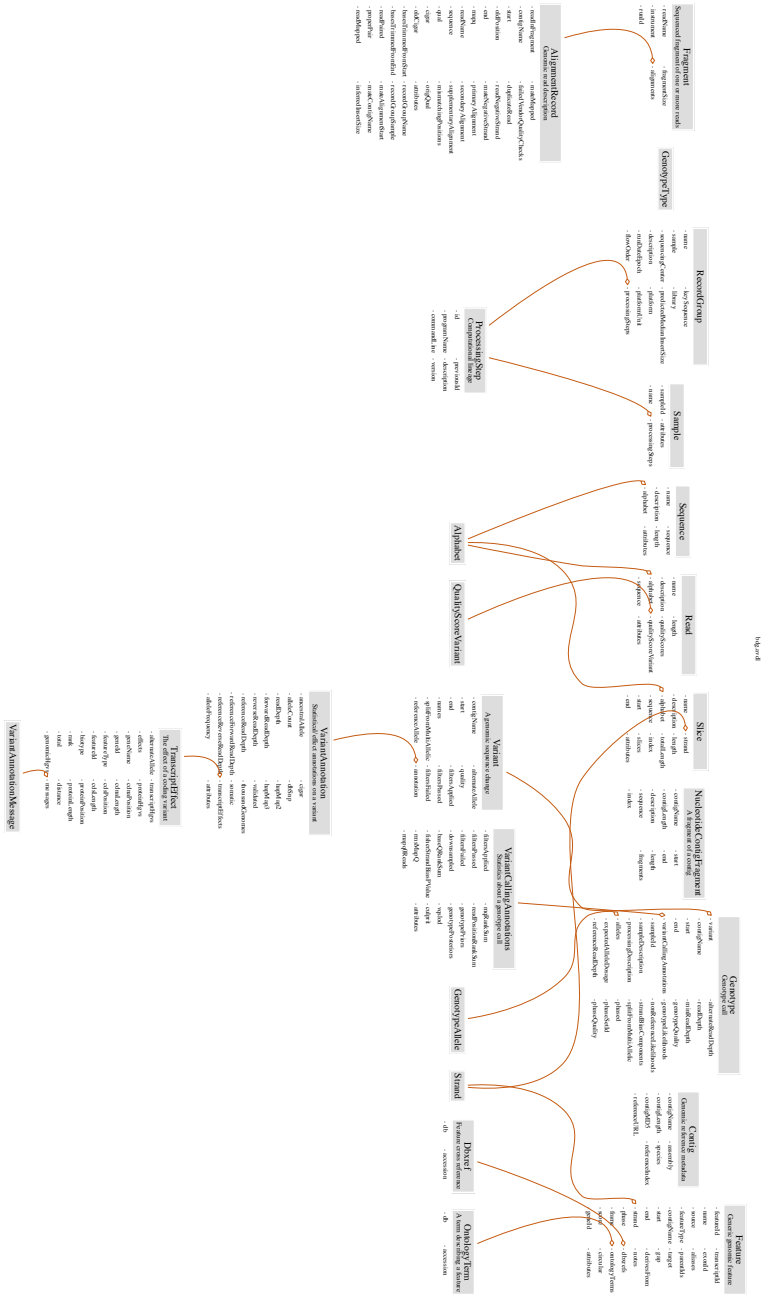

Figure 4.2: A UML diagram showing the dependencies and structure of the ADAM schemas. ADAM's read schema is fairly flat, which contrasts with the other schemas which employ nesting. The variant and feature schemas are deeply nested. This allows both rich annotation and the definition of the complex feature hierarchies necessary for describing genomic annotations [46].

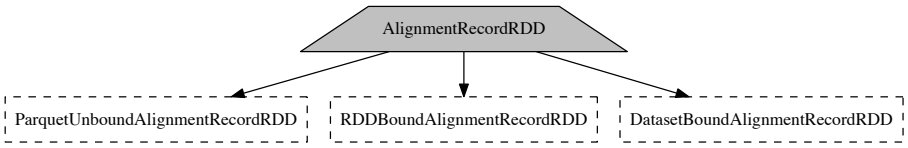

Figure 4.3: With the move to support Spark SQL, we added a binding scheme to the GenomicRDD hierarchy. This figure depicts the alternative bindings available for the AlignmentRecordRDD, which is representative of the core GenomicRDD types. The dotted lines around the concrete implementations of the abstract AlignmentRecordRDD class signify that these classes are protected and cannot be instantiated outside of the ADAM library.

Part III

Algorithms and Tools

Chapter 5

Automatic Parallelization of Legacy Tools with Cannoli

While ADAM provides a general API for implementing genomic analyses using Apache Spark, we will never be able to fully eliminate single node tools from genomic workflows. Beyond the small set of tools discussed in §2.3, the majority of genomics tools are designed to run on a single node. Additionally, due to memory access and allocation patterns, tools like aligners pay a steep performance penalty when moving out of C/C++ to the JVM. An example of this is the SNAP aligner, which was originally written on Apache Spark, before being rewritten in C++ for performance [157]. Finally, while it is possible to reimplement common genomic analyses like variant calling using ADAM, it is not feasible to do this across the hundreds of extant genomic analyses.

As discussed in §2.3, there are many tools that provide customized solutions to this problem. The GATK's Queue engine [41] could be used to run the GATK in parallel, and the FreeBayes variant caller [53] can be run in parallel through the SpeedSeq pipeline [27]. Additionally, a variety of tools are parallelized by Hadoop-ecosystem wrappers. These include tools like the Seal tool which parallelizes BWA using Pydoop [74, 112], Rail-RNA [99], which was built on top of Hadoop, and tools built on top of Hadoop Streaming like CrossBow [71] for SNP calling and Cloudburst [122] for alignment. Additionally, several tools have built on top of Apache Spark, including CloudScale-BWAMEM [25] and BWASpark [4].

Instead of expecting developers to create a proliferation of custom wrappers for parallelizing tools on distributed analysis systems, we should be able to build a general infrastructure to automatic this process. Since many genomics tools are built around a streaming paradigm, if we can provide automated chunking, process setup, distribution of reference files, and streaming data, we should be able to automatically parallelize a large number of genomic analysis tools.

In this chapter, we introduce a two part architecture for automatically parallelizing genomic analyis tools. First, we describe ADAM's pipe API, which performs the automated chunking, resource distribution, and process coordination. Then, we describe the Cannoli tool, which contains wrappers for a set of common genomic analysis tools. This approach

is general but not universal: while the pipe API can be used to parallelize the majority of analysis steps, it cannot be used for tools that require an all-reduce. All-reduce patterns are found in RNA-seq quantification tools like Kallisto [20] or Salmon [109], or tools that perform a global normalization, like the XHMM copy number variant caller [52]. However, this approach works well for tools whose computation can be described as mapping over unaligned reads, or mapping over data aligned to a genomic locus range.

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

The pipe API is an API in ADAM that provides a simple mechanism for automatically parallelizing a command across a cluster running Apache Spark. The user specifies the command to be run, files that need to be distributed to each worker running the command, and any environment settings that are needed. ADAM then uses the attached genomic reference metadata (see §4.2) to infer the proper partitioning to apply to the data, partitions the data, and runs the user specified command. The input format to provide to the command and the output format to expect from the command is determined at compile-time in Scala and at runtime in R and Python.

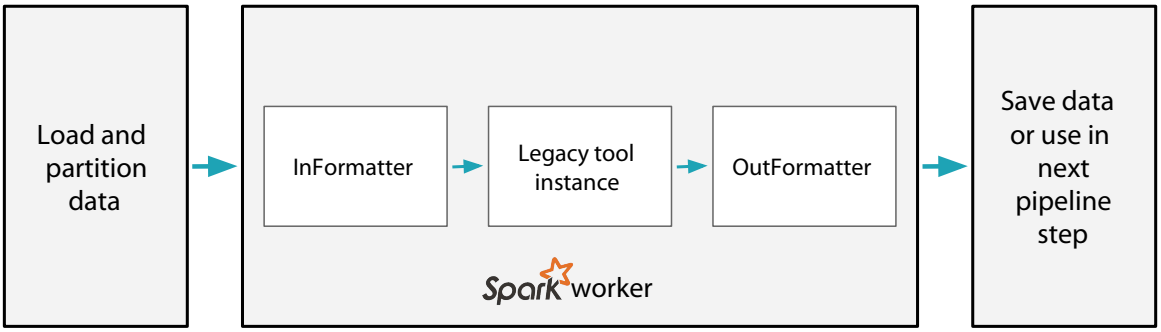

Figure 5.1: The implementation of the pipe API.

Figure 5.1 illustrates how the pipe API is executed by ADAM. The pipe implementation starts by checking the genome reference metadata (see §4.2) to see if the dataset is aligned to a reference genome. If the data is not aligned, then we assume that no specific partitioning scheme is needed, and skip repartitioning. If the data is aligned, then we use the partitioner used in the region join implementation (see §4.3) to chunk the data into partitions that represent contiguous blocks of the reference genome. After the partitioning phase, we open a subprocess per partition that is running the user specified command, and connect streams to the standard in and out pipes of the running process.

To handle a wide variety of file formats, we have abstracted the I/O subsystem into "formatters." We use the formatters to encode/decode the data going to and from the running process:

- To create the input for the running process, we use the InFormatter trait. This trait receives an iterator over a single partition of a GenomicRDD, and encodes that iterator into the format that the running process is expecting. Each implementation of the InFormatter trait is required to also implement the InFormatterCompanion trait, which defines a singleton object that can create an instance of the class. This pattern is necessary to expose the metadata stored in a GenomicRDD to the InFormatter by guaranting that there will be a static method that can build an InFormatter with access to the metadata stored in a GenomicRDD. This approach is necessary since Scala cannot guarantee that each implementation of a trait provides a constructor with a given type signature, and we need metadata to construct properly formatted BAM and VCF streams. The InFormatter runs asynchronously in a new thread.

- To read the data from the running process, we use the OutFormatter trait. This trait decodes the output of an invocation of the running process and creates an iterator. The OutFormatter does not need access to the metadata from a GenomicRDD, so we do not need an OutFormatterCompanion trait. The OutFormatter is run synchronously in the thread that called the process. The OutFormatter provides an iterator so that we can leverage Apache Spark's pulling model, which limits the number of records that must be materialized into memory at a given instant.

This API allows for a very concise, yet general framework. With the formatter pattern, we can support a broad range of formats while exposing a very simple API to the user. Specifically, we can pipe sequence fragments as interleaved FASTQ, reads as SAM/BAM/CRAM, variants/genotypes as VCF, and features as BED/GTF/GFF2/GFF3/IntervalList/NarrowPeak. If a program needs reference files to be available locally on each node, they can achieve this by passing an array containing the URLs of each file. We use an internal Apache Spark API to copy these files locally on each machine. Along with this, we provide a simple syntax that allows the user to symbolically add the local path to this file into their command, which we know at runtime.

To demonstrate how concise the API is, Listings 5.1 and 5.2 demonstrate how to use the pipe API to call BWA [78] from Python and FreeBayes [53] from R.

Listing 5.1: Calling BWA using the Python pipe API

```
from bdgenomics.adam.adamContext import ADAMContext

ac = ADAMContext(sc)

reads = ac.loadAlignments("reads.fastq").toFragments()
```

```
alignedReads = reads.pipe(
  "bwa mem -t 1 -p /data/hs38DH.fa -",
  "org.bdgenomics.adam.fragments.InterleavedFASTQInFormatter",
  "org.bdgenomics.adam.rdd.read.AnySAMOutFormatter",
  "org.bdgenomics.adam.api.java.FragmentsToAlignmentRecordsConverter")
```

Listing 5.2: Calling FreeBayes using the R pipe API

```
library(bdg.adam)

ac <- ADAMContext(sparkR.session())

reads <- loadAlignments(ac, "alignments.bam")

variants <- pipe(reads,
  "freebayes --fasta-reference /data/hs38DH.fa --stdin",
  "org.bdgenomics.adam.rdd.read.BAMInFormatter",
  "org.bdgenomics.adam.rdd.variant.VCFOutFormatter",
  "org.bdgenomics.adam.api.AlignmentRecordsToVariantContextsConverter")
```

In both examples, the user can run the parallelized tool with less than five lines of code. We believe that the power of this API lies in its simplicity. With a minimal API, we can replace the manual sharding that was historically implemented using workflow management systems [41, 27] or custom wrappers [71, 122, 112, 99]. Since this API is built in the query layer of the ADAM stack, it can run independent of both the file format used to save the data and the storage system used to store the data. This latter point is a significant improvement over workflow systems that expect a shared POSIX file system, as these systems often use indexed range query into a file to perform sharding [27]. Range queries are difficult to apply generally on a cloud file store like S3, as the tool being run in parallel must be able to natively query the cloud file system. Additionally, this approach makes it possible to integrate a command line program into a larger programmatic workflow. We discuss a possible use case that would leverage this pattern in §10.2.

5.2 Packaging Parallelized Single-node Tools in Cannoli

To make the automated parallelization of the pipe API more accessible to users, we introduced the Cannoli tool, which provides a command line interface (CLI) for executing common tools using the pipe API. Our goals were to provide a minimal wrapper around the pipe API that would:

- Allow the user to use reference files that were already present locally on all nodes in

the cluster, or automatically distribute the necessary reference files to all nodes.

- Support running the tool from a locally installed executable or a Docker container, as Docker is becoming a widely used method for distributing pre-built genomic analysis tools [148].

To support this, we have control logic that parses out the arguments passed to the CLI. If the user has requested that we distribute all of the files to the workers, we then elaborate out the paths needed by the tool and add the files to the pipe call. We do this since many tools that require prebuilt resources use a multitude of resource files that are all at a base path. For example, BWA [78] uses seven index files, SNAP [157] uses four index files, and SnpEff [32] uses a large directory of resource files. If the user has requested to run the tool via a Docker container, we then ensure that the directory containing the distributed index files are accessible within the container. Currently, we support a list of tools including but not limited to BWA, SNAP, and Bowtie2 [72] for alignment; FreeBayes [53] and mpileup [75] for variant calling; and SnpEff and BEDTools [115] for annotation. We evaluate the performance and concordance of the tools implemented in Cannoli in §8.3.

Chapter 6

Scalable Alignment Preprocessing with ADAM

In ADAM, we have implemented the three most commonly used pre-processing stages from the GATK pipeline [41]. 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 marking:** During the process of preparing DNA for sequencing, reads are duplicated erroreously 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 [139], 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 read pairs whose reads align to different reference chromosomes (known as chimeric read pairs, see Li, et al. [79]). 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. Local misalignments are typically caused by the presence of insertion/deletion (INDEL) variants [41]. 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 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:** 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. We evaluate the performance and concordance of these algorithms against other tools in §8.1.

6.1 BQSR Implementation

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

The original BQSR engine in ADAM was generic and placed no limitations on the number or type of covariates that could be applied. A covariate describes a parameter space where variation in the covariate parameter may be correlated with a sequencing error. However, supporting generic covariates reduced the throughput of the recalibration engine, and we found that recalibration was generally run with two specified covariates. Currently, we support two covariates that map to common sequencing errors [97]:

- 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. Additionally, a reagent error can occur in a way that corrupts bases from a single cycle.

- 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.

These covariates are calculated for each sequenced base. We then merge the covariates into a CovariateKey, which also includes the read group that the sample was from (see §4.2) and the quality score assigned to the base. 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. 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.

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.reduceByKey(merge)
return table
```

Originally, Algorithm 4 was implemented by running an aggregate where we merged all of the disparate covariates into the table at each step. By restricting the BQSR engine to two covariates, we achieved several performance optimizations:

- By hard-coding the types of covariates we used, we reduced the cost of identifying whether two CovariateKey objects represented the same covariate.
- Additionally, hard-coding the covariates greatly reduced the size of the CovariateKey object in memory, which allows us to achieve higher cache hit rates when caching the output of the first stage of BQSR.
- Merging the two base observations originally required updating and merging the new and old Observation values. By rewriting as a reduceByKey, we were able to eliminate the need for mutable objects. We did not move this codepath to Spark SQL, since this refactor predated the introduction of Spark SQL support into ADAM. However, this is a straightforward modification.
- In the process of rewriting the CovariateKey representation, we optimized the implementation of the code that assigned dinucleotide and cycle covariates to a given base. This yielded a 2× speedup when creating covariates.

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.

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

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. However, this process is expensive if implemented naïvely. To handle covariate specific errors, we use the hierarchical model from the GATK [41]. In this model, we generate the empirical quality of a CovariateKey by hierarchically computing the empirical error rates for the covariate groups that it falls into. We treat the read group as the highest order covariate, then the quality score, and then the two error covariates. Algorithm 5 demonstrates this approach.

In the original BQSR implementation, this inversion would be called on every read base. This is inefficient, as the empirical quality is equal across all bases from a given CovariateKey. We eliminated this by inverting all covariates after all of the observations had been collected. Then, evaluating the new empirical quality score for a base assigned to a covariate was performed by looking up the covariate key in a hash map, which is computationally inexpensive.

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

Algorithm 5 Inverting an element in the covariate table

$key \leftarrow$ covariate key to invert
$global \leftarrow$ the empirical quality of this read group
$qualities \leftarrow$ the empirical qualities of bases with a given quality score
$dinucs \leftarrow$ the empirical qualities of bases with a given context
$cycles \leftarrow$ the empirical qualities of bases at a given cycle
$p_1 \leftarrow$ compensate(key.quality, $global$)
$p_2 \leftarrow$ compensate(p_1, $qualities(key$.quality)
$d_d \leftarrow$ delta(p_2, $dinucs(key$.dinuc))
$d_c \leftarrow$ delta(p_2, $cycles(key$.cycle))
return $p_2 + d_d + d_c$

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 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 6, which is a data parallel algorithm for finding the maximal convex hulls that describe a genomic dataset.

Algorithm 6 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
```

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 7.

Algorithm 7 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 [126].
- *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 [130] 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. To improve performance, in the $q_{i,j}$ calculation, we terminate the loop once the running sum $\sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)]$ is larger than the prior $\min_i q_{i,j}$. 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.

To eliminate a shuffle, we leveraged an insight from SAMBLASTER [49], which runs directly on the output of an aligner, that will group all reads from a single fragment together (also known as query grouped). The fragment schema introduced in §4.2 describes a set of query grouped reads, and is equivalent to the output of the step in duplicate marking that groups reads by read name. When reads are grouped by sequencing fragment, we can eliminate a shuffle. Fragment-grouped reads are found when loading reads from a query grouped SAM/BAM/CRAM file, or when running on the output of reads aligned using Cannoli. We describe the performance benefit of this change in §8.1.

Chapter 7

Rapid Variant Calling with Avocado

This chapter introduces Avocado, a substitution and short INDEL variant caller that is built natively on top of the ADAM APIs. For highest accuracy, Avocado is run as a two phase tool. In the first phase, we reassemble or realign our reads around INDEL variants. In the second phase, we apply a probabilistic model built around a biallelic model to the reads to identify variants.

Avocado's INDEL reassembly process cleans up all reads that are aligned near INDEL variants. We do this as a two step process. In the first step, we pass over all the reads and use our novel indexed de Bruijn algorithm to extract and canonicalize all of the INDEL variants. For best accuracy, we use the ADAM INDEL realigner described in §6.2 to realign known INDELs. This approach improves variant calling accuracy over solely using the indexed de Bruijn algorithm. After realigning the INDELs, we run genotyping. 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. Genotyping is a four step process:

1. We extract all variants from the aligned reads by parsing the alignments.
2. 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.
3. We merge all of the per-read likelihoods per variant. Aggregating the per-read likelihoods gives us the final genotype likelihoods per each variant.
4. 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 [159, 158] using the ADAM library [85, 103].

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. 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. 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 [62]. 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 $\mathcal{O}(nd)$, where, $n = \max(\text{distance}_{\mathcal{S}_i}(s_1, s_2), \text{distance}_{\mathcal{S}_j}(s_1, s_2))$.

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 $m_i = distance_{\mathcal{S}_i}(s_1, s_2)$, and $m_j = distance_{\mathcal{S}_j}(s_1, s_2)$. 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. A substitution 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.

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 in $\mathcal{S}'_j$ relative to $\mathcal{S}'_i$, $k-1$ letters *after* index(s_1):

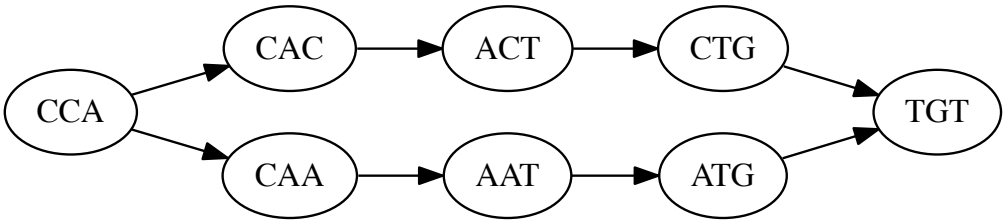

Figure 7.1: Subgraph Corresponding To a Single Nucleotide Edit

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.

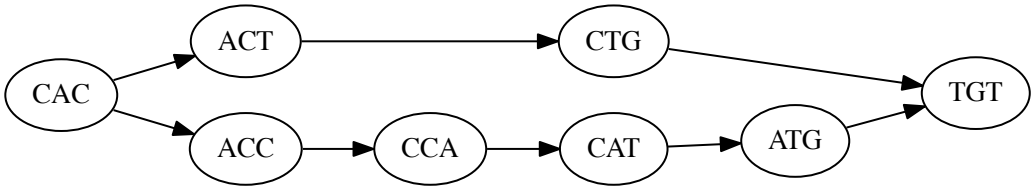

Figure 7.2: Subgraph Corresponding To a Canonical INDEL Edit

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 $s_1, \mathcal{L}_i, s_2$. This property holds true for $\mathcal{S}_j$ as well. The canonical edits merely result from special cases:

- 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. [43] or Smith and Waterman [130]), *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 [143]). 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 [80]. 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. MD tags allow 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 §8.1, 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 [85, 103]. This tool is based on the algorithms used in the GATK's INDEL realigner [41], 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 §8.1, 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 [75], 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 [111]. 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 [27]. 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 [27], 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. By default, 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 the overlap join primitive introduced in §4.3 to find all of the variants that a single read overlaps. 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.

Genotyping Model

Once we have joined our reads against our variants, we score each read using a variant of the biallelic genotyping model proposed by Li [75]. Our approach differs in several ways:

- Due to the limited numerical precision of double-length floating point numbers, we implement our likelihood model entirely in log space. To avoid computing logs repeatedly, we tabulate all the possible values of a single read's contribution to the likelihood of a single variant given a fixed range of base and mapping qualities.
- To handle multi-allelic sites, we decompose the likelihood model further. Specifically, we compute the likelihood that the read supports the reference allele, the alternate

allele under test, another observed alternate allele, or an allele that was not observed. We can use this approach to compute the likelihood of a compound heterozygous variant, and to make a partial no-call.

An early variant of Avocado used a strict biallelic model, but this generated incorrect calls for compound heterozygous sites and INDEL variants. This early version of Avocado performed the following process to score variants. For each variant, we check to see if the read supports the variant allele. 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. This version calculated 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 [75], but in log space.

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

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

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 [43]. 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 quasi-gVCF approach. Additionally, we support a gVCF mode where all sites are scored, even if they are not covered by a putative variant.

The new scoring model used in Avocado checks to see if the read matches the alternate allele we are testing, the reference allele, another alternate allele that overlaps with the allele we are testing, or none of the above. To do this, we use the "one sided" variant of Equation (7.1), which is given in Equation (7.3).

$$\log \mathcal{L}(g) = \sum_{i=0} jl_r(g, m-g, \epsilon_i) \tag{7.3}$$

For all sites, the log sum of the one-sided likelihood of the reference allele and the one-sided likelihood of the alternate allele under test will be equal to the value of the full biallelic likelihood model. However, the one-sided model can also be applied to sites with more than one alternate allele. For these sites, after computing all one-sided likelihoods, we then calculate the combined likelihoods for each pair of alleles, and emit a genotype call that corresponds to the highest likelihood of all admissible likelihood combinations.

We compute the likelihoods for each read in parallel. The scoring function maps over all of the reads. For each variant covered by the read being observed, the scoring function will emit a record that contains the mapping and base qualities, and whether the read supported the reference allele, the alternate allele under test, another alternate allele, or no known alleles. These records are converted into a Spark SQL DataFrame, and we join these values against a small table containing all of the pre-computed one-sided likelihoods. We then run an aggregation over all reads covering a given variant, for all variants in parallel. Once we have aggregated 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.

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.

Part IV

Evaluation

Chapter 8

Benchmarking the ADAM Stack

8.1 Benchmarking Preprocessing Algorithms

Benchmarking Duplicate Markers

Benchmarking INDEL Realigners

Benchmarking Recalibrators

8.2 Evaluating Compression Techniques

These representations achieve high compression versus the legacy formats. We provide a detailed breakdown of compression in §8.2. 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.

8.3 Benchmarking Cannoli

Chapter 9

The Simons Genome Diversity Dataset Recompute

Part V

Conclusion and Future Work

Chapter 10

Future Work

In this dissertation, we have articulated a set of overarching goals for ADAM and the Big Data Genomics project. We aimed to make it easier to write genomic analyses that could be run on large datasets using distributed computing. With these APIs, we started working on the Avocado variant caller, with the goal of making variant calling both faster and more accurate. From this work, we have identified several important future directions. In this section, we will discuss possible avenues for improving query performance in ADAM by providing more genomics specific query optimizations, extending the variant calling algorithms in Avocado, and mapping genomics analysis tasks directly into hardware.

10.1 Further Query Optimization in ADAM

ADAM provides several interfaces that either provide genomics-specific query APIs, or that perform a runtime optimization. These interfaces include the region join API described in §4.3, the pipe API described in §5.1, and the lazy binding optimization described in §**??**. We can improve the performance of the region join and pipe APIs by eliminating shuffles when data is already presorted. We can also improve the performance of the pipe API by applying the lazy binding techinque to the various genomic file formats. Additionally, we discuss how we can make it easier to understand the performance of distributed code in Apache Spark, and how we can make it easier to manipulate genomic metadata.

Preserving Sorted Knowledge in ADAM

An inefficiency in ADAM's region join API stems from the need to achieve an equivalent partitioning between the two datasets being joined. We recently added functionality to ADAM that saves metadata to disk if the dataset is being saved as a sorted table of Apache Parquet data, with the sort order being defined lexicographically by the contig names. While this approach can eliminate shuffles for some joins, it has several limitations:

- Since the region join API requires records that straddle the partition boundaries to be replicated across the boundary, this approach only works for Apache Parquet. Conceptually, we could implement this with a sharded codec, but Hadoop-BAM's support for sharded file formats is limited. As a result, we cannot eliminate shuffles for data coming in from a standard genomic file format that defines a sort order.
- Additionally, the current partitioning scheme requires a lexicographical contig ordering. This is inconsistent with the karyotypic ordering that is commonly used in the bioinformatics ecosystem, and which relies on being able to specifiy the ordering of contigs with an index..

We are working on an alternative approach that is compatible with all contig orderings and file formats. In this approach, we make several changes:

- Instead of replicating the records that straddle a partition boundary in the dataset that is saved to disk, we move these records using a shuffle-free approach. To do this, we run a first pass that looks as the records at the start of each partition and selects all records that need to be replicated to the end of the previous partition. These records are then collected to the driver and appended to the ends of the partitions prior to the join running. Since this approach doesn't require records to be replicated, it is compatible with all file formats.
- In the current implementation, once data is loaded as sorted, this is encoded by setting a field that is protected to the GenomicRDD trait. This makes it more difficult to implement methods that can be optimized to make use of sort order. Specifically, if a method can optimize for sort order, the method needs to check this field and select the sorted or unsorted codepath at runtime. In the new implementation, we make sorted and unsorted traits. If an implementation of GenomicRDD has behavior that can be optimized for data that is known to be sorted, that implementation should provide the optimized variant in a class that implements the sorted trait, and an unoptimized variant in the unsorted trait.
- We drop the requirement for a lexicographic ordering, and use an indexed contig ordering. This supports both the lexicographic ordering that we typically use, as well as the karyotypic ordering that is commonly used in genomics.

Beyond broadening the opportunities for eliminating a shuffle in the region join codepath, an additional benefit of this new approach is that it allows us to harmonize the partitioning scheme used by the pipe API with the scheme used by the region join API. Currently, these two schemes are not harmonious because the region join's partitioning scheme does not allow a flanking length to be specified. In the pipe API, we allow the user to specify the width of flanking sequence to include in each call. Since this new partitioning scheme uses a flanking length to decide which record to copy from the start of a partition to the end of the prior

partition, we can unify these two schemes. We have prototyped this approach, but have not yet evaluated and critiqued this approach on read world datasets.

Lazily Binding To Genomic File Formats

An inefficiency in the pipe API occurs when reading data from a file stored in a legacy file format and then piping this data into a command using that same file format. An example of this would be loading reads from a SAM file and then pipeing these reads into a variant caller using SAM input. This is inefficient because we wind up converting the reads into the ADAM schema, just to convert the data back into the legacy file format.

One possible approach to eliminate this issue would be to change how we implement the conversion/view system in ADAM. Currently, the view mechanism is implemented using monolithic datatype conversion classes. These classes provide methods that convert a single record from a legacy data file into ADAM's schema records. The GATK4 [138] uses an alternative approach. In this approach, the GATK4 wraps the record types that it converts from in an interface that these records then implement. The conversions are then implemented through the methods in the wrapper classes.

Improving Debugging Capabilities for Distributed Systems

Debugging is a persistent problem in developing code that runs on distributed systems. While much work has focused on debugging errors [7, 163] in a distributed system, we are more interested in debugging performance anomalies. Due to the nature of genomic data, faults are typically easy to diagnose, as the faults are localized to a set of genomic loci. These loci can then be isolated, and used for debugging. A bigger challenge can be isolating performance anomalies, especially since many performance anomalies may be related to complex genomic loci. Additionally, Apache Spark's execution model pipelines multiple transformations into a single analysis stage [158], which can obfuscate the contribution of multiple functions to a performance anomaly.

In ADAM, we attempted to address the latter problem through a library enhancement that was contributed by an open source developer. In this approach, we wrap the RDD API in a way that wraps all RDD function calls in timers. These timers are high resolution timers and are specified at a very fine granularity (e.g., a timer might wrap a single function call inside of a map over the RDD). Once a job runs, the statistics collected by the timers are reported in a tabular format.

While this library is useful, it still doesn't have a sufficient faculty for aiding in the causal discovery of performance anomalies. One approach that would be useful would time slice the statistics further, and would couple them with timewise reports of events that indicate a performance issue, such as long or frequent GC pauses. By coupling this tool with a performance visualizaton tool like Ganglia [86], there would be an opportunity to visualize the instrumented performance of a method along with other indicators such as JVM garbage

collection epochs and stats, network/disk/memory bandwith utilization, CPU utilization, and other performance indicators.

Additionally, another dimension that would be useful to pair the performance instrumentation code with would be the primary key associated with the record being processed during an instrumented function invocation. This is particularly important for genomics, as many issues with work imbalance are due to pathological regions in the genome where a repeat or a section of low-complexity sequence cause several orders of magnitude increased coverage. It might be possible to use a statistical model that predicts performance [146] to isolate key-specific/skew-driven performance issues.

10.2 Extensions to Avocado

There are several extensions that we are considering to the Avocado variant caller. In this dissertation, we have discussed Avocado's INDEL reassembly and single sample germline genotyping methods. There is ongoing work to add more joint variant calling methods to Avocado, such as a pedigree-aware caller (currently only supports mother/father/offspring trios). In this section, we discuss support for incremental joint variant calling, tumor-normal variant calling, and methods for merging the output of multiple variant callers.

Incremental Joint Variant Calling

In a gVCF-based joint variant calling workflow, variant discovery is done per sample, prior to a joint genotyping phase. In discovery, gVCF files are generated that contain the likelihoods of the reference allele at all sites in the genome, and likelihoods for alternate alleles observed in that sample. In the joint genotyping phase, the gVCFs of all samples are then used to generate a prior probability for the variant being called as an allele in the population. While this process allows the expensive gVCF creation process to be done only once per sample, the joint genotyping must be repeated every time the cohort is updated.

This is unnecessarily expensive, since genotype priors are generally based upon allele frequency and should be stable as long as the composition of the cohort remains uniform as the cohort grows. Ideally, instead of recomputing the prior using the whole population, we could make an update to the current prior using the new samples that we have added. For some statistical models, there are well known techniques that can provide provably correct approaches [125] or approximate results [90].

One possible shortcoming of the gVCF approach is that it may result in reduced sensitivity when detecting INDEL variants. Specifically, for INDELs, longer variants that benefit from realignment or reassembly may be missed. However, if these variants are later discovered in a sample with strong support for the variant, we may be able to discover them in a sample that previously called the INDEL as a negative by realigning the reads that overlap the site. As such, we believe that it would be worth investigating this problem, either in the context of a pipeline that discovers all INDEL variants across all samples before joint

calling variants, or by reviewing the read evidence in every prior called sample whenever a novel INDEL variant is observed in the cohort.

Somatic Variant Calling

Although the reassembly algorithm introduced in §7.1 can run on data that does not have known copy number, the Avocado genotyper described in §7.2 requires known copy number to call a variant. While this genotyping model can be used as a first step in a variant caller that assumes a popuation structure, it is not compatible with somatic variant calling. In somatic mutation calling, we are looking for the variants that appear in one sequencing run from a sample that do not appear in another sequencing run from a sample. Typically, this technique is used to call mutations in a tumor, and it would be desirable to support somatic variant calling using Apache Spark. We have worked to port the Mutect [29] algorithm over to Avocado, but have not completed this effort.

One critical part of somatic variant calling is coming up with a statistical model that addresses low allele frequency variation, which is caused by tumor subclonality. Many tools address this by calling variants that are supported by a small number of high quality bases. One possible way to address this issue would be with an approach that mixes contamination estimation [30] with base quality recalibration. Specifically, we could use an advanced form of base quality recalibration to try to capture sequence specific error models, while using the germline variants as the known variants for masking. Then, we could use contamination estimation to try to identify a background rate of sequence contamination. We have implemented an ADAM-based variant of the ContEst [30] tool in Quinine, which is the Big Data Genomics' quality control tool. From here, we would need to merge these two approaches into a novel probabilistic model.

Efficient Consensus Methods

To improve the accuracy of variant calls generated from a single set of aligned reads, some pipelines use a consensus method [165]. In this approach, several variant callers are used, and a variant is accepted as a true positive if multiple variant callers confidently call the variant. While this approach can improve the accuracy of variant calling, it necessitates running multiple variant calling pipelines and can substantially increase computational cost.

One approach that may be tractible is to use a machine learning approach to run a limited consensus approach. A possible implementation might use a setup similar to the GATK's [41] variant quality score recalibration (VQSR) algorithm, which fits a Gaussian mixture model to annotated genotypes. To do this, we would use a read dataset that has good ground truth data, such as one of the GIAB truth sets [165]. We would then run the variant callers that we are interested in, and annotate all the variant calls as either true or false positives. Using the variant calling annotations on each called genotype, we would then train a statistical model that explained when a variant caller would call a variant correctly.

To use this model to decrease concensus calling runtime, we would use the tool that was the most sensitive as a first pass. Once we had called variants with this tool, we would look at all sites that appeared to be weak false positives, and would run one or more variant caller that the trained model indicated would perform well on these sites. This approach would also need the first variant caller to either emit a gVCF or to emit calls for all possibly variant sites. One natural extension would be to optimize for runtime. Specifically, if two variant callers have similar sensitivity, but one of the two variant callers achieves this sensitivity with significantly lower runtime, we would likely want to use the faster variant caller as our first pass tool. This variant of the problem may be more difficult to optimize, as the amount of time needed to call a variant may depend on the complexity of the variant and the reads around the variant site [17].

10.3 Hardware Acceleration for Genomic Data Processing

As we increase the I/O bandwidth that can be achieved by running many tasks in parallel across a horizontally scalable file system, single thread performance becomes an issue. As we have seen with the Cannoli tool, the JVM can provide good performance, but it is difficult to outperform C/C++ from within the JVM. However, the marginal performance gain from moving to a lower level language is unclear. To gain an even higher performance gain, we should consider moving to a hardware accelerator, which can be highly specialized to our specific application patterns [12]. There has already been commercial work towards building hardware accelerated solutions for genomic data analysis [44], as well as academic work towards accelerating the BWA alignment algorithm [25, 26] and alignment for de novo assembly [142].

To this end, we have been working on the Darwin project, which implements the preprocessing algorithms introduced in Chapter 6 in a library which can be synthesized to target both field programmable gate arrays (FPGAs) and application specific integrated circuits (ASIC). The FPGA form factor is especially desirable as commercial cloud vendors like AWS [124] and Microsoft Azure [114] have moved to support FPGAs in the datacenter. This library provides a 2–10× performance improvement over ADAM's read preprocessing library. In the immediate short term, we will continue to validate the algorithms in Genie, and will work to add variant calling.

While FPGAs hold promise for accelerating complex and computationally intensive genomic data analysis pipelines, they face the same accessibility problem as Apache Spark, which was described in Chapter 5. To address this problem, we have used the Q100 [155] architecture, which uses a SQL-like programming methodology to drive a high-level synthesis workflow. This greatly increases the accessibility of hardware design, which is currently restricted to people who are willing to learn a hardware description language, which typically have programming patterns that are divergent from software design [13]. One possible direc-

tion for future work is to use the Apache Spark SQL library to drive a high level synthesis flow, as the SQL query captured by the query engine could be turned into synthesizable hardware. This would make the programming model accessible to a larger group of programmers.

One significant remaining question is how to process user defined functions (UDFs) that are specified outside of the SQL dialect. UDFs cannot typically be optimized by a query planner [11, 67], but are critical for genomic data analysis. While the duplicate marker and base recalibrator can be implemented in Genie with modest extensions to SQL, the INDEL realigner is implemented as a monolithic UDF. One option is to use a dataflow methodology [16], which would map the dataflow described by a program into hardware. This would be compatible with a purely functional programming methodology that relies on list comprehensions, which is a idiomatic programming style in the Scala [104] language. This approach could be used to synthesize hardware from a UDF, which would make it increasingly possible to implement genomic analysis tasks in hardware.

Chapter 11

Conclusion

Part VI

Appendix

Appendix A

ADAM Schemas

A.1 Alignment Record Schema

Listing A.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;
}
```

A.2 Fragment Schema

Listing A.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 = [];
}
```

A.3 Variation Schemas

Listing A.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;
}
```

Variant Annotation Schemas

Listing A.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 = {};
}
```

Genotype Annotation Schema

Listing A.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 = {};
}
```

A.4 Feature Schema

Listing A.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 = {};
}
```

Bibliography

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

[2] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[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] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the International Conference on Management of Data (SIGMOD '15)*, pages 331–346. ACM, 2015.

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

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

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

[11] 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 International Conference on Management of Data (SIGMOD)*, 2015.

[12] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[13] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference (DAC '12)*, pages 1216–1225. ACM, 2012.

[14] 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.

[15] 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.

[16] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet. OpenDF: A dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, 2008.

[17] 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.

[18] 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.

[19] 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.

[20] N. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature biotechnology*, 34(5):525, 2016.

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

[22] Cancer Genome Atlas Research Network. Genomic and epigenomic landscapes of adult de novo acute myeloid leukemia. *New England Journal of Medicine*, 2013(368):2059–2074, 2013.

[23] 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.

[24] 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*, 18(9):906, 2016.

[25] 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 '16)*, June 2016.

[26] 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 '15)*, May 2015.

[27] 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.

[28] 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.

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

[30] K. Cibulskis, A. McKenna, T. Fennell, E. Banks, M. DePristo, and G. Getz. ContEst: Estimating cross-contamination of human samples in next-generation sequencing data. *Bioinformatics*, 27(18):2601–2602, 2011.

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

[32] 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.

[33] 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.

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

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

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

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

[47] 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.

[48] 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 medicine*, 6:89, 2014.

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

[50] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS '00)*, pages 390–398. IEEE, 2000.

[51] 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.

[52] M. Fromer, J. L. Moran, K. Chambert, E. Banks, S. E. Bergen, D. M. Ruderfer, R. E. Handsaker, S. A. McCarroll, M. C. O'Donovan, M. J. Owen, et al. Discovery and statistical genotyping of copy-number variation from whole-exome sequencing depth. *The American Journal of Human Genetics*, 91(4):597–607, 2012.

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

[54] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*, pages 1–11. IEEE, 2015.

[55] 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 (SC '14)*, pages 437–448. IEEE Press, 2014.

[56] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. merAligner: A fully parallel sequence aligner. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '15)*, pages 561–570. IEEE, 2015.

[57] 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.

[58] 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.

[59] 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 *Proceedings of the Symposium on Operating System Design and Implementation (OSDI '14)*, volume 14, pages 599–613, 2014.

[60] 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.

[61] 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 *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI '11)*, volume 11, pages 22–22, 2011.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] S. Leo and G. Zanetti. Pydoop: A Python MapReduce and HDFS API for Hadoop. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC '10)*, pages 819–825. ACM, 2010.

[75] 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.

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

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

[78] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

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

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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

[85] 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.

[86] M. L. Massie, B. N. Chun, and D. E. Culler. The gadenglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[87] 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.

[88] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the Python in Science Conference (SciPy '10)*, volume 445, pages 51–56. SciPy Austin, TX, 2010.

[89] 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.

[90] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[91] 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.

[92] 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.

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

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] A. Nellore, L. Collado-Torres, A. E. Jaffe, J. Alquicira-Hernández, C. Wilks, J. Pritt, J. Morton, J. T. Leek, and B. Langmead. Rail-RNA: Scalable analysis of RNA-seq splicing and coverage. *Bioinformatics*, 2016.

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

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] OpenCB. HPG BigData. `https://github.com/opencb/hpg-bigdata`.

[107] 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.

[108] 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.

[109] 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.

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

[111] 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.

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

[113] 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.

[114] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA '14)*, pages 13–24. IEEE, 2014.

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

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

[117] 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.

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

[119] 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.

[120] H. Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.

[121] 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.

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

[123] 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.

[124] A. W. Services. EC2 F1 instances with FPGAs – Now generally available. `https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/`, April 2017.

[125] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

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

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

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

[133] 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.

[134] 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.

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

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

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

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

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

[140] 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.

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

[142] Y. Turakhia, K. J. Zheng, G. Bejerano, and W. J. Dally. Darwin: A hardware-acceleration framework for genomic sequence alignment. *bioRxiv*, page 092171, 2017.

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

[144] 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.

[145] 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.

[146] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI '16)*, pages 363–378. USENIX Association, 2016.

[147] 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.

[148] 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.

[149] 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.

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

[151] 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, and Cancer Genome Atlas Research Network. The Cancer Genome Atlas pan-cancer analysis project. *Nature genetics*, 45(10):1113–1120, 2013.

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

[153] 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.

[154] R. Williams. Spark-BAM. https://github.com/hammerlab/spark-bam.

[155] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 49(4):255–268, 2014.

[156] 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.

[157] 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.

[158] 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.

[159] 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.

[160] 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.

[161] 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 *Proceedings of the International Conference on Big Data (BigData '15)*, pages 918–927. IEEE, 2015.

[162] 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.

[163] Z. Zhang, E. R. Sparks, and M. J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*, pages 143–153. ACM, 2017.

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

[165] 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.