Rethinking Data-Intensive Science Using Scalable Analytics Systems

Frank Austin Nothaft*, Matt Massie*, Timothy Danford*,‡, Zhao Zhang*, Uri Laserson°,
Carl Yeksigian‡, Jey Kottalam*, Arun Ahuja†, Jeff Hammerbacher†,°,
Michael Linderman†, Michael J. Franklin*, Anthony D. Joseph*, David A. Patterson*
*AMPLab, University of California, Berkeley, °Cloudera, San Francisco, CA
†Carl Icahn School of Medicine, Mount Sinai, New York, NY, ‡Genomebridge, Cambridge, MA
{fnothaft, massie, tdanford, zhaozhang, jey, franklin, adj, pattrsn}@berkeley.edu
{arun.ahuja, jeff.hammerbacher, michael.linderman}@mssm.edu
laserson@cloudera.com
cyeksig@genomebridge.org

ABSTRACT

"Next generation" data acquisition technologies are allowing scientists to collect exponentially more data at a lower cost. These trends are broadly impacting many scientific fields, including genomics, astronomy, and neuroscience. We can attack the problem caused by exponential data growth by applying horizontally scalable techniques from current analytics systems to accelerate scientific processing pipelines.

In this paper, we describe `ADAM`, an example genomics pipeline that leverages the open-source Apache `Spark` and `Parquet` systems to achieve a 28× speedup over current genomics pipelines, while reducing cost by 63%. From building this system, we were able to distill a set of techniques for implementing scientific analyses efficiently using commodity "big data" systems. To demonstrate the generality of our architecture, we then implement a scalable astronomy image processing system which achieves a 2.8–8.9× improvement over the state-of-the-art MPI-based system.

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

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2742787.

Keywords

Analytics, MapReduce, Genomics, Scientific Computing

1. INTRODUCTION

Major improvements in scientific data acquisition techniques are driving increased scientific data storage and processing needs [11, 43]. In fields like neuroscience [18] and genomics [48], particle physics, and astronomy, scientists routinely perform analyses that process terabytes (TB) to petabytes (PB) of data. While traditional scientific computing and storage platforms are optimized for fast linear algebra, many emerging domains make heavy use of statistical learning techniques and user defined functions (UDFs) on top of semistructured data. This move towards statistical techniques has been driven by the increase in the amount of data available to scientists. At the same time, commercial needs have led to the development of horizontally-scalable analytics systems like `MapReduce` [12, 13] and `Spark` [59], as well as statistical systems that are accessible to non-experts, such as `Scikit-learn` [40] and `MLI` [47].

Since the amount of scientific data being generated is growing so quickly, there is a good opportunity to apply modern, horizontally scalable analytics systems to science. New scientific projects such as the 100K for UK, which aims to sequence the genomes of 100,000 individuals in the United Kingdom [20] will generate three to four orders of magnitude more data than prior projects like the 1000 Genomes Project [46]. These projects use the current "best practice" genomic variant calling pipeline [6], which takes approximately 120 hours to process a single, high-quality human genome using a single, beefy node [49]. To address these challenges, scientists have started to implement computer systems techniques such as map-reduce [34] and columnar storage [19] in custom scientific compute/storage systems. While these systems have improved analysis cost and performance, current implementations incur significant overheads imposed by the legacy formats and codebases that they use.

In this paper, we demonstrate `ADAM` [33], a genomic data processing system built using Apache `Avro`, `Parquet`, and `Spark` [3, 4, 59], that achieves a 28× increase in read preprocessing throughput over the current best practice pipeline, while reducing analysis cost by 63%. In the process of creating this system, we developed a "narrow waisted" layering model for building similar scientific analysis systems. This narrow waisted stack is inspired by the OSI model for networked systems [61]. Our model uses the data schema as the narrow waist that separates data processing from data storage. ADAM additionally demonstrates that it is possible to achieve very high performance on scientific applications using free open source software (OSS). By shifting scientific data processing away from proprietary high performance computing (HPC) environment and into a commodity/OSS world, we can democratize scientific analysis.

To researchers and practitioners unfamiliar with current practice in computational science, the layering-based approach we propose may appear sufficiently straightforward as to be uncontroversial. In practice however—as discussed in the seminal "end-to-end" paper [42]—there is a common tendency to engage in "stack smashing" when building a large system to improve performance or to simplify implementation. Scientific applications are no different; in genomics, processing pipelines impose invariants on the layout of data on disk to improve performance, such as requiring data to be stored in a coordinate-sorted order. These invariants are rarely made explicit, which can lead to functional errors when composing multiple tools into a pipeline and necessitate slow sorting stages. In both genomics and astronomy, data is stored in flat file that centralize file/experiment metadata in order to improve the size of data on disk. This increases the cost of parallel metadata access and limits the scalability of current parallel astronomy pipelines (see §6.2). In this paper, we demonstrate that scientific pipelines *can* be decomposed without sacrificing computational cost through the use of the following techniques:

1. We make the schema the "narrow waist" of our stack and enforce data independence. We then devise algorithms for making common scientific processing patterns fast (e.g., coordinate-space joins, see §5.1).

2. To improve horizontal scalability, we push computation to the data. To support this, we use Parquet, a modern parallel columnar store based off of Dremel [35].

3. We use a denormalized schema to achieve O(1) parallel access to metadata.

We introduce the stack model in Figure 1 as a way to decompose scientific systems. In addition to the genomics application described above, we demonstrate the generality of this model by using it to implement a system for processing astronomy images and achieve a 2.8–8.9× performance improvement over a state-of-the-art Message Passing Interface (MPI) based pipeline.

While the stack smashing used in genomics to accelerate common access patterns is undesirable because it violates data independence and can lead to errors when composing multiple tools into a pipeline, we also find that it can lead to correctness errors inside of a single tool. As noted above, tools built using the current Sequence/Binary Alignment and Map (SAM/BAM [31]) formats for storing genomic alignments apply constraints about record ordering to enable specific computing patterns. Our implementation (described in §4.1) identifies errors in two current genomics processing stages that occur *because* of the sorted data layout invariant. Our implementations of these stages do not make use of sort order, and achieve high performance while eliminating these errors.

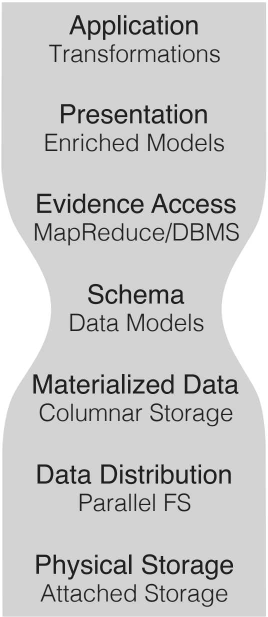

Figure 1: A stack model for scientific computing

We have made all of the software (source code and executables) described in this paper available free of charge under the permissive Apache 2 open-source license. Additionally, the setup for implementing the genomics experiments described in this paper has been made publicly available to enable reproducing our results. Access instructions are given in Appendices D and E.

2. BACKGROUND

Our work is at the intersection of computational science, 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 papers by Dean and Ghemawat [12, 13] that described Google's MapReduce system. Over a similar timeframe, scientific fields have taken advantage of improvements in data acquisition technologies. Since the Human Genome Project finished in 2001 [28], the price of genomic sequencing has dropped by 10,000× [38]. This drop in cost has enabled the capture of petabytes of sequence data and enabled significant population genomics experiments like the 1000 Genomes project [46] and The Cancer Genome Atlas (TCGA, [53]). These changes are not unique to genomics; indeed, fields such as neuroscience [11] and astronomy [23, 52, 56] are experiencing similar trends.

Although there has been significant progress in the development of systems for processing large datasets—the development of first generation map-reduce systems [12], followed by iterative map-reduce systems like Spark [59], as well as parallel and columnar DBMS [1, 27]—the uptake of these systems in the scientific world has been slow. Most implementations have either used map-reduce as an inspiration for API design [34], or have been systems that have used map-reduce to parallelize existing toolkits [29, 44]. These approaches are perilous for several reasons:

- A strong criticism of the map-reduce model is that the API is insufficiently expressive for describing complex tasks. As a consequence of this, tools like the GATK [34] that adopt map-reduce as a programming model force significant restrictions on algorithm implementors. For example, a GATK walker is provided with a single view over the data (a sorted iterator) with limited reduce functionality.

- A major contribution of systems like MapReduce [13] and Spark [59, 58] is the ability to reliably distribute parallel tasks across a cluster in an automated fashion. While the GATK uses map-reduce as a programming abstraction, it does not use map-reduce as an execution strategy. To run tools like the GATK across a cluster, organizations use workflow management systems for sharding and persisting intermediate data, and managing failures and retries. This *ad hoc* approach is a source of inefficiency during execution: since the execution system cannot pipeline jobs together, iterative stages in the GATK must spill to disk and become bottlenecked by I/O performance.

- The Hadoop-based implementations in Crossbow [29] and Cloudburst [44] run unmodified legacy tools on top of Hadoop. This approach does achieve speedups, but does not attack overhead. High overhead occurs due to duplicated loading of indices and poor broadcast performance.

Recent work by Diao et al [15] has looked at optimizing map-reduce systems for processing genomic data. They adapt strategies from the query optimization literature to reorder computation to minimize data shuffling. While this approach does improve shuffle traffic, several preprocessing stages cannot be transposed. For instance, reversing the order of realignment and base quality score recalibration (see §4.1) will change the inferred quality score distribution. Additionally, we believe that the shuffle traffic that Diao et al observe is an artifact caused by the abstraction inversion discussed in §1. As we demonstrate in §4.1, these penalties can be eliminated by restructuring the preprocessing algorithms.

One notable area where modern data management 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 that uses columnar storage techniques and special compression algorithms to achieve a $> 30\%$ reduction in size over the compressed BAM format [19]. While CRAM achieves high ($\gg 50\%$) compression, it imposes restrictions on the ordering and structure of the data, and does not provide support for predicates or projection. Additionally, to achieve compression ratios greater than 30%, CRAM uses lossy compression codecs. We perform a more comprehensive comparison against CRAM in §6.3.

One interesting trend of note is the development of databases specifically for scientific applications. The exemplar is SciDB, which provides an array-driven storage model with efficient linear algebra routines [8]. While arrays accelerate many linear algebra based routines, they are not a universally great fit. Due to the semistructured nature of genomics datasets, complex UDFs are needed to import genomic data into array databases. Other systems like the Genome Query Language (GQL, [26]) have extended SQL to provide efficient query semantics across genomic coordinates. While GQL achieves 10× performance improvements for certain algorithms, SQL is not an attractive language for scientific analyses that make heavy use of UDFs.

3. CHARACTERISTICS OF SCIENTIFIC ANALYSIS SYSTEMS

Most prior work on scientific computing has been focused on linear algebra and other problems that can be structured as a matrix or network. However, in several of the emerging data-driven scientific disciplines, data is less rigorously structured. As discussed in §2, scientists have developed custom solutions to process this data. In this section, we discuss the common characteristics of workloads in these emerging scientific areas. Given these characteristics, we describe a way to decompose data processing and storage systems so that we can efficiently implement important processing patterns while providing a wide range of data access methods.

3.1 Layering

The processing patterns being applied to scientific data shift widely as the data itself ages. Because of this change, we want to design a scientific data processing system that is flexible enough to accommodate our different use cases. At the same time, we want to ensure that the components in the system are well isolated so that we do not bleed functionality across the stack. If we did bleed functionality across layers in the stack, this violation of the end-to-end principle would make it more difficult to implement different applications using our stack [42]. Additionally, as we discuss in §4.1, improper separation of concerns can lead to errors in our application.

These concerns are very similar to the factors that led to the development of the Open Systems Interconnection (OSI) model and Internet Protocol (IP) stack for networking services [61]. The networking stack models were designed to allow the mixing and matching of different protocols, all of which existed at different functional levels. The success of the networking stack model can largely be attributed to the "narrow waist" of the stack, which simplified the integration of a new protocol or technology by ensuring that the protocol only needed to implement a single interface to be compatible with the rest of the stack.

Unlike conventional scientific systems that leverage custom data formats like BAM, SAM, or CRAM [19, 31], we believe that the use of an explicit schema for data interchange is critical. In our stack model shown in Figure 1, the schema becomes the "narrow waist" of the stack. Most importantly, placing the schema as the narrow waist enforces a strict separation between data storage/access and data processing. The seven layers of our stack model are decomposed as follows, and are numbered in ascending order from bottom to top:

1. **Physical Storage** coordinates data writes to physical media.

2. **Data Distribution** manages access, replication, and distribution of the files that have been written to storage media.

3. **Materialized Data** encodes the patterns for how data is encoded and stored. This layer determines I/O bandwidth and compression.

4. **Data Schema** specifies the representation of data, and forms the narrow waist of the stack that separates access from execution.

5. **Evidence Access** provides primitives for processing data, and enables the transformation of data into different views and traversals.

6. **Presentation** enhances the data schema with convenience methods for performing common tasks and accessing common derived fields from a single element.

7. **Application**s use the evidence access and presentation layers to compose algorithms for performing an analysis.

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. With careful design in the data format and data access layers, we can seamlessly support conventional whole file access patterns, while also allowing easy access to small slices of files. By treating the compute substrate and storage as separate layers, we also drastically increase the portability of the APIs that we implement.

As we discuss in more detail in §4.1, current scientific systems bleed functionality between stack layers. The `BAM`, `SAM` and `CRAM` formats are exemplars, as they expect data to be sorted by genomic coordinate. This modifies the layout of data on disk (level 3, Materialized Data) and constrains how applications traverse datasets (level 5, Evidence Access). Beyond constraining applications, this leads to bugs in applications that are difficult to detect.[1] To resolve this conflict, we demonstrate several ways to efficiently implement conventional scientific traversals without imposing a sort order invariant in §5. These traversals are implemented above the evidence access layer, and are independent of anything below the schema.

The idea of decomposing scientific applications into a stack model is not new; Bafna et al [7] made a similar suggestion in 2013. We borrow some vocabulary from Bafna et al, but our approach is differentiated in several critical ways:

- Bafna et al consider the stack model specifically in the context of data management systems for genomics; as a result, they bake current technologies and design patterns into the stack. Instead, a stack design should serve to abstract layers from methodologies/implementations, lest technology trends render the stack obsolete.

- Bafna et al define a binary data format as the narrow waist in their stack, instead of a schema. While these two seem interchangeable, they are not in practice. A schema provides a view of the logical content of data, while a binary data format provides a view of the physical layout of data.

- Notably, Bafna et al use this stack model to motivate `GQL` [26]. While a query system should provide a way to process and transform data, Bafna et al instead move this system down to the data materialization layer. We feel that this approach inverts the semantics that a user of `GQL` would expect.

Our stack enables us to serve the use cases we outline in §3.2. By using `Parquet` as a storage format, we are able to process the data in many `Hadoop`-based systems. We implement high performance batch and interactive processing with `Spark`, and can delegate to systems like `Impala` [25] and `Spark SQL` [5] for data warehousing.

3.2 Workloads

There are several common threads that unify the diverse set of applications that make up scientific computing. When looking at the data that is used in different fields, several trends pop out:

1. Scientific data tends to be rigorously associated with coordinates in some domain. These coordinate systems vary, but can include:
   - Time (e.g., fMRI data, particle simulations)
   - Chromosomal position (e.g., genomic read alignments and variants)
   - Position in space (imaging data, some sensor datasets, see `Hadoop-GIS` [2] and `Spatial-Hadoop` [16])

2. For aggregated data, we frequently want to slice data into many different views. For example, for time domain data aggregated from many sensors, scientists may want to perform analyses by slicing across a single point in time, or by slicing across a single sensor. In genomics, we frequently aggregate data across many people from a given population. Once we have done this first aggregation, we may want to then slice the data by subsets of the population, or by regions of the genome (e.g., specific genes of interest).

There are two important consequences of the characteristics above. First, since data is attached to a coordinate system, the coordinate system itself may impose logical processing patterns. For example, for time domain data, we may frequently need to run functions that pass a sliding window across the dataset (e.g., for convolution). Second, the slicing of aggregated data is frequently used to perform analyses across subsets of a larger dataset. This slicing is common if we want to study a specific phenomenon, like the role of a gene in a disease (a common analysis in the TCGA [53]), or the measured activity in a single lobe of the brain while performing a task. Since the datasets we are processing are large,[2] it may be uneconomical to colocate data with processing nodes, because of either the number of nodes that would need to be provisioned, or the amount of storage that would need to be provisioned per node.

[1] Current BQSR and Duplicate Marking implementations fail in certain corner-case alignments, due to an improper sort order invariant.

[2] For example, the Acute Myeloid Leukemia subset of the TCGA alone is over 4 TB in size, and is only one of 20 cancers in the TCGA.

For scientific fields that process very large datasets, the exact processing techniques and algorithms vary considerably, but common processing trends do exist:

1. There is increasing reliance on statistical methods. The Thunder pipeline makes heavy use of the MLI/MLLib statistical libraries [18, 47], and tools like the GATK perform multiple rounds of statistical refinement [14].

2. Many scientific applications are data parallel. This parallelism varies across applications; in some applications (like genomics), we may leverage the independence of sites across a coordinate system and process individual coordinate regions in parallel. For other systems, we may have matrix calculations that can be parallelized [47], or we may be able to run processing in parallel across samples or traces.

Additionally, there are several different emerging use cases for scientific data processing and storage systems. These different use cases largely correspond to different points in the lifecycle of the data:

- **Batch processing:** After the initial acquisition of raw sensor data (e.g., raw DNA reads, brain electrode traces, telescope images), a batch processing pipeline (e.g., Thunder or the GATK) performs dimensionality reduction or statistical summarization of the data. This processing is generally used to extract notable features from the data, such as turning raw genomic reads into variant alleles, or identifying areas of activity in neuroscience traces. These tasks are unlikely to have any interactive component, and are likely to be long running compute jobs.

- **Ad hoc exploration:** Batch processing is often followed by exploratory processing of the results. For example, when studying disease genetics, a geneticist may use the variant/genotype statistics to identify genomic sites with statistically significant links to the disease phenotype. Data exploration tasks have a significant user facing/interactive nature, and are generally performed by scientists who may be programming laypeople.

- **Data warehousing:** In large scientific projects, it is common to make data available to the members of the scientific community through some form of warehouse service (e.g., the Cancer Genomics Hub, CGHub, for the TCGA). As is the case for all data warehousing, this implies that point queries must be made reasonably efficient, even though the data is expected to be cold. To reduce the cost of storing data, we may prioritize compression here; this has led to the creation of compressed storage formats like CRAM [19].

In this paper, we design a system that can achieve all of the above goals. The genomics and astronomy pipelines we demonstrate achieve improvements in batch processing performance, and allow for interactive/exploratory analysis through both Scala and Python. Through the layering principles we lay out in the next section and the performance optimizations we introduce in §5.2, we make our system useful for warehousing scientific data.

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 [59], Avro [3], and Parquet [4]. We have chosen these two applications as they fit in different areas in the design space. Specifically, the genomics pipeline makes heavy use of statistical processing techniques over semistructured data, while the astronomy application has a traditional matrix structure.

Corresponding to the stack model that was introduced in Figure 1, we use the following technologies to implement both of our applications:

1. **Physical Storage:** We have designed our system to run on top of local or distributed drives, as well as block stores.

2. **Data Distribution:** Our system is designed to operate on top of the Hadoop Distributed File System (HDFS), or to coordinate data distribution backed by an Amazon S3 bucket. We describe these optimizations in §5.2.

3. **Materialized Data:** We store data using the open source Parquet columnar store [4].

4. **Schema:** We manage our schemas (and data serialization) via the Avro serialization framework [3]. Our schemas are described in Appendix B.

5. **Evidence Access:** We use Spark's Resilient Distributed Dataset (RDD, [58]) abstraction to provide parallel processing over the data. We enhance this with the join patterns we describe in §5.1.

6. **Presentation:** In our genomics application, we provide several rich datatypes that implicitly wrap our schemas to provide convenience methods for metadata access. This is not as crucial in the astronomy application.

In the remainder of this section, we describe the applications that we have implemented, and the optimizations we have made to improve the horizontal scalability of these algorithms.

4.1 Genomics Pipeline

Contemporary genomics has been revolutionized by "next generation" sequencing technologies (NGS), which have driven a precipitous drop in the cost of genomic assays [38]. Although there are a variety of sequencing technologies in use, the majority of sequence data comes from the Illumina sequencing platform, which uses a "sequencing-by-synthesis" technique to generate short read data [36]. Short reads are genomic subsequences 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 65× coverage will produce approximately 1.4 billion reads of 150 base length. This 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. Most sequencing assays sequence "read pairs" where two reads are known to have come from a single fragment of DNA, with a known approximate distance between each read.

One of the most common genomic analyses is *variant calling*, which is a statistical process to infer the sites where a single individual varies from the reference genome.[3] To call variants, we perform the following 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 that has focused on indexing strategies for improving alignment performance [30, 32, 57]. 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. These adjustments 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 [14].

3. **Variant calling:** Variant calling is a statistical process that 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, may involve complex traversals of the genomic coordinate space, or may require us to fit a statistical model to all or part of the dataset. While we do not present work on variant filtration in this paper, variant filtration has motivated the coordinate space joins presented in §5.1.

This process is very expensive in time to run; the current best practice pipeline uses the `BWA` tool [30] for alignment and the `GATK` [14, 34] 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 [49]. Recent projects have achieved 5–10× improvements in alignment and variant calling performance [41, 57], which makes the pre-processing stages the performance bottleneck. Our experimental results have corroborated this, as the four pre-processing stages take approximately 35 hours to run on a clinical quality human genome when run on an Amazon EC2 `i2.8xlarge` machine. We have focused on implementing the four most-commonly used pre-processing stages, as well as Flagstat, a command used for validating the quality of an aligned sample. Flagstat operates by performing an aggregate over boolean fields attached to each read. In the remainder of this section, we describe the stages that we have implemented, and the techniques we have used to improve performance and accuracy.

1. **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 [59]; we solely need to define an ordering for genomic coordinates, which is well defined.

2. **Duplicate Removal:** During the process of preparing DNA for sequencing, errors during sample preparation and sequencing can lead to the duplication of some 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, all but the highest quality read are marked as duplicates.

   We have validated our duplicate removal code against `Picard` [51], which is used by the `GATK` for marking duplicates. Our implementation is fully concordant with the `Picard`/`GATK` duplicate removal engine, except we are able to perform duplicate marking for chimeric read pairs.[4] Specifically, because `Picard`'s traversal engine is restricted to processing linearly sorted alignments, `Picard` mishandles these alignments.

3. **Local Realignment:** In local realignment, we correct areas where variants cause reads to be locally misaligned from the reference genome.[5] In this algorithm, we first identify regions as targets for realignment. In the `GATK`, this is done by traversing sorted read alignments. In our implementation, we generate targets from reads, and then compute the convex hull of overlapping targets. We introduce a parallel algorithm for this in Appendix C.

   After we have generated the targets, we associate reads to the overlapping target, if one exists. After associating reads to realignment targets, we run a heuristic realignment algorithm that works by minimizing the quality-score weighted number of bases that mismatch against the reference.

4. **Base Quality Score Recalibration (BQSR):** During the sequencing process, systemic errors occur that lead to the incorrect assignment of base quality scores. In this step, BQSR labels each sequenced base with an *error covariate*, and counts the total number of bases and the number of bases that mismatched against the reference genome per covariate bin. The correction is applied by estimating the error probability for each set of covariates under a beta-binomial model with uniform prior:

$$\mathbf{E}(P_{err}|cov) = \frac{\texttt{\#errors}(cov) + 1}{\texttt{\#observations}(cov) + 2} \quad (1)$$

   We have validated the concordance of our BQSR implementation against the original implementation in

[3] The reference genome represents the "average" genome for a species. The Human Genome Project [28] assembled the first human reference genome.

[4] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [30].

[5] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al [14].

the GATK [14]. Across both tools, only 5000 of the ~180B bases (< 0.0001%) in the high-coverage NA12878 genome dataset differ (dataset information in Appendix E). After investigating this discrepancy, we have determined that this is an error in the GATK caused by a sort order invariant. Specifically, paired-end reads are mishandled if the two reads in the pair overlap.

For current implementations of these read processing steps, performance is limited by disk bandwidth [15]. This bottleneck exists because the operations read in a SAM/BAM file, perform a small amount of processing, and write the data to disk as a new SAM/BAM file. We achieve a performance bump 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. Additionally, by rethinking the design of our algorithms, we are able to reduce overhead in several other ways:

1. Current algorithms require the reference genome to be present on all nodes. This assembly is then used to look up the reference sequence that overlaps all reads. The reference genome is several gigabytes in size, and performing a lookup in the reference genome can be costly due to its size. Instead, we leverage the mismatchingPositions field in our schema to embed information about the reference in each read. This optimization allows us to avoid broadcasting the reference, and provides O(1) lookup.

2. Shared-memory genomics applications tend to be impacted significantly by false sharing of data structures [57]. Instead of having data structures that are modified in parallel, we restructure our algorithms so that we only touch data structures from a single thread, and then merge structures in a reduce phase. The elimination of sharing improves the performance of covariate calculation during BQSR and the target generation phase of local realignment.

3. In a naïve implementation, the local realignment and duplicate marking tasks can suffer from stragglers. The stragglers occur due to a large amount of reads that either do not associate to a realignment target, or that are unaligned. We pay special attention to these cases by manually randomizing the partitioning for these reads. This randomization resolves load imbalance and mitigates stragglers.

4. For the Flagstat command, we are able to project a limited subset of fields. Flagstat touches fewer than 10 fields, which account for less than 10% of space on disk. We discuss the performance implications of this further in §6.3.

These techniques allow us to achieve a 28× performance improvement over current tools, and scalability beyond 128 machines. We perform a detailed performance review in §6.1.

4.2 Astronomy Image Processing

The Montage [24] application is an astronomy image processing pipeline that builds "mosaic" images by combining small image tiles obtained from telescopes. Montage has the requirement of preserving the energy quantity and position of each pixel between the input and output images. The pipeline has the following four phases:

1. **Tile Reprojection** reprojects the raw images with the scale and rotation required for the final mosaic.

2. **Background Modeling** smooths out the background levels between each pair of overlapped images and fits a plane to each of them. This phase can be further divided into overlap calculation, difference image creation, and plane-fitting coefficient calculation.

3. **Background Matching** removes the backgrounds from the reprojected images. The best solution from the previous phase is used to smooth out the overlap between tiled images.

4. **Tile Mosaicing** uses a smoothing function to merge all corrected images into a aggregated mosaic file, after applying background matching to the reprojected images. This phase also includes a metadata processing stage prior to mosaicing.

The tile reprojection, background modeling, and background matching phases are embarrassingly parallel, and each task (both computation and I/O) can run independent of other tasks in the same phase. We care the most about the tile mosaicing phase because the MPI implementation requires a preceding stage to summarize the metadata of all corrected images to produce a metadata table containing the tile positioning information. This particular stage is inefficient because it has to read all input files into memory, but only accesses a small portion of the file. In addition, the current implementation parallelizes the computation with each matrix row as the element, which results in an inefficient replication of input images when executed in a distributed environment. We address the metadata issue by explicitly integrating the metadata into the denormalized image data schema. We then store the images in a columnar store which allows all input images to be loaded into memory a single time. All subsequent computation proceeds in memory.

5. DATA ACCESS OPTIMIZATIONS FOR SCIENTIFIC PROCESSING

In §3.2, we discussed several processing patterns that were important for scientific data processing. In this section, we introduce optimizations for two important use cases. First, we present a join pattern that enables processing that traverses a coordinate system. Region joins enable distributed implementations of important genomics algorithms. Second, we implement an efficient method for applying predicates and projections into data that is stored in a remote block store. Enabling predicates and projections on remote data allows us to defer as much remote data movement as is possible and improves the efficiency of accessing remotely staged data.

5.1 Coordinate System Joins

There are a wide array of experimental techniques and platforms in genome informatics, but many of these methods produce datapoints that are tied to locations in the genome through the use of genomic coordinates. Each cell contains a copy of the genome with one molecule per chromosome.

Each molecule is a collection of DNA polymers coated with (and wrapped around) proteins and packed into the nucleus in a complex 3-dimensional shape. In practice, computational biologists abstract this complexity by storing a single long string that represents the nucleotides of the chromosome. We can then connect a datapoint or observation to the genome by associating the data with the chromosome name and a point or interval on a 1-dimensional space.

A platform for scientific data processing in genomics needs to understand these 1-dimensional coordinate systems because these become the basis on which data processing is parallelized. For example, when calling variants from sequencing data, the sequence data that is localized to a single genomic region (or "locus") can be processed independently from the data localized to a different region, as long as the regions are far enough apart.

Beyond parallelization, many of the core algorithms and methods for data aggregation in genomics are phrased in terms of geometric primitives on 1-D intervals and points where we compute distance, overlap, and containment. An algorithm for calculating quality control metrics may try to calculate "coverage," a count of how many reads overlap each base in the genome. A method for filtering and annotating potential variants might assess the validity of a variant using the quality characteristics of all reads that overlap the putative variant.

To support these algorithms, we provide a "region" or "spatial" join primitive. The algorithm used is described in algorithm 1 and takes as input two sets (RDDs, see Zaharia et al [58]) of `ReferenceRegions`, a data structure that represents intervals along the 1-D genomics coordinate space. It produces the set of all overlapping `ReferenceRegion` pairs. The *hulls* variable contains the set of convex hulls and is broadcasted to all compute nodes during the join.

Algorithm 1 Partition And Join Regions via Broadcast

```
$left \leftarrow$ input dataset; left side of join
$right \leftarrow$ input dataset; right side of join
$regions \leftarrow left$.map($data \Rightarrow$generateRegion($data$))
$regions \leftarrow regions$.groupBy($region \Rightarrow region.name$)
$hulls \leftarrow regions$.findConvexHull()
$hulls$.broadcast()
$keyLeft \leftarrow left$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
$keyRight \leftarrow right$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
$joined \leftarrow keyLeft$.join($keyRight$)
$truePositives \leftarrow joined$.filter($r1, r2 \Rightarrow r1$.overlaps($r2$))
return $truePositives$
```

To find the maximal set of non-overlapping regions, we must find the convex hull of all regions emitted. We present a distributed algorithm for finding convex hulls in Appendix C. The distributed convex hull computation problem is important because it is used both for computing regions for partitioning during a region join and for performing INDEL realignment.

While the join described above is a broadcast join, a region join can also be implemented via a straightforward shuffle-based approach, which is described in Algorithm 2. The `partitionJoinFn` function maintains two iterators (one each from both the left and right collections), along with a buffer. This buffer is used to track all key-value pairs from the right collection iterator that *could* match to a future key-value pair from the left collection iterator. We prune this buffer every time that we advance the left collection iterator. For simplicity, the description of Algorithm 2 ignores the complexity of processing keys that cross partition boundaries. In our implementation, we replicate keys that cross partition boundaries into both partitions.

Algorithm 2 Partition And Join Regions via Shuffle

```
$left \leftarrow$ input dataset; left side of join
$right \leftarrow$ input dataset; right side of join
$partitions \leftarrow left$.getPartitions()
$left \leftarrow left$.repartitionAndSort($partitions$)
$right \leftarrow right$.repartitionAndSort($partitions$)
$joined \leftarrow left$.zipPartitions($right$, partitionJoinFn)
return $joined$
```

These joins serve as a core that we can use to build other abstractions with. For example, self-region joins and multi-region joins are common in genomics, and can be easily implemented using the above implementations. We are currently working to implement further parallel spatial functions such as sliding windows, using techniques similar to the shuffle-based join. We are working to characterize the performance differences between the two join strategies described above. In the future, we hope to enable the use of the region join in a SQL based system such as `Spark SQL` [5].

5.2 Loading Remote Data

Another challenge faced by scientific systems is where to store the initial data files and how to load them efficiently. Today, `Spark` is usually run in conjunction with the `HDFS` portion of the `Hadoop` stack—`HDFS` provides data locality, access to local disk on each node of the `Spark` cluster, and robustness to node failure. However, `HDFS` imposes significant constraints on running a `Spark` system in virtualized or commodity computing (e.g. "cloud") environments. It is easy to scale an `HDFS`-based system up to larger numbers of nodes, but harder to remove nodes when the capacity is no longer needed.

If we are willing to forgo the advantages of local disk and data locality provided by `HDFS`, however, we may be able to relax some of these other restrictions and build a `Spark`-based cluster whose size is more easily adjusted to the changing demands of the computation. By storing our data in higher-latency, durable, cheaper block storage (e.g., S3) we can also exploit the varying requirements of data availability—not all datasets need to be kept "hot" in `HDFS` at all times, but can be accessed in a piecemeal or parallelized manner through S3 interfaces.

`Spark` provides a particularly convenient abstraction for writing these new data access methods. By implementing our own data-loading RDD, we are able to allow a `Spark` cluster to access `Parquet` files stored in S3 in parallel (each partition in the RDD reflects a row group in the corresponding `Parquet` file). For `Parquet` files containing records that reflect known genomics datatypes (that are mapped to genomic locations, for example) we generate simple index files for each `Parquet` file. Each index file lists the complete set of row groups for the `Parquet` file, as well which genomic regions contain data points within each row group. Our parallelized data loader reads this index file and restricts the partitions in the data loading RDD it creates to only those `Parquet` row groups that possibly contain data relevant to the user's query.

6. PERFORMANCE

Thus far, we have discussed ways to improve the performance of scientific workloads that are being run on commodity map-reduce systems by rethinking how we decompose and build algorithms. In this section, we review the improvements in performance that we are able to unlock. We achieve near-linear speedup across 128 nodes for a genomics workload, and achieve a 3× performance improvement over the current best MPI-based system for the Montage astronomy application. Additionally, both systems achieve 25-50% compression over current file formats when storing to disk.

6.1 Genomics Workloads

Table 1 previews our performance versus current systems. The tests in this table are run on the high coverage NA12878 full genome BAM file that is available from the 1000 Genomes project (access information in Appendix E). These tests have been run on the EC2 cloud, using the instance types listed in Table 3. We evaluated ADAM against the GATK [14], SAMtools [32], Picard [51], and Sambamba [50]. We evaluated the performance of BQSR, INDEL realignment (IR), duplicate marking (DM), sort, and Flagstat (FS). Blank entries (—) indicate that a tool did not implement that feature. For the ADAM runs, we present speedup relative to the fastest legacy tool.

Table 1: Summary Performance on NA12878

Tool	EC2	BQSR	IR	DM	Sort	FS	Total
[14]	1†	**1283m**	**658m**	—	—	—	2075m1
[32]	1†	—	—	509m	203m	54m41	
[50]	1†	—	—	**44m50**	**83m**	**6m11**	
[51]	1†	—	—	160m	562m	—	
ADAM	1†	1602m 1/1.25×	366m 1.7×	143m 1/3.8×	108m 1/1.3×	2m17 2.7×	2221m17 1/1.07×
ADAM	32⋆	74m 17×	64m 10×	34m56 1.2×	39m23 2.1×	0m43 8.6×	223m2 9.3×
ADAM	64⋆	41m52 30×	35m39 18×	21m35 2.0×	18m56 4.3×	0m49 7.5×	118m51 17×
ADAM	128⋆	25m59 49×	20m27 32×	15m27 2.9×	10m31 7.9×	1m20 4.3×	73m44 28×

We compute the cost of running each experiment by multiplying the number of instances used by the total wall time for the run by the cost of running a single instance of that type for an hour, which is the process Amazon uses to charge customers. This data is shown in Table 2. Although ADAM is more expensive than the best legacy tool (Sambamba [50]) for sorting and duplicate marking, ADAM is less expensive for all other stages. In total, using ADAM reduces the end-to-end analysis cost by 63% over a pipeline constructed out of solely legacy tools.

Table 2: Cost on NA12878

Stage	Legacy		ADAM	
	Tool	Cost	EC2	Cost
BQSR	[14]	$132.57	32⋆	**$27.62**
IR	[14]	$67.99	32⋆	**$23.89**
DM	[50]	**$4.63**	32⋆	$13.04
Sort	[50]	**$8.57**	64⋆	$14.13
FS	[50]	$0.63	1†	**$0.24**
Total		$214.39		$78.92

Table 3 describes the instance types. Memory capacity is reported in Gibibytes (GiB), and the cost reported is the cost of one hour on one machine. 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 i2.8xlarge instance is chosen to represent a workstation. Network bandwidth is constant across all instances.

Table 3: AWS Machine Types

	Machine	Cost	Description
†	i2.8xlarge	$6.20	32 proc, 244G RAM, 8 SDD
⋆	r3.2xlarge	$0.70	8 proc, 61G RAM, 1 SDD

As can be seen from these results, ADAM is within 7% of the state of the art when running on a single machine. However, ADAM achieves superlinear speedup when increasing the cluster size by 8–16×, and near linear speedup when increasing the cluster size by 32× to 128 nodes. This conclusion is not necessarily clear from Table 1, as we change instance sizes when scaling the cluster, but Figure 2 presents a per-core speedup plot for the NA12878 high coverage genome.

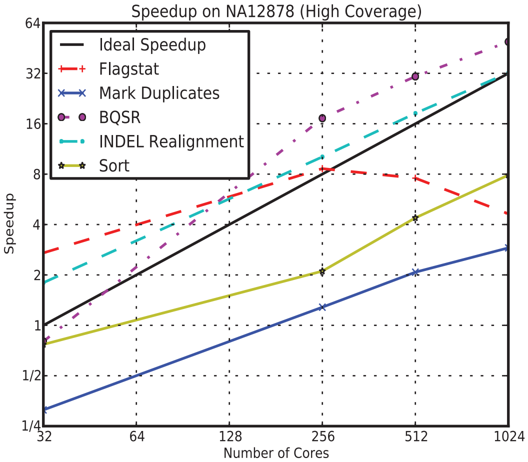

Figure 2: Speedup on NA12878

When testing on NA12878, we achieve near-linear speedup out through 1024 cores using 128 r3.2xlarge nodes. In this test, our performance is limited by several factors:

- Although columnar stores have very high read performance, they have poor write performance. Our tests exaggerate the penalty of poor write performance since we write the same amount of data as we read. In a typical variant calling pipeline, the input will be a large read file, but the pipeline's output will be a variant call file that is approximately two orders of magnitude smaller. Since the amount of data written is much smaller, the penalty of the reduced write performance is decreased. In practice, we also use in-memory caching to chain stages together. This amortizes write time across several stages of computation.

- Additionally, for large clusters, straggler elimination is an issue. However, we have made optimizations to both the duplicate marking and INDEL realignment code to eliminate stragglers by randomly rebalancing reads that are unmapped/do not map to a target across partitions.

We do note that the performance of Flagstat degrades going from 32 to 128 `r3.2xlarge` nodes. Flagstat executes in two minutes on a single node. By increasing the number of machines we use to execute this query, we increase scheduling overhead, which leads to degraded performance. When running on the 128 machine cluster, approximately 75% of the runtime was spent scheduling and broadcasting the task to be run.

6.2 Astronomy Workloads

To evaluate the mosaicing application, we use the 2MASS data[6] and the `Montage` test case of 3x3 degree mosaicing with Galaxy m101 as the center. The tile mosaicing phase converts 1.5 GB of input data into a 1.2 GB aggregated output file. We compare the `Spark-mAdd` performance against the HPC styled MPI-based parallel implementation from `Montage` v3.3 (`MPI-mAdd`). We performed the test on 1, 4, and 16 Amazon `c3.8xlarge` instances. We chose the `c3.8xlarge` instances for this test because they provided HPC-optimized networking, which is a prerequisite for good MPI performance. We use `OrangeFS` v2.8.8—a successor of `PVFS` [9]—as the shared file system when running `MPI-mAdd`. All 32 cores on each instance are used for both `Spark-mAdd` and `MPI-mAdd`.

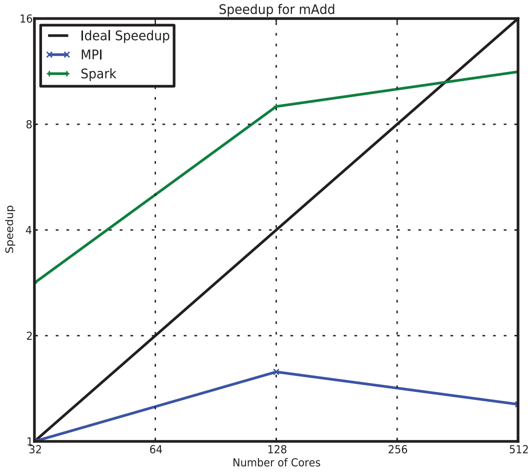

Figure 3: Speedup when running *mAdd* using MPI and Spark

As shown in Figure 3, `Spark-mAdd` runs 2.8x, 5.7x, 8.9x faster than `MPImAdd` on 1, 4, and 16 instances. In the single machine case, `MPI-mAdd` achieves a cost of $0.17 per analysis. `Spark-mAdd` costs $0.06 to run on a single instance, which is a 2.8× improvement in cost. `Spark-mAdd` is still cheaper than `MPI-mAdd` by a factor of 2.2× when running on four nodes. `Spark-mAdd` is only more expensive than a single node of `MPI-mAdd` when running on 16 nodes, and even then is only 40% more expensive than the lowest cost `MPI-mAdd` run while providing a 8.9× performance improvement over the fastest `MPI-mAdd` run.

The performance improvement is caused by multiple factors. We are able to reduce the amount of I/O performed, while also reducing contention in the I/O system and improving data locality. By denormalizing the metadata into our data schema, we are able to combine the metadata processing stage with the mAdd stage. This combination allows us to only load the input dataset a single time. `Parquet` also compresses the input and output data, which reduces the volume of I/O performed. Additionally, the MPI implementation is bound by contention when trying to write all output to a single file in a shared file system, while `Parquet` writes output files into `HDFS` in a contention free manner. Finally, `Spark` allows the computation to benefit from data locality, while MPI distributes the computation across the available resources without optimizing for data placement.

While the dataset used is a small dataset, larger datasets are commonplace. For example, the Large Synoptic Survey Telescope (LSST, [23]) has been used to collect terabyte sized datasets [37]. In future work, we plan to tackle these very large astronomy datasets using our framework.

6.3 Column Store Performance

Earlier in this paper, we motivated the use of a column store as it would allow us to better push processing to the data. Specifically, we can use predicate pushdown and projections to minimize the amount of I/O that we perform. Additionally, column stores provide compressed storage and allow us to minimize both the required I/O bandwidth and space on disk. In this section, we look at the read performance and compression achieved by using a columnar store. We will not look extensively at write performance; for genomic data, write performance is not a bottleneck because our workflow computes a summarization of a large dataset. As a result, our output dataset tends to be O(100 MB) while our input dataset is in the range of O(10 GB)–O(100GB).

6.3.1 Compression

The `Parquet` columnar store [4] supports several compression features. Beyond block-level compression, `Parquet` supports run length encoding for repeated values, dictionary encoding, and delta encoding. Currently, we make use of run length encoding to compress highly repeated metadata value, and dictionary encoding to compress fields that can take a limited range of values. Dictionary encoding provides substantial improvements for genomic data; specifically, the majority of genomic sequence data can be represented with three bits per base.[7] Three bits are an improvement over our in-memory string representation that allocates a byte per base.

Table 4 shows the compression we achieve on the `NA12878` and `HG00096` human genome sequencing samples (datasets are described in Appendix E). We compare against the GZIP compressed `BAM` [31] format, and the `CRAM` format [19]. We achieve approximately a 1.25× improvement in storage. This is not as impressive as the result achieved by the `CRAM` project, but `CRAM` applies genome-specific compression tech-

[6] Available from `http://irsa.ipac.caltech.edu/applications/2MASS/IM/`.

[7] Although DNA only contains four bases (A, C, G, and T), *sequenced* DNA uses disambiguation codes to indicate that a base was read in error. As a result, we cannot achieve the ideal two-bits per base.

niques that make use of the read alignment. Specifically, CRAM only stores the read bases that *do not* appear in the reference genome. As we only expect a genomic variant at one in every 1000 bases, and a read error at one in every 50 bases, this allows them to achieve significant compression of the sequenced bases. Additionally, CRAM applies lossy compression to quality scores. This compression approach is significant, as quality scores are 60% of the data stored on disk in the ADAM/Parquet format.

Table 4: Genomic Data Compression

	NA12878	
Format	**Size**	**Compression**
GZIP BAM	234 GB	—
CRAM	112 GB	2.08×
Parquet	185 GB	1.26×
	HG00096	
Format	**Size**	**Compression**
GZIP BAM	14.5 GB	—
CRAM	3.6 GB	4.83×
Parquet	11.4 GB	1.27×

The astronomy datasets achieve higher compression ratios. Table 5 compares our storage system against the legacy FITS [54] format. We measured the aggregate compression of the image files provided as input to our system, and the compression of our pipeline output.

Table 5: Astronomy Data Compression

	Input Dataset	
Format	**Size**	**Compression**
FITS	1.5 GB	—
Parquet	0.55 GB	2.75×
	Output Dataset	
Format	**Size**	**Compression**
FITS	1.2 GB	—
Parquet	0.88 GB	1.35×

For genomic datasets, our compression is limited by the sequence and base quality fields, which respectively account for approximately 30% and 60% of the space spent on disk. Quality scores are difficult to compress because they are high entropy. We are currently looking into computational strategies to address this problem; specifically, we are working to probabilistically estimate the quality scores without having observed quality scores. This estimation would be performed via a process that is similar to the base quality score recalibration algorithm presented earlier in this paper.

6.3.2 Horizontal Scalability

The representation Parquet uses to store data to disk is optimized for horizontal scalability in several ways. Specifically, Parquet is implemented as a hybrid row/column store where the whole set of records in a dataset are partitioned into row groups that are then serialized in a columnar layout. This partitioning provides us with two additional benefits:

1. We are able to perform parallel access to Parquet row groups without consulting metadata or checking for a file split.
2. Parquet achieves very even balance across partitions. On the HG00096 dataset, the average partition size was 105 MB with a standard deviation of 7.4 MB. Out of the 116 partitions in the file, there is only one partition whose size is not between 105–110MB.

Parquet's approach is preferable when compared to Hadoop-BAM [39], a project that supports the direct usage of legacy BAM files in Hadoop. Hadoop-BAM must pick splits, which adds non-trivial overhead. Additionally, once Hadoop-BAM has picked a split, there is no guarantee that the split is well placed. It is only guaranteed that the split position will not cause a functional error. Finally, although BAM metadata is centralized, Hadoop-BAM punts on metadata distribution, and users must manually broadcast the metadata.

6.3.3 Projection and Predicate Performance

We use the Flagstat workload to evaluate the performance of predicates and projections in Parquet. We define three projections and four predicates, and test all of these combinations. In addition to projecting the full schema (see Appendix B.1), we also use the following two projections:

1. We project the read sequence and all of the flags (40% of data on disk).
2. We only project the flags (10% of data on disk).

Beyond the null predicate (which passes every record), we evaluate the following three predicates:

1. We pass only uniquely mapped reads (99.06% of reads).
2. We pass only the first pair in a paired end read (50% of reads, see §4.1 for definition of "paired end").
3. We pass only *un*mapped reads (0.94% of reads).

Table 6: Predicate/Projection Speedups

	0	1	2
0	—	1.7	1.9
1	1.0	1.7	1.7
2	1.3	2.2	2.6
3	1.8	3.3	4.4

Table 6 documents the speedup we achieve by combining predicate pushdown and projections. Projections are arranged in the columns of the table while predicates are assigned to rows. We achieve a 1.7× speedup by moving to a projection that eliminates the deserialization of our most complex field (the quality scores that consume 60% of space on disk), while we only get a 1.3× performance improvement when running a predicate that filters 50% of records. This difference can be partially attributed to overhead from predicate pushdown; we must first deserialize a column, process the filter, and then read all records who passed the pushdown filter. If we did not perform this step, we would be able to do a straight scan over all of the data in each partition.

7. DISCUSSION AND FUTURE WORK

Similar to what we propose, the Thunder system was developed as a novel map-reduce-based system for processing terabytes of neuroscience imaging data [18]. Thunder performs a largely statistical workload, and the significant tasks in terms of execution time are clustering and regression. 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 [47]. Thunder uses Spark's filtering primitives to allow scientists to cut problems into subproblems. This slicing and dicing is a common trend across scientific analyses, and is one of the reasons that we advocate for the use of a columnar store with efficient predicate pushdown.

There has also been work to optimize map-reduce systems for processing geospatial data. Significant projects include SpatialHadoop [16] and Hadoop-GIS [2]. These projects have focused on indexing strategies that can accelerate range queries against spatial data and improve work balance for map-reduce tasks run in Hadoop on spatial data. These approaches are similar to the region join that we propose in §5.1, but restricted to geospatial and imaging data.

Our genomics work leverages columnar storage to improve performance and compression of data on disk, with special emphasis on repetitive fields that can be run length encoded (RLE). While this improves disk performance, it has the side effect of making data consume significantly more space in memory than on disk. We are currently investigating techniques that leverage the immutability of data in our applications to reduce memory consumption and have modified Parquet's deserialization codec. For every value that is RLE'd, we allocate the value once in memory and share the value across all records which contained that value. This allocation pattern enables denormalizing repeated metadata.

It is worth noting that there are many significant scientific applications (such as genome assembly) that are expressed as traversal over graphs. Recent work by Simpson et al (ABySS, [45]) and Georganas et al [21] has focused on using MPI or Unified Parallel C (UPC) to implement their own distributed graph traversal. Both systems find that synchronization via message passing is a significant cost. By building our system using Spark, we are able to leverage the GraphX processing library [22, 55]. We are in the process of developing a genome assembler using this library system, and believe that we can achieve improved performance through careful graph partitioning. This partitioning involves algorithmic changes to the graph creation and traversal phases to bypass "knotted" sections of the graph.

In this paper, we focus on the GATK [14] as an example of a genome processing pipeline that needs to be distributed to improve analysis latency and throughput. While the GATK is widely used, there is significant debate as to whether the expensive methods employed by the GATK are necessary. Several new "minimal preprocessing" based methods such as SpeedSeq [10] have achieved achieved results that are comparable to the GATK, while eliminating the computationally expensive BQSR and INDEL realignment stages. While these approaches do improve throughput, they do not tackle parallelism beyond a single machine, which will be necessary for analyzing very large cohorts. Additionally, the proponents of "minimal" pipelines frequently validate accuracy on high quality, whole genome sequencing datasets. Since whole genome sequencing methods contain fewer sources of error than targeted/whole exome sequencing panels, a more comprehensive evaluation is needed to validate the impact of removing expensive stages.

8. CONCLUSION

In this paper, we have advocated for an architecture for decomposing the implementation of a scientific system, and then demonstrated how to efficiently implement genomic and astronomy processing pipelines using the open source Avro, Parquet, and Spark systems [3, 4, 59]. We have identified common characteristics across scientific systems, like the need to run queries that touch slices of datasets and the need for fast access to metadata. We then enforced data independence through a layering model that uses a schema as the "narrow waist" of the stack, and used optimizations to make common, coordinate-based processing fast. By using Parquet, a modern columnar store, we use predicates and projections to minimize I/O, and denormalize our schemas to improve the performance of accessing metadata.

By rethinking the architecture of scientific data management systems, we have been able to achieve parity on single node systems, while providing linear strong scaling out to 128 nodes. By making it easy to scale scientific analyses across multiple commodity machines, we enable the use of smaller, less expensive computers, leading to a 63% cost improvement and a 28× improvement in read preprocessing pipeline latency. On the astronomy workload, we achieve speedup between 2.8–8.9× speedup over the current best MPI-based solution at various scales. By applying our techniques to both astronomy and genomics, we have demonstrated that the techniques are applicable to both traditional matrix-based scientific computing, as well as novel scientific areas that have less structured data.

9. ACKNOWLEDGEMENTS

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, NIH BD2K Award 1-U54HG007990-01, NIH Cancer Cloud Pilot Award HHSN261201400006C and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata, VMware, and Yahoo!. Author FAN is supported by a National Science Foundation Graduate Research Fellowship.

We would also like to thank our colleagues who have provided feedback and engineering effort to the ADAM system, including André Schumacher, Christopher Hartl, Karen Feng, Eric Tu, Kristal Curtis, Taylor Sittler, Neal Sidhwaney, Ryan Williams, Ravi Pandya, and the UCSC Genome Bioinformatics group. As ADAM is an open source project, we also would like to thank the community members who have contributed code and use cases to the project, and would especially like to thank Neil Ferguson, Andy Petrella, Xavier Tordior, and Michael Heuer. Additionally, the authors would like to thank the anonymous reviewers and our shepherd, Ken Yocum, for their helpful commentary on this paper.

APPENDIX

A. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, pages 671–682. ACM, 2006.

[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

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

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

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015.

[6] 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.

[7] 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.

[8] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 963–968. ACM, 2010.

[9] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proceedings of the Linux Showcase & Conference*, pages 28–28. USENIX Association, 2000.

[10] C. Chiang, R. M. Layer, G. G. Faust, M. R. Lindberg, D. B. Rose, E. P. Garrison, G. T. Marth, A. R. Quinlan, and I. M. Hall. SpeedSeq: Ultra-fast personal genome analysis and interpretation. *bioRxiv*, page 012179, 2014.

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

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

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

[14] 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.

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

[16] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '15)*. IEEE, 2015.

[17] G. G. Faust and I. M. Hall. SAMBLASTER: fast duplicate marking and structural variant read extraction. *Bioinformatics*, page btu314, 2014.

[18] 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.

[19] 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.

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

[21] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, 2014.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI '14)*. ACM, 2014.

[23] Z. Ivezic, J. Tyson, E. Acosta, R. Allsman, S. Anderson, J. Andrew, R. Angel, T. Axelrod, J. Barr, A. Becker, et al. LSST: from science drivers to reference design and anticipated data products. *arXiv preprint*, 2008.

[24] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

[25] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russel, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '15)*, 2015.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[36] M. L. Metzker. Sequencing technologies—the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

[37] M. Moyers, E. Soroush, S. C. Wallace, S. Krughoff, J. Vanderplas, M. Balazinska, and A. Connolly. A demonstration of iterative parallel array processing in support of telescope image analysis. *Proceedings of the VLDB Endowment*, 6(12):1322–1325, 2013.

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

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

[40] 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.

[41] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. Twigg, A. O. Wilkie, G. McVean, G. Lunter, WGS500 Consortium, et al. Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature Genetics*, 46(8):912–918, 2014.

[42] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[43] 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.

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

[45] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

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

[47] 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 *Proceedings of the IEEE International Conference on Data Mining (ICDM '13)*, pages 1187–1192. IEEE, 2013.

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

[49] 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.

[50] A. Tarasov, A. J. Vilella, E. Cuppen, I. J. Nijman, and P. Prins. Sambamba: fast processing of NGS alignment formats. *Bioinformatics*, 2015.

[51] The Broad Institute of Harvard and MIT. Picard. `http://broadinstitute.github.io/picard/`, 2014.

[52] 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.

[53] 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, Cancer Genome Atlas Research Network, et al. The Cancer Genome Atlas pan-cancer analysis project. *Nature Genetics*, 45(10):1113–1120, 2013.

[54] D. Wells, E. Greisen, and R. Harten. FITS—a flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363, 1981.

[55] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*, page 2. ACM, 2013.

[56] D. G. York, J. Adelman, J. E. Anderson Jr, S. F. Anderson, J. Annis, N. A. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579, 2000.

[57] 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*, 2011.

[58] 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 USENIX Conference on Networked*

Systems Design and Implementation (NSDI '12), page 2. USENIX Association, 2012.

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

[60] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. Foster. MTC Envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*, pages 37–48. ACM, 2013.

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

B. SCHEMAS

Here, we present the schemas that we have used for these two systems. To clarify the schemas, we have grouped our fields into a simpler logical schema ordering, and we have also removed the Avro syntactic sugar used to ensure that all fields are nullable, which is required to enable arbitrary projections. These schemas are implemented using Avro [3], and data is stored to disk via Parquet [4]. In-memory (de-)serialization is provided via a custom wrapper around Avro's serialization framework.

B.1 Genomics Schema

The schema used for storing genomic short read data is described below:

```
record AlignmentRecord {
  /** Alignment position and quality */
  Contig contig;
  long start;
  long oldPosition;
  long end;

  /** read ID, sequence, and quality */
  string readName;
  string sequence;
  string qual;

  /** alignment details */
  string cigar;
  string oldCigar;
  int mapq;
  int basesTrimmedFromStart;
  int basesTrimmedFromEnd;
  boolean readNegativeStrand;
  boolean mateNegativeStrand;
  boolean primaryAlignment;
  boolean secondaryAlignment;
  boolean supplementaryAlignment;
  string mismatchingPositions;
  string origQual;

  /** Read status flags */
  boolean readPaired;
  boolean properPair;
  boolean readMapped;
  boolean mateMapped;
  boolean firstOfPair;
  boolean secondOfPair;
  boolean failedVendorQualityChecks;
  boolean duplicateRead;

  /** optional attributes */
  string attributes;

  /** record group metadata */
  string recordGroupName;
  string recordGroupSequencingCenter;
  string recordGroupDescription;
  long recordGroupRunDateEpoch;
  string recordGroupFlowOrder;
  string recordGroupKeySequence;
  string recordGroupLibrary;
  int recordGroupPredictedMedianInsertSize;
  string recordGroupPlatform;
  string recordGroupPlatformUnit;
  string recordGroupSample;

  /** Mate pair alignment information */
  long mateAlignmentStart;
  long mateAlignmentEnd;
  Contig mateContig;
}
```

All of the metadata from the sequencing run and prior processing steps are packed into the record group metadata fields. The program information describes the processing lineage of the sample and is expected to be uniform across all records, thus it compresses extremely well. The record group information is not guaranteed to be uniform across all records, but there are a limited number number of record groups per sequencing dataset. This metadata is string heavy, which makes proper deserialization from disk important. Although the information consumes less than 5% of space on disk, a poor deserializer implementation may replicate a string per field per record.

We have defined common projections and predicates to operate on these records. For example, tools that perform quality control for sequenced data commonly only access the read status flags. Additionally, it is common to run predicates on the read position, or whether the read is mapped or not. We have implemented code that allows us to apply these predicates to legacy datasets that do not support direct predicate pushdown. On legacy data, we only get the functionality of the predicate, not the performance improvement.

B.2 Astronomy Schema

We use the following schema for storing astronomy pixel values:

```
record PixelValue {
  /** pixel position */
  int xPos;
  int yPos;

  /** pixel value */
  float value;

  /** file metadata */
  int start;
  int end;
  int offset;
  int height;
}
```

This schema is derived from the legacy Flexible Image Transport System (FITS, [54]), which defines an interchange

Algorithm 3 Find Convex Hulls in Parallel

```
  $data \leftarrow$ input dataset
  $regions \leftarrow data.\text{map}(data \Rightarrow \text{generateTarget}(data))$
  $regions \leftarrow regions.\text{sort}()$
  $hulls \leftarrow regions.\text{fold}(r_1, r_2 \Rightarrow \text{mergeTargetSets}(r_1, r_2))$
  **return** $hulls$
```

format for astronomy images. During the `mAdd` processing kernel described in §6.2, we access the file metadata from each pixel. In current systems, metadata access becomes a significant performance bottleneck as we are performing metadata access across thousands of files [60].

C. CONVEX-HULL FINDING

A frequent pattern in our application is identifying the maximal convex hulls across sets of regions. For a set R of regions, we define a maximal convex hull as the largest region $\hat{r}$ that satisfies the following properties:

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{3}$$

$$\hat{R} \subset R \tag{4}$$

In our problem, we seek to find all of the maximal convex hulls, given a set of regions. For genomics, the convexity constraint described by (2) is trivial to check: specifically, the genome is assembled out of reference sequences that define disparate 1-D coordinate spaces. If two regions exist on different sequences, they are known not to overlap. If two regions are from a single single, we simply check to see if they overlap in that 1-D coordinate plane. We define the data-parallel Algorithm 3 to find the maximal convex hulls that describe a genomic dataset.

The `generateTarget` function projects each datapoint into a Red-Black tree which contains a single region. The performance of the fold depends on the efficiency of the merge function. We achieve efficient merges with the tail-call recursive `mergeTargetSets` function which is described in algorithm 4.

For a region join (see §5.1), we can use the maximal convex hull set to define partitioning for the join. Alternatively, for INDEL realignment (see §4.1), we use this set as an index for mapping reads directly to targets.

D. AVAILABILITY

The source code of both `ADAM` and `SparkMontage`, and our schemas are released under the Apache 2 license. `ADAM` is available at `https://www.github.com/bigdatagenomics/adam`, and `SparkMontage` is at `https://www.github.com/zhaozhang/SparkMontage`. `ADAM`'s schemas are available at `https://www.github.com/bigdatagenomics/bdg-formats`. `ADAM` is released via the Apache Maven Central repository, with the following dependencies:

```
<dependency>
  <groupId>org.bdgenomics.bdg-formats</groupId>
  <artifactId>bdg-formats</artifactId>
</dependency>
<dependency>
  <groupId>org.bdgenomics.adam</groupId>
  <artifactId>adam-distribution</artifactId>
</dependency>
```

As of the final submission of this paper, the latest releases of `ADAM` and `bdg-formats` were 0.16.0 and 0.4.0.

E. EXPERIMENTAL SETUP

The datasets and scripts used in the genomics experiments in this paper are all freely available online and can be used to reproduce our analyses. We made use of the `NA12878` and `HG00096` datasets from the 1000 Genome project when running our experiments [46]. These two datasets are available from either of the 1000 Genome project anonymous FTP servers (`ftp.1000genomes.ebi.ac.uk/vol1/ftp/` or `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp`), and are also hosted publicly on Amazon S3 (`s3://1000genomes`). These datasets reside at the following locations on those servers:

- `NA12878`: `data/NA12878/high_coverage_alignment/`
- `HG00096`: `data/HG00096/alignment/`

Our experiments were run on Amazon EC2 using either default or publicly available machine images. Scripts used to run the experiments are available under the Apache 2 license from the `bdg-recipes` repository, at `https://www.github.com/bigdatagenomics/bdg-recipes`. Our experimental scripts assemble `ADAM` release 0.16.0, `GATK` release 3.3 [14], `Sambamba`'s docker image [50], `SAMBLASTER` 0.1.21 [17], and `Picard` `b2a94f7` [51]. These were the latest versions of these tools at the time of final submission. These evaluation scripts run and time the experiments listed in this paper.

Algorithm 4 Merge Hull Sets

```
  $first \leftarrow$ first target set to merge
  $second \leftarrow$ second target set to merge
**Require:** $first$ and $second$ are sorted
  **if** $first = \emptyset \wedge second = \emptyset$ **then**
    **return** $\emptyset$
  **else if** $first = \emptyset$ **then**
    **return** $second$
  **else if** $second = \emptyset$ **then**
    **return** $first$
  **else**
    **if** $\text{last}(first) \cap \text{head}(second) = \emptyset$ **then**
      **return** $first + second$
    **else**
      $mergeItem \leftarrow (\text{last}(first) \cup \text{head}(second))$
      $mergeSet \leftarrow \text{allButLast}(first) \cup mergeItem$
      $trimSecond \leftarrow \text{allButFirst}(second)$
      **return** $\text{mergeTargetSets}(mergeSet, trimSecond)$
    **end if**
  **end if**
```