ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing

Matt Massie[1], Frank Austin Nothaft[1], Chris Hartl[1,2], Christos Kozanitis[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

Abstract

Current genomics applications are dominated by the movement of data to and from disk. This data movement pattern is a significant bottleneck that prevents these applications from scaling well to distributed computing clusters. In this report, we introduce a new set of data formats and programming patterns for genomics applications using in-memory MapReduce frameworks. These formats and frameworks improve application performance, data storage efficiency, and programmer productivity.

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 [23]. 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 [21].

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]. By rewriting these applications to make use of modern in-memory MapReduce frameworks like Apache Spark [39], these transformations can now be completed in FIXME[2] hours.

In this paper, we introduce ADAM, which is a programming framework 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 programing model (§6). Finally, we review the performance and compression gains we achieve (§7), and outline future enhancements to ADAM that we are working on (§8).

Performance teasers.

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 [21]. BAM files can be operated on in several languages, using either the SAMtools ([21], C++), Picard ([3], Java), and Hadoop-BAM ([24], Hadoop MapReduce through Java) APIs. BAM provides efficient access and compression over the SAM file format, as its binary encoding reduces several fields into a single byte, and eliminates text processing on load. However, the file format has been criticized as it is difficult to process — the three main APIs that implement the format each note that they do not fully implement the format due to its com-

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

[2]FIXME

plexity. 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 8 machines.

In response to the growing size of sequencing files[3], a variety of compression methods have come to light [19, 13, 32, 25, 6, 11, 36, 17, 16]. SlimGene [19], cSRA [33], and CRAM [13] use reference based compression techniques to losslessly represent reads while 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 presented recently a systematic approach of quality score removal in [16] which safely ignores quality scores from predictable nucleotides; these are bases that always appear after certain words. It is also 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. These steps minimize the amount of erroneous data in the input read set by eliminating duplicate data, verifying the alignment of short inserts/deletions (indels), and calibrating the quality scores assigned to bases (base quality score recalibration, BQSR). The typical pipeline for variant calling is illustrated in figure 1.

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. A breakdown of stage execution time is shown in table 1 for the version 2.7 of the Genome Analysis Toolkit (GATK), a popular variant calling pipeline [22]. The numbers in the table are derived from running on the NA12878 high-coverage human genome.

To provide the readers with a background about how the stages of this pipeline work, we will discuss

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

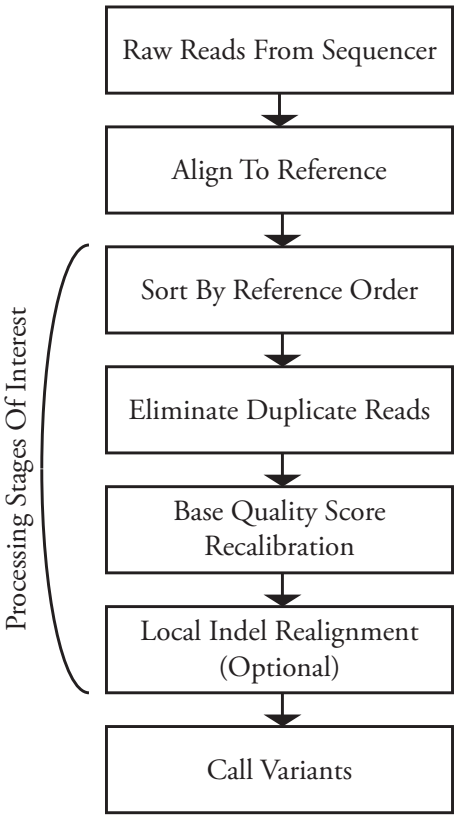

Figure 1: Variant Calling Pipeline

the algorithms that implement the intermediate read processing stages. For a detailed discussion of these algorithms, we refer readers to DePristo et al [12].

Sorting: This phase performs a pass over the reads and sorts them by the reference position at the start of their alignment.

Duplicate Removal: An insufficient number of sample molecules immediately prior to 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 66% of duplicates marked in this way are true duplicates caused by PCR-induced duplication while the remaining 33% are caused by the random distribution of read ends[8]. There is currently no way to separate "true" duplicates from

Table 1: Processing Stage Times for GATK Pipeline

Stage	GATK 2.7/NA12878
Alignment	52 hr
Sort	
Mark Duplicates	13 hr
BQSR	6 hr
Realignment	33 hr
Call Variants	4 hr
Total	**67 hr**

randomly occurring duplicates.

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.

Local Realignment: For performance reasons, all modern aligners use algorithms that provide approximate alignments. This 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 [27]) 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 that is implemented is not susceptible to these local alignment errors.

For current implementations of these read processing steps, performance is limited by disk bandwidth. This is 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 by performing our processing iteratively in memory. The four read processing stages can then be chained together, eliminating three dumps to disk and an additional three long reads from disk.

4 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 [40]. 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. We present a similar decomposition of services for genomics data in figure 2.

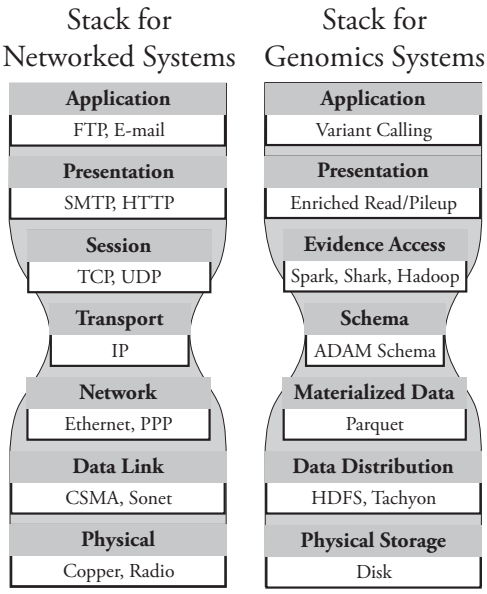

Figure 2: Stack Models for Networking and Genomics

REVIEW STACK

The seven levels of our stack model are decomposed as follows:

1. **Physical Storage:** This layer coordinates data writes to physical media.
2. **Storage:** 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.
4. **Data Schema:** This level 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 sits on top of the data access layer and 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 [24], and new access patterns must be carefully retrofitted to the data format that is in use [18].

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 latency and throughput, our systems must be able to scale efficiently to tens to hundreds of nodes.

2. **Bioinformaticans should be concerned with data, not formats:** The SAM/BAM formats do not provide a schema for the data they

   REFACTOR THIS

   store. This 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 [21, 3, 24].

3. **Openness and flexibility are critical to adoption:** As the cost of sequencing continues to drop [23], 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.

In the next few 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 discuss the greater 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. In this section, we discuss the frameworks used to store this data. We then introduce the representations, and discuss the content of 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 Scala, Java, C/C++/C#, Python, Ruby, and php. This 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 [21, 3].

We layer this data format inside of the Parquet column store [30]. Parquet is an open source columnar storage format that was designed by Cloudera and Twitter, which 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 enable predicate pushdown [20], which minimizes the amount of data read from disk. 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.

- Column stores achieve extremely high compression [4]. This reduces storage space on disk, and also improves serialization performance inside of MapReduce frameworks. We are able to achieve a 0.75× lossless compression ratio when compared to BAM, and have especially impressive results on quality scores. This is discussed in more detail in §7.3.
- Varied data projections can be achieved with column stores. This means that we can choose to only read several fields from a record. This 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 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.

A visualization of how ADAM in Parquet compares with BAM can be seen in figure 3. We remove the file header and instead distribute these values across all of the records stored. This eliminates global information and makes the file much simpler to distribute across machines. This 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 [30], see Parquet format readme) — this allows parallelism across rows by enabling the columnar store to be split into independent groups of reads.

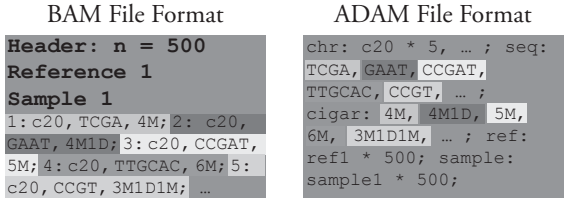

Figure 3: Comparative Visualization of BAM and ADAM File Formats

As discussed in §4, the internal data types presented in this section make up the narrow waist of our proposed genomics stack. 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.

5.1 Read Oriented Storage

Our default read oriented data format is defined in table 2. This format provides all of the fields supported by the SAM format. 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.

5.2 Reference Oriented Storage

In addition to storing sequences as reads, we provide a storage format for reference oriented (pileup) data. This format is documented in table 3. This pileup format is also used to implement a data storage format similar to the GATK's ReducedReads format [12].

We treat inserts as an inserted sequence range at the locus position. For bases that are an alignment match against the reference[4], 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.

As noted earlier, this datatype supports an operation similar to the GATK's ReducedRead datatype. We refer to these datatypes as aggregated pileups. We perform aggregation 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

[4]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.

Table 2: 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

soft clipped bases that show up at this location, and the amount of bases that are mapped to the reverse strand.

5.3 Variant and Genotype Storage

Chris H.

Variant type is one of SNP, multiple nucleotide polymorphism (MNP), insertion, deletion, structural variant (SV), or complex.

Table 3: 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

Table 4: Genotype Format

Field	Type
Sample ID	String
Genotype	Array[Int]
Likelihood (Phred)	Array[Int]
Format	String

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[5], it significantly increases the latency of the processing pipeline.

For the processing pipeline we have built on top

[5]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 [38].

Table 5: Variant Format

Field	Type
Reference Name	String
Position	Long
Variant ID	String
Reference Allele	String
Alternate Alleles	Array[String]
Allele Count	Array[Int]
Chromosome Count	Int
Type	Variant Type
Info	String
Filter	String
Genotypes	Array[AdamGenotype]

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 review several testcases 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.
- **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.

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 [24], 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.

7.1.1 Read/Write Performance

The decision to use a columnar store was based off of 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. The results of this benchmark can be found in Figure 4.

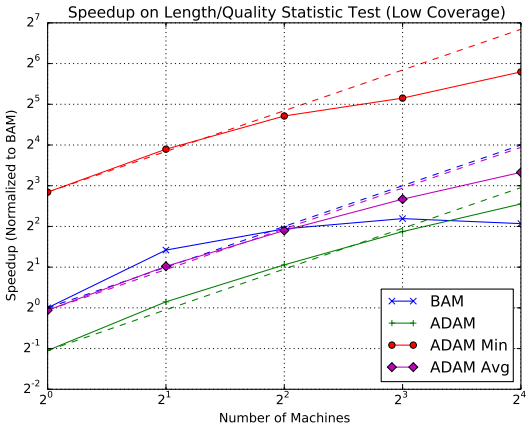

Figure 4: Read Length and Quality Speedup for HG00096

7.1.2 Predicate Pushdown

Predicate pushdown is one of the significant advantages afforded to us through the use of Parquet [30]. 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.

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:

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

By using predicate pushdown, we reduce this total to:

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

$Pred$ 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 $Proj = (Pred \cup (Proj \setminus Pred))$.

□

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.

7.2 Applications

In this section, we review the performance of the Sort and Mark Duplicate applications. Although implementations of BQSR and local realignment exist, we have not been able to fully characterize their performance. As such, we describe these implementations in the end of the paper, but we do not discuss their current performance. We will report on their performance in a forthcoming article.

7.2.1 Sort

Matt

7.2.2 Mark Duplicates

Matt

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.

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 which will allow sequencing data to be queried from the popular SQL based warehouse software such as Shark [35], 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 [30].

Of course there is a lot of 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 [18] and also text based search features on fields such as the read sequence and the CIGAR string. To keep storage 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.

9 Discussion

Table to go here.

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 [29]. 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.

However, it is conceivable 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 §4 allows us to easily change our implementation. Specifically, we can swap out columnar storage in the data access layer with row oriented storage [6]. 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 Dancehall vs. Datacenter

Matt

9.3 New Applications Enabled By ADAM

The ADAM format and environment provides a significant performance improvement over current BAM based solutions. By reducing processing latency, ADAM will make genomic data available for use in latency-critical applications, such as genomic driven treatment of acute diseases. Additionally, the genomics stack design we proposed in §4 unlocks new computational patterns that can be applied to genomic data. In this section, we introduce a few of these novel applications, discuss why they are important, and explain how these applications are enabled by ADAM.

9.3.1 Personalized Treatment of Acute Diseases

There is significant interest in using genomic data as another layer of evidence for the treatment of diseases that have genetic causes [14, 31]. While chronic hereditary diseases like Huntington's Disease may not

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

be sensitive to the latency of genomic processing pipelines, reduced latency is critical for neonatal diagnosis and for the treatment of some cancers. In acute myeloid leukemia (AML), there exists a well defined set of mutations that can be used for guiding patient treatment [26]. However, due to the rapid progression of AML, treatment decisions must be made in under one week. As illustrated in §3, this target is not within the reach of current processing pipelines.

It is worth noting that the numbers presented in §3 would be optimistic in reality. To improve accuracy, the current state of the art in mutation calling uses very high coverage sequencing data ($\sim 100 - 150\times$) for the tumor cell line, along with high coverage sequencing data ($\sim 30\times$) for the normal cell line [10]. We are developing a variant and mutation calling pipeline that uses the ADAM processing pipeline and the high-performance SNAP aligner [37] to achieve performance that will satisfy the latency requirements for use in acute disease diagnosis.

9.3.2 Viral Outbreak Monitoring

There is significant interest in tools for the analysis of viral outbreaks in both the medical and defense epidemiology communities [28, 9]. Genomic techniques can be used for identifying infection vectors in epidemics that feature rapidly mutating diseases, as well as for the rapid identification of unknown infectious agents in an attack using biological weapons. Similar to the acute genetic driven diseases discussed in §9.3.1, both of these applications are latency sensitive; the United States Department of Defense (DoD) has set a goal of developing a system which can identify an unknown infectious agent within one hour with the use of genomic data.

In addition to providing significantly lower latency than current pipelines, there are other advantages to ADAM. Current approaches to tracking viral outbreaks perform a graph based analysis that uses variant data and temporal data to identify possible modes of disease transmission [28]. As ADAM is implemented on top of Spark, data in ADAM format can be processed as a graph by using the GraphX distributed graph analysis framework [34].

9.3.3 Ad Hoc Genomic Analysis

ADAM is implemented on top of the Spark MapReduce framework. A major contribution of the Spark framework was the introduction of a command line interface that allows for the real-time ad hoc analysis of data kept in memory [38]. This interactive framework can be coupled with the ability to perform interactive queries in Shark to allow for rapid data aggregation and analysis. Several applications for this approach are presented by Kozanitis et al. [18] such as allowing direct evidence access to physicians who are working in a clinical setting with patients with hereditary diseases. However, GQL performance limited the performance of these queries to $\sim$ 45 minutes. We are still working on the implementation of our database interface (see §8.1), but expect to achieve improved performance.

In addition to clinical applications, interactive data access could be applied to pathway analysis. Specifically, if a biologist has genetic data for organisms who demonstrate a specific phenotype, interactive data processing could be used for correlating lab observations with genotypes. Instead of genotyping the full genome/exome of all reference organisms, the raw reads for all organisms could be aligned in stored in a database. Once lab measurements and cytological observations have allowed the biologist to hypothesize that a specific pathway had failed, the database of raw reads could be queried for all genes of interest (genes that influence the pathway under study). This would lead to a significant reduction in the amount of genomic data that needed to be processed, which would allow for interactive read processing and variant calling.

9.4 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 [38]. 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 level 6 in the stack we proposed in section §4. 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 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 provide a 2× improvement in programmer productivity. This is demonstrated through GAParquet, which demonstrates some of the functionality in the GATK on top of the ADAM stack [15]. 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. A LOC comparison of GATK/GAParquet and the read transformation pipeline is found in table 6.

Table 6: Lines of Code for ADAM and Original Implementation

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

10 Summary

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

```
TBD
```

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

B.1 Sort Implementation

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 re-

gions 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"[7]. 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
}
```

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

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.

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"[8] by the length of the indel. 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
  }
}
```

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

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

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] Beard, J. DARPA's bio-revolution. *Biomedical services* (2008), 4–6.

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

[11] 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.

[12] 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.

[13] 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.

[14] Ghosh, D., and Poisson, L. M. "omics" data and levels of evidence for biomarker discovery. *Genomics 93*, 1 (2009), 13–16.

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

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

[17] 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).

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] Smith, C. C., Wang, Q., Chin, C.-S., Salerno, S., Damon, L. E., Levis, M. J., Perl, A. E., Travers, K. J., Wang, S., Hunt, J. P., et al. Validation of ITD mutations in FLT3 as a therapeutic target in human acute myeloid leukaemia. *Nature 485*, 7397 (2012), 260–263.

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

[28] Snitkin, E. S., Zelazny, A. M., Thomas, P. J., Stock, F., Henderson, D. K., Palmore, T. N., Segre, J. A., et al. Tracking a hospital outbreak of carbapenem-resistant klebsiella pneumoniae with whole-genome sequencing. *Science translational medicine 4*, 148 (2012), 148ra116–148ra116.

[29] 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.

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

[31] Van Allen, E. M., Wagle, N., and Levy, M. A. Clinical analysis and interpretation of cancer genome data. *Journal of Clinical Oncology 31*, 15 (2013), 1825–1833.

[32] 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.

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

[34] Xin, R. S., Gonzalez, J. E., Franklin, M. J., Stoica, I., and AMPLab, E. GraphX: A resilient distributed graph system on Spark.

[35] 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.

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

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

[38] 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.

[39] 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.

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