ADAM Design Specification

Matt Massie

December 18, 2014

Contents

Introduction

ADAM is a genome analysis platform built using Apache Hadoop, Spark, Parquet and Avro. All software that ADAM requires is licensed under Apache 2.0 and ADAM itself is shared with a permissive Apache 2.0 license.

These systems provide many benefits when used together.

- Spark allows developers to focus on algorithms without needing to worry about distributed systems failures, data locality, on-disk formats, etc.,
- Spark enable jobs to be run locally on a single machine, on an in-house cluster or in the cloud *without modifications*,

- Parquet compresss legacy genomic formats using standard columnar techniques (e.g. RLE, dictionary encoding) and provide predicate pushdown and projection for performance,
- Avro and Parquet provide an agile way of customizing and evolving data formats over time,
- Spark and Hadoop are designed to scale linearly on commodity hardware,
- All systems are shared with a standard Apache 2.0 license.

Here's a sample snippet of code to count k-mers using ADAM, e.g.

```
// Load reads from 'inputPath' into an RDD for analysis
val adamRecords: RDD[AlignmentRecord] = sc.loadAlignments(args.inputPath,
  // Filter out all low quality reads that failed vendor quality checks
  predicate = Some(classOf[HighQualityReadsPredicate]),
  // Only materialize the 'sequence' from each record
  projection = Some(Projection(AlignmentRecordField.sequence)))

// The length of kmers we want to count
val kmerLength = 21

// Process the reads into an RDD of tuples with kmers and counts
val kmers: RDD[(String, Long)] = adamRecords.flatMap(read => {
  read.getSequence
    .toString
    .sliding(kmerLength)
    .map(k => (k, 1L))
}).reduceByKey { case (a, b) => a + b}

// Print the kmers as a text file to the 'outputPath'
kmers.map { case (kmer, count) => s"$count,$kmer"}
  .saveAsTextFile(args.outputPath)
```

When run on sample `NA21144`, chromosome 11 from the 1000 Genomes project, this job outputs:

```
AAAAAAAAAAAAAAAAAAAAAA, 124069
TTTTTTTTTTTTTTTTTTTTTT, 120590
ACACACACACACACACACACAC, 41528
GTGTGTGTGTGTGTGTGTGTGT, 40905
CACACACACACACACACACACA, 40795
TGTGTGTGTGTGTGTGTGTGTG, 40329
TAATCCCAGCACTTTGGGAGGC, 32122
TGTAATCCCAGCACTTTGGGAG, 31206
CTGTAATCCCAGCACTTTGGGA, 30809
GCCTCCCAAAGTGCTGGGATTA, 30716
```

```
TCCCAAAGTGCTGGGATTACAG, 30420
GTAATCCCAGCACTTTGGGAGG, 30253
CTCCCAAAGTGCTGGGATTACA, 30184
CCTGTAATCCCAGCACTTTGGG, 29157
...
```

This snippet provides a simple example which highlights the features high-level features of ADAM. ADAM implicitly converts a Spark context, `sc` in this case, into an `ADAMContext` which include the `adamLoad` method which takes an optional Parquet predicate and projection. The `inputPath` can be a single file or a file set.

The `HighQualityReadsPredicate` predicate (not shown) is a simple class is passed a record and returns a `boolean`. If the return value is `false`, the record is ignored. Only the record fields needed by the predicate are materialized for performance, e.g. if the predicate looked at aligned read flags, only the flags are materialized.

The projection defines the exact fields that the analysis needs to load into an `RDD`. In the case of this example, we only need the `sequence` from each record to count kmers. Parquet will only deserialize the sequence data and quickly skip over all other fields. Serialization/deserialization is an expense step so this dramatically helps performance and reduced the size of records in memory.

In Spark, an `RDD` is a resilient distributed dataset. You can think of it as an array of records distributed across your entire cluster. An `RDD` has a number of operations, e.g. `flatMap`, `reduceByKey`, which return a new `RDD` with new records. In this example, the `adamRecords` `RDD` is converted into the `kmers` `RDD` comprised of a tuple with each k-mer and the number of times it was seen.

Finally, an `RDD` has operators for saving records from an `RDD` back to an underlying file system e.g. `s3`, `HDFS` or local.

Once compiled, this code can be run on a laptop, a cluster of machines on Amazon's EC2, Google's Cloud or Microsoft Azure without modification. It can also read from a variety of file systems, e.g. `file://`, `hdfs://`, `s3://`, `ftp://`, without modification. Spark can load data from any data source which provides a Hadoop `InputFormat` which makes integration with legacy or custom storage systems possible.

Assumptions/ Constraints/ Standards

Parquet data can be accessed by Java and C. Since ADAM is written in Scala, we haven't done much testing of the C support. However, the C library used by Cloudera to read Parquet into Impala is open-source and broadly used and mature. We assume that multi-language support will be straight-forward but that is just an assumption.

There is also an assumption that the people deploying ADAM are experienced at devops and that the people writing new methods are experienced with big data systems. Over time, we hope to provide better documentation and utilities to help less experienced users. Misconfiguration is a problem with distributed systems in general and ADAM is not an exception, e.g. block size mismatch between `HDFS` filesystem and `Parquet` files increases data movement at shuffle.

System Design

The ADAM design emphasizes separation of concerns and clean interfaces between layers allowing users maximum flexibility at deployment. For example, ADAM supports both Hadoop 1.x and Hadoop 2.x and runs on Amazon, Google and Microsoft clouds without needing to modify code. The physical and block layers can be changed without effecting the layers above.

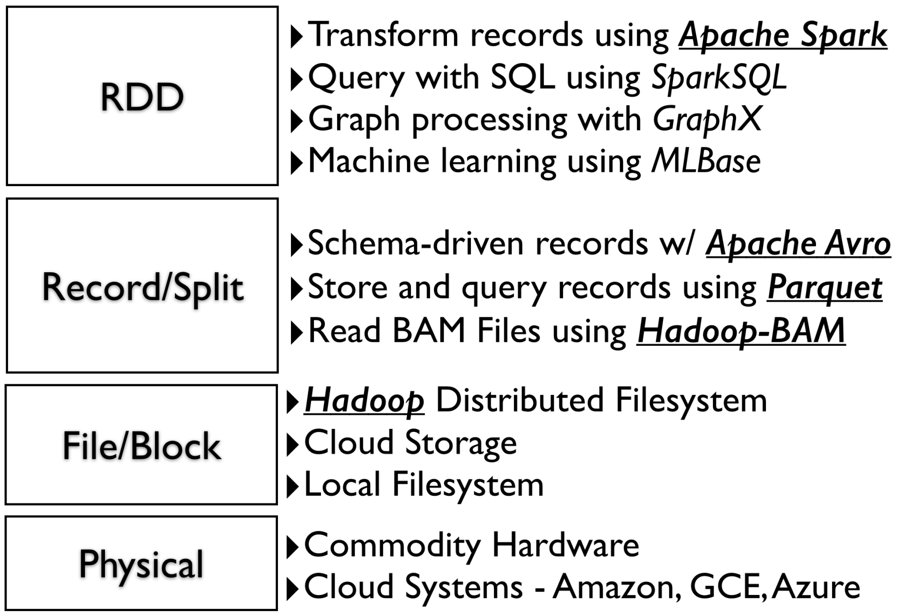

RDD
Transform records using Apache Spark
Query with SQL using SparkSQL
Graph processing with GraphX
Machine learning using MLBase
Record/Split
Schema-driven records w/ Apache Avro
Store and query records using Parquet
Read BAM Files using Hadoop-BAM
File/Block
Hadoop Distributed Filesystem
Cloud Storage
Local Filesystem
Physical
Commodity Hardware
Cloud Systems - Amazon, GCE, Azure

Figure 1: The Layered Designed of ADAM

The system is designed to use standard open-source interfaces that exist in the Hadoop ecosystem, e.g. Hadoop `InputFormat`, Hadoop `OutputFormat`, Spark `RDD`, as well as open data standards such a Parquet and Avro. This design prevents "lock-in" and allows the system to be deployed on any cloud vendor. We publish the schemas for all ADAM objects on github in Avro IDL format. This interface description language (IDL) can generate code for a number of different programming languages. Avro and Parquet also provide features that make it easy to change schemas while maintaining backward compatibility.

This approach has been adopted by The Global Alliance for Genomics and Health. All biological data and methods are described in Avro IDL at in the schemas repository of the GA4GH github account.

Parquet was chosen as a storage format because it allows access to the data *outside* of ADAM. Additionally, Parquet is simply a file format which makes it easy to use utilities like `rsync` and `distcp` in Hadoop to synchronize data between filesystems. Once ADAM has converted legacy data into Parquet, it is available to any system that can read from a Hadoop `InputFormat` since Parquet provides a `ParquetInputFormat` implementation. It should be noted that an `InputFormat` is simply an iterator pattern for fetching records from a data source. Parquet also provides code customized for systems like Apache Pig, Hive, Scrooge, etc which provides feature beyond iterating over records.

This openness allows users to access ADAM data as a graph, a SQL table (using e.g. Impala or SparkSQL), an `RDD` of in-memory records and use the built-in machine learning in Spark, `MLlib`. Users are not locked into a specific framework for analysis and can use their favorite environment for processing ADAM data.

The Google Dataflow JavaSDK is an implementation of the `FlumeJava` API which currently only runs on data in the Google Cloud. Any alternative Apache-licensed implementation of `FlumeJava` is Apache Crunch which supports building workflows in-memory (on a single node), on Hadoop MapReduce or Apache Spark. It would be possible to run ADAM as part of an Apache Crunch workflow.

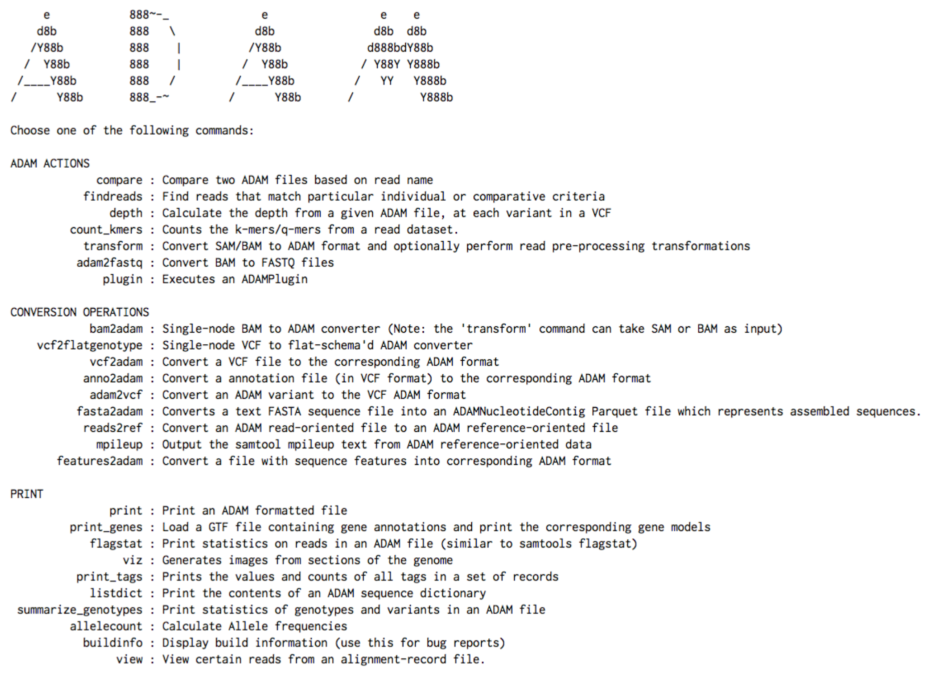
There are also more details about the ADAM design in the Berkeley tech report ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing.

Use-Cases

ADAM provides a platform for biological data analysis as well as a `CLI` for basic operations (see Figure 2 below). There are commands for comparing files, finding reads, calculating depth, converting data, executing ADAM plugins, printing statistics, generating k-mer and q-mer counts, calculating allele counts, visualizing reads via an embedded genome browser, etc. The `transform` command does more than just convert from BAM to ADAM but also includes sorting, mark duplicates, base quality score recalibration and indel realignment pre-processing methods to convert aligned reads into analysis-ready reads. The `transform` command is Extract-Transform-Load (ETL) component of ADAM.

There are other projects that build on ADAM to provide more functionality.

- RNAdam provides an RNA pipeline on top of ADAM with isoform quantification and fusion transcription detection,



```
     e            888~-_              e                 e    e
    d8b           888   \           d8b                d8b  d8b
   /Y88b          888    |         /Y88b              d888bdY88b
  /  Y88b         888    |        /  Y88b           / Y88Y Y888b
 /____Y88b        888   /        /____Y88b        /   YY   Y888b
/      Y88b       888_-~         /      Y88b      /          Y888b

Choose one of the following commands:

ADAM ACTIONS
             compare : Compare two ADAM files based on read name
           findreads : Find reads that match particular individual or comparative criteria
               depth : Calculate the depth from a given ADAM file, at each variant in a VCF
         count_kmers : Counts the k-mers/q-mers from a read dataset.
           transform : Convert SAM/BAM to ADAM format and optionally perform read pre-processing transformations
          adam2fastq : Convert BAM to FASTQ files
              plugin : Executes an ADAMPlugin

CONVERSION OPERATIONS
            bam2adam : Single-node BAM to ADAM converter (Note: the 'transform' command can take SAM or BAM as input)
     vcf2flatgenotype : Single-node VCF to flat-schema'd ADAM converter
            vcf2adam : Convert a VCF file to the corresponding ADAM format
           anno2adam : Convert a annotation file (in VCF format) to the corresponding ADAM format
            adam2vcf : Convert an ADAM variant to the VCF ADAM format
         fasta2adam : Converts a text FASTA sequence file into an ADAMNucleotideContig Parquet file which represents assembled sequences.
           reads2ref : Convert an ADAM read-oriented file to an ADAM reference-oriented file
             mpileup : Output the samtool mpileup text from ADAM reference-oriented data
       features2adam : Convert a file with sequence features into corresponding ADAM format

PRINT
               print : Print an ADAM formatted file
         print_genes : Load a GTF file containing gene annotations and print the corresponding gene models
            flagstat : Print statistics on reads in an ADAM file (similar to samtools flagstat)
                 viz : Generates images from sections of the genome
          print_tags : Prints the values and counts of all tags in a set of records
            listdict : Print the contents of an ADAM sequence dictionary
 summarize_genotypes : Print statistics of genotypes and variants in an ADAM file
         allelecount : Calculate Allele frequencies
           buildinfo : Display build information (use this for bug reports)
                view : View certain reads from an alignment-record file.
```

Figure 2: The ADAM CLI

- Avocado is a variant caller built on top of ADAM for germline and somatic calling,
- PacMin is an assembler for PacBio reads,
- A `Mutect` port is nearly complete,
- Read error correction,
- Short read assembly
- Variant filtration (train model via `MLlib`)

All these project are shared using an Apache 2.0 licensed.

ADAM has also used for *k*-means for population stratification.

Database Design

ADAM uses Apache Avro and Parquet for serialization, storage and querying of data. All ADAM objects are explicitly defined on github in Avro IDL. The ADAM build system, Maven, uses the `avro-maven-plugin` to automatically generate Java code for the defined objects. Avro can also generate code in C++, Python, Ruby, etc.

Parquet integrates well with Avro. It converts the Avro schema into Parquet schema to store Avro objects inside of the Parquet columnar container file. Alex Holmes has a great blog post with the details of how this integration works for Avro, Thrift and Protocol Buffers.

Parquet support pushdown predicates and schema projection to allow developers to only materialize the data they need, since (de)serialization is expensive.

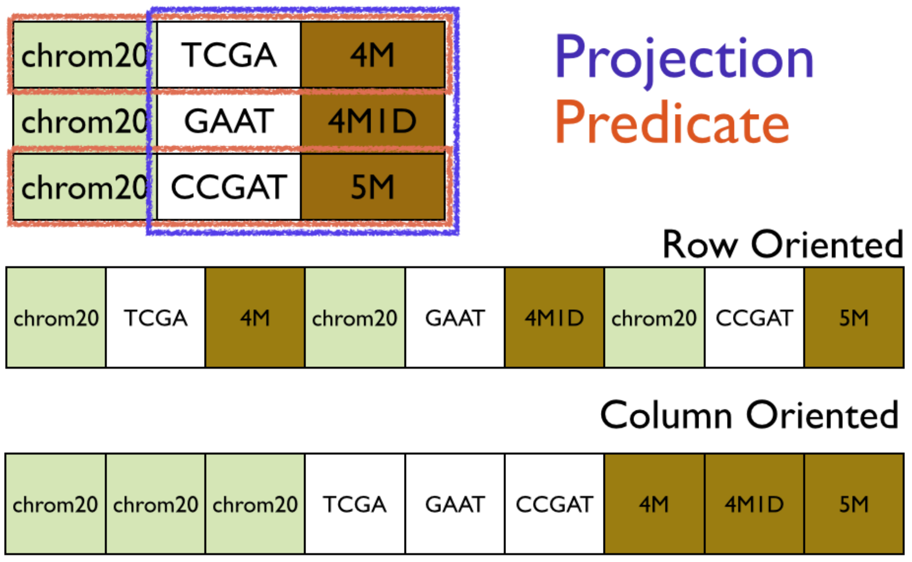


Figure 3: Predicates and Projection in Parquet

In this image, you can see that predicates allow you to choose specific *rows* of a table. Note, the predicates will only deserialize the fields in the row needed for the predicate. If the predicate passes (returns `true`), then Parquet will materialize the record according the projection in effect.

A projection is can be all fields or a limited number of fields in each record. In the image above, the projection is for the `sequence` and `cigar` fields and the `referenceName` is ignored. Of the nine cells in the image, the predicate and projection together only materialize the four cells.

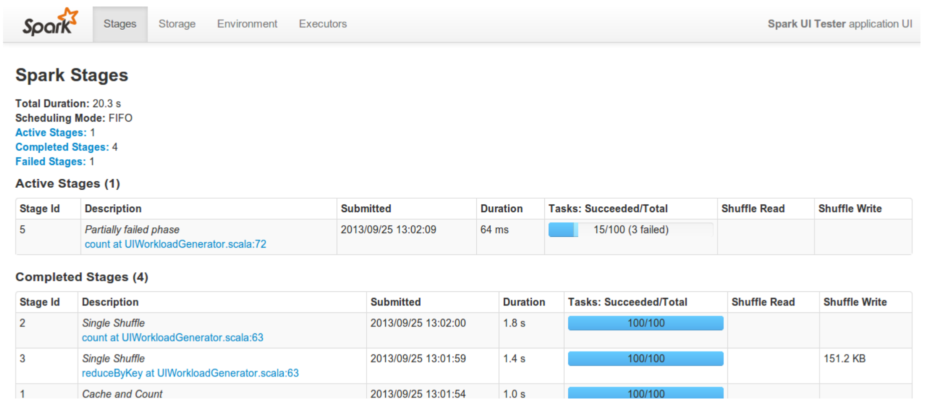
Parquet is column-oriented meaning that each field is storage in a contiguous area on disk. As you can see from the image above, this approach keeps similar data (e.g. `sequence` string) together which improves performance and compression. ADAM files are about 20% smaller than compressed BAM files with no loss of data. New columnar compression techniques coming in Parquet 2.0 will likely dramatically improve compression of quality scores which take about 70% of on-disk space.

Application Program Interfaces

ADAM uses the standard `InputFormat`, `OutputFormat` and `RDD` interfaces that are widely used in the Hadoop ecosystem.

User Interface Design

ADAM jobs can be monitored and killed using the Spark UI which by default run on port `4040` on the Spark master machine, e.g.



Spark Stages

Total Duration: 20.3 s
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 4
Failed Stages: 1

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
5	*Partially failed phase* count at UIWorkloadGenerator.scala:72	2013/09/25 13:02:09	64 ms	15/100 (3 failed)		

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
2	*Single Shuffle* count at UIWorkloadGenerator.scala:63	2013/09/25 13:02:00	1.8 s	100/100		
3	*Single Shuffle* reduceByKey at UIWorkloadGenerator.scala:63	2013/09/25 13:01:59	1.4 s	100/100		151.2 KB
1	*Cache and Count*	2013/09/25 13:01:54	1.0 s	100/100		

Figure 4: Spark UI

This UI allows users to track the progress and performance of jobs in flight including the exact line of code being run, shuffle sizes, environment variables, executor memory, etc.

Spark Notebook provides an iPython/Project Jupyter-like, user interface to creating, launching and visualizing ADAM workflows. This workflows can be saved and shared between users. The Spark Notebook `README` provides example of the visualization and how to integrate with ADAM.

Performance

ADAM is built on Hadoop, Spark, Parquet and Avro. ADAM performance benefits immediately from updates to those underlying technologies. For example, Databricks, a Spark startup spun out the AMPLab, recently announced that it crushed the Daytona GraySort record achieving a **3x** boost in performance using **10x** less machines. Moreover, Databricks used Amazon EC2 instead of bare metals machines which were used for all previous GraySort records. All the performance improvements they added to win this benchmark were folded back into the open-source implementation of Spark to benefit the community (and ADAM).

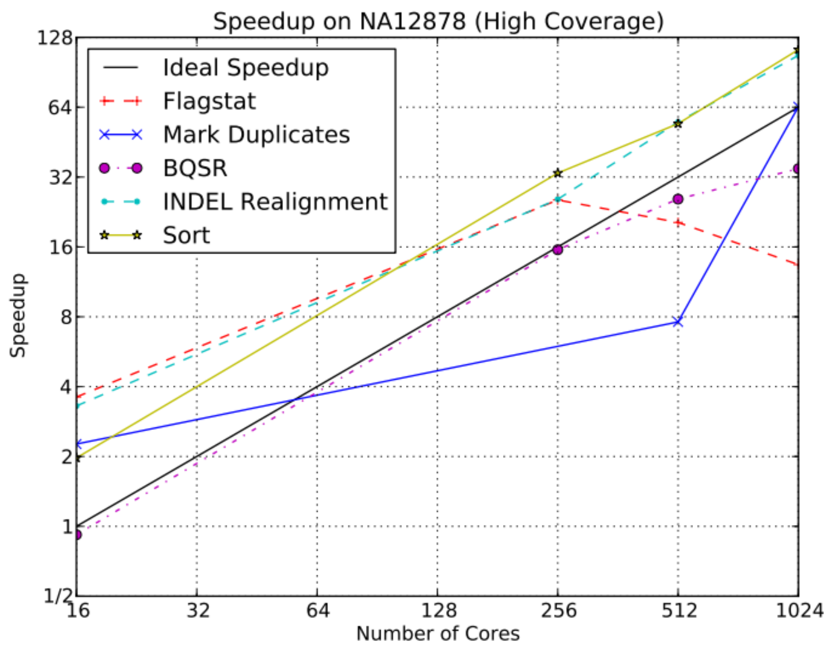


Figure 5: Speedup as a function of cores

Figure 5 demonstrates how the ADAM design benefits from the horizontal scalability of Spark.

For more details about the ADAM performance, see the Berkeley tech report ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing.

It should be noted that performance is influenced by the underlying block/physical layers. While ADAM can integrate with any data source, the performance will vary especially when processing is not data local.

Technologies to Evaluate and Criteria

1. Evaluate Apache Crunch as a workflow for ADAM to allows users to run on any cloud vendor, in-house or on a single machine.
2. Create a Hadoop `InputFormat` for some of the current data source at The Broad Institute to demonstrate ease of integration. ADAM can read from any data source that provides an `InputFormat`.