BIOINFORMATICS

Vol. 00 no. 00 2014
Pages 1–7

ADAM: A Data Format And Pipeline For Cloud Scale Genomic Processing

Matt Massie [1,*], Frank Austin Nothaft [1,*], Christopher Hartl [2], Christos Kozanitis [1], André Schumacher [3], Timothy Danford [4], Carl Yeksigian [4], Jey Kottalam [1], Arun Ahuja [5], Neal Sidhwaney [5], Jeffery Hammerbacher [5], Michael Linderman [5], Anthony D. Joseph [1], and David Patterson [1*]

[1]Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA
[2]The Broad Institute of MIT and Harvard, Cambridge, MA
[3]International Computer Science Institute (ICSI), University of California, Berkeley, CA
[4]GenomeBridge, Cambridge, MA
[5]Carl Icahn School of Medicine at Mount Sinai, New York, NY

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Motivation: Current genomics data formats and processing pipelines are not designed to process large datasets quickly. As the quantity of genetic data continues to increase, it is desirable to be able to process this data in the cloud. We introduce a new set of file formats that are designed for cloud computing as a potential successor to the BAM and VCF standards.

Results: On a high coverage (60×) 250GB NA12878 BAM file, we are able to perform Sorting and Duplicate Marking in under 50 minutes on 100 node cluster (a 50× performance improvement). On a single node, we are twice as fast as Picard and Samtools. ADAM files are up to 25% smaller than equivalent compressed BAM files.

Availability: The ADAM project website is at `http://adam.cs.berkeley.edu`. ADAM is open source under the Apache 2 license, is deployed through Maven, and the source is available through GitHub.

Contact: {massie,fnothaft,pattrsn}@berkeley.edu

1 INTRODUCTION

The Sequence/Binary Alignment/Map (SAM/BAM) and Variant Call Format (VCF) file formats were designed before multi-node processing and cloud computing were in vogue—thusly, they were designed for single node processing (Li *et al.*, 2009). As noted by McPherson (2009), as next generation sequencing (NGS) methods continue to improve, the latency of alignment and variant calling is becoming increasingly costly for scientists who are using genomics in a clinical setting, or who are working on very large datasets. To address this issue, it is desirable to be able to compute on genomic data across many machines, but characteristics of the BAM and VCF file formats practically limit scalability to 8 nodes (Niemenmaa *et al.*, 2012).

As the scalability limitations are inherent to the file formats[1], we choose to rethink file formats for genomics, instead of looking to incrementally improve either BAM or VCF. We view the following points as critical design questions:

- BAM and VCF optimize for row-oriented, flat file access. Is this access pattern the best pattern for genomics?
- Scientists collectively want to process data in several different programming languages. How can we avoid incompatibilities between data processed using different languages or libraries?
- Genomic data will be processed on various computing systems, including single workstations, large dedicated clusters, and using cloud computing services. How can we jointly optimize for these diverse platforms?
- The file formats we use must be flexible enough to support new fields, but should not become unmanageable. Is there a better way to support this extensibility than attribute maps?

To address these problems, we introduce ADAM, a set of formats, application programming interfaces (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 (Apache, 2012a) and Parquet (Twitter and Cloudera, 2013) for data storage. Our reference pipeline is implemented on top of Spark, a high performance in-memory map-reduce system (Zaharia *et al.*, 2010). This combination provides the following advantages:

- Avro provides an explicit data schema, which enables fully compatible accesses in C/C++/C#, Java/Scala, Python, php, and Ruby;

*to whom correspondence should be addressed

[1] This is discussed in more detail in §2.

- Parquet allows efficient distributed access and use by database systems like Impala and Shark; and
- Spark improves performance through in-memory caching and reducing disk I/O.

These changes lead to significant performance improvements, and improve the format's cross-platform portability. 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, fulfilling the promise of scalability of ADAM. We provide an extensive review of ADAM's performance in §3.3.

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. Additionally, any application that implements the ADAM schema is compatible with ADAM. This 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 1). We consider this stack model to be the greatest contribution of ADAM.

In this paper, we start by introducing why a new approach is necessary, and what our new approach is in §2. We review the design and performance of our pipeline in §3 and §3.3. Finally, we provide a comprehensive comparison of ADAM against other data formats and discuss the future growth of ADAM in §4.

2 APPROACH

As we noted in the introduction, instead of trying to extend the BAM and VCF file formats, we choose to reimagine these formats. Despite several attempts to adapt the BAM and VCF file formats for distributed processing (see Niemenmaa *et al.*, 2012), they have been limited in their ability to scale to larger cluster sizes due to several issues intrinsic to BAM and VCF:

- BAM and VCF both depend on centralized headers, which must be globally distributed; and
- BAM and VCF both have irregular record sizes, which limits the efficiency of row-parallel reading.

These characteristics limit the performance of applications that desire parallel access to all records of a BAM or VCF file. However, this points at another problem: some modern bioinformatics file formats lock users into a single processing platform or technique. There are several reasons that traditional flat file formats are not ideal for genomics, such as:

- **Database Queries:** Extending BAM to provide standard database queries required a significant extension (Kozanitis *et al.*, 2013). No such extension exists for VCF data.
- **Efficient Predication:** Typically, a user would only like to process a subset of records. Currently, for both BAM and VCF, a user must load all records and then perform a filter, which is inefficient. BAM supports the inclusion of an index file (BAM Index, BAI), but this index file can only accelerate reference position based filtering patterns[2].
- **Projection of Specific Fields:** It is typical for an application to only read some fields of a BAM record across all records, especially for statistics collection tools like SAMtools' `flagstat` (Li *et al.*, 2009). Row-oriented formats like BAM and VCF cannot support these access patterns.

Moving forward, we seek to meet two major goals: we want to provide an extensible architecture that will allow for efficient future extension of the file format, and we want to provide an initial implementation of that architecture that addresses the problems introduced above. In the remainder of this section, we review our high-level architecture that provides extensibility, and then we provide a technical overview of our reference pipeline.

2.1 Architectural Overview

The primary takeaway of the criticisms of BAM and VCF are that the formats are difficult to specialize for certain processing patterns without changing the implementation of the formats themselves. This issue is similar to the problems addressed during the development of the Open Systems Interconnection (OSI) model and Internet Protocol (IP) stack for networking services (Zimmermann, 1980). Specifically, the developers of the OSI model and the IP stack needed to make many different technologies and systems function in unison—to do this, they introduced the concept of a "narrow waist," which guaranteed that a new protocol or technology would be compatible with the rest of the system if it implemented one specific interface.

We draw inspiration from the development of networking standards—we believe that our largest contribution is the explicit ADAM schema, which is the "narrow waist" in our stack. This schema allows components to be cleanly interchanged as long as they implement the ADAM schema. Figure 1 shows our stack.

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.
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 sequential/parallel reading of records from a flat file.
6. **Presentation:** The presentation layer provides the application developer with efficient and straightforward methods for

[2] Typically, the first step of variant calling involves a filter that removes low quality reads. These filters are based on position-independent data (e.g. mapping quality, boolean flags) and cannot be accelerated with the BAI.

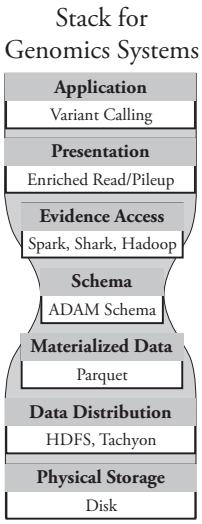

Fig. 1. A Stack Model for Genomics

querying the characteristics of individual portions of genetic data.

7. **Application:** Applications like variant calling and alignment are implemented in this layer.

The ADAM schema is represented using Apache Avro (Apache, 2012a), which is an open-source, cross-platform data serialization framework, similar to Apache Thrift and Google's Protocol Buffers (Apache, 2012c; Google, 2012). Avro was chosen as the interchange format for several reasons:

- Avro is an open-source framework covered by the Apache 2 license. This means that Avro can be used with both open and closed source software.
- Avro has broad cross-platform support. Natively, Avro supports C/C++/C#, Java, Scala, Python, Ruby, and php.
- Avro provides a clear and human readable language for explicitly describing the schema of an object (Avro Description Language, AVDL).
- Avro is natively supported by several common map-reduce frameworks and database systems.
- Avro schemas can be updated without breaking compatibility with objects written using a previous version of the schema.

As noted above, for a system to implement the ADAM format, it must read/write ADAM objects that are defined by Avro schemas. Schemas minimize tool/vendor lock-in.

A well defined software stack has several other significant advantages. By limiting application interactions with layers lower than the presentation layer, application developers are given a clear and consistent view of the data they are processing. 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. By treating the compute substrate and storage as separate layers, we also drastically increase the portability of the APIs that we implement.

2.2 Reference Implementation

We present a reference implementation of a read processing pipeline (`adam-core`). This implementation is fully open source and is designed for a distributed, in-memory map-reduce framework. We do not optimize for a specific computing platform—rather, we have designed agnostically for a commodity cloud computing platform. We discuss this decision in §4.1. Our reference implementation provides several read processing stages that can be either run from a command line, or executed by another application through an API, as well as an API for enriched datatypes. These APIs are similar to those provided by SAMtools (Li *et al.*, 2009) and Picard (Picard, 2013).

As mentioned in §1, we built our system using the Avro data serialization format, the Parquet columnar store, and the Spark map-reduce framework (Apache, 2012a; Twitter and Cloudera, 2013; Zaharia *et al.*, 2010). We have already explained the reason for using Avro, we now give the rationale for using Spark and Parquet.

Spark, In-Memory Map-Reduce: As introduced in Zaharia *et al.* (2010) and further described in Zaharia *et al.* (2012), Spark is an in-memory, distributed map-reduce framework that is compatible with the Hadoop ecosystem. Spark differs from conventional map-reduce frameworks like Hadoop (Apache, 2012b) and Google MapReduce (Dean and Ghemawat, 2008), as it avoids writing data to disk whenever possible and instead caches data in memory. This provides a dramatic speedup for iterative computations—Spark sees a 30–100× speedup on iterative jobs by eliminating disk I/O and by improving resource utilization by pipelining computation. This pattern is a good fit for genomics because:

- With the increasing size of genomic datasets, disk I/O is a bottleneck for modern genomics systems. By persisting data in memory between processing stages, we are able to lessen the disk I/O penalty and improve latency.
- Many genomic processing stages are iterative. For many applications, iteration implies that stages can be pipelined, which improves system performance on large parallel systems[3].

An additional benefit of Spark is that its programming interface supports Java, Scala, and Python[4].

Parquet, Columnar File Format: Parquet is a columnar file format that is designed for distributed computing. Columnar stores differ from conventional file formats as they group data together by field instead of by record[5]. This orientation enables very high read performance, improved compression (see Abadi *et al.*, 2006), and the efficient serialization of a subset of fields. Parquet has otherbenefits:

[3] Performance is improved as pipelining reduces the number of executors that become idle at the start and end of stages by overlapping the two stages.

[4] A programming interface for the R programming language will be available in the first half of 2014.

[5] In other words, all values for a single field are stored sequentially on disk.

- Parquet operates by splitting a single "file" into many smaller files that contain *row groups*[6]. This division enables both parallelism by column and by row which increases read parallelism.
- Parquet provides *predicate pushdown*, where user defined filters are applied while records are being read. This feature can lead to a $>2\times$ performance increase. A detailed analysis of Parquet's predicate pushdown functions for genomics applications can be found in Massie *et al.* (2013).

Figure 2 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.

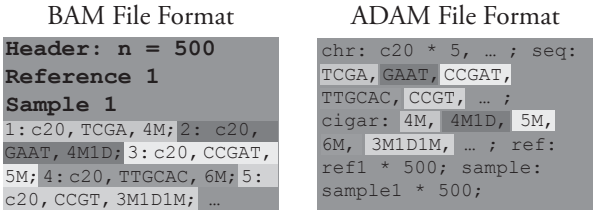

Fig. 2. Comparative Visualization of BAM and ADAM File Formats

Our reference pipeline implementation provides two components:

- `adam-cli`, a command line toolkit for converting data to/from the ADAM format, and for performing transformations, and
- `adam-core`, an API for processing genomic data that is stored in the ADAM format and is contained in memory.

We further describe the functionality provided by the command line toolkit and the API in §3.2.

3 METHODS

ADAM contains formats for storing read and reference oriented sequence information, and variant/genotype data. The read oriented sequence format is both 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 specific data formats that we provide. Additionally, we describe the `adam-cli` toolkit and `adam-core` API which were introduced in §2.2.

[6] In Parquet, a *row group* contains a subset of rows of data from the dataset. The row groups are then stored in columnar format.

3.1 Data Format and Schema

The data representation for the ADAM format is described using the open source Apache Avro data serialization system (Apache, 2012a). 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 (Li *et al.*, 2009) and for Java through Picard (Picard, 2013). Additionally, there are known incompatibilities between these different implementations—by using Avro, we eliminate these platform compatibility issues.

The data types presented in this section make up the narrow waist of our proposed genomics stack in Figure 1. We provide the following primary datatypes:

- **ADAMRecord:** This datatype is for storing read data, and provides similar semantics to the read storage in SAM/BAM;
- **ADAMPileup:** This datatype stores reference-oriented pileup data, broken out by individual read base;
- **ADAMGenotype:** This datatype represents the genotype of a single chromosome of a single sample; and
- **ADAMVariant:** This datatype represents a single variant allele which segregates at a site.

We do not provide a full description of all fields in our standard in this paper—a full description of the fields of the data types is available in the ADAM technical report (see Massie *et al.*, 2013, §5). We note that the internal representations of the data may change at any point in time—for the most up to date schema, we refer readers to the ADAM repository.

We next discuss novel details of our format implementation, and contrast the structure of our formats against the current BAM and VCF formats.

ADAMRecord, DNA Sequence Data: An individual ADAMRecord is equivalent to a single read stored in SAM/BAM format. To improve parallelism, we eliminate the need for header references. This simplification is done by replicating the information that is contained in the header across all records. While this would be too expensive to implement in a traditional row-oriented file format, the use of a columnar storage format like Parquet allows us to use run-length encoding (RLE) to make this tractable. RLE is an efficient method that encodes that a single piece of information is replicated across many (not necessarily all) records. Even though we replicate the header information across all records, ADAMRecord data consumes less space than BAM—ADAMRecord files are between 2% and 25% smaller than a compressed BAM file.

ADAMPileup, Locus-Oriented DNA Data: ADAM's pileup storage format stores individual read bases. This approach contrasts with traditional pileup formats that instead store all the bases which are found at a single reference position. There are several reasons that inform this choice:

- By storing individual bases separately, it is easier to use database methods to query across reference oriented data.

- We provide more intuitive formats for storing reference oriented insertion/deletion data. Specifically, our pileup storage format provides gap-based indexing, which disambiguates the length of a deletion or insertion.
- From the single read base format we provide, read data can be reconstructed losslessly from pileup data.

We do realize that many users are interested in viewing bases through the traditional pileup view. We provide an internal representation (*ADAMRod*) that provides efficient access to this data.

Additionally, we provide *pileup aggregation*, a technique that is similar to `ReducedReads` (see McKenna *et al.*, 2010). Pileup aggregation takes all bases that share a base type (e.g. cytosine, guanine, etc.), and averages their statistical values. This technique is useful in some circumstances (e.g. data archival) as it leads to a reduction in the amount of data stored that scales approximately as $O(1/coverage)$.

ADAMGenotype, Single Chromosome Genotypes, and ADAMVariant, Multi-Sample Variant Data: ADAM implements genotypes without relying on the context of a variant. An individual ADAMGenotype record describes a called genotype on a single chromosome of a sample. The metrics fields of a genotype (e.g. quality) mirror the metrics fields of the variant type. This enables genotypes to be expressed without the centralized notion of a variant—this is useful for data warehousing, where the genotypes of many samples may be gathered. At a later point in time, it may be desirable to collect and distribute variant calls for a subset of these samples; this subset may span multiple variant calling runs. By mirroring fields between Genotype and Variant records, we allow ourselves to compute variant statistics from a set of genotypes without having access to the raw read data. This transformation is implemented in both the `adam-cli` and the `adam-core` API.

Annotations are not a first class citizen of the VCF format—annotations are shoehorned in to the format through the use of the attributes field. For ADAMGenotypes and ADAMVariants, we allow users to define record schemas that are not part of the core variant/genotype schemas. We then additionally provide an internal ADAMVariantContext class that unifies annotations with specific genotypes and variants. Annotations are typically unified through the use of user defined *joins*, which are a database construct. There are several advantages to this process:

- Since users are defining records for annotations, annotated data can now be more complex—specifically, annotated data can contain multiple fields, or complex fields.
- As the *joins* are user defined, no limitations are placed on the forms of annotations that are provided. Annotations can be specific to a site, variant, genotype, sample, or etc.

Additionally, since annotations are defined through an explicit schema, annotation maintenance becomes significantly simpler.

ADAM Sequence Dictionaries: **Timothy, can you write this?**

3.2 Data Transformations

In this section, we provide a brief discussion of the transforms and utilities implemented inside of ADAM. For an extended description of the *Sorting*, *Mark Duplicates*, *BQSR*, and *Indel Realignment* algorithms, we refer readers to the appendix of Massie *et al.* (2013) which discusses these algorithms in more detail. We discuss the command line utilities that we provide, and give a brief overview of some of the transformations that our API provides.

Command Line Utilities: The `adam-cli` suite of command line utilities implements several important features. They include:

- Converters between:
  - ADAMRecord ↔ SAM/BAM
  - ADAMVariant and ADAMGenotype ↔ VCF
  - ADAMRecord → ADAMPileup
  - ADAMRecord → SAMtools style pileups
- *AggregatePileups*, which aggregates pileup data (see §3.1),
- *CompareAdam*, which compares two ADAM read files,
- *ComputeVariants*, which computes variant data from a set of genotypes (see §3.1),
- *Flagstat*, which is analogous to SAMtools' `flagstat` command,
- *ListDict*, which lists the content of an ADAM sequence dictionary,
- *PrintAdam*, which prints the contents of an ADAMRecord file, and
- *Transform*, which allows users to perform sorting, duplicate marking, and BQSR on read data.

The translations between ADAM and SAM/BAM/VCF use the Hadoop-BAM framework (see Niemenmaa *et al.*, 2012) to read and write SAM/BAM/VCF data—this allows for these translations to be distributed, which accelerates the translations. Similarly, all of the commands that operate on read data can operate on ADAMRecord, SAM, or BAM data; if SAM or BAM data is provided, it is transformed on-demand into ADAMRecords.

API Transformations: The `adam-core` API provides an extensive set of transformations that are performed on data stored in Resilient Distributed Dataset (RDD) form. RDDs were introduced in Zaharia *et al.* (2012) as a fault-tolerant abstraction for distributing an array of data across multiple machines. The API transformations we provide form the base on which the `adam-cli` is built. Some important transformations provided by our API include:

- Read data:
  - Sort by reference position
  - Mark Duplicates
  - BQSR
  - Realign Indels
  - Collect reference sequence dictionary
  - Collect metrics
  - Translate reads to pileups

- Pileup data:
  - Aggregate pileup data
  - Group pileups by reference position
  - Seperate reference position grouped data by samples
  - Compute coverage metrics
- Genotype data:
  - Validate multi-sample genotype data
  - Compute variant data from genotypes

Additionally, our API provides several enriched datatypes that provide better access semantics for developers. These enriched types include:

- *ADAMVariantContext*, a type which unifies variant, genotype, and annotation data;
- *RichADAMRecord*, a wrapper around ADAMRecords that converts some fields into richer record types (e.g. CIGAR/MD strings);
- *ADAMRod*, a type that wraps pileup bases at a single locus position;
- *MdTag*s, an enriched class for accessing alignment mismatch data; and,
- *ReferencePosition*, *ReferencePositionPair*, and *ReferenceRegion*, which are helper classes that provide rich semantics for describing the relationship between an object and it's location on the reference genome.

The `adam-core` API also provides partitioners for reference-aligned data. These partitioners work with the Spark map-reduce framework to efficiently co-locate data that has spatial locality. This improves the performance of applications that operate on data with high spatial locality by improving the likelihood that two objects that have high locality on the reference genome will be on the same machine. This reduces the amount of data that must be fetched across the network when performing distributed computing.

3.3 Performance

Table 1 previews the performance of ADAM for *Sort* and *Mark Duplicates*.

Table 1. *Sort* and *Mark Duplicates* Performance on 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

Table 2. Comparison of File Formats

	BAM	ADAM
Size	1.0×	0.75-0.9×
Scalability	<8 machines	>100 machines
	VCF	**ADAM**
Scalability	<8 machines	>100 machines

4 DISCUSSION

4.1 Single Node Processing vs. Dedicated Clusters vs. Cloud Computing

A central goal during the development of ADAM was to not force users to follow specific patterns. As such, we have also made efforts to design the ADAM format and command line utilities to be portable across different computing installations. We envision that end users will be using ADAM on single workstations, dedicated computing clusters, or through a cloud computing provider like Amazon EC2 or Microsoft Azure. As can be seen in §3.3, ADAM performs admirably on all platforms, achieving a 2× speedup over the current state-of-the-art on a single node, and near-linear speedup on when distributed.

As traditional genomics workflows could not easily or efficiently use compute clusters, many users have performed the bulk of their processing on beefy workstations. We predict that the rise of distributed computing tools for genomics will soon render workstations unattractive. Instead, we expect that clinical/research centers will either process consistently high volumes of genomic data and build and maintain their own dedicated compute farms, or will use a cloud computing platform. The benefits of cloud computing:

- Commercial cloud platforms are economically attractive as they do not present any capital acquisition costs, nor do they have maintenance costs.
- Additionally, cloud platforms tend to offer several levels of service differentiation. Users can trade cost for performance, as the number of machines and performance of machines can be selected. Additionally, cloud platforms also frequently auction unused slots off at below-market prices[7], which can further reduce costs.
- System setup can be simplified through the use of systems like *Docker* (Docker, 2013) for distributing consistent application images.

[7] For Amazon EC2, these are known as *spot instances*.

- Inexpensive pay-as-you-go storage solutions are available which provide good performance when used with a cloud service.

We anticipate that formats like ADAM will ease this transition, as ADAM can be used efficiently on all platforms. Additionally, ADAM's stack model explicitly allows for significant flexibility in the format. This will allow ADAM to easily adapt to any future computing revolutions.

4.2 Evolution of Standard

ADAM's read, pileup, variant, and genotype formats are designed to evolve easily. The formats are simple to evolve because they are expressed as an explicit schema in Avro. Avro allows for an application that has been compiled against a specific version of the schema to read data that has been written in an older or newer version of the schema, as long as a simple process is followed when updating the schema.

Additionally, we have designed the formats to allow for user defined fields to be added without changing the core schema. Read data uses "attributes," in a fashion similar to the SAM/BAM format. For variant and genotype data, our variant context implementation (see §3.1) supports several different and expressive ways for defining new fields. The variant context method is superior to the annotations provided by the VCF standard, as it provides higher expressivity while promoting better organization.

ADAM is a community effort, and all members of the greater bioinformatics community are invited to modify both the data formats and the processing pipelines. The ADAM team has established a Request For Comment (RFC) system that is similar to the one used by the Internet Engineering Task Force (IETF). Here, modifications are proposed and opened for public discussion for a set time frame. After a consensus is reached, the format can be amended. The RFC platform is hosted at our project website[8].

5 CONCLUSION

This paper presents ADAM, a new data storage format and processing pipeline for genomics data. Our reference implementation 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 genomics pipelines. 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 transformations 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 at scale.

[8] http://adam.cs.berkeley.edu/rfc

ACKNOWLEDGEMENT

The authors would like to thank their many colleagues who provided feedback on early implementations of ADAM, and on drafts of the manuscript. Additionally, we would like to thank Uri Laserson, David Haussler, Adam Novak, Mauricio Carnerio, and Joel Thibault for their feedback on technical direction and format implementation.

Funding: This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, a NSF Graduate Research Fellowship (DGE-1106400) and gifts from Amazon Web Services, Google, SAP, Apple, Inc., Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, GameOnTalis, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

REFERENCES

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

Apache (2012a). Avro. http://avro.apache.org.

Apache (2012b). Hadoop. http://hadoop.apache.org.

Apache (2012c). Thrift. http://thrift.apache.org.

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

Docker (2013). Docker. https://www.docker.io/.

Google (2012). Protocol buffers. https://developers.google.com/protocol-buffers/docs/overview?csw=1.

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

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

Massie, M., Nothaft, F. A., Hartl, C., Kozanitis, C., Schumacher, A., Joseph, A. D., and Patterson, D. A. (2013). ADAM: Genomics formats and processing patterns for cloud scale computing. Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley.

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

McPherson, J. D. (2009). Next-generation gap. *Nature Methods*, **6**, S2–S5.

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

Picard (2013). Picard. http://picard.sourceforge.org.

Twitter and Cloudera (2013). Parquet. http://www.parquet.io.

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

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

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