# Rethinking Scientific Analysis Using Modern Scalable Systems


## ABSTRACT

Revolutions in data acquisition are drastically changing how science conducts experiments. For example, "next-generation" sequencing technologies have driven exponential growth in the volume of genomic data, and similar trends impact many fields which rely on imaging, such as astronomy and neuroscience. Although there have been early attempts to use MapReduce systems to accelerate the processing of these datasets, they have conceded efficiency in favor of using legacy storage formats and software.

Since the amount of scientific data that is being captured is increasing exponentially, we have a good opportunity to rethink how we process and store these datasets. In this paper, we introduce a set of principles for implementing scientific analyses efficiently using commodity "big data" systems. We motivate these principles with an example genomics pipeline which leverages open-source MapReduce and columnar storage techniques to achieve a $> 50\times$ speedup over traditional genomics systems, at half the cost.



## Categories and Subject Descriptors

L.4.1 [**Applied Computing**]: Life and medical sciences—*Computational biology*; H.1.3.2 [**Information Systems**]: Data management systems—*Database management system engines, parallel and distributed DBMSs*; E.3.2 [**Software and its Engineering**]: Software creation and management—*Software Development Process Management*

## General Terms

Design

## Keywords

Analytics, MapReduce, Genomics, Scientific Computing


## 1. INTRODUCTION

With major improvements in scientific data acquisition techniques, data storage and processing have become major problems for scientists [30, 8]. In fields like neuroscience [15] and genomics [34], scientists routinely perform experiments that use terabytes (TB) to petabytes (PB) of data. While traditional scientific computing platforms are optimized for fast linear algebra, many emerging domains make heavy use of statistical learning techniques coupled with user defined operations on top of semistructured data. This move towards statistical techniques has been driven by the increase in the amount of data available to scientists, as well as the rise of statistical systems which are accessible to non-experts, such as `Scikit-learn` [29] and `MLI` [33].

While the increase in the amount of scientific data available is a boon for scientists, it puts significant stress on existing tool chains. Using the current "best practice" genomics software [5], it takes approximately 120 hours to process a single, high-quality human genome using a single, beefy node [35]. To address these challenges, scientists have started to apply computer systems techniques such as MapReduce [26, 31, 21] and columnar storage [16] to custom scientific compute/storage systems. While these systems have improved the analysis cost and performance, they incur significant overheads due to constraints of the legacy formats and codebases that they use.

Since the amount of scientific data being generated is growing so quickly, we cannot afford to be saddled by legacy software and formats. New scientific projects such as the "100K for UK," which aims to sequence the genomes of 100,000 individuals in the United Kingdom (UK, [14]) will generate three to four orders of magnitude more data than prior "massive" projects such as the 1000 Genomes Project [32]. While it is important to still be able to use data stored in legacy formats, the massive amount of *new* data provides us with an opportunity to rethink how we compose our systems. By choosing the correct mix of computing systems, we can provide better performance and scalability than custom systems, while enhancing the abstractions exposed to scientists.

In this paper, we demonstrate a system built using Apache Avro, Parquet, and Spark [2, 3, 41], which achieves a $50\times$ increase in throughput over the current best practice pipeline for processing genomic data. In the process of creating this system, we developed a "narrow waisted" layering model for building similar scientific analysis systems. This narrow waisted model is inspired by the OSI model for networked systems [42]. We then demonstrate the generality of this

model by using it to implement a system for processing astronomy images.

A subtle problem with earlier custom scientific processing and storage systems is that characteristics of the data format on disk would bleed into the computing model. For example, in the current Sequence/Binary Alignment and Map (SAM/BAM [23]) formats for storing genomic alignments, constraints about record ordering are required in order to enable specific computing patterns. We believe that this is an abstraction inversion, which makes it difficult to perform some other access patterns. In §3, we elucidate why this is a significant problem, and then in §5, we then introduce efficient algorithms for supporting these computational patterns without forcing constraints on the storage layer.

# 2. BACKGROUND

As our work exists at the intersection of computational science and data management and processing systems, our architectural approach is informed by recent trends in both areas. The design of large scale data management has changed dramatically since the landmark papers by Dean and Ghemawat [9, 10] which described Google's `MapReduce` system. Over a similar timeframe, scientific fields have moved to take advantage of improvements in data acquisition technologies. For example, since the Human Genome Project finished in 2001 [20], the price of genomic sequencing has dropped by 10,000× [28]. This drop in cost has enabled the capture of petabytes of sequence data, which has enabled significant population-scale genomics experiments like the 1000 Genomes project [32], and The Cancer Genome Atlas (TCGA, [37]). These changes are not unique to genomics; indeed, fields such as neuroscience [8] and astronomy [36] are experiencing similar changes.

There are significant opportunities to apply novel data management systems techniques to scientific problems. In the rest of this section, we'll look at the current state of the art and opportunities in both fields.

## 2.1 Genomic Data and Analyses

Contemporary genomics has been revolutionized by "next generation" sequencing technologies (NGS), which have driven a precipitous drop in the cost of running genomic assays [28]. Although there are a variety of sequencing technologies in use, the majority of sequence data comes from the Illumina/Solexa sequencing platform, which uses a "sequencing-by-synthesis" technique to generate *short read* data [27], where a sequencing run will generate many reads that are between 50 and 250 bases in length. In addition to adjusting the length of the reads, we can control the amount of the data that is generated by changing the amount of the genome that we sequence, or the amount of redundant sequencing that we perform (the average number of reads that covers each base, or *coverage*). A single human genome sequenced at 60× coverage will produce approximately 1.4 billion reads, which is approximately 600 GB of raw data, or 225 GB of compressed data. For each read, we also are provided *quality scores*, which represent the likelihood that the base at a given position was observed.

One of the most common genomic analyses is *variant calling*, which is a statistical process to infer the sites at which a single individual varies from the *reference genome*.[1] This process consists of the following general steps:

1. **Alignment:** For each read, we find the position in the genome that the read is most likely to have come from. As an exact search is too expensive, there has been an extensive amount of research which has focused on indexing strategies for improving alignment performance [22, 24, 39]. This process is parallel per sequenced read.
2. **Pre-processing:** After reads have been aligned to the genome, we perform several preprocessing steps to eliminate systemic errors in the reads. This may involve recalibrating the observed quality scores for the bases, or locally optimizing the read alignments. We will present a description of several of these algorithms in §4.1; for a more detailed discussion, we refer readers to DePristo et al [11] and Massie et al [25].
3. **Variant calling:** Variant calling is a statistical process which uses the read alignments and the observed quality scores to compute whether a given sample matches or diverges from the reference genome. This process is typically parallel per position or region in the genome.
4. **Filtration:** After variants have been called, we want to filter out false positive variant calls. We may perform queries to look for variants with borderline likelihoods, or we may look for clusters of variants, which may indicate that a local error has occurred. This process may be parallel per position, or may involve complex traversals of the genomic coordinate space.

This process is very expensive to run; the current best practice pipeline uses the BWA tool [22] for alignment and the GATK [26, 11] for pre-processing, variant calling, and filtration. Current benchmark suites have measured this pipeline as taking between 90 and 130 hours to run end-to-end [35]. In §4.1, we will demonstrate a distributed pipeline which achieves a $> 50\times$ speedup for the pre-processing component of this pipeline.

## 2.2 Scientific Data Processing

Although there has been significant progress in the development of systems for processing large datasets (e.g., the development of first generation MapReduce systems [9], followed by iterative MapReduce systems like Spark [41], as well as parallel and columnar DBMS [1, 19]), the uptake of these systems in the scientific world has been slow. Most implementations have either used MapReduce as an inspiration for programming API design [26], or have been limited systems which have used MapReduce to naïvely parallelize existing toolkits [21, 31]. These approaches are perilous for several reasons:

- A strong criticism levied against the map-reduce model is that the API is insufficiently expressive for describing complex tasks. As a consequence of this, tools like

[1]The reference genome represents the "average" genome for a species. The Human Genome Project [20] assembled the first human reference genome.

the GATK [26] which adopt MapReduce as a programming model (but *not an execution strategy!!!*) force significant restrictions on algorithm implementors. For example, a GATK `walker` is provided with a single view over the data (a sorted iterator over a specified region), and is allowed limited reduce functionality.

- A major contribution of systems like MapReduce [10] and Spark [41, 40] is the ability to reliably distribute parallel tasks across a cluster in an automated fashion. In practice, to run tools like the GATK across a cluster, organizations have rolled their own systems for sharding and persisting intermediate data, and managing failures and retries. This is not only an inefficient duplication of work, but it is also a source of inefficiency during execution: the performance of iterative stages in the GATK is bottlenecked by I/O performance. Additionally, the sharding techniques used limit scale-up to a 10× speedup on 23 machines.
- The naïve Hadoop-based implementations in Crossbow [21] and Cloudburst [31] lead to good speedups, but add significant overhead. Several of the methods that they parallelize incur high overhead due to duplicated loading of indices (for fast aligners, loading of large indices can be a primary I/O bottleneck) and poor broadcasting of data.

A notable exception is the `Thunder` system, which was developed for processing neuroscience imaging data [15]. `Thunder` performs a largely statistical workload, where clustering and regression are significant computational tasks. The system is constructed using Spark and Python and is designed to process datasets larger than 4 TB, and leverages significant functionality from the MLI/MLLib libraries [33].

Recent work by Diao et al [12] has looked at optimizations to MapReduce systems for processing genomic data.

One interesting trend of note is the development of databases specifically for scientific applications. The exemplar is SciDB, which provides an array based storage model as well as efficient linear algebra routines [7]. While arrays accelerate many linear algebra based routines, they are not a universally great fit. For many genomics workloads, data is semistructured and may consist of strings, boolean fields, and an array of tagged annotations. Other systems like the Genome Query Language [18] have extended SQL to provide efficient query semantics across genomic coordinates. While GQL achieves performance improvements of up to 10× for certain algorithms, SQL is not an attractive language for many scientific domains, which make heavy use of user designed functions (UDFs), which may be difficult to implement through SQL.

One notable area where database techniques have been leveraged by scientists is in the data storage layer. Due to the storage costs of large genomic datasets, scientists have introduced the CRAM format which uses columnar storage techniques and special compression algorithms to achieve a 30% reduction in size over the original BAM format [16]. While CRAM achieves good compression, it imposes restriction on the ordering and structure of the data, and does not provide efficient support for predicates or projection.

# 3. PRINCIPLES FOR SCIENTIFIC ANALYSIS SYSTEMS

## 3.1 Workloads

Traditionally, work in the scientific computing domain has focused on linear algebra, particle simulations, and optimization-style problems. The communication and

1. Characteristics of data
   (a) Scientific data tends to be sparse
   (b) Different users want to look at different subsets of both rows and columns
   (c) Data may not always be in a single site, or stored locally
   (d) *Experimental data* is immutable.
   (e) What are access patterns?
2. Characteristics of a ideal storage system:
   (a) Efficient support for projection of different columns
   (b) Efficient support for per-record predicates
   (c) Should not relegate user to a single execution environment
3. Processing:
   (a) Workloads are highly variable by field
   (b) For genomics, workloads are trivially data-parallel
   (c) Similar for fields with heavy image processing workloads
   (d) Simulation based fields are tougher; have all-to-all computation pattern, run on supercomputer
   (e) Defer discussion to §**??**
   (f) Ideally, cross-platform.

## 3.2 Layering

We can improve the efficiency of building scientific data storage and processing systems by choosing the correct abstractions.

The BAM and VCF 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 [42]. 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:

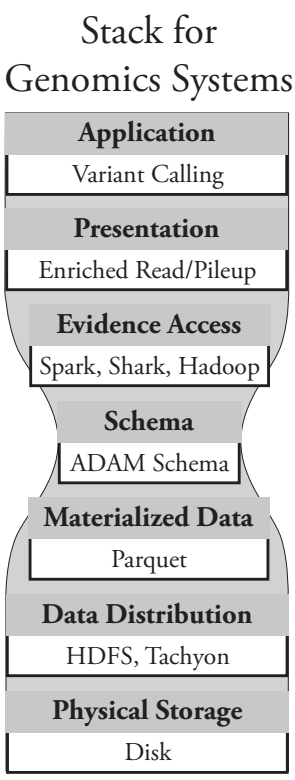


**Figure 1: A Stack Model for Genomics**

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 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 [2], an open-source, cross-platform data serialization framework similar to Apache Thrift and Google's Protocol Buffers [4, 17]. Avro was chosen as the interchange format for several reasons:

- Avro is an open-source framework covered by the Apache 2 license, which 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.

For a system to implement the ADAM format, it must read/write ADAM objects that are defined by Avro schemas. As such the schemas provide the common ground for tools to interoperate, thereby allowing the user to interchange one tool for another and minimize tool or 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, and this view of the data is independent of whether the data is local or distributed across a cluster or cloud. By separating the API from the data access layer, we improve flexibility. With careful design in the data format and data access layers, we can seamlessly support conventional 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.

Talk about data storage bleeding into programming model.

We need to:

1. Show how current systems fit into the stack model, and how our proposed stack is different
2. Elucidate why it is more efficient to build systems that are decomposed as per our stack above (reference networking stack and protocol interchange), see Bafna et al [6], talk about costs of programming without good stack model

# 4. CASE STUDIES

To validate our architectural choices, we have implemented pipelines for processing short read genomic data and astronomy image processing. Both of these pipelines are implemented using Spark [41], Avro [2], and Parquet [3]. We have chosen these two applications as they fit in different areas in the design space.

## 4.1 Genomics Pipeline

*Sorting.* This phase sorts all reads by the position of the start of their alignment. The implementation of this algorithm is trivial, as Spark provides a sort primitive [41]; we solely need to define an ordering for genomic coordinates, which is well defined.

*Duplicate Removal.* During the process of preparing DNA for sequencing, errors in the sample preparation and polymerase chain reaction (PCR) stages can cause the duplication of reads. Detection of duplicate reads requires matching all reads by their position and orientation after read alignment. Reads with identical position and orientation are assumed to be duplicates. When a group of duplicate reads is found, each read is scored, and all but the top-scoring read are marked as 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, we build a statistical model for read errors

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

For current implementations of these read processing steps, performance is limited by disk bandwidth. This bottleneck occurs because the operations read in a SAM/BAM file, perform a bit of processing, and write the data to disk as a new SAM/BAM file. We address this problem by performing our processing iteratively in memory. The four read processing stages can then be chained together, eliminating three long writes to disk and an additional three long reads from disk. The stages themselves cannot be performed in parallel, but the operations inside each stage are data parallel.

## 4.2 Astronomy Image Processing

# 5. DATA ACCESS OPTIMIZATIONS FOR SCIENTIFIC PROCESSING

## 5.1 Coordinate System Joins

This will be a compare/contrast discussion of the multiple join algorithms we've created. TBD.

## 5.2 Loading Remote Data

1. Data may not be kept locally
   (a) Too much data to keep locally
   (b) Not all data is hot
2. May push data off local disks into block store
3. Manually re-staging data has high latency cost → impacts throughput
4. What do we need to do to accommodate this?
   (a) Efficient indexing
   (b) Remote push-down predicate
5. Discuss S3/Parquet interaction

# 6. PERFORMANCE

## 6.1 Genomics Workloads

Table 1 previews the performance of ADAM for *Sort*, *Mark Duplicates*, and *Flagstat*. The tests in this table are run on the high coverage *NA12878* full genome BAM file that is available from the 1000 Genomes project; the HG00096 low coverage BAM from 1000 Genomes is used later in this section[2]. These tests have been run on the EC2 cloud, using the instance types listed. We compute the cost of running each experiment by multiplying the number of instances used by the total wall time for the run and by the cost of running a single instance of that type for an hour, which is the process Amazon uses to charge customers.

**Table 1: *Sort*, *Mark Duplicates*, and *Flagstat* Performance on NA12878**

| | | ***Sort*** | | |
|---|---|---|---|---|
| **Software** | **EC2 profile** | **Wall Time** | **Speedup** | **Cost** |
| Picard 1.103 | 1 `hs1.8xlarge` | 17h 44m | 1× | $81.57 |
| ADAM 0.5.0 | 1 `hs1.8xlarge` | 8h 56m | 2× | $41.09 |
| ADAM 0.5.0 | 32 `cr1.8xlarge` | 33m | 32× | $61.60 |
| ADAM 0.5.0 | 100 `m2.4xlarge` | 21m | 52× | $56.00 |
| | | ***Mark Duplicates*** | | |
| **Software** | **EC2 profile** | **Wall Time** | **Speedup** | **Cost** |
| Picard 1.103 | 1 `hs1.8xlarge` | 20h 22m | 1× | $93.68 |
| ADAM 0.5.0 | 100 `m2.4xlarge` | 29m | 42× | $79.26 |
| | | ***Flagstat*** | | |
| **Software** | **EC2 profile** | **Wall Time** | **Speedup** | **Cost** |
| SAMtools 0.1.19 | 1 `hs1.8xlarge` | 25m 24s | 1× | $1.95 |
| ADAM 0.5.0 | 32 `cr1.8xlarge` | 0m 46s | 33× | $1.43 |

Table 2 describes the instance types. Memory capacity is reported in Gibibytes (GiB), where 1 GiB is equal to $2^{30}$ bytes. Storage capacities are not reported in this table because disk capacity does not impact performance, but the number and type of storage drives is reported because aggregate disk bandwidth does impact performance. In our tests, the `hs1.8xlarge` instance is chosen to represent a workstation. Network bandwidth is constant across all instances.

**Table 2: AWS Machine Types**

| Machine | Cost | Description |
|---|---|---|
| `hs1.8xlarge` | $4.60/hr/machine | 16 cores, 117GiB RAM, 24× HDD |
| `cr1.8xlarge` | $3.50/hr/machine | 32 cores, 244GiB RAM, 2× SDD |
| `m2.4xlarge` | $1.64/hr/machine | 8 cores, 68.4GiB RAM, 2× HDD |

[2]The files used for these experiments can be found on the 1000 Genomes ftp site, `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/NA12878/high_coverage_alignment/` for NA12878, and in directory `/vol1/ftp/data/HG00096/alignment/` for HG00096.

As can be seen from these results, the ADAM pipeline is approximately twice as fast as current pipelines when running on a single node. Additionally, ADAM achieves speedup that is close to linear. This point is not clear from Table 1, as we change instance types when also changing the number of instances used. To clarify, Figure 2 presents speedup plots for the NA12878 high coverage genome and the HG00096 low coverage genome (16 GB BAM). These speedup measurements are taken on a dedicated cluster of 82 machines where each machine has 24 Xeon cores, 128GB of RAM, and 12 disks.

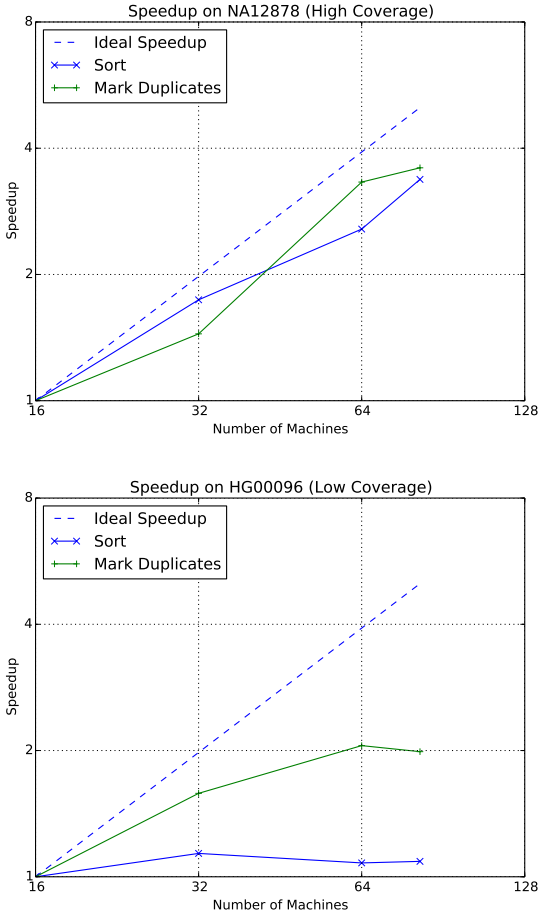


**Figure 2: Speedup when running *Sort* and *Mark Duplicates* on NA12878 and HG00096**

NA12878 sees linear speedup for both *Sort* and *Mark Duplicates* through 82 nodes. With 82 nodes, it takes 8.8 minutes to sort reads, and 14 minutes to mark duplicate reads across the 250GB file. HG00096 is a significantly smaller genome at 16 GB. Although the speedup from sorting diminishes after 16 nodes, duplicate marking sees speedup through 64 nodes. With HG00096 on 64 nodes, each machine in the cluster is responsible for processing only 250MB of data. Sorting completes in 3.3 minutes, and duplicate marking completes in 2.3 minutes. The speedup is limited by several factors:

- Although columnar stores have very high read performance, they are limited by write performance. Our tests exaggerate this penalty—as a variant calling pipeline will consume a large read file, but then output a variant call file that is approximately two orders of magnitude smaller, the write penalty will be reduced.
- Additionally, for large clusters, straggler elimination is an issue. Phases of both *Sort* and *MarkDuplicates* currently suffer from stragglers—we are in the process of addressing these issues.

However, as noted above, speedup continues until we reach approximately 1GB of data per node for sorting, or 250MB of data per node for duplicate marking. For a high coverage whole genome like NA12878, this should theoretically allow speedup through 250 nodes for sorting and 1,000 nodes for duplicate marking.

## 6.2 Astronomy Workloads

## 6.3 Column Store Performance

Compare to CRAM.

# 7. DISCUSSION

## 7.1 Evolution of Computing Infrastructure

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 for infrequent data processing needs. The benefits of cloud computing include:

- Commercial cloud platforms are economically attractive as they do not present 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 (*spot instances*, which can further reduce costs.
- System setup can be simplified through the use of systems like `Docker` [13] for distributing consistent application images.
- Inexpensive pay-as-you-go storage solutions are available which provide good performance when used with a cloud service.

We anticipate that our design decisions will make the transition to cloud services for scientific data systems more attractive.

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, which will allow ADAM to easily adapt to future computing innovations.

## 7.2 Syntax vs. Semantics

When selecting a storage format, it is important to address both the problems of syntax and semantics. A storage backend that is *syntactically good* will be easily adaptable to multiple platforms, provide good programming abstractions and APIs, and will be performant. A format that is *semantically good* will provide data representations that closely mirror the scientific and algorithmic processes they represent.

The core of the ADAM project is focused on improving the syntax used for representing biological data. Our focus on semantics is limited to introducing semantic improvements for variant and genotype data that allows for better representation of joint called data and annotations §**??**. However, while we do not address the greater issues of semantics, we provide a clean platform for tacking these issues because of the ADAM stack and its ability to support broader access and computational patterns. For example, ADAM data can easily be accessed using databases like Spark-SQL and Impala. Additionally, ADAM can be used as the base of a graph-based data representation through GraphX [38]. This design decision allows for the easy and clean extension of ADAM's semantics.

## 7.3 Advantages of Commodity Systems

In §3.2, we advocate for a cleanly decomposed system. In our case studies in §4, we then used the open source Avro, Parquet, and Spark systems to implement our stack [2, 3, 41]. An obvious advantage of using commodity systems is code reuse, but the benefits do not start and end there. By using components that are well accepted in the Hadoop ecosystem, we gain several major features for free:

1. Beyond efficient parallel performance, Parquet is also supported as a data format by several database-like systems, like Spark-SQL and Impala. This allows us to support efficient database style processing, without needing to manually retrofit a tool like GQL to the data [18].

2. While the native Spark Scala API provides rich programming abstractions, scientists may prefer other language environments, like Python or R. Other scientific systems like SciDB [7] support native Python and R bindings. Likewise, we provide distributed processing using Python through PySpark.

# 8. CONCLUSION

In the end, we conclude.

# APPENDIX

# A. REFERENCES


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

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

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

[4] Apache. Thrift. `http://thrift.apache.org`, 2012.

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

[6] V. Bafna, A. Deutsch, A. Heiberg, C. Kozanitis, L. Ohno-Machado, and G. Varghese. Abstractions for genomics. *Communications of the ACM*, 56(1):83–93, 2013.

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

[8] J. P. Cunningham. Analyzing neural data at huge scale. *Nature methods*, 11(9):911–912, 2014.

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

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

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

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

[13] Docker. Docker. `https://www.docker.io/`, 2013.

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

[15] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature methods*, 11(9):941–950, 2014.

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

[17] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/docs/overview?csw=1`, 2012.

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

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

[20] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar,

M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[21] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134, 2009.

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

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

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

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

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

[27] M. L. Metzker. Sequencing technologiesÑthe next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

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

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

[30] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics*, 11(9):647–657, 2010.

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

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

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

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

[35] A. Talwalkar, J. Liptrap, J. Newcomb, C. Hartl, J. Terhorst, K. Curtis, M. Bresler, Y. S. Song, M. I. Jordan, and D. Patterson. SMASH: A benchmarking toolkit for human genome variant calling. *Bioinformatics*, page btu345, 2014.

[36] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A multi-code analysis toolkit for astrophysical simulation data. *The Astrophysical Journal Supplement Series*, 192(1):9, 2011.

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

[38] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

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

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association, 2012.

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

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