ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing

Matt Massie
Frank Nothaft
Christopher Hartl
Christos Kozanitis
André Schumacher
Anthony D. Joseph
David A. Patterson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-207
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-207.html

December 15, 2013

ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing

Matt Massie[1], Frank Austin Nothaft[1], Christopher Hartl[1,2], Christos Kozanitis[1], André Schumacher[3], Anthony D. Joseph[1], and David Patterson[1]

[1]Department of Electrical Engineering and Computer Science, University of California, Berkeley
[2]The Broad Institute of MIT and Harvard
[3]International Computer Science Institute (ICSI), University of California, Berkeley

Executive Summary

Current genomics data formats and processing pipelines are not designed to scale well to large datasets. The current Sequence/Binary Alignment/Map (SAM/BAM) formats were intended for single node processing [18]. There have been attempts to adapt BAM to distributed computing environments, but they see limited scalability past eight nodes [22]. Additionally, due to the lack of an explicit data schema, there are well known incompatibilities between libraries that implement SAM/BAM/Variant Call Format (VCF) data access.

To address these problems, we introduce ADAM, a set of formats, APIs, and processing stage implementations for genomic data. ADAM is fully open source under the Apache 2 license, and is implemented on top of Avro and Parquet [5, 26] for data storage. Our reference pipeline is implemented on top of Spark, a high performance in-memory map-reduce system [32]. This combination provides the following advantages: 1) Avro provides explicit data schema access in C/C++/C#, Java/Scala, Python, php, and Ruby; 2) Parquet allows access by database systems like Impala and Shark; and 3) Spark improves performance through in-memory caching and reducing disk I/O.

In addition to improving the format's cross-platform portability, these changes lead to significant performance improvements. On a single node, we are able to speedup sort and duplicate marking by 2×. More importantly, on a 250 Gigabyte (GB) high (60×) coverage human genome, this system achieves a 50× speedup on a 100 node computing cluster (see Table 1), fulfilling the promise of scalability of ADAM.

The ADAM format provides explicit schemas for read and reference oriented (pileup) sequence data, variants, and genotypes. As the schemas are implemented in Apache Avro—a cross-platform/language serialization format—they eliminate the need for the development of language-specific libraries for format decoding/encoding, which eliminates the possibility of library incompatibilities.

A key feature of ADAM is that any application that implements the ADAM schema is compatible with ADAM. This is important, as it prevents applications from being locked into a specific tool or pattern. The ADAM stack is inspired by the "narrow waist" of the Internet Protocol (IP) suite (see Figure 2). We consider the explicit use of a schema in this format to be the greatest contribution of the ADAM stack.

In addition to the advantages outlined above, ADAM eliminates the file headers in modern genomics formats. All header information is available inside of each individual record. The variant and genotype formats also demonstrate two significant improvements. First, these formats are co-designed so that variant data can be seamlessly calculated from a given collection of sample genotypes. Secondly, these formats are designed to flexibly accommodate annotations without cluttering the core variant/genotype schema. In addition to the benefits described above, ADAM files are up to 25% smaller on disk than compressed BAM files without losing any information.

The ADAM processing pipeline uses Spark as a compute engine and Parquet for data access. Spark is an in-memory MapReduce framework which minimizes I/O accesses. We chose Parquet for data storage as it is an open-source columnar store that is designed for distribution across multiple computers with high compression. Additionally, Parquet supports efficient methods (predicates and projections) for accessing only a specific segment or fields of a file, which can provide significant (2-10×) additional speedup for genomics data access patterns.

1 Introduction

Although the cost of data processing has not historically been an issue for genomic studies, the falling cost of genetic sequencing will soon turn computational tasks into a dominant cost [21]. The process of transforming reads from alignment to variant-calling ready reads involves several processing stages including duplicate marking, base score quality recalibration, and local realignment. Currently, these stages have involved reading a Sequence/Binary Alignment Map (SAM/BAM) file, performing transformations on the data, and writing this data back out to disk as a new SAM/BAM file [18].

Because of the amount of time these transformations spend shuffling data to and from disk, these transformations have become the bottleneck in modern variant calling pipelines. It currently takes three days to perform these transformations on a high coverage BAM file[1].

Byrewriting these applications to make use of modern in-memory MapReduce frameworks like Apache Spark [32], these transformations can now be completed in under two hours. Sorting and duplicate mapping alone can be accelerated from 1.5 days on a single node to take less than one hour on a commodity cluster.

Table 1 previews the performance of ADAM for sort and duplicate marking.

Table 1: Sort and Duplicate Marking for NA12878

Software	EC2 profile	Wall Clock Time
Picard 1.103	1 `hs1.8xlarge`	17h 44m
ADAM 0.5.0	1 `hs1.8xlarge`	8h 56m
ADAM 0.5.0	32 `cr1.8xlarge`	33m
ADAM 0.5.0	100 `m2.4xlarge`	21m
Software	**EC2 profile**	**Wall Clock Time**
Picard 1.103	1 `hs1.8xlarge`	20h 22m
ADAM 0.5.0	100 `m2.4xlarge`	29m

The format we present also provides many programming efficiencies, as it is easy to change evidence access techniques, and it is directly portable across many programming languages.

In this paper, we introduce ADAM, which is a programming framework, a set of APIs, and a set of data formats for cloud scale genomic processing. These frameworks and formats scale efficiently to modern cloud computing performance, which allows us to parallelize read translation steps that are required between alignment and variant calling. In this paper, we provide an introduction to the data formats (§2) and pipelines used to process genomes (§3), introduce the ADAM formats and data access application programming interfaces (APIs) (§5) and the programing model (§6). Finally, we review the performance and compression gains we achieve (§7), and outline future enhancements to ADAM on which we are working (§8).

2 Current Genomics Storage Standards

The current de facto standard for the storage and processing of genomics data in read format is BAM. BAM is a binary file format that implements the SAM standard for read storage [18]. BAM files can be operated on in several languages, using APIs either from SAMtools[18] (in C++), Picard[3] (in Java), and Hadoop-BAM[22] (in Hadoop MapReduce via Java).

BAM provides more efficient access and compression than the SAM file format, as its binary encoding reduces several fields into a single byte, and eliminates text processing on load. However, BAM has been criticized because it is difficult to process. For example, the three main APIs that implement the format each note that they do not fully implement the format due to its complexity. Additionally, the file format is difficult to use in multiprocessing environments due to its use of a centralized header; the Hadoop-BAM implementation notes that it's scalability is limited to distributed processing environments of less than eight machines.

In response to the growing size of sequencing files[2], a variety of compression methods have come to light [16, 11, 27, 23, 6, 9, 30, 14, 13]. SlimGene [16], cSRA [28], and CRAM [11] use reference based compression techniques to losslessly represent reads. However, they advocate in favor of lossy quality value representations. The former two use lower quantization levels to represent quality values and CRAM uses user defined budgets to store only fractions of a quality string. In the same spirit, Illumina recently presented a systematic approach of quality score removal in [13] which safely ignores quality scores from predictable nucleotides; these are bases that always appear after certain words. It is also

[1] A 250GB BAM file at 30× coverage. See §7.2 for a longer discussion

[2] High coverage BAM files can be approximately 250 GB for a human genome.

worth mentioning that the standard configurations of cSRA and CRAM discard the optional fields of the BAM format and also simplify the QNAME field.

3 Genomic Processing Pipelines

After sequencing and alignment, there are a few common steps in modern genomic processing pipelines for producing variant call ready reads. Figure 1 illustrates the typical pipeline for variant calling. These steps minimize the amount of erroneous data in the input read set by eliminating duplicate data, calibrating the quality scores assigned to bases (base quality score recalibration (BQSR), and verifying the alignment of short inserts/deletions (indels).

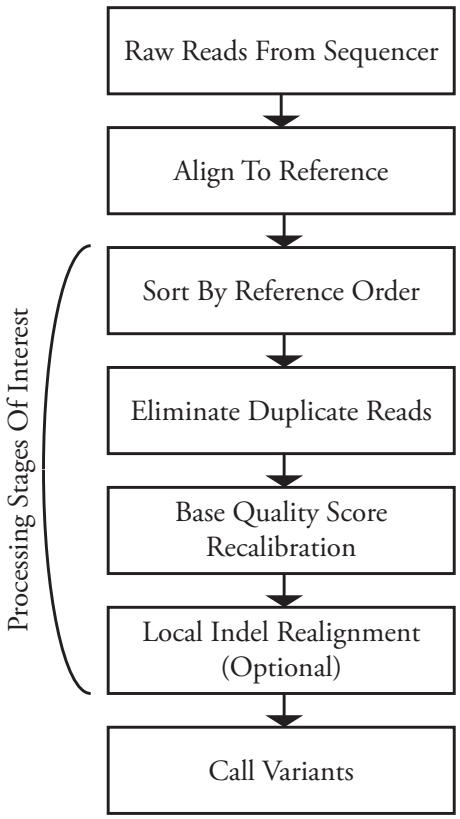

Figure 1: Variant Calling Pipeline

Traditionally, bioinformaticians have focused on improving the accuracy of the algorithms used for alignment and variant calling. There is obvious merit to this approach—these two steps are the dominant contributors to the accuracy of the variant calling pipeline. However, the intermediate read processing stages are responsible for the majority of execution time. Table 2 shows a breakdown of stage execution time for the version 2.7 of the Genome Analysis Toolkit (GATK), a popular variant calling pipeline [19]. The numbers in the table came from running on the NA12878 high-coverage human genome.

Table 2: Processing Stage Times for GATK Pipeline

Stage	**GATK 2.7/NA12878**
Mark Duplicates	13 hours
BQSR	9 hours
Realignment	32 hours
Call Variants	8 hours
Total	**62 hours**

To provide the readers with a background about how the stages of this pipeline work, we discuss the four algorithms that implement the intermediate read processing stages. For a detailed discussion of these algorithms, we refer readers to DePristo et al [10].

1. **Sorting:** This phase performs a pass over the reads and sorts them by the reference position at the start of their alignment.
2. **Duplicate Removal:** An insufficient number of sample molecules immediately prior to polymerase chain reaction (PCR) can cause the generation of duplicate DNA sequencing reads. Detection of duplicate reads requires matching all reads by their $5'$ 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 top-scoring read are marked as duplicates. Approximately two-thirds of duplicates marked in this way are true duplicates caused by PCR-induced duplication while the remaining third are caused by the random distribution of read ends[8]. There is currently no way to separate "true" duplicates from randomly occurring duplicates.
3. **Base Quality Score Recalibration:** During the sequencing process, systemic errors occur that lead to the incorrect assignment of base quality scores. In this step, a statistical model of the quality scores is built and is then used to revise the measured quality scores.
4. **Local Realignment:** For performance reasons, all modern aligners use algorithms that provide

approximate alignments. This approximation can cause reads with evidence of indels to have slight misalignments with the reference. In this stage, we use fully accurate sequence alignment methods (Smith-Waterman algorithm [24]) to locally realign reads that contain evidence of short indels. This pipeline step is omitted in some variant calling pipelines if the variant calling algorithm used is not susceptible to these local alignment errors.

For current implementations of these read processing steps, performance is limited by disk bandwidth. This bottleneck occurs because the operations read in a SAM/BAM file, perform a bit of processing, and write the data to disk as a new SAM/BAM file. We address this problem 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. The stages themselves cannot be performed in parallel, but the operations inside each stage are data parallel.

4 ADAM Design Philosophy

Modern bioinformatics pipelines have been designed without a model for how the system should grow or for how components of the analytical system should connect. We seek to provide a more principled model for system composition.

Our system architecture was inspired heavily by the Open Systems Interconnection (OSI) model for networking services [33]. This conceptual model standardized the internal functionality of a networked computer system, and its "narrow waisted" design was critical to the development of the services that would become the modern internet. Figure 2 presents a similar decomposition of services for genomics data.

The seven layers of our stack model are decomposed as follows, traditionally numbered from bottom to top:

1. **Physical Storage:** This layer coordinates data writes to physical media, usually magnetic disk.
2. **Data Distribution:** This layer manages access, replication, and distribution of the genomics files that have been written to disk.
3. **Materialized Data:** This layer encodes the patterns for how data is encoded and stored. This provides read/write efficiency and compression.

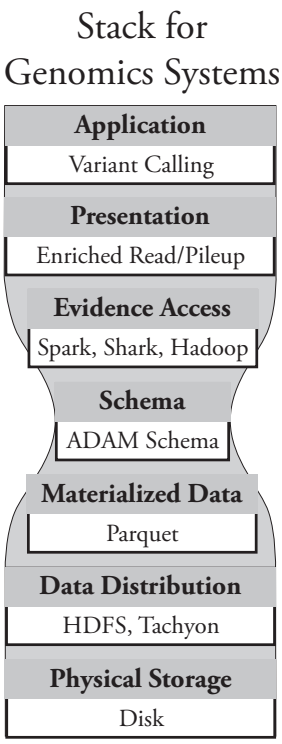

Figure 2: Stack Models for Networking and Genomics

4. **Data Schema:** This layer specifies the representation of data when it is accessed, and forms the narrow waist of the pipeline.
5. **Evidence Access:** This layer implements efficient methods for performing common access patterns such as random database queries, or sequentially reading records from a flat file.
6. **Presentation:** The presentation layer provides the application developer with efficient and straightforward methods for querying the characteristics of individual portions of genetic data.
7. **Application:** Applications like variant calling and alignment are implemented in this layer.

A well defined software stack has several significant advantages. By limiting application interactions with layers lower than the API, application developers are given a clear and consistent view of the data they are processing. By divorcing the API from the data access layer, we unlock significant flexibility. With careful design in the data format and data access layers, we can seamlessly support conventional flat file access patterns, while also allowing easy access to data with database methods (see §8.1). By treating the

compute substrate and storage as separate layers, we also drastically increase the portability of the APIs that we implement.

This approach is a significant improvement over current approaches which intertwine all layers together — if all layers are intertwined, new APIs must be developed to support new compute substrates [22], and new access patterns must be carefully retrofitted to the data format that is in use [15].

We can distill our philosophy into three observations that drive our design decisions:

1. **Scalability is a primary concern for modern genomics systems:** Processing pipelines operate on data that can range in size from tens of gigabytes to petabytes. We cannot remove processing stages, as this has an unacceptable impact on accuracy. To increase pipeline throughput and thereby reduce latency, our systems must be able to parallelize efficiently to tens to hundreds of compute nodes.

2. **Bioinformaticans should be concerned with data, not formats:** The SAM/BAM formats do not provide a schema for the data they store. This omission limits visibility into the structure of data being processed. Additionally, it significantly increases implementation difficulty: by making the format a primary concern, significant work has been required to introduce programming interfaces that allow these formats to be read from different programming languages and environments [18, 3, 22].

3. **Openness and flexibility are critical to adoption:** As the cost of sequencing continues to drop [21], the processing of genomics data will become more widespread. By maintaining an open source standard that can be flexibly modified, we allow all the members of the community to contribute to improving our technologies. Developing a flexible stack is also important; genetic data will be processed in a host of heterogenous computing environments, and with a variety of access patterns. By making our format easy to adopt to different techniques, we maximize its use.

We believe that the greatest contribution of our stack design is the explicit schema we have introduced. This innovation alone drives the flexibility of the stack: on top of a single exposed schema, it is trivial to change the evidence access layer to represent the relevant data in array or tabular form. A schema is essential, as it makes it inexpensive to support novel access patterns, which can enable new algorithms and applications.

In the next two sections, we discuss the implementation of ADAM, and how it was designed to satisfy these goals. We then analyze the performance of our system and the impact of our design philosophy.

5 Data Format and API

ADAM contains formats for storing read and reference oriented sequence information, and variant/genotype data. The read oriented sequence format is forwards and backwards compatible with BAM/SAM, and the variant/genotype data is forwards and backwards compatible with VCF. We provide two APIs:

1. A data format/access API implemented on top of Apache Avro and Parquet

2. A data transformation API implemented on top of Apache Spark

In this section, we provide brief introductions to these two APIs. We then introduce the data representations, discuss the content of each representation, and introduce the transforms that we provide for each representation.

The data representation for the ADAM format is described using the open source Apache Avro data serialization system [5]. The Avro system also provides a human readable schema description language that can auto-generate the schema implementation in several common languages including C/C++/C#, Java/Scala, php, Python, and Ruby. This flexibility provides a significant cross-compatibility advantage over the BAM/SAM format, where the data format has only been implemented for C/C++ through Samtools and for Java through Picard [18, 3]. To complicate matters, in BAM/SAM there are well known incompatibilities between these APIs implemented in different languages. This problem persists across APIs for accessing variant data (discussed in §5.3).

We layer this data format inside of the Parquet column store [26]. Parquet is an open source columnar storage format that was designed by Cloudera and Twitter. It can be used as a single flat file, a distributed file, or as a database inside of Hive, Shark, or Impala. Columnar stores like Parquet provide several significant advantages when storing genomic data:

- **Column stores achieve high compression** [4]. Compression reduces storage space on

disk, and also improves serialization performance inside of MapReduce frameworks. We are able to achieve up to a 0.75× lossless compression ratio when compared to BAM, and have especially impressive results on quality scores. Compression is discussed in more detail in §7.3.

- **Column stores enable predicate pushdown, which minimizes the amount of data read from disk.** [17] When using predicate pushdown, we deserialize specific fields in a record from disk, and apply them to a predicate function. We then only read the full record if it passes the predicate. This is useful when implementing filters that check read quality.
- **Varied data projections can be achieved with column stores.** This option means that we can choose to only read several fields from a record, which improves performance for applications that do not read all the fields of a record. Additionally, we do not pay for null fields in records that we store. This feature allows us to easily implement lossy compression on top of the ADAM format, and also allows us to eliminate the FASTQ standard.

By supporting varied data projections with low cost for field nullification, we can implement lossy compression on top of the ADAM format. We discuss this further in §7.3.

Figure 3 shows how ADAM in Parquet compares to BAM. We remove the file header from BAM, and instead distribute these values across all of the records stored. This dissemination eliminates global information and makes the file much simpler to distribute across machines. This distribution is effectively free in a columnar store, as the store just notes that the information is replicated across many reads. Additionally, Parquet writes data to disk in regularly sized row groups (from [26], see Parquet format readme); this data size allows parallelism across rows by enabling the columnar store to be split into independent groups of reads.

The internal data types presented in this section make up the narrow waist of our proposed genomics stack in Figure 2, as discussed in the prior section. In §8.1 and §9.1, we review how this abstraction allows us to support different data access frameworks and storage techniques to tailor the stack for the needs of a specific application.

For application developers who are using Apache Spark, we provide an additional API. This API consists of transform steps for RDDs (see Zaharia et al [31] for a discussion of RDDs) containing genomic data, and several enriched types. These transform steps provide several frequently used processing stages, like sorting genomic data, deduplicating reads, or transforming reads into pileups. By implementing these transformations on top of RDDs, we allow for efficient, distributed in-memory processing and achieve significant speedups. In this section, we describe the transformations that we provide per each datatype. In section §7.2, we discuss the performance characteristics of sorting and duplicate marking. Detailed descriptions of several of the algorithms that underly these transformations are provided in appendix B.

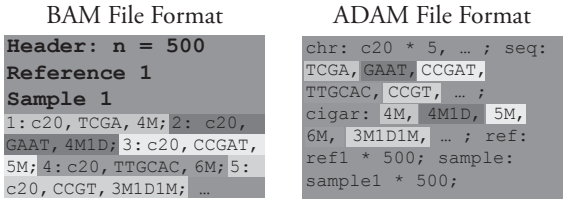

Figure 3: Comparative Visualization of BAM and ADAM File Formats

5.1 Read Oriented Storage

Table 3 defines our default read oriented data format. This format provides all of the fields supported by the SAM format. As mentioned above, to make it easier to split the file between multiple machines for distributed processing, we have eliminated the file header. Instead, the data encoded in the file header can be reproduced from every single record. Because our columnar store supports dictionary encoding and these fields are replicated across all records, replicating this data across all records has a negligible cost in terms of file size on disk.

This format is used to store all data on disk. In addition to this format, we provide an enhanced API for accessing read data, and several transformations. This enhanced API converts several string fields into native objects (e.g. CIGAR, mismatching positions, base quality scores), and provides several additional convenience methods. We supply several transformations for RDD data:

- **Sort:** Sorts reads by reference position
- **Mark Duplicates:** Marks duplicate reads
- **Base Quality Score Recalibration:** Normalizes the distribution of base quality scores

Table 3: Read Oriented Format

Group	Field	Type
General	Reference Name	String
	Reference ID	Int
	Start	Long
	Mapping Quality	Int
	Read Name	String
	Sequence	String
	Mate Reference	String
	Mate Alignment Start	Long
	Mate Reference ID	Int
	Cigar	String
	Base Quality	String
Flags	Read Paired	Boolean
	Proper Pair	Boolean
	Read Mapped	Boolean
	Mate Mapped	Boolean
	Read Negative Strand	Boolean
	Mate Negative Strand	Boolean
	First Of Pair	Boolean
	Second Of Pair	Boolean
	Primary Alignment	Boolean
	Failed Vendor Checks	Boolean
	Duplicate Read	Boolean
Attributes	Mismatching Positions	String
	Other Attributes	String
Read Group	Sequencing Center	String
	Description	String
	Run Date	Long
	Flow Order	String
	Key Sequence	String
	Library	String
	Median Insert	Int
	Platform	String
	Platform Unit	String
	Sample	String

- **Realign Indels:** Locally realigns indels that have been misaligned during global alignment
- **Read To Reference:** Transforms data to be reference oriented (creates pileups)
- **Flag Stat:** Computes statistics about the boolean flags across records
- **Create Sequence Dictionary:** Creates a dictionary summarizing data across multiple samples

The performance of sort and mark duplicates are discussed in §7.2. The implementations of sort, mark duplicates, BQSR, and indel realignment are discussed in appendix B.

5.2 Reference Oriented Storage

In addition to storing sequences as reads, we provide a storage format for reference oriented (pileup) data. Table 4 documents this format. This pileup format is also used to implement a data storage format similar to the GATK's ReducedReads format [10].

Table 4: Reference Oriented Format

Group	Field	Type
General	Reference Name	String
	Reference ID	Int
	Position	Long
	Range Offset	Int
	Range Length	Int
	Reference Base	Base
	Read Base	Base
	Base Quality	Int
	Mapping Quality	Int
	Number Soft Clipped	Int
	Number Reverse Strand	Int
	Count At Position	Int
Read Group	Sequencing Center	String
	Description	String
	Run Date	Long
	Flow Order	String
	Key Sequence	String
	Library	String
	Median Insert	Int
	Platform	String
	Platform Unit	String
	Sample	String

We treat inserts as an inserted sequence range at the locus position. For bases that are an alignment match against the reference[3], we store the read base and set the range offset and length to 0. For deletions, we perform the same operation for each reference position in the deletion, but set the read base to null. For inserts, we set the range length to the length of the insert. We step through the insert from the start of the read to the end of the read, and increment the range offset at each position.

[3]Following the conventions for CIGAR strings, an alignment match does not necessarily correspond to a sequence match. An alignment match simply means that the base is not part of an indel. The base may not match the reference base at the loci.

As noted earlier, this datatype supports an operation similar to the GATK's ReducedRead datatype. We refer to these datatypes as aggregated pileups. The aggregation transformation is done by grouping together all bases that share a common reference position and read base. Within an insert, we group further by the position in the insert. Once the bases have been grouped together, we average the base and mapping quality scores. We also count the number of soft clipped bases that show up at this location, and the amount of bases that are mapped to the reverse strand. In our RDD API, we provide an additional ADAMRod datatype which represents pileup "rods," which are pileup bases grouped by reference position.

5.3 Variant and Genotype Storage

The VCF specification more closely resembles an exchange format than a data format. In particular, the per-site and per-sample fields ("INFO" and "FORMAT") define arbitrary key-value pairs, for which parsing information is stored in the header of the file. This format enables side computation—such as classifying a particular SNP as likely or unlikely to damage the structure of a protein—to operate on a VCF and store their results. A typical analysis pipeline may comprise of several steps which read in the VCF produced by the previous step, add additional annotations to the INFO and FORMAT fields, and write a new VCF.

Maintaining full compatibility with VCF presents a structural challenge. Each (properly constructed) VCF file contains the data schema for its INFO and FORMAT fields within the file header. Potentially, every VCF file utilizes a different schema.

Maintaining support for these catch-all fields by incorporating the file schema within an ADAM-VCF Record violates the layer separation that is a part of the design goals: the Evidence Access layer (layer 5 in Figure 2) would take on the role of reading format information from a record, and using that to parse the contents of an information blob. Furthermore, the format information is "hidden" from the Schema layer (layer 4 in Figure 2): the system cannot know what information the record should contain until it looks at the format information for the record. This approach presents challenges to

- tools, which cannot check whether a field they expect is part of the record schema (until runtime);
- developers, who will have to write their own parsers for these unstructured blobs; and
- repository maintainers, who need to adjudicate whose (complex) fields and parsers are widely-used enough to merit inclusion and official support.

The "narrow waist" of the ADAM Schema divides the world of parsing and file formats from the world of data presentation and processing. Relying on the allegory of the network stack, just as the transport layer need not worry about how to retrieve data from across a network boundary, the access layer of the bioinformatic process should not need to worry about parsing or formatting raw data blobs.

The VCF format has repeated violations of this principle. While the VCF format is supposed to render a schema for all key value pairs in its INFO and FORMAT fields, a number of bioinformatics tools do not provide this parsing information, and so types must be inferred. In addition, some tools (in particular annotation tools such as VariantEffectPredictor[20]) serialize rich data types (such as nested maps) into the INFO field, describing their format as simply "String".

By adhering rigidly to specific schema, not only do we encourage separation of concerns, we also limit extensibility abuse, and the practice of looking at online documentation to understand how to parse a field in a VCF. The schema is the documentation, and the parsing is not the tool writer's concern!

These issues extend beyond tools that produce malformed VCFs or uninterpretable schema; the format is ill-supported by APIs that intend to support them. For instance, PyVCF always splits on commas, even if the number of values is defined as 1. Thus, it mis-parses VariantPredictorOutput which is a single (though doubly-delimited by "," and "—") field. Similarly to facilitate compression, per-sample fields (are not required to propagate all values.

To address this issue, we eliminate the attributes from the Variant and Genotype schemas. Instead, we provide an internal relational model that allows additional fields to be provided by specifying an explicit schema. At runtime, an ADAMVariantContext class is provided that handles the correct joining of this data. This class eliminates the need for an attributes section and the parsing incompatibilities related to this section, while maintaining the ability to arbitrarily introduce new fields as necessary. This approach provides all the flexibility of the current VCF format, while eliminating the pain points.

An additional benefit of the new Variant and Genotype schemas is that they have been designed to ensure that all Variant call information can be reconstructed using the called sample genotypes.

Table 5: Genotype Format

Field	Type
Reference ID	String
Reference Name	String
Reference Length	Long
Reference URL	String
Position	Long
Sample ID	String
Ploidy	Int
Haplotype Number	Int
Allele Variant Type	Variant Type
Allele	String
Is Reference?	Boolean
Reference Allele	String
Expected Allele Dosage	Double
Genotype Quality	Int
Depth	Int
Phred Likelihoods	String
Phred Posterior Likelihoods	String
Ploidy State Genotype Likelihoods	String
Haplotype Quality	Int
RMS Base Quality	Int
RMS Mapping Quality	Int
Reads Mapped Forward Strand	Int
Reads Mapped, MapQ == 0	Int
SV Type	SV Type
SV Length	Long
SV Is Precise?	Boolean
SV End	Long
SV Confidence Interval Start Low	Long
SV Confidence Interval Start High	Long
SV Confidence Interval End Low	Long
SV Confidence Interval End High	Long
Is Phased?	Boolean
Is Phase Switch?	Boolean
Phase Set ID	String
Phase Quality	Int

Table 6: Variant Format

Field	Type
Reference ID	String
Reference Name	String
Reference Length	Long
Reference URL	String
Position	Long
Reference Allele	String
Is Reference?	Boolean
Variant	String
Variant Type	Variant Type
ID	String
Quality	Int
Filters	String
Filters Run?	Boolean
Allele Frequency	String
RMS Base Quality	Int
Site RMS MapQ	Double
MapQ=0 Count	Int
Samples With Data	Int
Total Number Of Samples	Int
Strand Bias	Double
SV Type	SV Type
SV Length	Long
SV Is Precise?	Boolean
SV End	Long
SV Confidence Interval Start Low	Long
SV Confidence Interval Start High	Long
SV Confidence Interval End Low	Long
SV Confidence Interval End High	Long

This functionality is important for genomics projects where many samples are being sequenced and called together, but where different views of the sample population are wished to be provided to different scientists. In this situation, the total call set would be filtered for a subset of samples. From these collected genotypes, we would then recompute the variant calls and variant call statistics, which would then be provided as an output. This transformation is provided by our RDD-based API. Additionally, we provide an ADAMVariantContext datatype, which aggregates variant, genotype, and annotation data at a single genomic locus.

6 In-Memory Programming Model

As discussed in §3, the main bottleneck in current genomics processing pipelines is packaging data up on disk in a BAM file after each processing step. While this process is useful as it maintains data lineage[4], it significantly decreases the throughput of the processing pipeline.

For the processing pipeline we have built on top of the ADAM format, we have minimized disk accesses (read one file at the start of the pipeline, and write one file after all transformations have been completed). Instead, we cache the reads that we are transforming in memory, and chain multiple transformations together. Our pipeline is implemented on top of the Spark MapReduce framework, where reads are stored as a map using the Resilient Distributed Dataset (RDD) primitive.

7 Performance

This section reviews performance metrics pertinent to the ADAM file format, and applications written on top of ADAM. Our performance analysis is partitioned into three sections:

- **7.1. Microbenchmarks:** In this section, we compare several test cases that have little computation, and review ADAM against Hadoop-BAM. This section looks to evaluate the disk access performance of ADAM. In this section, we are most interested in validating scalability and building intuition for the efficiencies provided by variable projection and predicate pushdown.
- **7.2. Applications:** The main goal of ADAM is to accelerate the read processing pipelines that were introduced in §3. In this section, we review the throughput of our processing pipeline for both single nodes and for clusters, despite the focus of ADAM being the latter.
- **7.3. Compression:** Here, we review the reductions in disk space that we are able to achieve when compared against BAM. We also review how we can achieve further compression through aggregating pileup data, and how to implement both lossy and reference-based compression top of ADAM.

[4]Since we store the intermediate data from each processing step, we can later redo all pipeline processing after a certain stage without needing to rerun the earlier stages. This comes at the obvious cost of space on disk. This tradeoff makes sense if the cost of keeping data on disk and reloading it later is lower than the cost of recomputing the data. This does not necessarily hold for modern MapReduce frameworks [31].

7.1 Microbenchmarks

To validate the performance of ADAM, we created several microbenchmarks. These microbenchmarks are meant to demonstrate the pure read/write throughput of ADAM, and to show how the system scales in a cluster.

To validate these benchmarks, we implemented the relevant tests in both ADAM and Hadoop-BAM [22], running on Spark on an Amazon Elastic Compute 2 (EC2) cluster. We used the $m2.4xlarge$ instance type for all of our tests. This instance type is an 8 core memory optimized machine with 68.4 Gibibytes (GiB) of memory per machine.

We ran these benchmarks against HG00096, a low-coverage human genome (15 GB BAM) from the 1,000 Genomes Project. The file used for these experiments can be found on the 1000 Genomes ftp site, `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/HG00096/alignment/`.

7.1.1 Read/Write Performance

The decision to use a columnar store was based on the observation that most genomics applications are read-heavy. In this section, we aim to quantify the improvements in read performance that we achieve by using a column store, and how far these gains can be maximized through the use of projections.

Read Length and Quality: This microbenchmark scans all the reads in the file, and collects statistics about the length of the read, and the mapping quality score. Figure 4 shows the results of this benchmark. Ideal speedup curves are plotted along

with the speedup measurements. The speedups in this graph are plotted relative to the single machine Hadoop-BAM run.

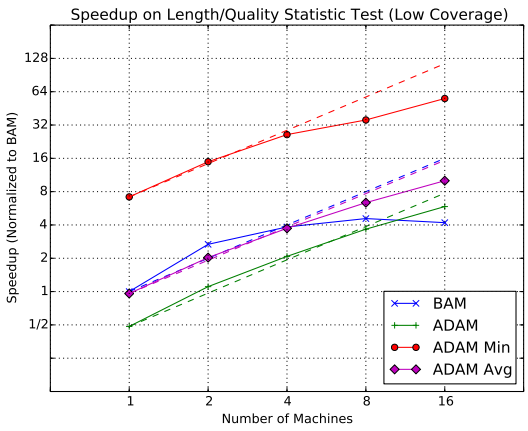

Figure 4: Read Length and Quality Speedup for HG00096

This benchmark demonstrates several things:

- ADAM demonstrates superior scalability over Hadoop-BAM. This performance is likely because ADAM files can be more evenly distributed across machines. Hadoop-BAM must guess the proper partitioning when splitting files [22]. The Parquet file format that ADAM is based on is partitioned evenly when it is written[26]. This paritioning is critical for small files such as the one used in this test — in a 16 machine cluster, there is only 1 GB of data per node, so imbalances on the order of the Hadoop file split size (128 MB) can lead to 40% differences in loading between nodes.
- Projections can significantly accelerate computation, if the calculation does not need all the data. By projecting the minimum dataset necessary to calculate the target of this benchmark, we can accelerate the benchmark by 8×. Even a more modest projection leads to a 2× improvement in performance.

7.1.2 Predicate Pushdown

Predicate pushdown is one of the significant advantages afforded to us through the use of Parquet [26]. In traditional pipelines, the full record is deserialized from disk, and the filter is implemented as a Map/Reduce processing stage. Parquet provides us with predicate pushdown: we deserialize only the columns needed to identify whether the record should be fully deserialized. We then process those fields; if they pass the predicate, we then deserialize the full record.

We can prove that this process requires no additional reads. Intuitively, this is true as the initial columns read would need to be read if they were to be used in the filter later. However, this can be formalized. We provide a proof for this in Section C of the Appendix. In this section, we seek to provide an intuition for how predicate pushdown can improve the performance of actual genomics workloads. We apply predicate pushdown to implement read quality predication, mapping score predication, and predication by chromosome.

Quality Flags Predicate: This microbenchmark scans the reads in a file, and filters out reads that do not meet a specified quality threshold. We use the following quality threshold:

- The read is mapped.
- The read is at its primary alignment position.
- The read did not fail vendor quality checks.
- The read has not been marked as a duplicate.

This threshold is typically applied as one of the first operations in any read processing pipeline.

Gapped Predicate: In many applications, the application seeks to look at a region of the genome (e.g. a single chromosome or gene). We include this predicate as a proxy: the gapped predicate looks for 1 out of every n reads. This selectivity achieves performance analogous to filtering on a single gene, but provides an upper bound as we pay a penalty due to our predicate hits being randomly distributed.

To validate the performance of predicate pushdown, we ran the predicates described above on the HG00096 genome on a single AWS $m2.4xlarge$ instance. The goal of this was to determine the rough overhead of predicate projection.

Figure 5 shows performance for the gapped predicate sweeping n.

There are several conclusions to be drawn from this experiment:

- As is demonstrated by equation (2) in Section C of the Appendix, we see the largest speedup when our predicate read set is significantly smaller than our projection read set. For

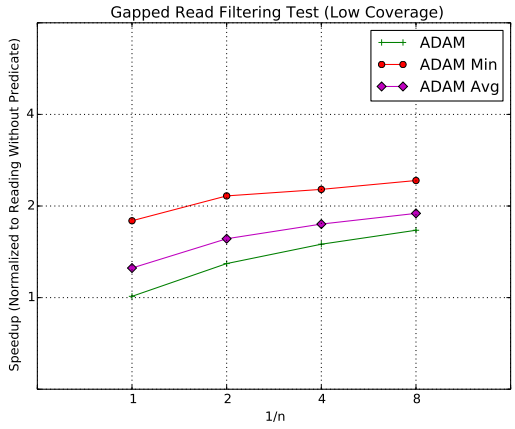

Figure 5: Gapped Predicate on HG00096

smaller projections, we see a fairly substantial reduction in speedup.

- As the number of records that passes the predicate drops, the number of reads done to evaluate the predicate begins to become a dominant term. This insight has one important implication: for applications that plan to access a very small segment of the data, accessing the data as a flat file with a predicate is likely not the most efficient access pattern. Rather, if the predicate is regular, it is likely better to access the data through a indexed database.

We additionally calculated the performance improvement that was achieved by using predicate pushdown instead of a Spark filter. Table 7 lists these results.

Table 7: Predicate Pushdown Speedup vs. Filtering

Predicate	Min	Avg	Max	Filtered
Locus	1.19	1.17	0.94	3%
Gapped, $n = 1$	1.0	1.0	1.01	0%
Gapped, $n = 2$	1.22	1.27	1.29	50%
Gapped, $n = 4$	1.31	1.42	1.49	75%
Gapped, $n = 8$	1.37	1.58	1.67	87.5%

These results are promising, as they indicate that predicate pushdown is faster than a Spark filter in most cases. We believe that the only case that showed a slowdown (max projection on Locus predicate) is due to an issue in our test setup. We therefore can conclude that filtering operations that can be performed statically at file load-in should be performed using predicate pushdown.

7.2 Applications

ADAM relies on Apache Spark for in-memory, map-reduce style processing. At the heart of Spark is a programming construct, called a Resilient Distributed Dataset (RDD), which represents a fault-tolerant collection of elements that are operated on in parallel.

Programming with an RDD is very similar to programming with any standard Scala collection like `Seq` or `List`. This polymorphism allows developers to focus on the algorithm they're developing without concerning themselves with the details of distributed computation. The example code presented in this section highlight the succinctness and expressiveness of Spark Scala code. While ADAM is currently written exclusively in Scala, Apache Spark also has Java and Python support.

Code written in Apache Spark can be run multi-threaded on a single machine or across a large cluster without modification. For these applications to be run in a distributed fashion, the data needs to be loaded onto the Hadoop Distribute File System (HDFS). HDFS breaks the file into blocks, distributes them across the cluster and maintains information about block (and replica) locations. Breaking large files like BAM files into blocks allows systems like Spark to create tasks that run across the entire cluster processing the records inside of each block.

The NA12878 high-coverage BAM file used for these experiments can be found on the 1000 Genomes ftp site, `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/NA12878/high_coverage_alignment/`.

The next two subsections compare BAM performance of these applications on a single node to ADAM on a single node and then to ADAM on a cluster.

7.2.1 Single Node Performance

A single `hs1.8xlarge` Amazon EC2 instance was used to compare the performance of ADAM to Picard Tools 1.103. This `hs1.8xlarge` instance has 8 cores, 117 GB Memory, and 24 2,048 GB HDD drives that can deliver 2.4 GB/s of 2 MB sequential read performance and 2.6 GB/s of sequential write performance.

The 24 disk were configured into a single RAID 0 configuration using `mdadm` to provide a single 44TB partition for tests. Measurements using `hdparm`

showed buffered disk reads at 1.4 GB/sec and cached reads at 6.1 GB/sec.

The AMI used ()`ami-be1c848e`) was a stock Amazon Linux AMI for x86_64 paravirtualized and EBS backed. Aside from creating the RAID 0 partition above, the only modification to the system was increasing the limit for file handles from 1024 to 65535. This increase was needed to handle the large number of files that Picard and ADAM spill to disk during processing. All single-node tests were performed on the same EC2 instance.

OpenJDK 1.6.0_24 (Sun Microsystems Inc.) was used for all single-node tests.

Picard Whole Genome Sorting: Picard's SamSort tool was used to sort the NA12878 whole genome BAM file using the following options as output by Picard at startup.

```
net.sf.picard.sam.SortSam \
INPUT=/mnt/NA12878.bam \
OUTPUT=/mnt/NA12878.sorted.bam \
SORT_ORDER=coordinate \
VALIDATION_STRINGENCY=SILENT \
MAX_RECORDS_IN_RAM=5000000 \
VERBOSITY=INFO \
QUIET=false \
COMPRESSION_LEVEL=5 \
CREATE_INDEX=false \
CREATE_MD5_FILE=false
```

The RAID 0 partition was also used as the java temporary directory using Java property `java.io.tmpdir` ensuring that Picard was spilling intermediate files to a fast and large partition. Picard was also given 64 GB of memory using the Java `-Xmx64g` flag. Because of the large amount of memory available, the `MAX_RECORDS_IN_RAM` was set to `5000000` which is ten times larger than the Picard default.

Picard took 17 hours 44 minutes wall clock time to sort the NA12878 whole genome BAM file spending 17 hours 31 minutes of user CPU time and 1 hour of system CPU time.

Since Picard isn't multi-threaded, only a single CPU was utilitized at nearly 100% during the entire sort. Memory usage didn't grow beyond 30GB (even though 64GB was available) which seems to indicate that `MAX_RECORDS_IN_RAM` could have been set even higher. `dstat` reported disk utilization of 40M/s reading and writing during the initial 4.5 hours of sorting which dropped to 5M/s reading and 8M/s writing during the last 13.25 hours when the final merging of spilled files occured. Picard spilled 386 GB of data during the sort.

ADAM Whole Genome Sorting: The following ADAM command was used to sort the NA12878 whole genome BAM file and save the results to the ADAM Avro/Parquet format.

```
$ java -Xmx111g \
-Dspark.default.parallelism=32 \
-Dspark.local.dir=/mnt/spark \
-jar adam-0.5.0-SNAPSHOT.jar \
transform -sort_reads \
NA12878.bam \
NA12878.sorted.adam
```

ADAM took 8 hours 56 minutes wall clock time to sort the whole genome using 6 hours of user CPU time and 2 hours 52 minutes of system CPU time.

During the initial key sort stage, ADAM was reading the BAM at 75M/s with about 60% CPU utilization for a little over 90 minutes. During the map phase, `dstat` reported 40M/s reading and 85M/s writing to disk. The shuffle wrote 452.6 GB of data to disk. During the final stage of saving the results to the Avro/Parquet format, all CPUs were at 100% usage and disk i/o was relatively lighter at 25M/s. RAM usage peaked at 73 GB of the 111 GB available to the VM.

Whole Genome Mark Duplicates: Picard's MarkDuplicates tool was used to mark duplicates in the NA12878 whole genome BAM file.

```
net.sf.picard.sam.MarkDuplicates \
INPUT=[/mnt/NA12878.sorted.bam] \
OUTPUT=/mnt/NA12878.markdups.bam \
METRICS_FILE=metrics.txt \
VALIDATION_STRINGENCY=SILENT \
PROGRAM_RECORD_ID=MarkDuplicates \
PROGRAM_GROUP_NAME=MarkDuplicates \
REMOVE_DUPLICATES=false \
ASSUME_SORTED=false \
MAX_SEQUENCES_FOR_DISK_READ_ENDS_MAP=50000 \
MAX_FILE_HANDLES_FOR_READ_ENDS_MAP=8000 \
SORTING_COLLECTION_SIZE_RATIO=0.25 \
READ_NAME_REGEX=
[a-zA-Z0-9]+:[0-9]:([0-9]+):([0-9]+):([0-9]+).* \
OPTICAL_DUPLICATE_PIXEL_DISTANCE=100 \
VERBOSITY=INFO \
QUIET=false \
COMPRESSION_LEVEL=5 \
MAX_RECORDS_IN_RAM=500000 \
CREATE_INDEX=false \
CREATE_MD5_FILE=false
```

Single Node Performance Summary: ADAM essentially doubles performance over BAM on a single node, although cluster performance is the real goal of ADAM. (Single node mark duplicate performance is still being collected.)

Table 8: Single Node Runtime Comparison

Application	Runtime	ADAM Runtime
SamSort	17 h 44 m	8 h 56 m

7.2.2 Cluster Performance

Unlike current genomic software systems, ADAM can scale out to reduce the time for data analysis by distributing computation across a cluster. Thus, cluster performance is the critical test for ADAM.

Flagstat: Even the simplest application can benefit from the ADAM format and execution environment. For example, the samtools `flagstat` command prints read statistics on a BAM file, e.g.

```
$ samtools flagstat NA12878.bam
1685229894 + 0 in total (QC-passed reads + QC-failed reads)
0 + 0 duplicates
1622164506 + 0 mapped (96.26\%:-nan\%)
1685229894 + 0 paired in sequencing
842614875 + 0 read1
842615019 + 0 read2
1602206371 + 0 properly paired (95.07\%:-nan\%)
1611286771 + 0 with itself and mate mapped
10877735 + 0 singletons (0.65\%:-nan\%)
3847907 + 0 with mate mapped to a different chr
2174068 + 0 with mate mapped to a different chr (mapQ>=5)
```

ADAM has a `flagstat` command which provides identical information in the same text format.

Table 9: Time to Run Flagstat on High-Coverage NA12878

Software	Wall clock time
Samtools 0.1.19	25m 24s
ADAM 0.5.0	0m 46s

The ADAM `flagstat` command was run on EC2 with 32 `cr1.8xlarge` machines which is, not suprisingly, why ADAM ran 34 times faster. While the samtools `flagstat` command is multi-threaded, it can only be run on a single machine.

Sort: When coordinate sorting a BAM file, reads are reordered by reference id and alignment start position. This sorting makes it easier for downstream tools to find reads for a specific area of the reference and makes finding mates in paired-ends reads more efficient. Many utilities will only except BAM files that have been sorted, e.g. Picard MarkDuplicates.

Each read is mapped to a tuple with the two values: position and read. Unmapped reads that a value of `None` for `ReferencePosition` are sent to the end of the file.

Table 10: Sort NA12878

Software	EC2 profile	Wall clock time
Picard Tools 1.103	1 `hs1.8xlarge`	17h 44m
ADAM 0.5.0	32 `cr1.8xlarge`	33m
ADAM 0.5.0	100 `m2.4xlarge`	21m

ADAM was run on an EC2 cluster with 32 `cr1.8xlarge` machines and a cluster with 100 `m2.4xlarge` instances.

Monitoring the Spark console during sorting revealed that the `sortByKey` operation (on line 10 above) took a total of 1 min 37 secs. The `map` operation (at line 2 above) took 1 min 48 secs, and the remainder of the time, 29 min 25 secs, was spent writing the results to HDFS. There were a total of 1586 tasks with a majority finishing in under 10 minutes.

Mark Duplicates: The ADAM algorithm for marking duplicate is nearly identical to Picard's MarkDuplicates command.

Table 11: Mark Duplicates for NA12878

Software	EC2 profile	Wall clock time
Picard 1.103	1 `hs1.8xlarge`	20 h 6 m
ADAM 0.5.0	100 `m2.4xlarge`	29m

7.2.3 Tuning Spark on EC2

All applications were run on Amazon EC2 with minimal tuning. The performance numbers in this report should not be considered best case performance numbers for ADAM.

An important Apache Spark property is `spark.default.parallelism` which controls the number of tasks to use for distributed shuffle operations. This value needs to be tuned to match the underlying cluster that Spark is running on.

The following graph shows CPU time series for two sort operations on a 32-node `cr1.8xlarge` EC2 cluster running back to back with different values for

Table 12: Time to Sort NA12878

spark.default.parallelism	Wall clock time
500	47 min 10 secs
48	32 min 38 secs

`spark.default.parallelism`. The early CPU profile is when `spark.default.parallelism` is set to 500 and the latter profile is when it's set to 48. You can see from the graph that with the higher level of parallelism there is a significantly higher amount of system CPU (in red) as 32 CPUs contend for access to the two disks in each `cr1.8xlarge` instance. While `cr1.8xlarge` instances provide plenty of memory, they do not have enough disks to help distribute the I/O load.

CPU Graph.png

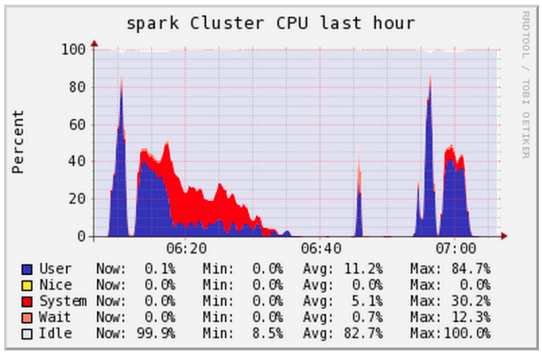

Figure 6: Comparison of 500 to 48 parallelism

7.3 Compression

By using dictionary coding, run length encoding with bit-packing, and GZIP compression, ADAM files can be stored using less disk space than an equivalent BAM file. Typically, ADAM is up to 75% of the size of the compressed BAM for the same data. If one is performing pileup aggregation, compression can improve to over 50%.

8 Future Work

Beyond the initial release of the ADAM data formats, API, and read transformations, we are working on several other extensions to ADAM. These extensions strive to make ADAM accessible to more users and to more programming patterns.

8.1 Database Integration

ADAM format plays a central role in a library that we develop that will allow sequencing data to be queried from the popular SQL based warehouse software such as Shark [29], Hive [1] and Impala [2]. In fact, the Avro/Parquet storage backend enables the direct usage of ADAM files as SQL tables; this integration is facilitated through libraries that integrate Parquet with the aforementioned software [26].

Of course, there is more work remaining before Shark provides biologically relevant functionality, such as the one that is outlined in Bafna et al [7].

Shark needs to be enhanced with libraries that provide efficient indexing, storage management and interval query handling. The indexing scheme that we develop enables range based read retrieval, such as the indexing mechanism of samtools, quick access to mate pair reads such as the indexing mechanism of GQL [15] and also text based search features on fields such as the read sequence and the CIGAR string.

To keep storage size under control, we implement a set of heuristics that prevent the materialization of queries with reproducible output. Finally, given that most biological data is modelled after intervals on the human genome, our library includes user defined functions that implement interval operators such as the IntervalJoin operator of GQL.

8.2 API Support Across Languages

Currently, the ADAM API and example read processing pipeline are implemented in the Scala language. However, long term we plan to make the API accessible to users of other languages. ADAM is built on top of Avro, Parquet, and Spark. Spark has bindings for Scala, Java, and Python, and Parquet implementations exist for Java and C++. As the data model is implemented on top of Avro, the data model can also be autogenerated for C/C++/C#, Python, Ruby, and php. We hope to make API implementations available for some of these languages at a later date.

8.3 BQSR and Indel Realignment

BQSR and indel realignment implementations are available in ADAM, but have not yet been fully validated for correctness nor has their performance been fully characterized. We anticipate this work to be complete in early 2014.

8.4 Variant and Genotype Format

Although we have defined a draft variant and genotype format, we are currently working with a multi-institution team to refine these formats. Our aim is to clarify problems present with the current VCF spec, and to make the variant and genotype formats more amenable to computational analysis, curation and distribution, and clinical use. We plan to release a follow-on report when this format is stable.

9 Discussion

As discussed in this paper, ADAM presents a significant jump for genomics data processing. To summarize, the benefits are as follows:

- ADAM achieves approximately a 30× to 50× speedup when compared to other modern processing pipelines.
- ADAM can successfully scale to clusters larger than 100 nodes.
- ADAM presents an explicit data schema which makes it straightforward for the data access layer (layer 3 in Figure 2) to be changed to support new access patterns.

In this section, we discuss a few tradeoffs inherent to ADAM.

9.1 Columnar vs. Row Storage

In our current implementation, we use the Parquet columnar store to implement the data access layer. We chose to use columnar storage as we believe that columnar storage is a better fit for bioinformatics workloads than flat row storage; this discussion is introduced in §5.

Typically, column based storage is preferable unless our workload is write heavy; write heavy workloads perform better with row based storage formats [25]. Genomics pipelines tend to skew in favor of reading data — pipelines tend to read a large dataset, prune/reduce data, perform an analysis, and then write out a significantly smaller file that contains the results of the analysis.

It is conceivable, however, that emerging workloads could change to be write heavy. We note that although our current implementation would not be optimized for these workloads, the layered model proposed in Figure 2 allows us to easily change our implementation. Specifically, we can swap out columnar storage in the data access layer with row oriented storage [5]. The rest of the implementation in the system would be isolated from this change: algorithms in the pipeline would be implemented on top of the higher level API, and the compute substrate would interact with the data access layer to implement the process of reading and writing data.

9.2 Programming Model Improvements

Beyond improving performance by processing data in memory, our programming model improves programmer efficiency. We build our operations as a set of transforms that extend the RDD processing available in Spark. These transformations allow a very straightforward programming model that has been demonstrated to significantly reduce code size [31]. Additionally, the use of Scala, which is an object functional statically typed language that supports type inference couples provides further gains.

Additional gains come from our data model. We expose a clear schema, and build an API that expands upon this schema to implement commonly needed functions. This API represents layer 6 in the stack we proposed in Figure 2). To reduce the cost of this API, we make any additional transformations from our internal schema to the data types presented in our API lazy. This transformation eliminates the cost of providing these functions if they are not used. Additionally, if an additional transformation is required, we only pay the cost the first time the transformation is performed.

In total, these improvements can double programmer productivity. This gain is demonstrated through GAParquet, which demonstrates some of the functionality in the GATK on top of the ADAM stack [12]. In the DiagnoseTargets stage of GATK, the number of lines of code (LOC) needed to implement the stage dropped from 400 to 200. We also see a reduction in lines of code needed to implement the read transformation pipeline described in §3. Table 13 compares LOC for GATK/GAParquet and the read transformation pipeline.

10 Summary

In this technical report, we have presented ADAM which is a new data storage format and processing pipeline for genomics data.

[5] It is worth noting that the Avro serialization system that we use to define ADAM is a performant row oriented system.

Table 13: Lines of Code for ADAM and Original Implementation

Application	Original	ADAM
GATK Diagnose Targets		
Walker	326	134
Subclass	93	66
Total	**419**	**200**

ADAM makes use of efficient columnar storage systems to improve the lossless compression available for storing read data, and uses in-memory processing techniques to eliminate the read processing bottleneck faced by modern variant calling pipelines. On top of the file formats that we have implemented, we also present APIs that enhance developer access to read, pileup, genotype, and variant data.

We are currently in the process of extending ADAM to support SQL querying of genomic data, and extending our API to more programming languages.

ADAM promises to improve the development of applications that process genomic data, by removing current difficulties with the extraction and loading of data and by providing simple and performant programming abstractions for processing this data.

A Availability

The ADAM source code is available at Github at `http://www.github.com/bigdatagenomics/adam`, and the ADAM project website is at `http://adam.cs.berkeley.edu`. Additionally, ADAM is deployed through Maven with the following dependencies:

```
<dependency>
  <groupId>edu.berkeley.cs.amplab.adam</groupId>
  <artifactId>adam-format</artifactId>
  <version>0.5.0-SNAPSHOT</version>
</dependency>

<dependency>
  <groupId>edu.berkeley.cs.amplab.adam</groupId>
  <artifactId>adam-commands</artifactId>
  <version>0.5.0-SNAPSHOT</version>
</dependency>
```

At publication time, the current version of ADAM is 0.5.0. ADAM is open source and is released under the Apache 2 license.

B Algorithm Implementations

As ADAM is open source, the source code for the system is freely available. However, the implementations of the algorithms are not always trivial. In this section, we highlight the algorithms that underly our read processing pipeline.

B.1 Sort Implementation

The ADAM Scala code for sorting a BAM file is succinct and relatively easy to follow. It is provided below for reference.

```
def adamSortReadsByReferencePosition():
    RDD[ADAMRecord] = {
  rdd.map(p => {
    val referencePos = ReferencePosition(p) match {
      case None =>
        // Move unmapped reads to the end
        ReferencePosition(Int.MaxValue,
          Long.MaxValue)
      case Some(pos) => pos
    }
    (referencePos, p)
  }).sortByKey().map(p => p._2)
}
```

B.2 BQSR Implementation

Base Quality Score Recalibration is an important early data-normalization step in the bioinformatics pipeline, and after alignment it is the next most costliest step. Since quality score recalibration can vastly improve the accuracy of variant calls — particularly for pileup-based callers like the UnifiedGenotyper or Samtools mpileup. Because of this, it is likely to remain a part of bioinformatics pipelines.

BQSR is also an interesting algorithm in that it doesn't neatly fit into the framework of map reduce (the design philosophy of the GATK). Instead it is an embarrassingly parallelizable aggregate. The ADAM implementation is:

```
def computeTable(rdd: Records, dbsnp: Mask) :
  RecalTable = {

  rdd.aggregate(new RecalTable)(
    (table, read) => { table + read },
    (table, table) => { table ++ table })
}
```

The ADAM implementation of BQSR utilizes the MD field to identify bases in the read that mismatch

the reference. This enables base quality score recalibration to be entirely reference-free, avoiding the need to have a central Fasta store for the human reference. However, dbSNP is still needed to mask out positions that are polymorphic (otherwise errors due to real variation will severely bias the error rate estimates).

B.3 Indel Realignment Implementation

Indel realignment is implemented as a two step process. In the first step, we identify regions that have evidence of an insertion or deletion. After these regions are identified, we generate candidate haplotypes, and realign reads to minimize the overall quantity of mismatches in the region. The quality of mismatches near an indel serves as a good proxy for the local quality of an alignment. This is due to the nature of indel alignment errors: when an indel is misaligned, this causes a temporary shift in the read sequence against the reference sequence. This shift manifests as a run of several bases with mismatches due to their incorrect alignment.

B.3.1 Realignment Target Identification

Realignment target identification is done by converting our reads into reference oriented "rods"[6]. At each locus where there is evidence of an insertion or a deletion, we create a *target* marker. We also create a target if there is evidence of a mismatch. These targets contain the indel range or mismatch positions on the reference, and the range on the reference covered by reads that overlap these sites.

After an initial set of targets are placed, we merge targets together. This is necessary, as during the read realignment process, all reads can only be realigned once. This necessitates that all reads are members of either one or zero realignment targets. Practically, this means that over the set of all realignment targets, no two targets overlap.

The core of our target identification algorithm can be found below.

```
def findTargets (reads: RDD[ADAMRecord]):
    TreeSet[IndelRealignmentTarget] = {

  // convert reads to rods
  val processor = new Read2PileupProcessor
  val rods: RDD[Seq[ADAMPileup]] = reads.flatMap(
      processor.readToPileups(_))
    .groupBy(_.getPosition).map(_._2)

  // map and merge targets
  val targetSet = rods.map(
      IndelRealignmentTarget(_))
    .filter(!_.isEmpty)
    .keyBy(_.getReadRange.start)
    .sortByKey()
    .map(new TreeSet()(TargetOrdering) + _._2)
    .fold(new TreeSet()(TargetOrdering))(
      joinTargets)

  targetSet
}
```

To generate the initial unmerged set of targets, we rely on the ADAM toolkit's pileup generation utilities (see S5.2). We generate realignment targets for all pileups, even if they do not have indel or mismatch evidence. We eliminate pileups that do not contain indels or mismatches with a filtering stage that eliminates empty targets. To merge overlapping targets, we map all of the targets into a sorted set. This set is implemented using Red-Black trees. This allows for efficient merges, which are implemented with the tail-call recursive *joinTargets* function:

```
@tailrec def joinTargets (
  first: TreeSet[IndelRealignmentTarget],
  second: TreeSet[IndelRealignmentTarget]):
    TreeSet[IndelRealignmentTarget] = {

  if (!TargetOrdering.overlap(first.last,
      second.head)) {
    first.union(second)
  } else {
    joinTargets (first - first.last +
      first.last.merge(second.head),
      second - second.head)
  }
}
```

As we are performing a fold on an RDD which is sorted by the starting position of the target on the reference sequence, we know a priori that the elements in the "first" set will always be ordered earlier relative to the elements in the "second" set. However, there can still be overlap between the two sets, as this ordering does not account for the end positions of the targets. If there is overlap between the last target in the "first" set and the first target in the "second" set, we merge these two elements, and try to merge the two sets again.

[6] Also known as pileups: a group of bases that are all aligned to a specific locus on the reference.

B.3.2 Candidate Generation and Realignment

Candidate generation is a several step process:

1. Realignment targets must "collect" the reads that they contain.
2. For each realignment group, we must generate a new set of candidate haplotype alignments.
3. Then, these candidate alignments must be tested and compared to the current reference haplotype.
4. If a candidate haplotype is sufficiently better than the reference, reads are realigned.

The mapping of reads to realignment targets is done through a tail recursive function that performs a binary search across the sorted set of indel alignment targets:

```
@tailrec def mapToTarget (read: ADAMRecord,
  targets: TreeSet[IndelRealignmentTarget]):
    IndelRealignmentTarget = {

  if (targets.size == 1) {
    if (TargetOrdering.equals (targets.head, read)) {
      targets.head
    } else {
      IndelRealignmentTarget.emptyTarget
    }
  } else {
    val (head, tail) = targets.splitAt(
      targets.size / 2)
    val reducedSet = if (TargetOrdering.lt(
        tail.head, read)) {
      head
    } else {
      tail
    }
    mapToTarget (read, reducedSet)
  }
}
```

This function is applied as a groupBy against all reads. This means that the function is mapped to the RDD that contains all reads. A new RDD is generated where all reads that returned the same indel realignment target are grouped together into a list.

Once all reads are grouped, we identify new candidate alignments. However, before we do this, we left align all indels. For many reads that show evidence of a single indel, this can eliminate mismatches that occur after the indel. This involves shifting the indel location to the "left"[7] by the length of the indel.

[7] To a lower position against the reference sequence.

After this, if the read still shows mismatches, we generate a new consensus alignment. This is done with the *generateAlternateConsensus* function, which distills the indel evidence out from the read.

```
def generateAlternateConsensus (sequence: String,
  start: Long, cigar: Cigar): Option[Consensus] = {
  var readPos = 0
  var referencePos = start

  if (cigar.getCigarElements.filter(elem =>
       elem.getOperator == CigarOperator.I ||
       elem.getOperator == CigarOperator.D
     ).length == 1) {
    cigar.getCigarElements.foreach(cigarElement =>
      { cigarElement.getOperator match {
        case CigarOperator.I => return Some(
          new Consensus(sequence.substring(readPos,
            readPos + cigarElement.getLength),
            referencePos to referencePos))
        case CigarOperator.D => return Some(
          new Consensus("",
          referencePos until (referencePos +
            cigarElement.getLength)))
        case _ => {
          if (cigarElement.getOperator
                .consumesReadBases &&
              cigarElement.getOperator
                .consumesReferenceBases
              ) {
            readPos += cigarElement.getLength
            referencePos += cigarElement.getLength
          } else {
            return None
          }
        }
      }
    })
    None
  } else {
    None
  }
}
```

From these consensuses, we generate new haplotypes by inserting the indel consensus into the reference sequence. The quality of each haplotype is measured by sliding each read across the new haplotype, using *mismatch quality*. Mismatch quality is defined for a given alignment by the sum of the quality scores of all bases that mismatch against the current alignment. While sliding each read across the new haplotype, we aggregate the mismatch quality scores. We take the minimum of all of these scores and the mismatch quality of the original alignment. This sweep is performed using the *sweepReadOverReferenceForQuality* function:

```
def sweepReadOverReferenceForQuality (
    read: String,reference: String,
    qualities: Seq[Int]): (Int, Int) = {
  var qualityScores = List[(Int, Int)]()

  for (i <- 0 until (reference.length -
      read.length)) {
    qualityScores = (
      sumMismatchQualityIgnoreCigar(
        read,
        reference.substring(i, i + read.length),
          qualities),
        i) :: qualityScores
  }

  qualityScores.reduce ((p1: (Int, Int),
      p2: (Int, Int)) => {
    if (p1._1 < p2._1) {
      p1
    } else {
      p2
    }
  })
}
```

If the consensus with the lowest mismatch quality score 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.

B.4 Duplicate Marking Implementation

The following ADAM code, reformatted for this report, expresses the algorithm succinctly in 42 lines of Scala code.

```
for (((leftPos, library), readsByLeftPos) <-
    rdd.adamSingleReadBuckets()
       .keyBy(ReferencePositionPair(_))
       .groupBy(leftPositionAndLibrary);

buckets <- {

leftPos match {
 // These are all unmapped reads.
 // There is no way to determine if
 // they are duplicates
 case None =>
    markReads(readsByLeftPos.unzip._2,
      areDups = false)
 // These reads have their left position mapped
 case Some(leftPosWithOrientation) =>
   // Group the reads by their right position
   val readsByRightPos = readsByLeftPos.groupBy(
     rightPosition)
   // Find any reads with no right position
   val fragments = readsByRightPos.get(None)
   // Check if we have any pairs
   // (reads with a right position)
   val hasPairs = readsByRightPos.keys
     .exists(_.isDefined)
   if (hasPairs) {
     // Since we have pairs,
     // mark all fragments as duplicates
     val processedFrags = if (fragments.isDefined
     ) {
       markReads(fragments.get.unzip._2,
         areDups = true)
     } else {
       Seq.empty
     }
     val processedPairs =
         for (buckets <- (readsByRightPos - None)
           .values;
             processedPair <-
                scoreAndMarkReads(buckets.unzip._2))
        yield processedPair
     processedPairs ++ processedFrags
   } else if (fragments.isDefined) {
     // No pairs. Score the fragments.
     scoreAndMarkReads(fragments.get.unzip._2)
   } else {
     Seq.empty
   }
};
read <- buckets.allReads) yield read
```

For lines 1-4, all reads with the same record group name and read name are collected into buckets. These buckets contain the read and optional mate or secondary alignments. Each read bucket is then keyed by 5 $'$ position and orientation and grouped together by left (lowest coordinate) position, orientation and library name.

For lines 8-41, we processed each read bucket with a common left $5'$ position. Unmapped reads are never marked duplicate as their position is not known. Mapped reads with a common left position are separated into paired reads and fragments. Fragments, in this context, are reads that have no mate or should have a mate but it doesn't exist.

If there are pairs in a group, all fragments are marked duplicates and the paired reads are grouped by their right $5'$ position. All paired reads that have a common right and left $5'$ position are scored and all but the highest scoring read is marked a duplicate.

If there are no pairs in a group, all fragments are scored and all but the highest scoring fragment are marked duplicates.

C Predicate Pushdown Proof

Theorem 1. *Predicate pushdown requires no additional disk accesses.*

Proof. If R is the set of records to read, $Proj$ is the projection we are using, and $size(i, j)$ returns the amount of data recorded in column j of record i, the total data read without predicate pushdown is:

$$\sum_{m}^{R} \sum_{i}^{Proj} size(m, i) \quad (1)$$

By using predicate pushdown, we reduce this total to:

$$\sum_{m}^{R} \sum_{i}^{Pred} size(m, i) + \sum_{n}^{R_{p=True}} \sum_{i}^{Proj \setminus Pred} size(n, i) \quad (2)$$

$Pred$ represents the set of columns used in the predicate. The set $R_{p=True}$ represents the subset of records in R that pass the predicate function. We can show that if no records fail the predicate (i.e. $R = R_{p=True}$), then equations 1 and 2 become equal. This is because by definition $Proj = (Pred \cup (Proj \setminus Pred))$.

□

References

[1] Apache Hive. `http://hive.apache.org/`.

[2] Cloudera Impala. `http://www.cloudera.com/content/cloudera/en/products/cdh/impala.html`.

[3] Picard. `http://picard.sourceforge.org`.

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

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

[6] Asnani, H., Bharadia, D., Chowdhury, M., Ochoa, I., Sharon, I., and Weissman, T. Lossy Compression of Quality Values via Rate Distortion Theory. *ArXiv e-prints* (July 2012).

[7] Bafna, V., Deutsch, A., Heiberg, A., Kozanitis, C., Ohno-Machado, L., and Varghese, G. Abstractions for Genomics: Or, which way to the Genomic Information Age? *Communications of the ACM* (2013). (To appear).

[8] Bainbridge, M., Wang, M., Burgess, D., Kovar, C., Rodesch, M., D'Ascenzo, M., Kitzman, J., Wu, Y.-Q., Newsham, I., Richmond, T., Jeddeloh, J., Muzny, D., Albert, T., and Gibbs, R. Whole exome capture in solution with 3 Gbp of data. *Genome Biology 11*, 6 (2010), R62.

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

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

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

[12] Hartl, C. GAParquet. `http://www.github.com/chartl/gaparquet`.

[13] Janin, L., Rosone, G., and Cox, A. J. Adaptive reference-free compression of sequence quality scores. *Bioinformatics* (Jun 2013).

[14] Jones, D. C., Ruzzo, W. L., Peng, X., and Katze, M. G. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res.* (Aug 2012).

[15] Kozanitis, C., Heiberg, A., Varghese, G., and Bafna, V. Using genome query language to uncover genetic variation. *Bioinformatics* (2013).

[16] Kozanitis, C., Saunders, C., Kruglyak, S., Bafna, V., and Varghese, G. Compressing genomic sequence fragments using SlimGene. *J. Comput. Biol. 18*, 3 (Mar 2011), 401–413.

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

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

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

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

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

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

[23] Popitsch, N., and von Haeseler, A. NGC: lossless and lossy compression of aligned high-throughput sequencing data. *Nucleic Acids Res.* (Oct 2012).

[24] Smith, T. F., and Waterman, M. S. Comparison of biosequences. *Advances in Applied Mathematics 2*, 4 (1981), 482–489.

[25] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 553–564.

[26] Twitter, and Cloudera. Parquet. `http://www.parquet.io`.

[27] Wan, R., Anh, V. N., and Asai, K. Transformations for the compression of FASTQ quality scores of next-generation sequencing data. *Bioinformatics 28*, 5 (Mar 2012), 628–635.

[28] Wheeler, D. L., et al. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Res. 36* (Jan 2008), 13–21.

[29] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 13–24.

[30] Yanovsky, V. ReCoil - an algorithm for compression of extremely large datasets of dna data. *Algorithms Mol Biol 6* (2011), 23.

[31] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. 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* (2012), USENIX Association, p. 2.

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

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