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 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 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 [11]. 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 [9].

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 Spark [16], 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 speedup 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).

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 [9]. BAM files can be operated on in several languages, using either the SAMtools ([9], C++), Picard ([3], Java), and Hadoop-BAM ([12], Hadoop MapReduce through Java) APIs. BAM provides efficient access and compression over the SAM file format, as it's 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 it's complexity. Additionally, the file format is difficult to use in multiprocessing environments due to it's use of a centralized header; the Hadoop-BAM implementa-

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

[2]FIXME

tion notes that it's scalability is limited to distributed processing environments of less than 8 machines.

In response to the growing size of BAM files[3], the CRAM format has been proposed as a standard for archival purposes [6]. This format is a binary format that implements similar fields as BAM but with better compression. Lossy compression is achieved by only storing differences from the reference, and by binning quality scores to reduce the amount of data that needs to be saved. This format provides it's own API, and has some compatibility with the existing Picard API for Java.

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 [10]. The numbers in the table are derived from running on the NA12878 high-coverage human genome.

Table 1: Processing Stage Times for GATK Pipelines

Stage	**GATK 2.7/NA12878**
Alignment	6 hr
Sort	
Mark Duplicates	
BQSR	
Realignment	
Call Variants	

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

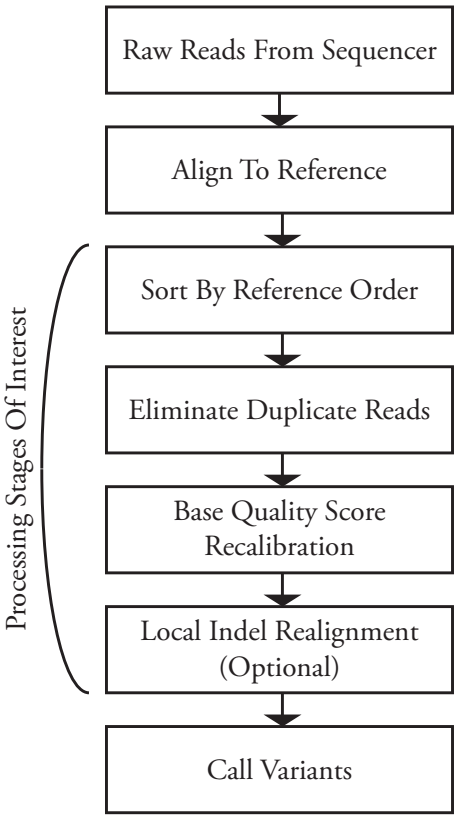

Figure 1: Variant Calling Pipeline

To provide the readers with a background about how the stages of this pipeline work, we will discuss the algorithms that implement the intermediate read processing stages. For a detailed discussion of these algorithms, we refer readers to DePristo et al [5].

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

Duplicate Removal: Due to imperfections during the sequencing process, an unknown number of reads are optically duplicated. If these duplicate reads are not removed, they will then incorrectly bias the variant caller, and can lead to incorrect variant calls. In this stage, we review all reads mapped at a specific loci, and discard reads that have identical sequence information.

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 [13]) 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 [17]. This conceptual model standardized the internal functionality of a networked computer system, and 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.

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

1. **Storage:** This layer manages access, replication, and distribution of the genomics files that have been written to disk.
2. **Data Format:** This layer defines how data is written to disk, and how data is serialized in a distributed computing environment.
3. **Compute Substrate:** The compute substrate layer provides efficient mechanisms for implementing control flow and distributing work.

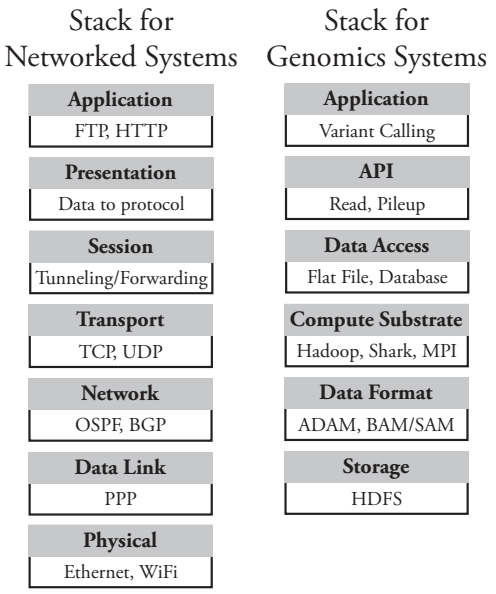

Figure 2: Stack Models for Networking and Genomics

4. **Data Access:** This layer provides access to the data stored on disk, and implements efficient methods for performing common access patterns such as random database queries, or sequentially reading records from a flat file.
5. **API:** The API 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.
6. **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 [12], and new access patterns

must be carefully retrofitted to the data format that is in use [7].

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 [1]. 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 [9, 3].

We layer this data format inside of the Parquet column store [2]. Parquet is an open source columnar storage format that was designed by 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 [8], 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.

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 [5].

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.

[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

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

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

5.3 Variant and Genotype Storage

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

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],

[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

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]

it significantly increases the latency 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

7.1 Microbenchmarks

7.2 Applications

7.3 Compression

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.

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 [15].

8.1 Database Integration

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

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 [14]. 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.

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

9.2 Big Data vs. Traditional Processing

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.

11 Availability

The ADAM source code is available at Github at `http://www.github.com/massie/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.

References

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

[2] Parquet. `http://www.parquet.io`.

[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] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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