Scalable Genome Resequencing with ADAM and avocado

Frank Nothaft

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-65
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-65.html

May 12, 2015

Scalable Genome Resequencing with `ADAM` and `avocado`

by

Frank Austin Nothaft

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2015

Abstract

Scalable Genome Resequencing with `ADAM` and `avocado`

by

Frank Austin Nothaft

Master of Science in Computer Science

University of California, Berkeley

Professor David Patterson, Chair

The decreased cost of genome sequencing technologies has made genome sequencing a viable tool for clinical and populations genomics applications. The efficiency of genome sequencing has been further improved through large projects like the Human Genome Project, which have assembled reference genomes for medically/agriculturally important organisms. These reference quality assemblies have enabled the creation of *genome resequencing pipelines*, where the genome of a single sample is computed by computing the *difference* between a given sample and the reference genome for the organism.

While sequencing cost has decreased by more than 10,000× since the Human Genome Project concluded in 2003, resequencing pipelines have struggled to keep pace with the growing volume of genomic data. These tools suffer from limited parallelism because they were not designed to use parallel or distributed computing techniques, and are limited by asymptotically inefficient algorithms. In this thesis, we introduce two tools, `ADAM` and `avocado`. `ADAM` provides an efficient framework for performing distributed genomic analyses, and `avocado` implements efficient local reassembly to discover genomic variants. `ADAM` presents high level APIs that allow for genomic analyses to be parallelized across more than 1,000 processors. Using these APIs, we are able to achieve linear speedups when parallelizing several common analysis stages.

Contents

Chapter 1

Variant Identification Pipelines for Genomic Data

1.1 Introduction

Since the completion of the Human Genome Project in 2003, genome sequencing costs have dropped by more than $10,000\times$ [41]. The rapidly declining cost of sequencing a single human genome has enabled large sequencing projects like the 1,000 Genomes Project [52] and the Cancer Genome Atlas (TCGA, [60]). As these large sequencing projects perform analysis that process terabytes to petabytes of genomic data, they have created a demand for genomic analysis tools that can efficiently process these scales of data [48, 55].

Over a similar time range, commercial needs led to the development of horizontally scalable analytics systems. The development and deployment of `MapReduce` at Google [13, 14] spawned the development of a variety of distributed analytics tools and the `Hadoop` ecosystem [4]. In turn, these systems led to other systems that provided a more fluent programming model [62] and higher performance [65]. The demand for these systems has been driven by the increase in the amount of data available to analysts, and has coincided with the development of statistical systems that are accessible to non-experts, such as `Scikit-learn` [46] and `MLI` [54].

With the rapid drop in the cost of sequencing a genome, and the accompanying growth in available data, there is a good opportunity to apply modern, horizontally scalable analytics systems to genomics. New projects such as the 100K for UK, which aims to sequence the genomes of 100,000 individuals in the United Kingdom [21], and the Department of Veterans Affairs' Million Veteran project [45] will generate three to four *orders of magnitude* more data than prior projects like the 1,000 Genomes Project [52]. Additionally, periodic releases of new reference datasets such as reference genomes necessitates the periodic re-analysis of these large datasets. These projects use the current "best practice" genomic variant calling pipelines [6], which take approximately 120 hours to process a single, high-quality human genome using a single, beefy node [56]. To address these challenges, scientists have started to apply computer systems techniques such as map-reduce [29, 36, 49] and columnar storage [19] to custom scientific compute/storage systems. While these systems have improved analysis cost and performance, current implementations incur significant overheads imposed by the legacy formats and codebases that they use.

In this thesis, we demonstrate `ADAM`, a genomic data processing and storage system built using Apache `Avro`, `Parquet`, and `Spark` [3, 5, 65], and `avocado`, a variant caller built on top of `ADAM`. This pipeline is parallel and achieves a 28× increase in throughput over the current best practice pipeline, while reducing analysis cost by 66%. In the process of creating `ADAM`, we developed a "narrow waisted" layering model for building scientific analysis systems. This narrow waisted stack is inspired by the OSI model for networked systems [66]. However, in our stack model, the data schema is the narrow waist that separates data processing from data storage. Our stack solves the following three problems that are common across current scientific analysis systems:

1. Current scientific systems improve the performance of common patterns by changing the data model (often by requiring data to be stored in a coordinate-sorted order).

2. Legacy data formats were not designed with horizontal scalability in mind.

3. The system must be able to efficiently access shared metadata, and to slice datasets for running targeted analyses.

We solve these problems with the following techniques:

1. We make a schema the "narrow waist" of our stack to enforce data independence and devise algorithms for making common genomics patterns fast.

2. To improve horizontal scalability, we use `Parquet`, a modern parallel columnar store based off of Dremel [37] to push computation to the data.

3. We use a denormalized schema to achieve O(1) parallel access to metadata.

We introduce the stack model in Figure 1.1 as a way to decompose scientific systems.

While the abstraction inversion used in genomics to accelerate common access patterns is undesirable because it violates data independence, we also find that it sacrifices performance and accuracy. The current Sequence/Binary Alignment and Map (SAM/BAM [33]) formats for storing genomic alignments apply constraints about record ordering to enable specific computing patterns. Our implementation (described in §1.3) identifies errors in two current genomics processing stages that occur *because* of the sorted access invariant. Our implementations of these stages do not make use of sort order, and achieve higher performance *while* eliminating these errors.

Additionally, this thesis describes the variant discovery and genotyping algorithms implemented in `avocado`. `avocado` introduces a new algorithm for local reassembly that eliminates

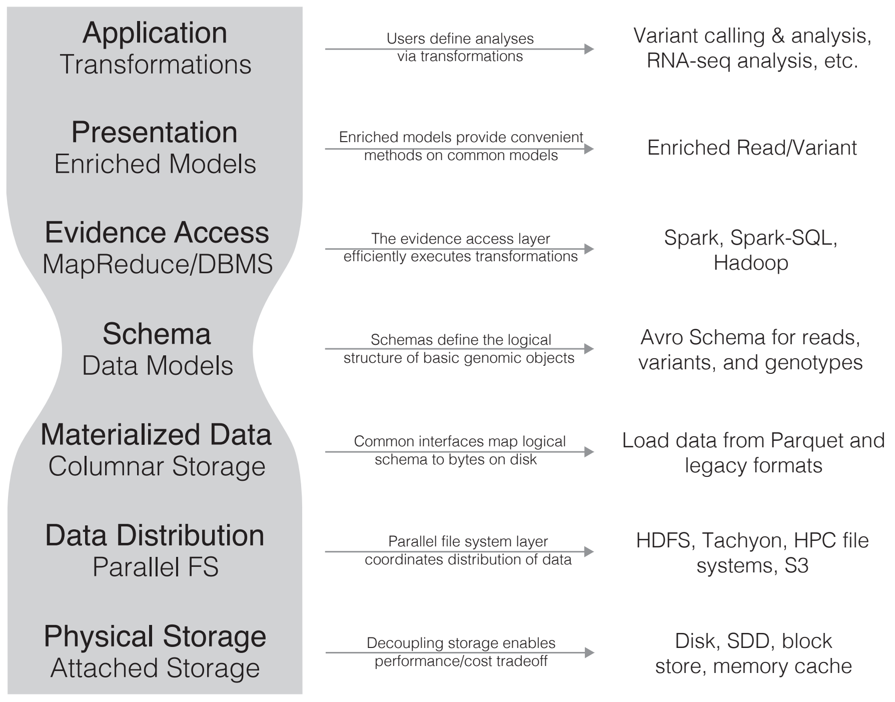
Application
Transformations
Users define analyses via transformations
Variant calling & analysis, RNA-seq analysis, etc.
Presentation
Enriched Models
Enriched models provide convenient methods on common models
Enriched Read/Variant
Evidence Access
MapReduce/DBMS
The evidence access layer efficiently executes transformations
Spark, Spark-SQL, Hadoop
Schema
Data Models
Schemas define the logical structure of basic genomic objects
Avro Schema for reads, variants, and genotypes
Materialized Data
Columnar Storage
Common interfaces map logical schema to bytes on disk
Load data from Parquet and legacy formats
Data Distribution
Parallel FS
Parallel file system layer coordinates distribution of data
HDFS, Tachyon, HPC file systems, S3
Physical Storage
Attached Storage
Decoupling storage enables performance/cost tradeoff
Disk, SDD, block store, memory cache

Figure 1.1: A Stack Model for Genomic Analyses

the expensive step of realigning reads to candidate haplotypes. Additionally, `avocado` introduces a novel statistical model for genotyping that eliminates errors caused by statistical models that optimistically assume the local independence of genomic loci.

All of the software (source code and executables) described in this thesis are available free of charge under the permissive Apache 2 open-source license. `ADAM` is available at `https://www.github.com/bigdatagenomics/adam`, and `avocado` is available at `https://www.github.com/bigdatagenomics/avocado`.

1.2 Background

This work is at the intersection of computational biology, data management, and processing systems. As such, our architectural approach is informed by recent trends in these areas. The design of large scale data management systems has changed dramatically since the papers by Dean and Ghemawat [13, 14] describing Google's `MapReduce` system. Over a similar timeframe, genomics has arisen due to improvements in data acquisition technologies. For example, since the Human Genome Project finished in 2001 [28], the price of genomic sequencing has dropped by 10,000× [41]. This drop in cost has enabled the capture of petabytes of sequence data, which has (in turn) enabled significant population genomics experiments like the 1,000 Genomes project [52] and The Cancer Genome Atlas (TCGA, [60]).

Although there has been significant progress in the development of systems for processing large datasets—the development of first generation map-reduce systems [13], followed by iterative map-reduce systems like `Spark` [65], as well as parallel and columnar DBMS [1, 27]—the uptake of these systems in genomics has been slow. `MapReduce`'s impact has been limited to tools that use the map-reduce programming model as an inspiration for API design [36], or have been limited systems that have used `Hadoop` to naïvely parallelize existing toolkits [29, 49]. These approaches are perilous for several reasons:

- A strong criticism levied against the map-reduce model is that the API is insufficiently expressive for describing complex tasks. As a consequence of this, tools like the GATK [36] that adopt map-reduce as a programming model force significant restrictions on algorithm implementors. For example, a GATK `walker`[1] is provided with a single view over the data (a sorted iterator over a specified region), and is allowed limited reduce functionality.
- A major contribution of systems like `MapReduce` [14] and `Spark` [65, 64] is the ability to reliably distribute parallel tasks across a cluster in an automated fashion. While the GATK uses map-reduce as a programming abstraction (i.e., as an interface for writing `walkers`), it does not use map-reduce as an execution strategy. To run tools like the GATK across a cluster, organizations use workflow management systems for sharding and persisting intermediate data, and managing failures and retries. This approach is not only an inefficient duplication of work, but it is also a source of inefficiency during execution: the performance of iterative stages in the GATK is bottlenecked by I/O performance.
- The naïve `Hadoop`-based implementations in Crossbow [29] and Cloudburst [49] use scripts to run unmodified legacy tools on top of `Hadoop`. This approach does achieve speedups, but it does not attack overhead. Several of the methods that they parallelize incur high overhead due to duplicated loading of indices[2] and poor broadcasting of data.

Recent work by Diao et al [16] has looked at optimizations to map-reduce systems for processing genomic data. They adapt strategies from the query optimization literature to reorder computation to minimize data shuffling. While this approach does improve shuffle traffic, several preprocessing stages cannot be transposed. For instance, reversing the order of indel realignment and base quality score recalibration (see §1.3) will change the inferred quality score distribution. Additionally, we believe that the shuffle traffic that Diao et al observe is an artifact caused by an abstraction inversion present in many genomics tools. This abstraction inversion requires that all genomic data is processed in sorted order, which

[1]The GATK provides `walkers` as an interface for traversing regions of the genome.

[2]For fast aligners, loading of large indices can be a primary I/O bottleneck.

necessitates frequent shuffles. As we demonstrate in §1.3, these penalties can be eliminated by restructuring the pre-processing algorithms.

One notable area where modern data management techniques have been leveraged by scientists is in the data storage layer. Due to the storage costs of large genomic datasets, scientists have introduced the CRAM format that uses columnar storage techniques and special compression algorithms to achieve a 30% reduction in size over the original BAM format [19]. While CRAM achieves high ($\gg$ 50%) compression, it imposes restrictions on the ordering and structure of the data, and does not provide support for predicates or projection. We perform a more comprehensive comparison against CRAM in §4.2.

One interesting trend of note is the development of databases specifically for scientific applications. The exemplar is SciDB, which provides an array based storage model as well as efficient linear algebra routines [10]. While arrays accelerate many linear algebra based routines, they are not a universally great fit. For many genomics workloads, data is semistructured and may consist of strings, Boolean fields, and an array of tagged annotations. Other systems like the Genome Query Language [25] have extended SQL to provide efficient query semantics across genomic coordinates. While GQL achieves performance improvements of up to 10$\times$ for certain algorithms, SQL is not an attractive language for many scientific domains, which make heavy use of user designed functions that may be cumbersome in SQL.

1.3 Pipeline Structure

This thesis targets the acceleration of *variant calling*, which is a statistical process to infer the sites at that a single individual varies from the reference genome.[3] Although there are a variety of sequencing technologies in use, the majority of sequence data used for variant calling and genotyping comes from the Illumina sequencing platform, which uses a "sequencing-by-synthesis" technique to generate short read data [38]. Short read refers to sequencing runs that generate many reads that are between 50 and 250 bases in length. In addition to adjusting the length of the reads, we can control the amount of the data that is generated by changing the amount of the genome that we sequence, or the amount of redundant sequencing that we perform (the average number of reads that covers each base, or *coverage*). A single human genome sequenced at 60$\times$ coverage will produce approximately 1.4 billion reads, which is approximately 600 GB of raw data, or 225 GB of compressed data. For each read, we also are provided *quality scores*, which represent the likelihood that the base at a given position was observed. In a variant calling pipeline, we perform the following steps:

1. **Alignment:** For each read, we find the position in the genome that the read is most likely to have come from. As an exact search is too expensive, there has been an extensive amount of research that has focused on indexing strategies for improving alignment performance [32, 34, 63]. This process is parallel per sequenced read.

[3]The reference genome represents the "average" genome for a species. The Human Genome Project [28] assembled the first human reference genome.

2. **Pre-processing:** After reads have been aligned to the genome, we perform several preprocessing steps to eliminate systemic errors in the reads. This transformation may involve recalibrating the observed quality scores for the bases, or locally optimizing the read alignments. We will present a description of several of these algorithms in §1.3; for a more detailed discussion, we refer readers to DePristo et al [15].

3. **Variant calling:** Variant calling is a statistical process that uses the read alignments and the observed quality scores to compute whether a given sample matches or diverges from the reference genome. This process is typically parallel per position or region in the genome.

4. **Filtration:** After variants have been called, we want to filter out false positive variant calls. We may perform queries to look for variants with borderline likelihoods, or we may look for clusters of variants, which may indicate that a local error has occurred. This process may be parallel per position, may involve complex traversals of the genomic coordinate space, or may require us to fit a statistical model to all or part of the dataset.

This process is very time consuming to run; the current best practice pipeline uses the BWA tool [32] for alignment and the GATK [15, 36] for pre-processing, variant calling, and filtration. Current benchmark suites have measured this pipeline as taking between 90 and 130 hours to run end-to-end [56]. Recent projects have achieved 5–10$\times$ improvements in alignment and variant calling performance [47, 63], which makes the pre-processing stages the performance bottleneck. Our experimental results have corroborated this, as the `GATK`'s four pre-processing stages take over 110 hours to run on a clinical quality human genome when run on an Amazon EC2 `cr1.8xlarge` machine.

For current implementations of these read processing steps, performance is limited by disk bandwidth [16]. This bottleneck exists because the operations read in a SAM/BAM file, perform a small amount of processing, and write the data to disk as a new SAM/BAM file. We achieve a performance bump by performing our processing iteratively in memory. The four read processing stages can then be chained together, eliminating three long writes to disk and an additional three long reads from disk. Additionally, by rethinking the design of our algorithms, we are able to reduce overhead in several other ways:

1. Current algorithms require the reference genome to be present on all nodes. This assembly is then used to look up the reference sequence that overlaps all reads. The reference genome is several gigabytes in size, and performing a lookup in the reference genome can be costly due to its size. Instead, wherever possible, we leverage the `mismatchingPositions` field in our schema to embed information about the reference in each read. This optimization allows us to avoid broadcasting the reference, and provides O(1) lookup. When this is not possible, we make use of a "region join" primitive, which enables us to make use of the reference genome with *minimal* duplication.

2. Shared-memory genomics applications tend to be impacted significantly by false sharing of data structures [63]. Instead of having data structures that are modified in parallel, we restructure our algorithms so that we only touch data structures from a single thread, and then merge structures in a reduce phase. The elimination of sharing improves the performance of covariate calculation during BQSR and the target generation phase of local realignment.

3. In a naïve implementation, the local realignment and duplicate marking tasks can suffer from stragglers. The stragglers occur due to a large amount of reads that either do not associate to a realignment target, or that are unaligned. We pay special attention to these cases by manually randomizing the partitioning for these reads. This resolves load imbalance and mitigates stragglers.

4. For the Flagstat command, we are able to project a limited subset of fields. Flagstat touches fewer than 10 fields, which account for less than 10% of space on disk. We discuss the performance implications of this further in §4.2.

These techniques allow us to achieve a $> 28\times$ performance improvement over current tools, and scalability beyond 128 machines. We perform a detailed performance review in §4.1.

1.4 Related Work

Several variant analysis toolkits exist, with the most well known analysis toolkit being the `GATK` [15]. Additional toolkits include `HugeSeq` [26], `STORMSeq` [23], and `SpeedSeq` [11]. These tools combine alignment, variant calling, and filtration into an easy to use package, and may also orchestrate work distribution across a set of distributed machines. For example, the `GATK` and `HugeSeq` make use of an improvised map-reduce model, while `STORMSeq` uses a grid engine to distribute work according to a provided partitioning function. These tools delegate to either the `GATK`'s `HaplotypeCaller` or `UnifiedGenotyper`, or `FreeBayes` [20] for calling germline point events and INDELs. `Platypus` [47] is an additional notable toolkit that directly integrates alignment with variant calling to improve computational efficiency.

Although earlier methods such as the `mpileup` caller assumed the statistical independence of sites [30] post-alignment, current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [31]. These methods take two different approaches to generate candidate sequences for realignment:

1. *Realignment-only:* Putative INDELs are extracted directly from the aligned reads, and the reads are locally realigned.

2. *Reassembly:* The aligned reads are reassembled into haplotypes, which the reads are aligned against.

The realignment-only approach is used in `UnifiedGenotyper`[4] and `FreeBayes`, while `HaplotypeCaller`, `Platypus`, and `Scalpel` [40] make use of reassembly. In both cases, we perform the following algorithmic steps:

1. Candidate haplotypes are generated for realignment.
2. Each read is realigned to each haplotype, typically using a pair Hidden Markov Model (HMM, see [17]).
3. A statistical model uses the read$\leftrightarrow$haplotype alignments to choose the haplotype pair that most likely represents the variants hypothesized to exist in the region.
4. The alignments of the reads to the chosen haplotype pair are used to generate statistics that are then used for genotyping.

In haplotype reassembly, step 1 is broken down into two further steps:

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

In both the *realignment* and *reassembly* 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. These approaches are algorithmically different from *global alignment* because they make use of local context when picking the sequence to align to, and the alignment search space is much smaller, which enables the use of more expensive alignment methods.

De novo assembly provides another promising approach to variant discovery. In the *de novo* formulation of assembly, the reads are not aligned to a reference genome. Instead, "novel" contiguous fragments of sequence are assembled from the reads. Variants are called by aligning these assemblies to the reference genome, and by realigning the reads against the novel assemblies. Several implementations of *de novo* variant calling exist, most notably the `Cortex` [22] and `Discovar` [61] assemblers. Although *de novo* assembly solves several important issues seen by traditional variant callers (reference bias, structural variant detection), *de novo* assembly is currently too computationally expensive for widespread use in genotyping.

[4]When used following `IndelRealignment`

Chapter 2

Genomic Data Storage and Preprocessing Using `ADAM`

2.1 Distributed Architectures for Genomics

Due to both the growing volume of genomic sequencing data and the large size of these datasets, sequencing centers face the increasingly difficult task of turning around genomic data analyses in a reasonable timeframe [48, 55]. While the per-run latency of current genomic pipelines such as the `GATK` can be improved by manually partitioning the input dataset and distributing work, native support for distributed computing is not provided. As a stopgap solution, projects like `Cloudburst` [49] and `Crossbow` [29] have ported individual analytics tools to run on top of `Hadoop`. While this approach has served well for proofs of concept, it provides poor abstractions for application developers and makes it difficult to create novel distributed genomic analyses, and does little to attack sources of inefficiency or incorrectness in distributed genomics pipelines.

To address these problems, we need to reconsider how to build software for processing genomic data. Modern genome analysis pipelines are built around monolithic architectures and flat file formats. These architectures are designed to efficiently run current algorithms on single node processing systems, but impose significant restrictions. These restrictions include:

- These implementations are locked to a single node processing model. Even the `GATK`'s "map-reduce" styled `walker` API [36] 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 `Hadoop` or `Spark` [64].
- 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 §2.4 and footnote 1.

- Additionally, while these invariants are intended to improve performance, it is not clear that these invariants *actually* improve performance. There are two common sort invariants used in genomics: sort by reference position and sort by read name. Changing between these two sort orders entails a full shuffle and resort of the dataset. Additionally, a sort is required after alignment to establish a sort order.

As noted above, current implementations are locked to a single node model. Projects like `Hadoop-BAM` [42], `SeqPig` [50], and `BioPig` [43] have attempted to build a distributed processing environment on top of current single node genomics APIs. However, several problems must be solved in order to make distributed processing of genomic data productive:

- Current genomics data formats rely on a centralized header for storing experiment metadata. Since the metadata is centralized, it must be replicated to all machines.
- A simple map-reduce or SQL-like API is insufficient for implementing genomic analyses. Rather, to enhance bioinformatician productivity, we need to define APIs that allow developers to conveniently express algorithms.

In `ADAM`, we have taken a more aggressive approach to the design of APIs for processing genomic data in a distributed system. Although modern genomics pipelines are built as monolithic applications, we have chosen a layered decomposition for `ADAM`, that uses a schema as a “narrow waist”. Instead of using a flat file format, as is traditional in genomics, we are using this schema with `Parquet` (a commodity columnar store [5]) to store genomic data in a way that both allows efficient distributed read/write and that achieves high compression. On top of this, we have added primitives that implement common genomic traversals in a distributed manner. We have then used `ADAM` to implement common preprocessing stages from commonly used genomics pipelines. `ADAM`’s preprocessing stages are between 1-5$\times$ faster than the equivalent `GATK` preprocessing stages, and achieve linear scaling out to 128 nodes.

2.2 Layering

The processing patterns being applied to scientific data shift widely as the data itself ages. Because of this change, we want to design a scientific data processing system that is flexible enough to accommodate our different use cases. At the same time, we want to ensure that the components in the system are well isolated so that we avoid bleeding functionality across the stack. If we bleed functionality across layers in the stack, we make it more difficult to adapt our stack to different applications. Additionally, as we discuss in §2.4, 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 [66]. 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.

Unlike conventional scientific systems that leverage custom data formats like BAM or SAM [33], or CRAM [19], we believe that the use of an explicit schema for data interchange is critical. In our stack model shown in Figure 1.1, the schema becomes the "narrow waist" of the stack. Most importantly, placing the schema as the narrow waist enforces a strict separation between data storage/access and data processing. Additionally, this enables literate programming techniques which can clarify the data model and access patterns. The seven layers of our stack model are decomposed as follows, and are numbered in ascending order from bottom to top:

1. **Physical Storage:** This layer coordinates data writes to physical media.
2. **Data Distribution:** This layer manages access, replication, and distribution of the files that have been written to storage media.
3. **Materialized Data:** This layer encodes the patterns for how data is encoded and stored. This layer determines I/O bandwidth and compression.
4. **Data Schema:** This layer specifies the representation of data, and forms the narrow waist of the stack that separates access from execution.
5. **Evidence Access:** This layer provides us with primitives for processing data, and allows us to transform data into different views and traversals.
6. **Presentation:** This layer enhances the data schema with convenience methods for performing common tasks and accessing common derived fields from a single element.
7. **Application:** At this level, we can use our evidence access and presentation layers to compose the algorithms to perform our desired analysis.

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

As we discuss in more detail in §2.4, 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.[1] These views of evidence should be implemented at the evidence access layer instead of in the layout of data on disk. This split enforces independence of anything below the schema.

The idea of decomposing scientific applications into a stack model is not new; Bafna et al [7] 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 [25]. 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 [66] 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.

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

2.3 Data Storage for 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 [12], SAM [33], BAM, and CRAM [19]. 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 [5] to materialize data on disk, the Apache Avro [3] 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. First, the schemas are fully denormalized, which reduces the cost of metadata access and simplifies metadata distribution. We are able to get these benefits without greatly increasing the cost of memory access because our backing store (Parquet) makes use of run length and dictionary encoding, which allows for a single object to be allocated for highly repetitive elements on read. 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 [35, 44].

We have reproduced the schemas used to describe reads, variants, and genotypes below. ADAM also contains schemas for describing assembled contigs, genomic features, and variant annotations, but we have not included them in this section.

Listing 2.1: ADAM read schema

```
record AlignmentRecord {
  /** Alignment position and quality */
  Contig contig;
  long start;
  long oldPosition;
  long end;

  /** read ID, sequence, and quality */
  string readName;
  string sequence;
  string qual;

  /** alignment details */
  string cigar;
  string oldCigar;
  int mapq;
  int basesTrimmedFromStart;
  int basesTrimmedFromEnd;
  boolean readNegativeStrand;
  boolean mateNegativeStrand;
  boolean primaryAlignment;
  boolean secondaryAlignment;
  boolean supplementaryAlignment;
  string mismatchingPositions;
  string origQual;

  /** Read status flags */
  boolean readPaired;
  boolean properPair;
  boolean readMapped;
  boolean mateMapped;
  boolean firstOfPair;
  boolean secondOfPair;
  boolean failedVendorQualityChecks;
  boolean duplicateRead;

  /** optional attributes */
  string attributes;

  /** record group metadata */
  string recordGroupName;
  string recordGroupSequencingCenter;
  string recordGroupDescription;
  long recordGroupRunDateEpoch;
  string recordGroupFlowOrder;
  string recordGroupKeySequence;
  string recordGroupLibrary;
  int recordGroupPredictedMedianInsertSize;
  string recordGroupPlatform;
  string recordGroupPlatformUnit;
  string recordGroupSample;

  /** Mate pair alignment information */
  long mateAlignmentStart;
  long mateAlignmentEnd;
  Contig mateContig;
}
```

Our read schema maps closely to the logical layout of data presented by `SAM` and `BAM`. The main modifications relate to how we represent metadata, which has been denormalized across the record. All of the metadata from the sequencing run and prior processing steps are packed into the record group metadata fields. The program information describes the processing lineage of the sample and is expected to be uniform across all records, thus it compresses extremely well. The record group information is not guaranteed to be uniform across all records, but there are a limited number number of record groups per sequencing dataset. This metadata is string heavy, which benefits from column-oriented decompression and makes proper deserialization from disk important. 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 increases the amount of memory allocated and the garbage collection (GC) load.

Listing 2.2: `ADAM` variant and genotype schemas

```
enum StructuralVariantType {
  DELETION,
  INSERTION,
  INVERSION,
  MOBILE_INSERTION,
  MOBILE_DELETION,
  DUPLICATION,
  TANDEM_DUPLICATION
}

record StructuralVariant {
  StructuralVariantType type;
  string assembly;

  boolean precise;
  int startWindow;
  int endWindow;
}

record Variant {
  Contig contig;
  long start;
  long end;

  string referenceAllele;
  string alternateAllele;
  StructuralVariant svAllele;

  boolean isSomatic;
}

enum GenotypeAllele {
  Ref,
  Alt,
  OtherAlt,
  NoCall
}

record VariantCallingAnnotations {
  float variantCallErrorProbability;

  array<string> variantFilters;

  int readDepth;
  boolean downsampled;
  float baseQRankSum;
  float clippingRankSum;
  float fisherStrandBiasPValue = null;
  float haplotypeScore;
  float inbreedingCoefficient;
  float rmsMapQ;
  int mapq0Reads;
  float mqRankSum;
  float variantQualityByDepth;
  float readPositionRankSum;

  array<int> genotypePriors;
  array<int> genotypePosteriors;

  float vqslod;
  string culprit;
  boolean usedForNegativeTrainingSet;
  boolean usedForPositiveTrainingSet;

  map<string> attributes;
}

record Genotype {
  Variant variant;
  VariantCallingAnnotations variantCallingAnnotations;

  string sampleId;
  string sampleDescription;
  string processingDescription;

  array<GenotypeAllele> alleles;

  float expectedAlleleDosage;
  int referenceReadDepth;
  int alternateReadDepth;
  int readDepth;
  int minReadDepth;
  int genotypeQuality;
  array<int> genotypeLikelihoods;
  array<int> nonReferenceLikelihoods;
  array<int> strandBiasComponents;

  boolean splitFromMultiAllelic;

  boolean isPhased;
  int phaseSetId;
  int phaseQuality;
}
```

The variant and genotype schemas present a larger departure from the representation used by the Variant Call Format (VCF). The most noticeable difference is that we have migrated away from VCF's variant oriented representation to a matrix representation. Instead of the variant record serving to group together genotypes, the variant record is embedded within the genotype. Thus, a record represents the genotype assigned to a sample, as opposed to a VCF row, where all individuals are collected together. The second major modification is to assume a biallelic representation[2]. This differs from VCF, which allows multiallelic records. By limiting ourselves to a biallelic representation, we are able to clarify the meaning of many of the variant calling annotations. If a site contains a multiallelic variant (e.g., in VCF parlance this could be a 1/2 genotype), we split the variant into two or more biallelic records. The sufficient statistics for each allele should then be computed under a reference model similar to the model used in genome VCFs. If the sample does contain a multiallelic variant at the given site, this multiallelic variant is represented by referencing to another record via the OtherAlt enumeration.

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

2.4 Read Preprocessing Algorithms

In ADAM, we have implemented the three most-commonly used pre-processing stages from the GATK pipeline [15]. In this section, we describe the stages that we have implemented,

[2]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.

and the techniques we have used to improve performance and accuracy when running on a distributed system. These pre-processing stages include:

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

   We have validated our duplicate removal code against Picard [58], which is used by the GATK for Marking Duplicates. Our implementation is fully concordant with the Picard/GATK duplicate removal engine, except we are able to perform duplicate marking for chimeric read pairs.[3] 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.[4] 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 (BQSR):** During the sequencing process, systemic errors occur that lead to the incorrect assignment of base quality scores. In this step, we label each base that we have sequenced with an *error covariate*. For each covariate, we count the total number of bases that we saw, as well as the total number of bases within the covariate that do not match the reference genome. From this data, we apply a correction by estimating the error probability for each set of covariates under a beta-binomial model with uniform prior.

   We have validated the concordance of our BQSR implementation against the GATK. Across both tools, only 5000 of the $\sim$180B bases ($< 0.0001\%$) in the high-coverage

[3] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [32].

[4] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al [15].

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.

BQSR Implementation

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

Our system is generic and places no limitation on the number or type of covariates that can be applied. A covariate describes a parameter space where variation in the covariate parameter may be correlated with a sequencing error. We provide two common covariates that map to common sequencing errors [39]:

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

To generate the covariate observation table, we aggregate together the number of observed and error bases per covariate. Algorithms 1 and 2 demonstrate this process.

Algorithm 1 Emit Observed Covariates

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

Algorithm 2 Create Covariate Table

```
$reads \leftarrow$ input dataset
$covariates \leftarrow$ covariates to use for recalibration
$sites \leftarrow$ known variant sites
$sites$.broadcast()
$observations \leftarrow reads$.map($read \Rightarrow$ emitObservations($read, covariates, sites$))
$table \leftarrow observations$.aggregate(CovariateTable(), mergeCovariates)
**return** $table$
```

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

Once we have computed the observations that correspond to each covariate, we estimate the observed base quality using equation (2.1). This represents a Bayesian model of the mismatch probability with Binomial likelihood and a Beta(1, 1) prior.

$$\mathbf{E}(P_{err}|cov) = \frac{\texttt{\#errors}(cov) + 1}{\texttt{\#observations}(cov) + 2} \tag{2.1}$$

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.

Indel Realignment Implementation

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

Realignment Target Identification

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

Convex-Hull Finding

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

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{2.2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{2.3}$$

$$\hat{R} \subset R \tag{2.4}$$

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 (2.2) is trivial to check: specifically, the genome is assembled out of reference contigs[5] 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 3, which is a data parallel algorithm for finding the maximal convex hulls that describe a genomic dataset.

Algorithm 3 Find Convex Hulls in Parallel

```
$data \leftarrow$ input dataset
$regions \leftarrow data.\text{map}(data \Rightarrow \text{generateTarget}(data))$
$regions \leftarrow regions.\text{sort}()$
$hulls \leftarrow regions.\text{fold}(r_1, r_2 \Rightarrow \text{mergeTargetSets}(r_1, r_2))$
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 4.

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:

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

Algorithm 4 Merge Hull Sets

```
  $first \leftarrow$ first target set to merge
  $second \leftarrow$ second target set to merge
Require: $first$ and $second$ are sorted
  if $first = \emptyset \wedge second = \emptyset$ then
    return $\emptyset$
  else if $first = \emptyset$ then
    return $second$
  else if $second = \emptyset$ then
    return $first$
  else
    if last($first$) $\cap$ head($second$) $= \emptyset$ then
      return $first + second$
    else
      $mergeItem \leftarrow$ (last($first$) $\cup$ head($second$))
      $mergeSet \leftarrow$ allButLast($first$) $\cup mergeItem$
      $trimSecond \leftarrow$ allButFirst($second$)
      return mergeTargetSets($mergeSet$, $trimSecond$)
    end if
  end if
```

- *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 [51].
- *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 [53] 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 (2.5).

$$q_{i,j} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j \in \{0, \dots, l_c - l_{r_i}\} \tag{2.5}$$

$$q_{i,R} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j = \mathrm{pos}(r_i|R) \tag{2.6}$$

$$q_i = \min(q_{i,R}, \min_{j \in \{0, \dots, l_c - l_{r_i}\}} q_{i,j}) \tag{2.7}$$

$$q_c = \sum_{r_i \in \mathbf{r}} q_i \tag{2.8}$$

In (2.5), $s(i)$ denotes the base at position i of sequence s, and l_s denotes the length of sequence s. We pick the consensus sequence that minimizes the q_c value. If the chosen consensus has a log-odds ratio (LOD) that is greater than 5.0 with respect to the reference, we realign the reads. This is done by recomputing the CIGAR and MDTag for each new alignment. Realigned reads have their mapping quality score increased by 10 in the Phred scale.

Duplicate Marking Implementation

Reads may be duplicated during sequencing, either due to clonal duplication via PCR before sequencing, or due to optical duplication while on the sequencer. To identify duplicated reads, we apply a heuristic algorithm that looks at read fragments that have a consistent mapping signature. First, we bucket together reads that are from the same sequenced fragment by grouping reads together on the basis of read name and record group. Per read bucket, we then identify the 5' mapping positions of the primarily aligned reads. We mark as duplicates all read pairs that have the same pair alignment locations, and all unpaired reads that map to the same sites. Only the highest scoring read/read pair is kept, where the score is the sum of all quality scores in the read that are greater than 15.

Chapter 3

Variant Calling via Reassembly Using avocado

3.1 Modular Approaches for Variant Calling

In this chapter, we present `avocado`, a variant caller built on top of `ADAM`. `avocado` has been designed to enable users to run an end-to-end variant caller that makes use of the `ADAM` preprocessing stages described in §2.4 along with state-of-the art variant calling methods, without needing to spill to disk. `avocado`'s general pipeline structure is shown in Figure 3.1.

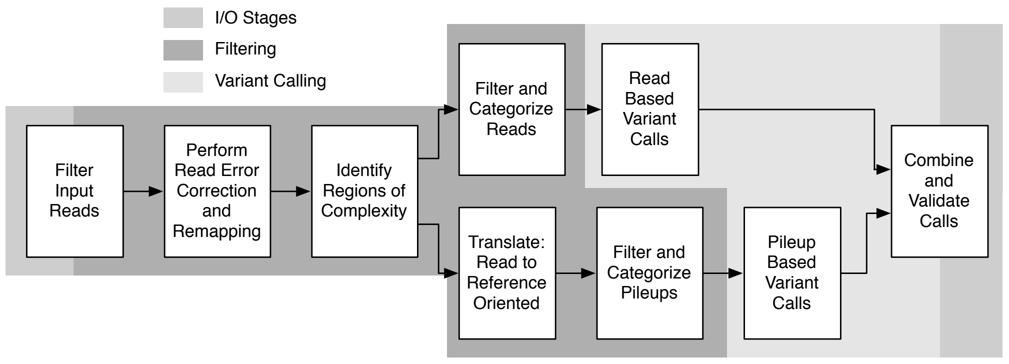
I/O Stages
Filtering
Variant Calling
Filter Input Reads
Perform Read Error Correction and Remapping
Identify Regions of Complexity
Filter and Categorize Reads
Read Based Variant Calls
Translate: Read to Reference Oriented
Filter and Categorize Pileups
Pileup Based Variant Calls
Combine and Validate Calls

Figure 3.1: The architecture of the `avocado` pipeline

In `avocado`, we support both read and reassembly based variant discovery, along with multiple statistical methods for genotyping. Although the methods described in this thesis target human germline variant calling, `avocado`'s local reassembly methods have been implemented so that they can support somatic variant calling, and variant calling on polyploid genomes.

In this chapter, we first talk about `avocado`'s novel local reassembler, which is able to reduce the computational complexity of local reassembly. This algorithmic reformulation also enables the reassembly of pooled/somatic and polyploid samples, as it makes no assumptions about path count through the assembly graph. We then cover the statistical model used for genotyping in `avocado`.

3.2 Efficient Reassembly via Indexed *de Bruijn* Graphs

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 [2]. Reassembly is a critical component of several prominent variant callers such as the Genome Analysis Toolkit's (GATK) `HaplotypeCaller` [15], `Scalpel` [40], and `Platypus` [47]. Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls [8], this accuracy comes at the cost of end-to-end runtime [56]. Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled [9] or by improving the performance of algorithms of the core algorithms used in local reassembly [47].

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 HMM
4. A statistical model uses the read$\leftrightarrow$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 section, we focus on steps one through three of the local reassembly problem, as there is wide variation in the algorithms used in stages four and five. Stage one (graph creation) has approximately $\mathcal{O}(rl_r)$ time complexity, and stage two (graph elaboration) has $\mathcal{O}(h \max(l_h))$ time complexity. The asymptotic time cost bound of local reassembly comes from stage three, where cost is $\mathcal{O}(hrl_r \max(l_h))$, where h is the number of haplotypes tested in this region[1], r is the number of reads aligned to this region, l_r is the read length[2], and $\min(l_h)$ is the length of the shortest haplotype that we are evaluating. This complexity comes from realigning r reads to h haplotypes, where realignment has complexity $\mathcal{O}(l_r l_h)$. We ignore the storage complexity of reassembly here, as our approach does not change the storage complexity of the graph.

In this section, 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.

Current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [31]. 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 earlier in this section. In non-assembly approaches like `FreeBayes` [20], 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 to varies between methods (e.g., the GATK's `IndelRealigner` uses a simple log-odds model [15], while methods like `FreeBayes` [20] and `Platypus` [47] 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 [31]. *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

[1]The number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see [15, 20]) allow users to limit the number of haplotypes evaluated to improve performance.

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

$\mathcal{O}(k(l_{\text{ref}} + l_{\text{variants}} + l_{\text{errors}}))$, where l_{ref} is the length of the reference haplotype in this region, l_{variants} is the length of true variant sequence in this region, l_{errors} is the length of erroneous sequence in this region, and k is the k-mer size. In practice, we can approximate both errors and variants as being random, which gives $\mathcal{O}(kl_{\text{ref}})$ storage complexity. From this graph, we must enumerate the haplotypes present in the graph. Starting from the first k-mer in the reference sequence for this region, we perform a depth-first search to identify all paths to the last k-mer in the reference sequence. Assuming that the graph is acyclic (a common restriction for local assembly), we can bound the best case cost of this search at $\Omega(h \min l_h)$.

The number of haplotypes evaluated, h, is an important contributor to the algorithmic complexity of reassembly pipelines, as it sets the storage and time complexity of the realignment scoring phase, the time complexity of the haplotype enumeration phase, and is related to the storage complexity of the *de Bruijn* graph. The best study of the complexity of assembly techniques was done by Kingsford et al. [24], 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 equation (3.1), 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.

$$h \sim f_v l_{\text{ref}} + \epsilon l_{\text{ref}} c \tag{3.1}$$

This model is naïve, as the coverage depth and rate of variation varies across sequenced datasets, especially for targeted sequencing runs [18]. 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 [15]. 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 [20], restricting realignment to INDELs (`IndelRealigner`, [15]), or by trimming paths from the assembly graph. Additionally, in an *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 (3.1) solely as a motivating approximation, and hope to study these characteristics in more detail in future work.

Formulation

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* $\textit{prefix}(s_1)^{\{1,k-2\}} + \textit{suffix}(s_1) = \textit{prefix}(s_2)$*.*

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 ($s_1 \to \cdots \to s_2$ 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 $\mathcal{O}(nd)$,[3]

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

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

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

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

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

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

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

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

To visualize the graph corresponding to a substitution, take the two example sequences `CCACTGT` and `CCAATGT`. These two sequences differ by a `C` $\leftrightarrow$ `A` edit at position three. With k-mer length $k = 3$, this corresponds to the graph in Figure 3.2.

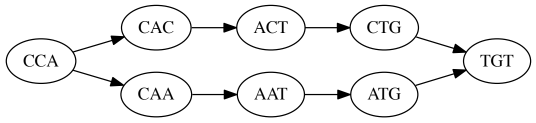

Figure 3.2: Subgraph Corresponding To a Single Nucleotide Edit

If $m_i = k-1, m_j \geq k$ or vice versa, we have a canonical INDEL edit (for convenience, we assume that $\mathcal{S}'_i$ contains the $k-1$ length path). Here, we can prove that there is a $m_j - m_i$ length insertion[4] in $\mathcal{S}'_j$ relative to $\mathcal{S}'_i$, $k-1$ letters *after* index(s_1):

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

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

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

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 3.3.

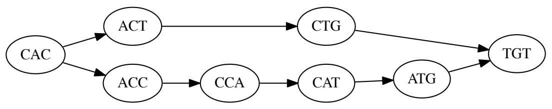

Figure 3.3: Subgraph Corresponding To a Canonical INDEL Edit

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 $s_1, \mathcal{L}_i, s_2$.[5] 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 [17, 53]), *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 [59]). By this, we can achieve $O(n(d + 1))$ cost.

Specialization for Local Reassembly

Thus far, we have assumed that we want to find the edits between two or more *known* sequences. However, when performing local reassembly for variant discovery/calling, our goal is to identify all possible variants and to associate probabilities to observations that contain these variants. These hypothesized variants are generated by examining the reads aligned to the reference at/near a given site.

However, we can adopt a "pooled" model that uses the *indexed de Bruijn* graph to discover alternate alleles without performing a search for all haplotypes. Here, we extract a substring $\mathcal{R}$ from a reference assembly, corresponding to the subsection of that reference that we would like to reassemble. Then, we create a pooled "sequence" $\mathcal{P}$, that is generated

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

from the k-mers present in the reads aligned to $\mathcal{R}$. However, since $\mathcal{P}$ is a composite of the pooled reads, we cannot assign indices to k-mers in $\mathcal{P}$. Instead, we will rely wholly on the path length properties demonstrated in §3.2 and the indices of k-mers in $\mathcal{R}$ to discover and anchor alleles. First, let us classify paths where $\mathcal{R}$ and $\mathcal{P}$ diverge into two types:

- **Spurs:** A spur is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where either s_1 or $s_n \in \mathcal{R}, \mathcal{P}$ and all other k-mers are $\notin \mathcal{R}, \in \mathcal{P}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$. If $s_1 \in \mathcal{R}$, then s_n must not have a successor. Alternatively, if $s_n \in \mathcal{R}$, than s_1 is required to not have a predecessor.
- **Bubbles:** A bubble is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where both s_1 and $s_n \in \mathcal{R}, \mathcal{P}$ and all other k-mers are $\notin \mathcal{R}, \in \mathcal{P}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$.

Currently, we disregard spurs. Spurs typically result from sequencing errors near the start or end of a read. Additionally, given a spur, we cannot put a constraint on what sort of edit it may be from the reference, which increases the computational complexity of processing the spur. We concede that this may not be the best approach, but plan to explore better options for for processing spurs in future work.

We can elaborate the graph and identify variants by walking the graph from the first k-mer in $\mathcal{R}$. Although haplotype elaboration algorithms have $\Omega(h \min l_h)$ cost where $\min l_h$ is the length of the shortest haplotype and h is the number of haplotypes described by the graph, we can limit our graph traversal to have $\mathcal{O}(n)$ runtime cost where $n = V(G)$ by introducing a tail-recursive finite state machine (FSM). Whenever we reach a branch point in the graph, our FSM will push state onto a stack, which allows us—with a few exceptions—to avoid making multiple traversals through a single k-mer in the graph. Our FSM has the following states:

- `R`, `Reference`: We are on a run of k-mers that are in $\mathcal{R}$.
- `A`, `Allele`: We are on a run of k-mers that have diverged off of the reference. We have a divergence start point, but have not connected back to the reference yet. This could be either a bubble or a spur.
- `C`, `ClosedAllele`: We were on a run of k-mers that had diverged off of the reference, but have just reconnected back to the reference and now know the start and end positions (in $\mathcal{R}$) of the bubble, as well as the non-reference sequence and length of the bubble.

We allow the FSM to make the following state transitions, which are depicted in Figure 3.4:

- `R` $\rightarrow$ `R`: We are continuing along a reference run.
- `R` $\rightarrow$ `A`: We were at a k-mer $\in \mathcal{R}$, and have seen a branch to a k-mer $\notin \mathcal{R}$.

- `A` $\rightarrow$ `A`: We are continuing along a non-reference run.
- `A` $\rightarrow$ `C`: We were on a non-reference run, and have just connected back to a reference k-mer.
- `C` $\rightarrow$ `R`: We have just closed out an allele, and are back at a k-mer $\in \mathcal{R}$.

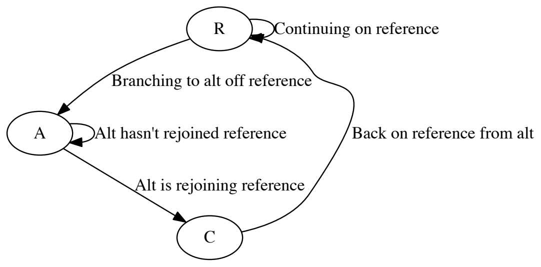
R
Continuing on reference
Branching to alt off reference
A
Alt hasn't rejoined reference
Back on reference from alt
Alt is rejoining reference
C

Figure 3.4: Finite State Machine for Pooled vs. Reference Assembly

We initialize the state machine to `R`, and start processing with the first k-mer from $\mathcal{R}$. Per k-mer, we evaluate the possible state transitions of each successor k-mer. If the set of successor k-mers contains a single k-mer, we continue to that state. If the successor set contains multiple k-mers, we choose a successor state to transition to, and push the branch context of all other successor states onto our stack. If the successor set is empty, we pop a branch context off of the stack, and switch to that context. We stop once we reach a k-mer whose successor set is empty, *and* our branch context stack is empty.

The implementation of the `R` and `A` states largely amount to bookkeeping. In the `R` state, we must track the current position in $\mathcal{R}$, and in `A`, we must record where we branched off of $\mathcal{R}$, and $\mathcal{L}'$, the bases we have seen since we branched[6] off of $\mathcal{R}$. If we are in the `A` state and walk to a k-mer $\in \mathcal{R}$, we then transition into the `C` state. Using the positions of our branch point and the current k-mer in $\mathcal{R}$, we are able to calculate the length of the reference path via Lemma 1, while the length of the non-reference path is given by $l_{\mathcal{L}'}$. The edited sequence $\mathcal{L}$ present between the branch point and the current k-mer is given by trimming the first $k-1$ bases from $\mathcal{L}'$. If we have a non-canonical edit, or a deletion in $\mathcal{P}$ relative to $\mathcal{R}$, we can look these bases up by indexing into $\mathcal{R}$.

To improve computational efficiency, we should emit statistics for genotyping *while* traversing the graph. By doing this, we can eliminate the need for realignment in haplotype reassembly. Since we have constructed a graph where every k-mer is anchored to a reference position or to a position in a variant, or is in a spur, if we log several statistics when processing reads from $\mathcal{P}$, we can directly emit the probability observations that support a reference base or a variant allele. Although the relevant statistics vary between different

[6]Although we are processing k-mers, we only need to reconstruct the sequence that the k-mers are describing by taking the first base from each successive k-mer.

variant calling/genotyping algorithms, common statistics include base and mapping quality, as well as strand of the observation. Additionally, genotyping algorithms that incorporate phasing may want to identify observations that came from the same allele. These statistics can be logged at a low cost when chopping the reads in $\mathcal{P}$ into k-mers.

As noted earlier in this section, we normally do *not* need to retrace through any k-mers in the graph when processing the graph. However, we may need to retrace k-mers if we have overlapping variants. For example, take the reference string `ACTGCCGTCT`, the SNV `ACTGCAGTCT`, and the complex variant `ACTGCAAGTCT`[7]. For $k = 4$, both variants share the 4-mer `AGTC`, but both take independent paths from the reference 4-mer `CTGC` to `AGTC`. In this case, we need to walk `AGTC` twice.

3.3 Statistical Models for Genotyping

The biallelic genotyping model in `avocado` is derived closely from the biallelic genotyping model used in the `samtools mpileup` variant caller [30]. However, we make several modifications. First, we apply a windowing approach to unify "complex" events. Second, we operate solely in log space. This is done in order to avoid the underflow issues discussed in §2.3.6 of Li 2011 [30].

Windowing Algorithm

The site independence assumption in `samtools mpileup` serves as a simplification for the statistical approaches that follow, but can be a source of error in the presence of INDELs and other complex events. As asserted earlier in this section, we believe that we can arrive at a canonical INDEL alignment via local reassembly. This canonicalization should resolve issues due to read evidence for an INDEL allele being misaligned across multiple reads. However, even after realignment, we cannot guarantee that we have observed the same alleles across *all* samples, nor can we guarantee that all alleles are at a single site on the reference genome. This is trivial to note: all deletions cover a *region* of the reference genome, not a single site. As such, grouping together alleles by start position is not sufficient for correctness. While performing a coordinate space self-join—as supported in `ADAM`—would be sufficient for *correctness*, it would not be performant due to the number of observations being scored in a large dataset. Instead, we apply the sweeping window approach given in Algorithm 5.

This algorithm maintains an open window that it uses to group together all observations that are in the window. It is worth noting that this is different from a self-join or a groupBy. Specifically, we group all observations into the smallest possible region that could fully contain those observations. If multiple observations from the same read are grouped together, we then combine the statistics from those observations, weighted by the number of bases from the read contributing to each respective observation.

[7]At `CCG`, we have inserted an `A` between the two `C`s, and changed the second `C` $\to$ `A`. Alternatively, this can be described as a deletion of the second `C` and an insertion of `AA` between `CG`.

Algorithm 5 Open Windows for Site Observations

```
$observations \leftarrow$ the set of allele observations
$contigs \leftarrow$ a description of the reference genome assembly
$observations \leftarrow observations$.repartitionAndSortWithinPartition($contigs$)
$sites \leftarrow \emptyset$
**for** $partition \in observations$ **do**
    $windowStart \leftarrow partition$.head.start
    $windowEnd \leftarrow partition$.head.end
    $site \leftarrow \emptyset$
    **for** $observation \in partition$ **do**
        **if** $windowEnd <= observation$.start **then**
            $site \leftarrow \{observation\}$
        **else**
            $window \leftarrow [windowStart, windowEnd)$
            $sites$.append(($window, site$))
            $site$.append($observation$)
        **end if**
    **end for**
    **if** $sites \neq \emptyset$ **then**
        $window \leftarrow [windowStart, windowEnd)$
        $sites$.append(($window, site$))
    **end if**
**end for**
**return** $sites$
```

Statistical Model

The biallelic genotyping model in `avocado` is derived directly from `mpileup` [30], except with a translation to log space. We calculate log likelihoods from the site observations, and use the major allele frequency (MAF)[8] as a prior when normalizing. We use equation (3.2) to calculate the log likelihood of the genotype state at a site. We define the terms used in equation (3.2) in Table 3.1.

$$l(g) = -k \log m + \sum_{i=1}^{j} \log(\epsilon_i(m-g) + (1-\epsilon_i)g) + \sum_{i=j+1}^{k} \log(\epsilon_i g + (1-\epsilon_i)(m-g)) \qquad (3.2)$$

We estimate the MAF via expectation-maximization. Equation (3.3) gives the update equation for n samples. c_s represents the contribution of sample s to the MAF, and $p_s(gl_{\text{MAF}})$ is the probability of genotype state g for sample s, given l_{MAF}. The MAF is treated as a binomial prior.

[8]The frequency of the reference allele as observed at this site across all samples.

Table 3.1: Equation (3.2) Terms

Term	Definition
m	The copy number of the site.
g	The *genotype state*, or number of reference alleles.
j	Reads $\in k$ that support the reference allele.
k	The total number of reads at the site
ϵ_i	The probability that observation i was in error.

$$l_{\text{MAF}} = s(c_1, \dots, c_n) \tag{3.3}$$
$$c_s = s(n_{s,i}, \dots) - s(d_{s,i}, \dots), i \in \{0, \dots, g\} \tag{3.4}$$
$$n_{s,i} = p_s(i|l_{\text{MAF}}) + \log i \tag{3.5}$$
$$d_{s,i} = p_s(i|l_{\text{MAF}}) \tag{3.6}$$
$$s(l_1, l_2) = l_1 + \log(1 + e^{\frac{l_2}{l_1}}) \tag{3.7}$$
$$s(l_1, l_2, \dots, l_n) = s(l_1, s(l_2, s(\dots s(l_{n-1}, l_n)))) \tag{3.8}$$

Equation (3.7) is a numerically stable algorithm for adding two log values to each other and returning a log result. This equation is from Durbin et al [17]. To sum together an array of log values, we compose equation (3.7) recursively into equation (3.8). In practice, to improve numerical stability we sort the array before performing the recursive sum.

Chapter 4

Performance and Accuracy Analysis

Thus far, we have discussed ways to improve the performance of scientific workloads that are being run on commodity map-reduce systems by rethinking how we decompose and build algorithms. In this section, we review the improvements in performance that we are able to unlock. `ADAM` achieves near-linear speedup across 128 nodes. Additionally, `ADAM` achieves 25-50% compression over current file formats when storing to disk.

4.1 Genomics Workloads

Table 4.1 previews our performance versus current systems. The tests in this table are run on the high coverage `NA12878` full genome BAM file that is available from the 1,000 Genomes project.[1] These tests have been run on the Amazon Web Services EC2 cloud, using the instance types listed in Table 4.2. We compute the cost of running each experiment by multiplying the number of instances used by the total wall time for the run and by the cost of running a single instance of that type for an hour, which is the process Amazon uses to charge customers.

Table 4.2 describes the instance types. Memory capacity is reported in Gibibytes (GiB). Storage capacities are not reported in this table because disk capacity does not impact performance, but the number and type of storage drives is reported because aggregate disk bandwidth does impact performance. In our tests, the `hs1.8xlarge` instance is chosen to represent a workstation. Network bandwidth is constant across all instances.

As can be seen from these results, our pipeline is at best three times faster than current pipelines when running on a single node; at worst, we are approximately at parity. Additionally, `ADAM` achieves speedup that is close to linear on a cluster. This point is not clear from Table 4.1, as we change instance types when also changing the number of instances used. Figure 4.1 presents speedup plots for the NA12878 high coverage genome.

[1]The file used for these experiments can be found on the 1,000 Genomes ftp site, `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/NA12878/high_coverage_alignment/` for NA12878.

Table 4.1: Summary Performance on NA12878

Tool	EC2	BQSR	IR	DM	Sort	FS	Total
[15]	1†	1283m	658m	—	—	—	2075m1
[33]	1†	—	—	509m	203m	54m41	
[57]	1†	—	—	44m50	83m	6m11	
[58]	1†	—	—	160m	562m	—	
`ADAM`	1†	1602m	366m	143m	108m	2m17	2221m17
		$^1/_{1.25}\times$	1.7×	$^1/_{3.8}\times$	$^1/_{1.3}\times$	2.7×	$^1/_{1.07}\times$
`ADAM`	32⋆	74m	64m	34m56	39m23	0m43	223m2
		17×	10×	1.2×	2.1×	8.6×	9.3×
`ADAM`	64⋆	41m52	35m39	21m35	18m56	0m49	118m51
		30×	18×	2.0×	4.3×	7.5×	17×
`ADAM`	128⋆	25m59	20m27	15m27	10m31	1m20	73m44
		49×	32×	2.9×	7.9×	4.3×	28×

Table 4.2: AWS Machine Types

	Machine	Cost	Description
†	`i2.8xlarge`	\$6.20	32 proc, 244G RAM, 8 SDD
⋆	`r3.2xlarge`	\$0.70	8 proc, 61G RAM, 1 SDD

When testing on NA12878, we achieve linear speedup out through 1024 cores; this represents 128 `m2.4xlarge` nodes. In this test, our performance is limited by several factors:

- Although columnar stores have very high read performance, their write performance is low. Our tests exaggerate this penalty; since a variant calling pipeline will consume a large read file, but then output a variant call file that is approximately two orders of magnitude smaller, the write penalty will be reduced. In practice, we also use in-memory caching to amortize write time across several stages.

- Additionally, for large clusters, straggler elimination is an issue. However, we have made optimizations to both the `Mark Duplicates` and `INDEL Realignment` code to eliminate stragglers by randomly rebalancing reads that are unmapped/do not map to a target across partitions.

We do note that the performance of `flagstat` degrades going from 256 to 1,024 cores. By increasing the number of machines we use to execute this query, we increase scheduling overhead, which leads to degraded performance.

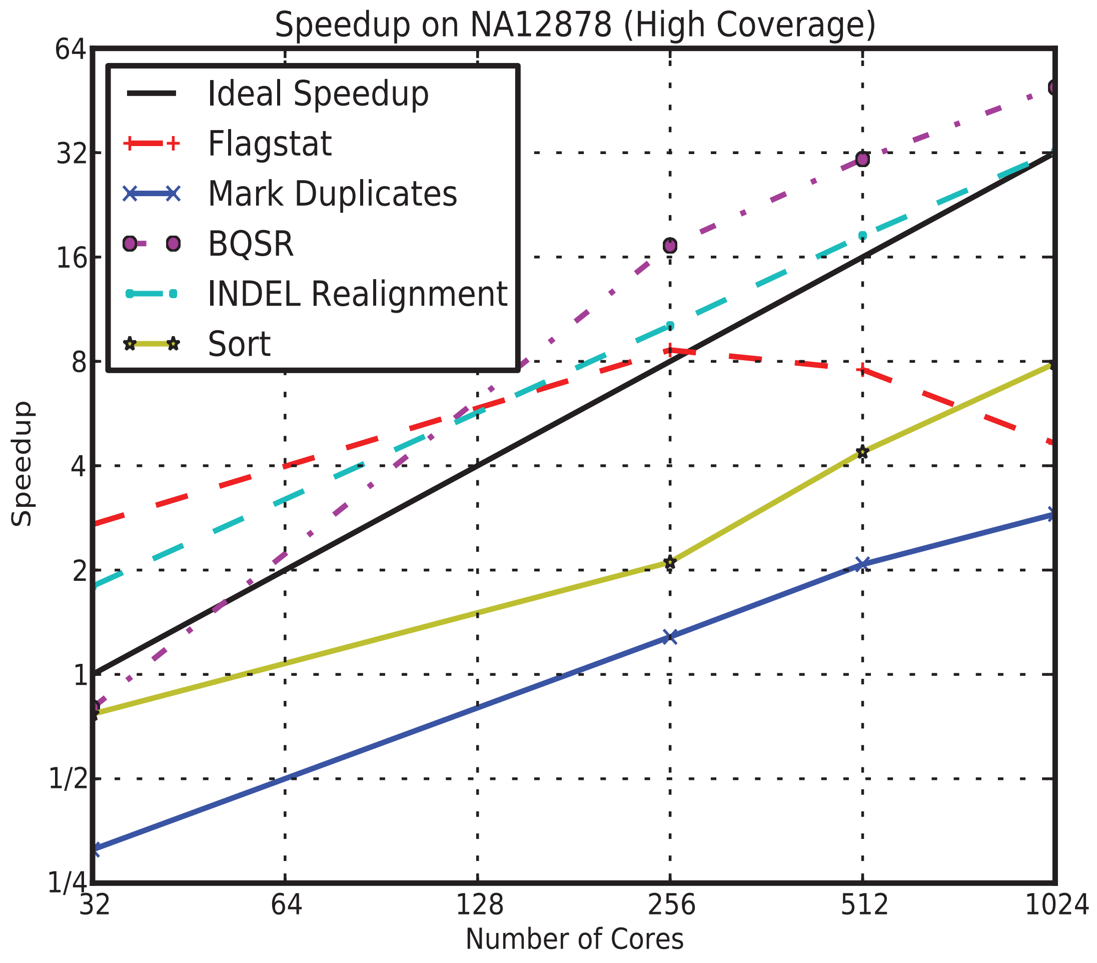

Figure 4.1: Speedup on NA12878

4.2 Column Store Performance

In §2.3, we motivated the use of a column store as it would allow us to better push processing to the data. Specifically, we can use predicate pushdown and projections to minimize the amount of I/O that we perform. Additionally, column stores provide compressed storage, which allows us to minimize both the required I/O bandwidth and space on disk. In this section, we'll look at the performance that our columnar store achieves in terms of read performance and compression. We will not look extensively at write performance; for genomic data, write performance is not a bottleneck because our workflow computes a *summarization* of a large dataset. As a result, our output dataset tends to be $\mathcal{O}(100\text{ MB})$ while our input dataset is in the range of $\mathcal{O}$(10 GB) to $\mathcal{O}$(100GB).

Compression

The `Parquet` columnar store [5] supports several compression features. Beyond traditional block-level compression, `Parquet` supports run length encoding for repeated values, dictionary encoding, and delta encoding. Currently, we make use of run length encoding to compress highly repeated metadata values, and dictionary encoding to compress fields that can take a limited range of values. Dictionary encoding provides substantial improvements for genomic data; specifically, the majority of genomic sequence data can be represented with three bits per base.[2] This size is an improvement over our in-memory string representation that allocates a byte per base.

In Table 4.3, we look at the compression we achieve on the `NA12878` and `HG00096`[3] human genome sequencing samples. We compare against the GZIP compressed BAM [33] format and the CRAM format [19]. We achieve approximately a 1.25× improvement in storage. This is not as impressive as the result achieved by the CRAM project, but the CRAM project applies specific compression techniques that make use of the read alignment. Specifically, CRAM only stores the read bases that *do not* appear in the reference genome. As a genomic variant is only expected at one in every 1,000 bases, and a read error at one in every 50 bases, this allows `CRAM` to achieve significant compression of the sequenced bases.

Table 4.3: Genomic Data Compression

	`NA12878`	
Format	**Size**	**Compression**
BAM	234 GB	—
CRAM	112 GB	2.08×
Parquet	185 GB	1.26×
	`HG00096`	
Format	**Size**	**Compression**
BAM	14.5 GB	—
CRAM	3.6 GB	4.83×
Parquet	11.4 GB	1.27×

For genomic datasets, our compression is limited by the sequence and base quality fields, which respectively account for approximately 30% and 60% of the space spent on disk. Quality scores are difficult to compress because they have high entropy. We plan to look into

[2]Although DNA only contains four bases (A, C, G, and T), *sequenced* DNA uses disambiguation codes to indicate that a base was read in error. As a result, we cannot achieve the ideal two-bits per base.

[3]A link to the `NA12878` dataset was provided earlier in this paper. The `HG00096` dataset is available from `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/HG00096/alignment/`.

computational strategies to address this problem; specifically, we are working to probabilistically estimate the quality scores *without* having observed quality scores. This estimate would be performed via a process that is similar to the base quality score recalibration algorithm presented earlier in this paper.

Horizontal Scalability

The representation `Parquet` uses to store data to disk is optimized for horizontal scalability in several ways. Specifically, `Parquet` is implemented as a hybrid row/column store where the whole set of records in a dataset are partitioned into row groups that are then serialized in a columnar layout. This hybridization provides us with two additional benefits:

1. We are able to perform parallel accesses to `Parquet` row groups without consulting metadata or checking for a file split.
2. `Parquet` achieves even balance across partitions. On the `HG00096` dataset, the average partition size was 105 MB with a standard deviation of 7.4 MB. Out of the 116 partitions in the file, there is only one partition whose size is not between 105–110MB.

`Parquet`'s approach is preferable when compared to `Hadoop-BAM` [42], a project that supports the direct usage of legacy BAM files in `Hadoop`. `Hadoop-BAM` must pick splits, which adds non-trivial overhead. Additionally, once `Hadoop-BAM` has picked a split, there is no guarantee that the split is well placed. It is only guaranteed that the split position will not cause a *functional error*.

Projection and Predicate Performance

We use the `flagstat` workload to evaluate the performance of predicates and projections in `Parquet`. We define three projections and four predicates, and test all of these combinations. In addition to projecting the full schema, we also use the following two projections:

1. We project the read sequence *and* all of the flags (40% of data on disk).
2. We only project the flags (10% of data on disk).

Beyond the null predicate (which passes every record), we evaluate the following three predicates:

1. We pass only uniquely mapped reads (99.06% of reads).
2. We pass only the first pair in a paired end read (50% of reads).
3. We pass only *un*mapped reads (0.94% of reads).

Table 4.4: Predicate/Projection Speedups

	0	1	2
0	—	1.7	1.9
1	1.0	1.7	1.7
2	1.3	2.2	2.6
3	1.8	3.3	4.4

Table 4.4 compares the performance of these projections. Projections are arranged in the columns of the table while predicates are assigned to rows.

We achieve a 1.7× speedup by moving to a projection that eliminates the deserialization of our most complex field (the quality scores that consume 60% of space on disk), while we only get a 1.3× performance improvement when running a predicate that filters 50% of records. This variation can be partially attributed to overhead from predicate pushdown; we must first deserialize a column, process the filter, and then read all records who passed the pushdown filter. If we did not perform this step, we could do a straight scan over all of the data in each partition.

Chapter 5

Conclusion

In this thesis, we have introduced the `ADAM` and `avocado` systems. These systems present a "green field" view of how we can store and process genomic datasets by using commodity, scalable analytics systems. In addition to developing systems that can be used to process genomic data on multiple nodes, we have used these systems to explore the programming interfaces provided to people who are developing genomic data processing algorithms. We believe that `ADAM` is a promising step towards a higher level data model and API for providing reference aligned genomics datasets.

5.1 Future Work

We are in the process of evaluating `avocado` on the 1,000 Genomes dataset. Our efforts on `avocado` are focused in two directions:

1. Improving system scalability. During the evaluation of `avocado`, we have identified areas in the `ADAM` API that need optimization. Specifically, we would like to explore general techinques for straggler mitigation. Although we have improved the performance of the local assembler through algorithmic enhancements, the region join that is used to create regions for reassembly suffers from stragglers due to uneven coverage across the genome. Additionally, we believe that we can improve `avocado`'s performance by eliminating unnecessary shuffles. There is an unnecessary shuffle between the reassembly and genotyping phase caused by a sort invariant in the windowing function (see §3.3).

2. Additionally, we are interested in improving the statistical models used for variant calling. Although `avocado` implements local assembly, we can further improve our accuracy by moving to a haplotype-aware genotyping model. Additionally, since our local assembler can handle pooled samples, we hope to add a statistical model that supports calling somatic variants.

Once we have added support for somatic variant calling, we plan to use `avocado` to re-call the TCGA [60] to stress-test the performance and scalability of the system. Beyond using such a large dataset to stress-test the performance of our systems, we believe that this provides us with a good opportunity to evaluate the design choices made by the analytics systems that we have built on (Apache `Spark`). While many large scale workloads run on top of the `Hadoop` ecosystem are proprietary, our workload and data are publically available. This enables us to perform a review of the performance characteristics of these systems at scale, without being encumbered by a proprietary system.

5.2 Conclusion

In this thesis, we have advocated for an architecture for decomposing the implementation of a scientific system, and then demonstrated how to efficiently implement genomic processing pipelines using the open source `Avro`, `Parquet`, and `Spark` systems [3, 5, 65]. We have identified common characteristics across genomics systems, like the need to run queries that touch slices of datasets and the need for fast access to metadata. We then enforced data independence through a layering model that uses a schema as the "narrow waist" of the stack, and used optimizations to make common, coordinate-based processing fast. By using `Parquet`, a modern columnar store, we use predicates and projections to minimize I/O, and are able to denormalize our schemas to improve the performance of accessing metadata. By rethinking the architecture of scientific data management systems, we have been able to achieve a 28× performance improvements over conventional genomics processing systems, along with linear strong scaling and a 63% cost improvement.

Bibliography

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data (SIGMOD '06)*, pages 671–682. ACM, 2006.

[2] 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.

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

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

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

[6] G. A. Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, et al. From FastQ data to high-confidence variant calls: The Genome Analysis Toolkit best practices pipeline. *Current Protocols in Bioinformatics*, pages 11–10, 2013.

[7] 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.

[8] 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.

[9] 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*, pages 20–34. Springer, 2014.

[10] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 963–968. ACM, 2010.

[11] 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. *bioRxiv*, page 012179, 2014.

[12] 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.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*. ACM, 2004.

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

[15] 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.

[16] Y. Diao, A. Roy, and T. Bloom. Building highly-optimized, low-latency pipelines for genomic data analysis. In *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR '15)*, 2015.

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

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

[19] 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.

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

[21] Genomics England. 100,000 genomes project. `https://www.genomicsengland.co.uk/`.

[22] 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.

[23] K. J. Karczewski, G. H. Fernald, A. R. Martin, M. Snyder, N. P. Tatonetti, and J. T. Dudley. STORMSeq: An open-source, user-friendly pipeline for processing personal genomics data in the cloud. *PloS one*, 9(1):e84860, 2014.

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

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

[26] H. Y. Lam, C. Pan, M. J. Clark, P. Lacroute, R. Chen, R. Haraksingh, M. O'Huallachain, M. B. Gerstein, J. M. Kidd, C. D. Bustamante, et al. Detecting and annotating genetic variations using the HugeSeq pipeline. *Nature biotechnology*, 30(3):226–229, 2012.

[27] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] Y. Li, A. Terrell, and J. M. Patel. WHAM: A high-throughput sequence alignment method. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, SIGMOD '11, pages 445–456, New York, NY, USA, 2011. ACM.

[35] 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.

[36] 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.

[37] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[38] M. L. Metzker. Sequencing technologies—the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 2015.

[45] U. D. of Veterans Affairs. Million veteran program (mvp). `http://www.research.va.gov/mvp`.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[48] 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.

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

[50] 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.

[51] 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.

[52] N. Siva. 1000 genomes project. *Nature Biotechnology*, 26(3):256–256, 2008.

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

[54] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for distributed machine learning. In *13th IEEE International Conference on Data Mining (ICDM '13)*, pages 1187–1192. IEEE, 2013.

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

[56] 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.

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

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

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

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

[61] N. I. Weisenfeld, S. Yin, T. Sharpe, B. Lau, R. Hegarty, L. Holmes, B. Sogoloff, D. Tabbaa, L. Williams, C. Russ, et al. Comprehensive variation discovery in single human genomes. *Nature genetics*, 2014.

[62] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[63] 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.

[64] 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 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association, 2012.

[65] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing (HotCloud '10)*, page 10, 2010.

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