bdgenomics.workflows Documentation

Release 0.23.0-SNAPSHOT

Big Data Genomics

Jan 04, 2018

Architecture

CHAPTER 1

Introduction

ADAM is a library and command line tool that enables the use of Apache Spark to parallelize genomic data analysis across cluster/cloud computing environments. ADAM uses a set of schemas to describe genomic sequences, reads, variants/genotypes, and features, and can be used with data in legacy genomic file formats such as SAM/BAM/CRAM, BED/GFF3/GTF, and VCF, as well as data stored in the columnar Apache Parquet format. On a single node, ADAM provides competitive performance to optimized multi-threaded tools, while enabling scale out to clusters with more than a thousand cores. ADAM's APIs can be used from Scala, Java, Python, R, and SQL.

1.1 The ADAM/Big Data Genomics Ecosystem

ADAM builds upon the open source Apache Spark, Apache Avro, and Apache Parquet projects. Additionally, ADAM can be deployed for both interactive and production workflows using a variety of platforms. A diagram of the ecosystem of tools and libraries that ADAM builds on and the tools that build upon the ADAM APIs can be found below.

As the diagram shows, beyond the *ADAM CLI*, there are a number of tools built using ADAM's core APIs:

- Avocado is a variant caller built on top of ADAM for germline and somatic calling
- Cannoli uses ADAM's *pipe* API to parallelize common single-node genomics tools (e.g., BWA, bowtie2, FreeBayes)
- DECA is a reimplementation of the XHMM copy number variant caller on top of ADAM/Apache Spark
- Gnocchi provides primitives for running GWAS/eQTL tests on large genotype/phenotype datasets using ADAM
- Lime provides a parallel implementation of genomic set theoretic primitives using the *region join API*
- Mango is a library for visualizing large scale genomics data with interactive latencies and serving data using the GA4GH schemas

1.1.1 Architecture Overview

ADAM is architected as an extensible, parallel framework for working with both aligned and unaligned genomic data using Apache Spark. Unlike traditional genomics tools, ADAM is built as a modular stack, where a set of schemas

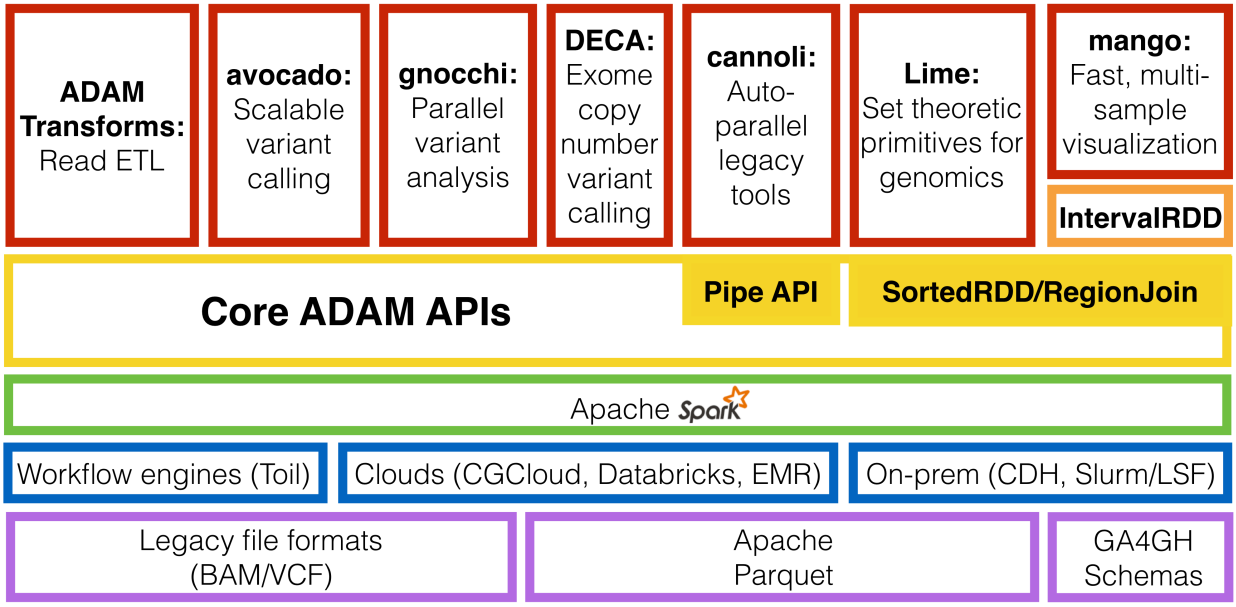

Fig. 1.1: The ADAM ecosystem.

define and provide a narrow waist. This stack architecture enables the support of a wide range of data formats and optimized query patterns without changing the data structures and query patterns that users are programming against.

ADAM's architecture was introduced as a response to the challenges processing the growing volume of genomic sequencing data in a reasonable timeframe (Schadt et al. 2010). While the per-run latency of current genomic pipelines such as the GATK could be improved by manually partitioning the input dataset and distributing work, native support for distributed computing was not provided. As a stopgap solution, projects like Cloudburst (Schatz 2009) and Crossbow (Langmead et al. 2009) have ported individual analytics tools to run on top of Hadoop. While this approach has served well for proofs of concept, this approach provides poor abstractions for application developers. These poor abstractions make it difficult for bioinformatics developers to create novel distributed genomic analyses, and does little to attack sources of inefficiency or incorrectness in distributed genomics pipelines.

ADAM's architecture reconsiders how we build software for processing genomic data by eliminating the monolithic architectures that are driven by the underlying flat file formats used in genomics. These architectures impose significant restrictions, including:

- These implementations are locked to a single node processing model. Even the GATK's "map-reduce" styled walker API (McKenna et al. 2010) is limited to natively support processing on a single node. While these jobs can be manually partitioned and run in a distributed setting, manual partitioning can lead to imbalance in work distribution and makes it difficult to run algorithms that require aggregating data across all partitions, and lacks the fault tolerance provided by modern distributed systems such as Apache Hadoop or Spark (Zaharia et al. 2012).
- Most of these implementations assume invariants about the sorted order of records on disk. This "stack smashing" (specifically, the layout of data is used to accelerate a processing stage) can lead to bugs when data does not cleanly map to the assumed sort order. Additionally, since these sort order invariants are rarely explicit and vary from tool to tool, pipelines assembled from disparate tools can be brittle.
- Additionally, while these invariants are intended to improve performance, they do this at the cost of opacity. If we can express the query patterns that are accelerated by these invariants at a higher level, then we can achieve both a better programming environment and enable various query optimizations.

At the core of ADAM, users use the *ADAMContext* to load data as *GenomicRDDs*, which they can then manipulate. In the GenomicRDD class hierarchy, we provide several classes that contain functionality that is applicable to all genomic datatypes, such as *coordinate-space joins*, the *pipe* API, and genomic metadata management.

1.1.2 The ADAM Stack Model

The stack model that ADAM is based upon was introduced in (Massie et al. 2013) and further refined in (F. A. Nothaft et al. 2015), and is depicted in the figure below. This stack model separates computational patterns from the data model, and the data model from the serialized representation of the data on disk. This enables developers to write queries that run seamlessly on a single node or on a distributed cluster, on legacy genomics data files or on data stored in a high performance columnar storage format, and on sorted or unsorted data, without making any modifications to their query. Additionally, this allows developers to write at a higher level of abstraction without sacrificing performance, since we have the freedom to change the implementation of a layer of the stack in order to improve the performance of a given query.

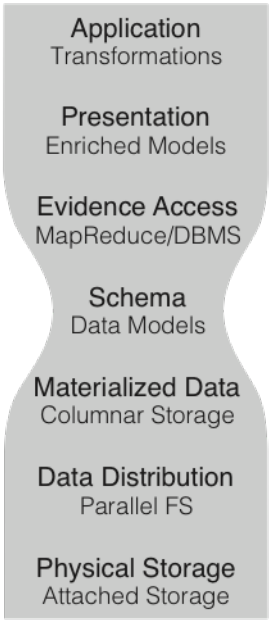

Fig. 1.2: The ADAM Stack Model

This stack model is divided into seven levels. Starting from the bottom, these are:

1. The *physical storage* layer is the type of storage media (e.g., hard disk/solid state drives) that are used to store the data.
2. The *data distribution* layer determines how data are made accessible to all of the nodes in the cluster. Data may be made accessible through a POSIX-compliant shared file system such as NFS (Sandberg et al. 1985), a non-POSIX file system that implements Hadoop's APIs (e.g., HDFS), or through a block-store, such as Amazon S3.
3. The *materialized data* layer defines how the logical data in a single record maps to bytes on disk. We advocate for the use of Apache Parquet, a high performance columnar storage format based off of Google's Dremel database (Melnik et al. 2010). To support conventional genomics file formats, we exchange the Parquet implementation of this layer with a layer that maps the given schema into a traditional genomics file format (e.g., SAM/BAM/CRAM, BED/GTF/GFF/NarrowPeak, VCF).
4. The *schema* layer defines the logical representation of a datatype.

5. The *evidence access* layer is the software layer that implements and executes queries against data stored on disk. While ADAM was originally built around Apache Spark's Resilient Distributed Dataset (RDD) API (Zaharia et al. 2012), ADAM has recently enabled the use of Spark SQL (Armbrust et al. 2015) for evidence access and query.
6. The *presentation* layer provides high level abstractions for interacting with a parallel collection of genomic data. In ADAM, we implement this layer through the *GenomicRDD* classes. This layer presents users with a view of the metadata associated with a collection of genomic data, and APIs for *transforming* and *joining* genomic data. Additionally, this is the layer where we provide cross-language support.
7. The *application* layer is the layer where a user writes their application on top of the provided APIs.

Our stack model derives its inspiration from the layered Open Systems Interconnection (OSI) networking stack (Zimmermann 1980), which eventually served as the inspiration for the Internet Protocol stack, and from the concept of data independence in database systems. We see this approach as an alternative to the "stack smashing" commonly seen in genomics APIs, such as the GATK's "walker" interface (McKenna et al. 2010). In these APIs, implementation details about the layout of data on disk (are the data sorted?) are propagated up to the application layer of the API and exposed to user code. This limits the sorts of queries that can be expressed efficiently to queries that can easily be run against linear sorted data. Additionally, these "smashed stacks" typically expose very low level APIs, such as a sorted iterator, which increases the cost to implementing a query and can lead to trivial mistakes.

1.1.3 The bdg-formats schemas

The schemas that comprise ADAM's narrow waist are defined in the bdg-formats project, using the Apache Avro schema description language. This schema definition language automatically generates implementations of this schema for multiple common languages, including Java, C, C++, and Python. bdg-formats contains several core schemas:

- The *AlignmentRecord* schema represents a genomic read, along with that read's alignment to a reference genome, if available.
- The *Feature* schema represents a generic genomic feature. This record can be used to tag a region of the genome with an annotation, such as coverage observed over that region, or the coordinates of an exon.
- The *Fragment* schema represents a set of read alignments that came from a single sequenced fragment.
- The *Genotype* schema represents a genotype call, along with annotations about the quality/read support of the called genotype.
- The *NucleotideContigFragment* schema represents a section of a contig's sequence.
- The *Variant* schema represents a sequence variant, along with statistics about that variant's support across a group of samples, and annotations about the effect of the variant.

The bdg-formats schemas are designed so that common fields are easy to query, while maintaining extensibility and the ability to interoperate with common genomics file formats. Where necessary, the bdg-formats schemas are nested, which allows for the description of complex nested features and groupings (such as the Fragment record, which groups together AlignmentRecords). All fields in the bdg-formats schemas are nullable, and the schemas themselves do not contain invariants around valid values for a field. Instead, we validate data on ingress and egress to/from a conventional genomic file format. This allows users to take advantage of features such as field projection, which can improve the performance of queries like *flagstat* by an order of magnitude.

1.1.4 Interacting with data through ADAM's evidence access layer

ADAM exposes access to distributed datasets of genomic data through the *ADAMContext* entrypoint. The ADAMContext wraps Apache Spark's SparkContext, which tracks the configuration and state of the current running Spark

application. On top of the SparkContext, the ADAMContext provides data loading functions which yield *GenomicRDD*s. The GenomicRDD classes provide a wrapper around Apache Spark's two APIs for manipulating distributed datasets: the legacy Resilient Distributed Dataset (Zaharia et al. 2012) and the new Spark SQL Dataset/DataFrame API (Armbrust et al. 2015). Additionally, the GenomicRDD is enriched with genomics-specific metadata such as computational lineage and sample metadata, and optimized genomics-specific query patterns such as *region joins* and the *auto-parallelizing pipe API* for running legacy tools using Apache Spark.

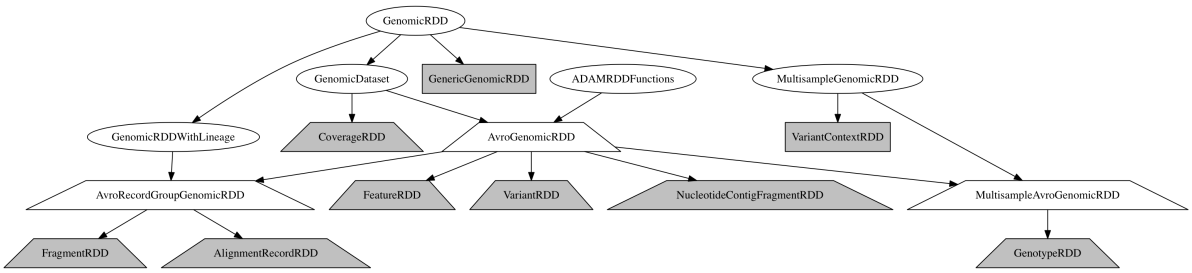

Fig. 1.3: The GenomicRDD Class Hierarchy

All GenomicRDDs include a sequence dictionary which describes the reference genome that the data in the RDD are aligned to, if one is known. Additionally, RecordGroupGenomicRDD store a dictionary with read groups that are attached to the reads/fragments. Similarly, the MultisampleGenomicRDD includes a list of samples who are present in the dataset.

1.1.5 Building ADAM from Source

You will need to have Apache Maven version 3.1.1 or later installed in order to build ADAM.

> **Note:** The default configuration is for Hadoop 2.7.3. If building against a different version of Hadoop, please pass `-Dhadoop.version=<HADOOP_VERSION>` to the Maven command. ADAM will cross-build for both Spark 1.x and 2.x, but builds by default against Spark 1.6.3. To build for Spark 2, run the `./scripts/move_to_spark2.sh` script.

```
git clone https://github.com/bigdatagenomics/adam.git
cd adam
export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=128m"
mvn clean package -DskipTests
```

Outputs

```
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.647s
[INFO] Finished at: Thu May 23 15:50:42 PDT 2013
[INFO] Final Memory: 19M/81M
[INFO] ------------------------------------------------------------------------
```

You might want to take a peek at the `scripts/jenkins-test` script and give it a run. It will fetch a mouse chromosome, encode it to ADAM reads and pileups, run flagstat, etc. We use this script to test that ADAM is working correctly.

Running ADAM

ADAM is packaged as an überjar and includes all necessary dependencies, except for Apache Hadoop and Apache Spark.

You might want to add the following to your `.bashrc` to make running ADAM easier:

```
alias adam-submit="${ADAM_HOME}/bin/adam-submit"
alias adam-shell="${ADAM_HOME}/bin/adam-shell"
```

`$ADAM_HOME` should be the path to where you have checked ADAM out on your local filesystem. The first alias should be used for running ADAM jobs that operate locally. The latter two aliases call scripts that wrap the `spark-submit` and `spark-shell` commands to set up ADAM. You will need to have the Spark binaries on your system; prebuilt binaries can be downloaded from the Spark website. Our continuous integration setup builds ADAM against Spark versions 1.6.1 and 2.0.0, Scala versions 2.10 and 2.11, and Hadoop versions 2.3.0 and 2.6.0.

Once this alias is in place, you can run ADAM by simply typing `adam-submit` at the command line.

```
adam-submit
```

Building for Python

ADAM can be installed using the Pip package manager, or from source. To build and test *ADAM's Python bindings*, enable the `python` profile:

```
mvn -Ppython package
```

This will enable the `adam-python` module as part of the ADAM build. This module uses Maven to invoke a Makefile that builds a Python egg and runs tests. To build this module, we require either an active Conda or virtualenv environment.

To setup and activate a Conda environment, run:

```
conda create -n adam python=2.7 anaconda
source activate adam
```

To setup and activate a virtualenv environment, run:

```
virtualenv adam
. adam/bin/activate
```

Additionally, to run tests, the PySpark dependencies must be on the Python module load path and the ADAM JARs must be built and provided to PySpark. This can be done with the following bash commands:

```
# add pyspark to the python path
PY4J_ZIP="$(ls -1 "${SPARK_HOME}/python/lib" | grep py4j)"
export PYTHONPATH=${SPARK_HOME}/python:${SPARK_HOME}/python/lib/${PY4J_ZIP}:$
↪{PYTHONPATH}

# put adam jar on the pyspark path
ASSEMBLY_DIR="${ADAM_HOME}/adam-assembly/target"
ASSEMBLY_JAR="$(ls -1 "$ASSEMBLY_DIR" | grep "^adam[0-9A-Za-z\.\_-]*\.jar$" | grep -v␣
↪-e javadoc -e sources || true)"
export PYSPARK_SUBMIT_ARGS="--jars ${ASSEMBLY_DIR}/${ASSEMBLY_JAR} --driver-class-
↪path ${ASSEMBLY_DIR}/${ASSEMBLY_JAR} pyspark-shell"
```

This assumes that the *ADAM JARs have already been built*. Additionally, we require pytest to be installed. The adam-python makefile can install this dependency. Once you have an active virtualenv or Conda environment, run:

```
cd adam-python
make prepare
```

Building for R

ADAM supports SparkR, for Spark 2.1.0 and onwards. To build and test *ADAM's R bindings*, enable the `r` profile:

```
mvn -Pr package
```

This will enable the `adam-r` module as part of the ADAM build. This module uses Maven to invoke the `R` executable to build the `bdg.adam` package and run tests. The build requires the `testthat`, `devtools` and `roxygen` packages

```
R -e "install.packages('testthat', repos='http://cran.rstudio.com/')"
R -e "install.packages('roxygen2', repos='http://cran.rstudio.com/')"
R -e "install.packages('devtools', repos='http://cran.rstudio.com/')"
```

The build also requires you to have the `SparkR` package installed, and the ADAM JARs must be built and provided to `SparkR`. This can be done with the following bash commands:

```
# put adam jar on the SparkR path
ASSEMBLY_DIR="${ADAM_HOME}/adam-assembly/target"
ASSEMBLY_JAR="$(ls -1 "$ASSEMBLY_DIR" | grep "^adam[0-9A-Za-z\_\.-]*\.jar$" | grep -v␣
↪javadoc | grep -v sources || true)"
export SPARKR_SUBMIT_ARGS="--jars ${ASSEMBLY_DIR}/${ASSEMBLY_JAR} --driver-class-path
↪${ASSEMBLY_DIR}/${ASSEMBLY_JAR} sparkr-shell"
```

Note that the `ASSEMBLY_DIR` and `ASSEMBLY_JAR` lines are the same as for the *Python build*. As with the Python build, this assumes that the *ADAM JARs have already been built*.

1.1.6 Installing ADAM using Pip

ADAM is available through the Python Package Index and thus can be installed using pip. To install ADAM using pip, run:

```
pip install bdgenomics.adam
```

Pip will install the bdgenomics.adam Python binding, as well as the ADAM CLI.

1.1.7 Running an example command

flagstat

Once you have data converted to ADAM, you can gather statistics from the ADAM file using *flagstat*. This command will output stats identically to the samtools `flagstat` command.

```
./bin/adam-submit flagstat NA12878_chr20.adam
```

Outputs:

```
51554029 + 0 in total (QC-passed reads + QC-failed reads)
0 + 0 duplicates
50849935 + 0 mapped (98.63%:0.00%)
51554029 + 0 paired in sequencing
25778679 + 0 read1
25775350 + 0 read2
49874394 + 0 properly paired (96.74%:0.00%)
50145841 + 0 with itself and mate mapped
704094 + 0 singletons (1.37%:0.00%)
158721 + 0 with mate mapped to a different chr
105812 + 0 with mate mapped to a different chr (mapQ>=5)
```

In practice, you will find that the ADAM `flagstat` command takes orders of magnitude less time than samtools to compute these statistics. For example, on a MacBook Pro, the command above took 17 seconds to run while `samtools flagstat NA12878_chr20.bam` took 55 seconds. On larger files, the difference in speed is even more dramatic. ADAM is faster because it is multi-threaded, distributed and uses a columnar storage format (with a projected schema that only materializes the read flags instead of the whole read).

Running on a cluster

We provide the `adam-submit` and `adam-shell` commands under the `bin` directory. These can be used to submit ADAM jobs to a spark cluster, or to run ADAM interactively.

1.1.8 Benchmarks

Algorithm Performance

To test the efficiency of ADAM's implementations of the various algorithms that are used in our variant calling pipeline, we ran strong scaling experiments. In this experiment, we used the high coverage genome NA12878 from the 1000 Genomes project. We held the executor configuration constant, and stepped from one to four executors before doubling until we reached 32 executors. Each executor had 16 hyperthreaded cores (32 concurrent threads) and was allocated 200 gigabytes of memory.

In general, most of the algorithms in ADAM scale linearly as the amount of compute resources is increased. The significant exception is BQSR, which is explained by the large broadcast which runs during BQSR. The broadcast introduces a serial step that runs for several minutes, as the multiple gigabyte mask table is broadcast to all of the nodes in the cluster. The figures below break these results down and compares them to the latest implementation of these algorithms in the GATK4.

As our benchmarks demonstrate, ADAM's implementations of the various algorithms in the standard germline reads-to-variants pipeline are more performant than the same algorithms implemented in the GATK. ADAM outperforms the GATK when running equivalent implementations of BQSR (1.4x speedup) and duplicate marking (1.1x speedup). To improve the performance of duplicate marking, we added a variant of the duplicate marking algorithm that operates on reads that are grouped by sequencing fragment. By optimizing for reads that are grouped with the other reads from their fragment, we can eliminate the first shuffle in duplicate marking, which reassembles reads by read-pair. This optimization is common in duplicate markers, as paired-end aligners will typically emit a SAM or BAM file where consecutive reads are from the same sequencing fragment.

1.1.9 Storage Size

ADAM uses Apache Parquet as a way to store genomic data. This is in addition to our support for conventional genomic file formats. Parquet is an efficient columnar storage system that is widely used in the analytics ecosystem, and integrates with a variety of data management tools and query engines. Parquet provides improved storage capacity

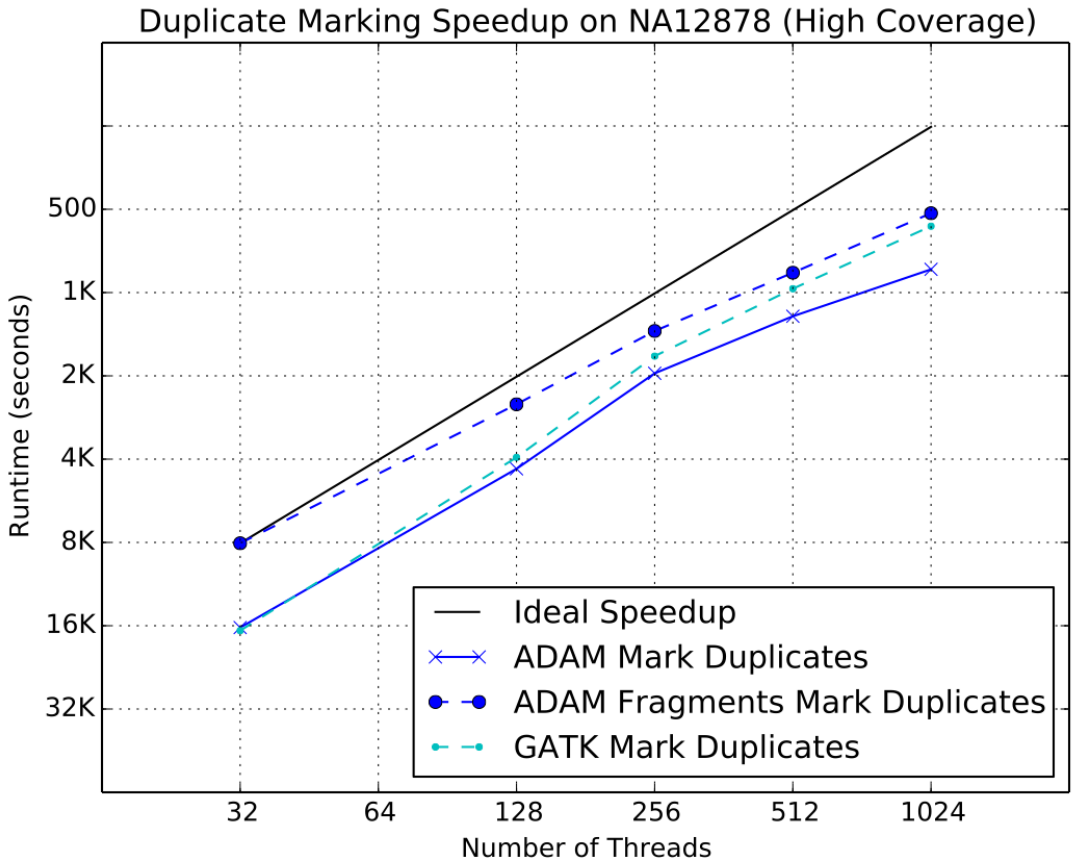

Fig. 1.4: Strong scaling characteristics of duplicate marking

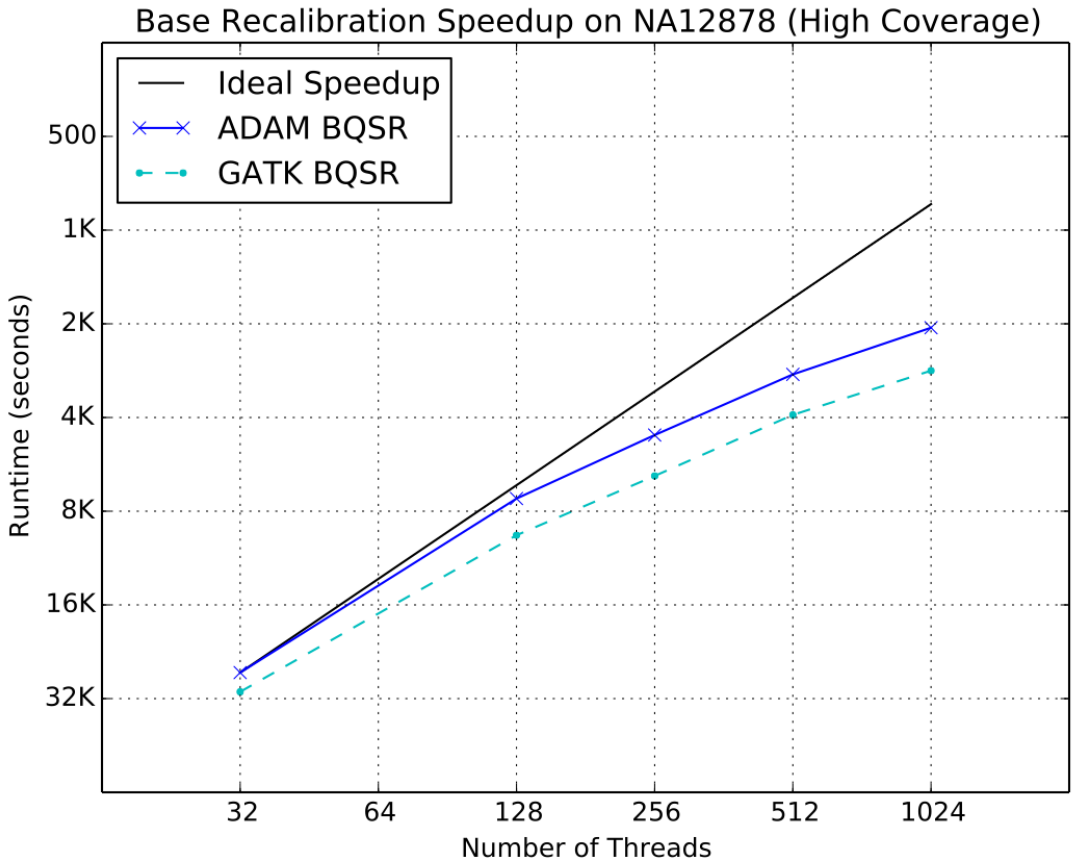

Fig. 1.5: Strong scaling characteristics of base quality recalibration

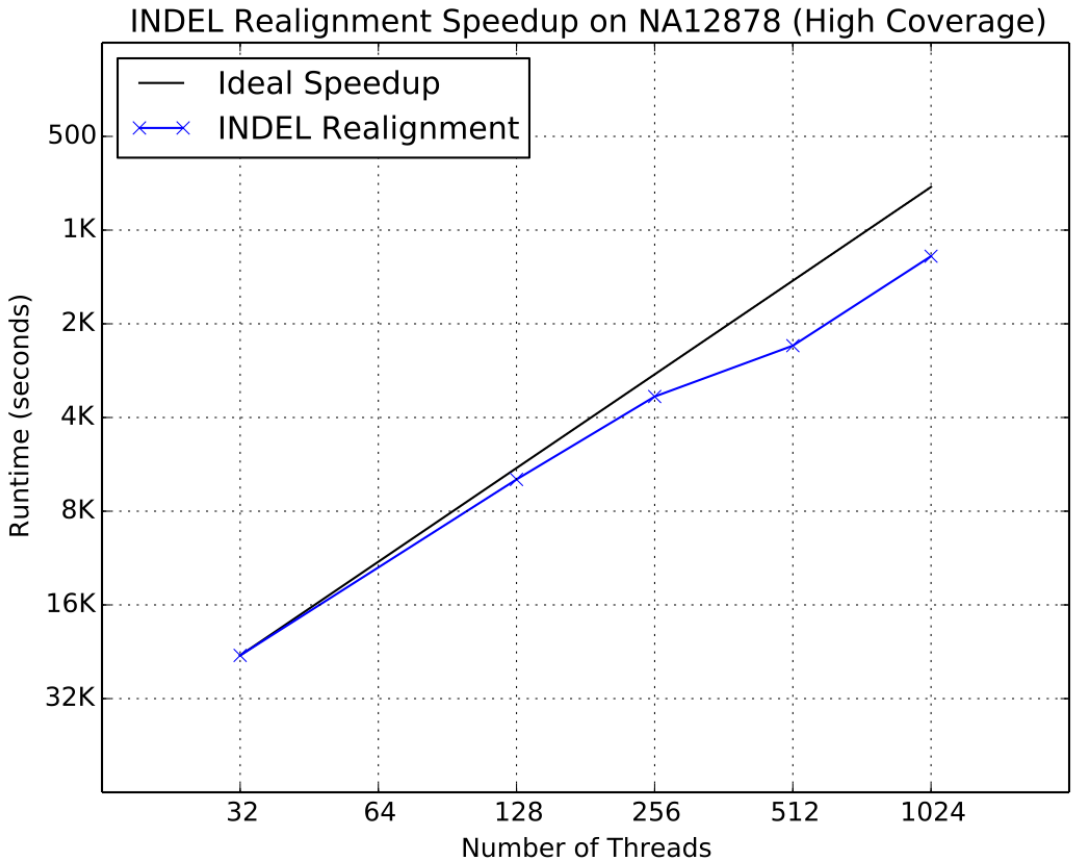

Fig. 1.6: Strong scaling characteristics of INDEL Realignment

relative to several conventional genomics data storage formats. Here, we look at the storage cost of aligned reads, features, and variants.

Aligned Reads

In this benchmark, we have stored a copy of NA12878 aligned to the GRCh37 reference genome using BWA. We store this genome in BAM, CRAM, and ADAM, using the default compression settings for each. BAM and CRAM files were generated using htslib. This read file was sequenced at approximately 60x coverage across the whole genome.

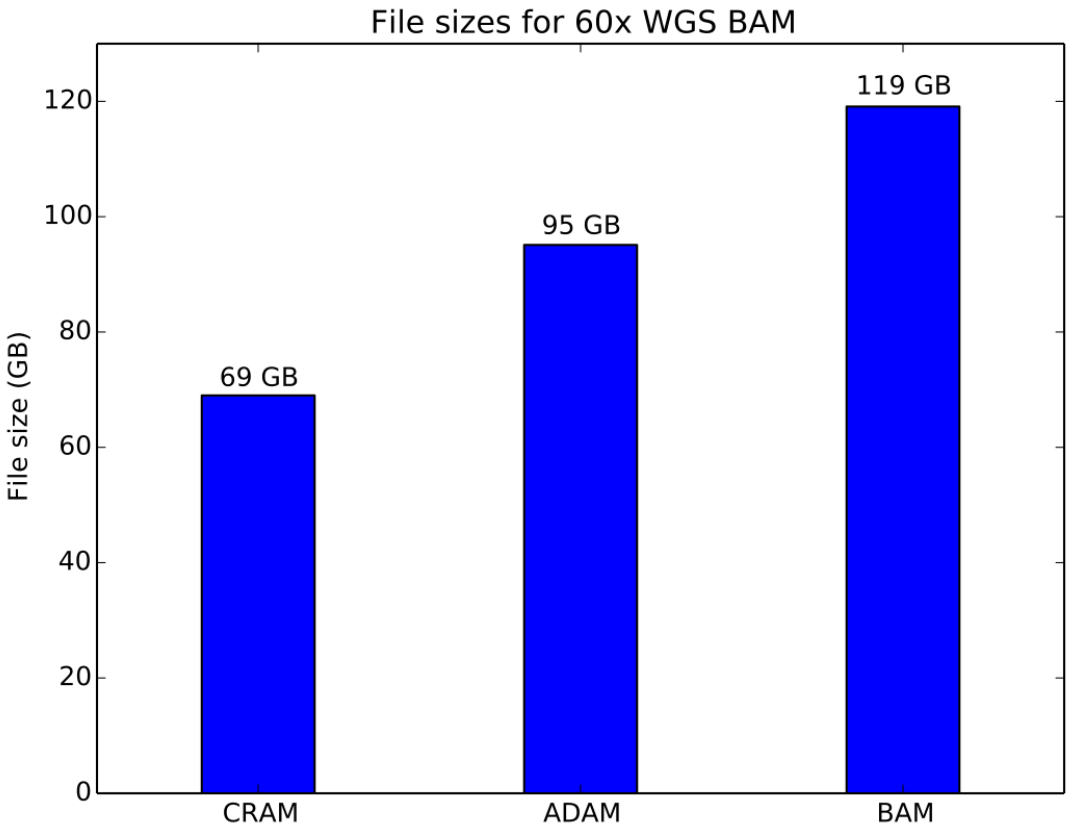

Fig. 1.7: Storage cost of a 60x coverage WGS aligned dataset

ADAM provides a 20% improvement in storage size over BAM, while CRAM achieves a 43% improvement in storage cost. CRAM achieves a higher compression ratio by using reference-based compression techniques to minimize the amount of data stored on disk.

Features

Here, we benchmark both the GFF3 and BED formats. For GFF3, we use the ENSEMBL GRCh38 genome annotation file. For BED, we use genome-wide coverage counts generated from the NA12878 dataset used in the *aligned read benchmarks*.

For the genome annotation file, ADAM provides a 20% improvement in storage size relative to the compressed GFF3 file.

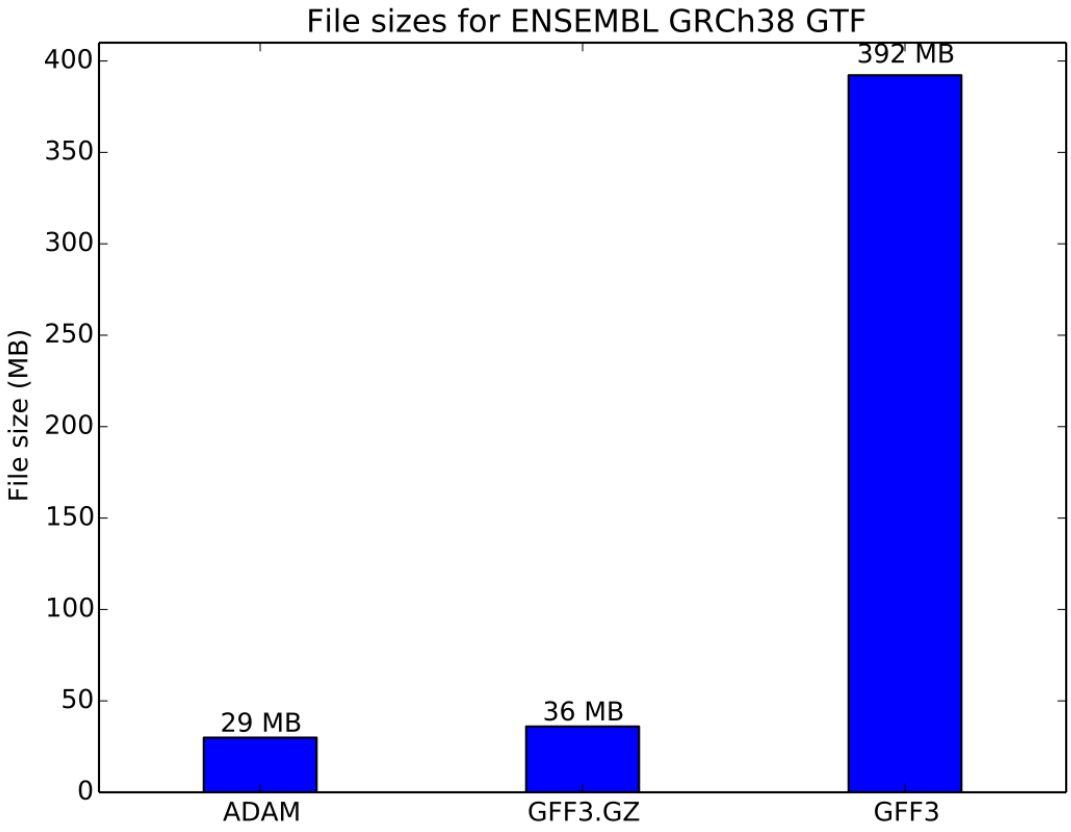

Fig. 1.8: Storage cost of genome annotations

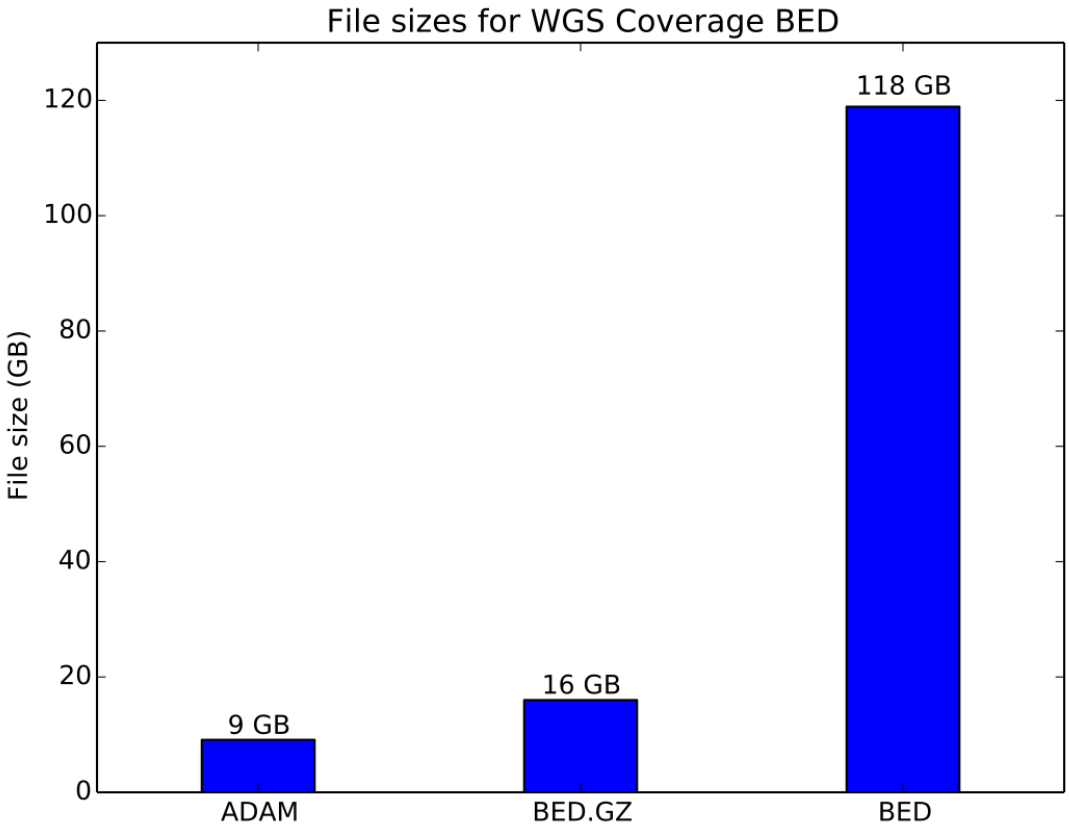

Fig. 1.9: Storage cost of coverage data

For the coverage data, ADAM provides a 45% improvement in storage size relative to the compressed BED file.

Genomic Variants

In this benchmark, we used the 1,000 Genomes phase 3 data release VCFs. We compared GZIP-compressed VCF and uncompressed VCF to ADAM.

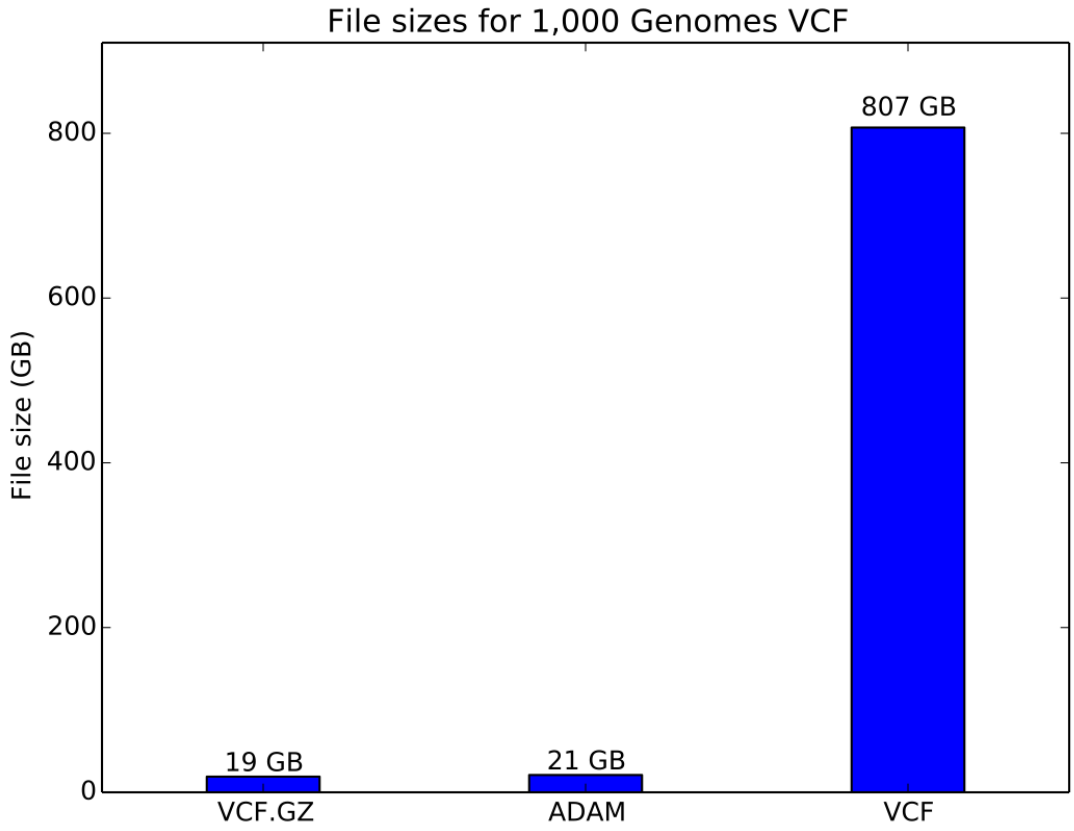

Fig. 1.10: Storage cost of variant data

Compressed VCF is approximately 10% smaller than genotype data stored as Parquet.

1.1.10 Deploying ADAM

Running ADAM on AWS EC2 using CGCloud

CGCloud provides an automated means to create a cluster on EC2 for use with ADAM.

CGCloud lets you automate the creation, management and provisioning of VMs and clusters of VMs in Amazon EC2. The CGCloud plugin for Spark lets you setup a fully configured Apache Spark cluster in EC2.

Prior to following these instructions, make sure you have set up your AWS account and know your AWS access keys. See https://aws.amazon.com/ for details.

Configure CGCloud

Begin by reading the CGcloud readme.

Next, configure CGCloud core and then install the CGcloud spark plugin.

One modification to CGCloud install instructions: replace the two pip calls
`pip install cgcloud-core` and `pip install cgcloud-spark` with the single command:

```
pip install cgcloud-spark==1.6.0
```

which will install the correct version of both cgcloud-core and cgcloud-spark.

Note, the steps to register your ssh key and create the template boxes only need to be done once.

```
cgcloud register-key ~/.ssh/id_rsa.pub
cgcloud create generic-ubuntu-trusty-box
cgcloud create -IT spark-box
```

Launch a cluster

Spin up a Spark cluster named `cluster1` with one leader and two worker nodes of instance type `m3.large` with the command:

```
cgcloud create-cluster spark -c cluster1 -s 2 -t m3.large
```

Once running, you can ssh to `spark-master` with the command:

```
cgcloud ssh -c cluster1 spark-master
```

Spark is already installed on the `spark-master` machine and slaves, test it by starting a spark-shell.

```
spark-shell
exit()
```

Install ADAM

To use the ADAM application on top of Spark, we need to download and install ADAM on `spark-master`. From the command line on `spark-master` download a release here. As of this writing, CGCloud supports Spark 1.6.2, not Spark 2.x, so download the Spark 1.x Scala2.10 release:

```
wget https://repo1.maven.org/maven2/org/bdgenomics/adam/adam-distribution_2.10/0.20.0/
↪adam-distribution_2.10-0.20.0-bin.tar.gz
tar -xvfz adam-distribution_2.10-0.20.0-bin.tar.gz
```

You can now run `./bin/adam-submit` and `./bin/adam-shell` using your EC2 cluster.

Input and Output data on HDFS and S3

Spark requires a file system, such as HDFS or a network file mount, that all machines can access. The CGCloud EC2 Spark cluster you just created is already running HDFS.

The typical flow of data to and from your ADAM application on EC2 will be:

- Upload data to AWS S3
- Transfer from S3 to the HDFS on your cluster
- Compute with ADAM, write output to HDFS
- Copy data you wish to persist for later use to S3

For small test files you may wish to skip S3 by uploading directly to spark-master using `scp` and then copying to HDFS using:

```
hadoop fs -put sample1.bam /datadir/
```

From the ADAM shell, or as a parameter to ADAM submit, you would refer to HDFS URLs like this:

```
adam-submit \
  transformAlignments \
  hdfs://spark-master/work_dir/sample1.bam \
  hdfs://spark-master/work_dir/sample1.adam
```

Bulk Transfer between HDFS and S3

To transfer large amounts of data back and forth from S3 to HDFS, we suggest using Conductor. It is also possible to directly use AWS S3 as a distributed file system, but with some loss of performance.

Directly accessing data stored in S3

To directly access data stored in S3, we can leverage one of the Hadoop FileSystem API implementations that access S3. Specifically, we recommend using the `S3a` file system. To do this, you will need to configure your Spark job to use S3a. If you are using a vendor-supported Spark distribution like Amazon EMR or Databricks, your Spark distribution may already have the S3a file system installed. If not, you will need to add JARs that contain the classes needed to support the S3a file system. For most Spark distributions built for Apache Hadoop 2.6 or higher, you will need to add the following dependencies:

- com.amazonaws:aws-java-sdk-pom:1.10.34
- org.apache.hadoop:hadoop-aws:2.7.4

Instead of downloading these JARs, you can ask Spark to install them at runtime using the `--packages` flag. Additionally, if you are using the S3a file system, your file paths will need to begin with the `s3a://` scheme:

```
adam-submit \
  --packages com.amazonaws:aws-java-sdk-pom:1.10.34,org.apache.hadoop:hadoop-aws:2.7.
↪4 \
  -- \
  transformAlignments \
  s3a://my_bucket/my_reads.adam \
  s3a://my_bucket/my_new_reads.adam
```

If you are loading a BAM, CRAM, or VCF file, you will need to add an additional JAR. This is because the code that loads data stored in these file formats uses Java's `nio` package to read index files. Java's `nio` system allows users to specify a "file system provider," which implements `nio`'s file system operations on non-POSIX file systems like HDFS or S3. To use these file formats with the `s3a://` scheme, you should include the following dependency:

- net.fnothaft:jsr203-s3a:0.0.1

You will need to do this even if you are not using the index for said format.

Terminate Cluster

Shutdown the cluster using:

```
cgcloud terminate-cluster -c cluster1 spark
```

CGCloud options and Spot Instances

View help docs for all options of the `cgcloud create-cluster` command:

```
cgcloud create-cluster -h
```

In particular, note the `--spot-bid` and related spot options to utilize AWS spot instances in order to save on costs. To avoid unintended costs, it is a good idea to use the AWS console to double check that your instances have terminated.

Accessing the Spark GUI

In order to view the Spark server or application GUI pages on port 4040 and 8080 on `spark-master`, go to Security Groups in the AWS console and open inbound TCP for those ports from your local IP address. Find the IP address of `spark-master`, which is part of the Linux command prompt. On your local machine, you can then open `http://ip_of_spark_master:4040/` in a web browser, where `ip_of_spark_master` is replaced with the IP address you found.

1.1.11 Running ADAM on CDH 5, HDP, and other YARN based Distros

Apache Hadoop YARN is a widely used scheduler in the Hadoop ecosystem. YARN stands for "Yet Another Resource Negotiator", and the YARN architecture is described in (Vavilapalli et al. 2013). YARN is used in several common Hadoop distributions, including the Cloudera Hadoop Distribution (CDH) and the Hortonworks Data Platform (HDP). YARN is supported natively in Spark.

The ADAM CLI and shell can both be run on YARN. The ADAM CLI can be run in both Spark's YARN `cluster` and `client` modes, while the ADAM shell can only be run in `client` mode. In the `cluster` mode, the Spark driver runs in the YARN `ApplicationMaster` container. In the `client` mode, the Spark driver runs in the submitting process. Since the Spark driver for the Spark/ADAM shell takes input on stdin, it cannot run in `cluster` mode.

To run the ADAM CLI in YARN `cluster` mode, run the following command:

```
./bin/adam-submit \
  --master yarn \
  --deploy-mode cluster \
  -- \
  <adam_command_name> [options] \
```

In the `adam-submit` command, all options before the `--` are passed to the `spark-submit` script, which launches the Spark job. To run in `client` mode, we simply change the `deploy-mode` to `client`:

```
./bin/adam-submit \
  --master yarn \
  --deploy-mode client \
  -- \
  <adam_command_name> [options] \
```

In the `adam-shell` command, all of the arguments are passed to the `spark-shell` command. Thus, to run the `adam-shell` on YARN, we run:

```
./bin/adam-shell \
  --master yarn \
  --deploy-mode client
```

All of these commands assume that the Spark assembly you are using is properly configured for your YARN deployment. Typically, if your Spark assembly is properly configured to use YARN, there will be a symbolic link at `${SPARK_HOME}/conf/yarn-conf/` that points to the core Hadoop/YARN configuration. Though, this may vary by the distribution you are running.

The full list of configuration options for running Spark-on-YARN can be found online. Most of the standard configurations are consistent between Spark Standalone and Spark-on-YARN. One important configuration option to be aware of is the YARN memory overhead parameter. From 1.5.0 onwards, Spark makes aggressive use of off-heap memory allocation in the JVM. These allocations may cause the amount of memory taken up by a single executor (or, theoretically, the driver) to exceed the `--driver-memory`/`--executor-memory` parameters. These parameters are what Spark provides as a memory resource request to YARN. By default, if one of your Spark containers (an executors or the driver) exceeds its memory request, YARN will kill the container by sending a `SIGTERM`. This can cause jobs to fail. To eliminate this issue, you can set the `spark.yarn.<role>.memoryOverhead` parameter, where `<role>` is one of `driver` or `executor`. This parameter is used by Spark to increase its resource request to YARN over the JVM Heap size indicated by `--driver-memory` or `--executor-memory`.

As a final example, to run the ADAM *transformAlignments* CLI using YARN cluster mode on a 64 node cluster with one executor per node and a 2GB per executor overhead, we would run:

```
./bin/adam-submit \
  --master yarn \
  --deploy-mode cluster \
  --driver-memory 200g \
  --executor-memory 200g \
  --conf spark.driver.cores=16 \
  --conf spark.executor.cores=16 \
  --conf spark.yarn.executor.memoryOverhead=2048 \
  --conf spark.executor.instances=64 \
  -- \
  transformAlignments in.sam out.adam
```

In this example, we are allocating 200GB of JVM heap space per executor and for the driver, and we are telling Spark to request 16 cores per executor and for the driver.

1.1.12 Running ADAM on Toil

Toil is a workflow management tool that supports running multi-tool workflows. Unlike traditional workflow managers that are limited to supporting jobs that run on a single node, Toil includes support for clusters of long lived services through the Service Job abstraction. This abstraction enables workflows that mix Spark-based tools like ADAM in with traditional, single-node tools. (Vivian et al. 2016) describes the Toil architecture and demonstrates the use of

Toil at scale in the Amazon Web Services EC2 cloud. Toil can be run on various on-premises High Performance Computing schedulers, and on the Amazon EC2 and Microsoft Azure clouds. A quick start guide to deploying Toil in the cloud or in an on-premises cluster can be found here.

toil-lib is a library downstream from Toil that provides common functionality that is useful across varied genomics workflows. There are two useful modules that help to set up an Apache Spark cluster, and to run an ADAM job:

- `toil_lib.spark`: This module contains all the code necessary to set up a set of Service Jobs that launch and run an Apache Spark cluster backed by the Apache Hadoop Distributed File System (HDFS).
- `toil_lib.tools.spark_tools`: This module contains functions that run ADAM in Toil using Docker, as well as Conductor, a tool for running transfers between HDFS and Amazon's S3 storage service.

Several example workflows that run ADAM in Toil can be found in toil-scripts. These workflows include:

- adam-kmers: this workflow was demonstrated in (Vivian et al. 2016) and sets up a Spark cluster which then runs ADAM's `` `countKmers` CLI <#countKmers>`__.
- adam-pipeline: this workflow runs several stages in the ADAM `` `transformAlignments` CLI <#transformAlignments>`__. This pipeline is the ADAM equivalent to the GATK's "Best Practice" read preprocessing pipeline. We then stitch together this pipeline with BWA-MEM and the GATK in the adam-gatk-pipeline.

An example workflow: `toil_scripts.adam_kmers.count_kmers`

For an example of how to use ADAM with Toil, let us look at the toil_scripts.adam_kmers.count_kmers module. This module has three parts:

- A main method that configures and launches a Toil workflow.
- A job function that launches both the Spark cluster service and the ADAM job.
- A child job function that calls ADAM and Conductor to transfer a BAM file from S3, convert that BAM file to Parquet, count *k*-mers, and upload the *k*-mer counts back to S3.

Configuring and launching Toil

Toil takes most of its configuration from the command line. To make this easy, Toil includes a function in the `toil.job.Job` class to register Toil's argument parsing code with the *Python standard ''argparse'* <https://docs.python.org/2/library/argparse.html>`__ library. E.g., *in ''count_kmers.py'* <https://github.com/BD2KGenomics/toil-scripts/blob/master/src/toil_scripts/adam_kmers/count_kmers.py#L183-L214>`__, we set up our arguments and then add the Toil specific arguments by:

```
parser = argparse.ArgumentParser()

# add parser arguments
parser.add_argument('--input_path',
                    help='The full path to the input SAM/BAM/ADAM/FASTQ file.')
parser.add_argument('--output-path',
                    help='full path where final results will be output.')
parser.add_argument('--kmer-length',
                    help='Length to use for k-mer counting. Defaults to 20.',
                    default=20,
                    type=int)
parser.add_argument('--spark-conf',
                    help='Optional configuration to pass to Spark commands. Either␣
↪this or --workers must be specified.',
                    default=None)
parser.add_argument('--memory',
```

```
                    help='Optional memory configuration for Spark workers/driver.␣
↪This must be specified if --workers is specified.',
                    default=None,
                    type=int)
parser.add_argument('--cores',
                    help='Optional core configuration for Spark workers/driver. This␣
↪must be specified if --workers is specified.',
                    default=None,
                    type=int)
parser.add_argument('--workers',
                    help='Number of workers to spin up in Toil. Either this or --
↪spark-conf must be specified. If this is specified, --memory and --cores must be␣
↪specified.',
                    default=None,
                    type=int)
parser.add_argument('--sudo',
                    help='Run docker containers with sudo. Defaults to False.',
                    default=False,
                    action='store_true')

Job.Runner.addToilOptions(parser)
```

Then, we parse the arguments and start Toil:

```
args = parser.parse_args()
Job.Runner.startToil(Job.wrapJobFn(kmer_dag,
                                   args.kmer_length,
                                   args.input_path,
                                   args.output_path,
                                   args.spark_conf,
                                   args.workers,
                                   args.cores,
                                   args.memory,
                                   args.sudo,
                                   checkpoint=True), args)
```

Note that we are passing the parsed arguments to the `Job.Runner.startToil` function. The other argument that we are passing is the Job that we would like Toil to run. In this example, Toil is wrapping the `kmer_dag` function (discussed in the next section) up as a Job. The `Job.wrapJobFn` call takes the `kmer_dag` function and all of the arguments that are being passed and serializes them so they can be run locally or on a remote node. Additionally, we pass the optional argument `checkpoint=True`. This argument indicates that the `kmer_dag` Job function is a "checkpoint" job. If a job is a checkpoint job and any of its children jobs fail, then we are saying that the workflow can be successfully rerun from this point. In Toil, service jobs should always be launched from a checkpointed job in order to allow the service jobs to successfully resume after a service job failure.

More detailed information about launching a Toil workflow can be found in the Toil documentation.

Launching a Spark Service

In the `toil_scripts.adam_kmers.count_kmers` example, we wrap the `kmer_dag` function as a job, and then use this function to launch a Spark cluster as a set of service jobs using the `toil_lib.spark` module. Once we've done that, we also launch a job to run ADAM by starting the `download_count_upload` child job function. We launch the Spark service cluster by calling the `spawn_spark_cluster`function, which was imported from the `toil_lib.spark` module:

```
master_hostname = spawn_spark_cluster(job,
                                      workers,
                                      cores)
```

This function takes in three parameters:

- `job`: A handle to the currently running Toil Job. This is used to enqueue the service jobs needed to start the Spark cluster.
- `workers`: The number of Spark workers to allocate.
- `cores`: The number of cores to request per worker/leader node.

When called, this method does not return a hostname string. Rather, it returns a promise for the hostname string. This promise is not valid inside of the `kmer_dag` job, but will be valid in the child job (`download_count_upload`) that runs Spark. Toil cannot guarantee that the Spark Service job will start until after the job that enqueues it completes.

Finally, we enqueue the child job that runs ADAM and Conductor:

```
job.addChildJobFn(download_count_upload,
                  masterHostname,
                  input_file, output_file, kmer_length,
                  spark_conf, memory, sudo)
```

Detailed documentation for the `toil_lib.spark` module can be found in the toil-lib docs.

Running ADAM and other Spark applications

Once we have enqueued the Spark service jobs and the child job that interacts with the services, we can launch Spark applications from the child job. In our example application, our child job function does the following work:

1. We check to see if the input file is already in HDFS:

```
if master_ip is not None:
    hdfs_dir = "hdfs://{0}:{1}/".format(master_ip, HDFS_MASTER_PORT)
else:
    _log.warn('Master IP is not set. If default filesystem is not set, jobs may fail.
↪')
    hdfs_dir = ""
```

2. If it is not in HDFS, we copy it in using Conductor:

```
# if the file is not already in hdfs, copy it in
hdfs_input_file = hdfs_dir
if input_file.startswith("s3://"):

    # append the s3 file name to our hdfs path
    hdfs_input_file += input_file.split("/")[-1]

    # run the download
    _log.info("Downloading input file %s to %s.", input_file, hdfs_input_file)
    call_conductor(master_ip, input_file, hdfs_input_file,
                   memory=memory, override_parameters=spark_conf)
```

3. We check to see if the file is a Parquet file, and convert it to Parquet if it is not:

```
# do we need to convert to adam?
if (hdfs_input_file.endswith('.bam') or
```

```
    hdfs_input_file.endswith('.sam') or
    hdfs_input_file.endswith('.fq') or
    hdfs_input_file.endswith('.fastq')):

    hdfs_tmp_file = hdfs_input_file

    # change the file extension to adam
    hdfs_input_file = '.'.join(hdfs_input_file.split('.')[:-1].append('adam'))

    # convert the file
    _log.info('Converting %s into ADAM format at %s.', hdfs_tmp_file, hdfs_input_file)
    call_adam(master_ip,
              ['transformAlignments',
               hdfs_tmp_file, hdfs_input_file],
              memory=memory, override_parameters=spark_conf)
```

4. We use the ADAM CLI to count the *k*-mers in the file:

```
# run k-mer counting
_log.info('Counting %d-mers in %s, and saving to %s.',
          kmer_length, hdfs_input_file, hdfs_output_file)
call_adam(master_ip,
          ['countKmers',
           hdfs_input_file, hdfs_output_file,
           str(kmer_length)],
          memory=memory, override_parameters=spark_conf)
```

5. If requested, we use Conductor to copy the *k*-mer counts back to S3:

```
# do we need to upload the file back? if so, run upload
if run_upload:
    _log.info("Uploading output file %s to %s.", hdfs_output_file, output_file)
    call_conductor(master_ip, hdfs_output_file, output_file,
                   memory=memory, override_parameters=spark_conf)
```

The `call_adam` and `call_conductor` functions are imported from the `toil_lib.tools.spark_tools` module. These functions run ADAM and Conductor using Docker containers from cgl-docker-lib.[1] These two functions launch the Docker containers using the `call_docker` function from the `toil_lib.programs` module, and do some basic configuration of the command line. In the ADAM example, all the user needs to pass is the exact arguments that they would like run from the ADAM CLI, and the Spark configuration parameters that are passed to the `adam-submit` script are automatically configured.

As you may have noticed, all of this functionality is contained in a single Toil job. This is important for fault tolerance. Toil provides tolerance against data loss through the use of a file store, which manages the persistance of local files to a persistant store (e.g., S3). Since we store intermediate files in HDFS, thus bypassing the file store, our intermediate results are not persistant, and thus individual Spark applications are not atomic.

Using PySpark in Toil

As an aside, a nice benefit of Toil is that we can run PySpark jobs inline with Toil workflows. A small demo of this is seen in the `toil_lib.spark` unit tests:

```
def _count_child(job, masterHostname):

    # noinspection PyUnresolvedReferences
```

[1] These containers are published on Quay.

```
    from pyspark import SparkContext

    # start spark context and connect to cluster
    sc = SparkContext(master='spark://%s:7077' % masterHostname,
                      appName='count_test')

    # create an rdd containing 0-9999 split across 10 partitions
    rdd = sc.parallelize(xrange(10000), 10)

    # and now, count it
    assert rdd.count() == 10000
```

1.1.13 Running ADAM on Slurm

For those groups with access to a HPC cluster with Slurm managing a number of compute nodes with local and/or network attached storage, it is possible to spin up a temporary Spark cluster for use by ADAM.

The full IO bandwidth benefits of Spark processing are likely best realized through a set of co-located compute/storage nodes. However, depending on your network setup, you may find Spark deployed on HPC to be a workable solution for testing or even production at scale, especially for those applications which perform multiple in-memory transformations and thus benefit from Spark's in-memory processing model.

Follow the primary instructions for installing ADAM into `$ADAM_HOME`. This will most likely be at a location on a shared disk accessible to all nodes, but could be at a consistant location on each machine.

Start Spark cluster

A Spark cluster can be started as a multi-node job in Slurm by creating a job file `run.cmd` such as below:

```
#!/bin/bash

#SBATCH --partition=multinode
#SBATCH --job-name=spark-multi-node
#SBATCH --exclusive

#Number of seperate nodes reserved for Spark cluster
#SBATCH --nodes=2
#SBATCH --cpus-per-task=12

#Number of excecution slots
#SBATCH --ntasks=2

#SBATCH --time=05:00:00
#SBATCH --mem=248g

# If your sys admin has installed spark as a module
module load spark

# If Spark is not installed as a module, you will need to specifiy absolute path to
# $SPARK_HOME/bin/spark-start where $SPARK_HOME is on shared disk or at a consistant␣
↪location
start-spark

echo $MASTER
sleep infinity
```

Submit the job file to Slurm:

```
sbatch run.cmd
```

This will start a Spark cluster containing two nodes that persists for five hours, unless you kill it sooner. The file `slurm.out` created in the current directory will contain a line produced by `echo $MASTER` above which will indicate the address of the Spark master to which your application or ADAM-shell should connect such as `spark://somehostname:7077`

Start adam-shell

Your sys admin will probably prefer that you launch your `adam-shell` or start an application from a cluster node rather than the head node you log in to. You may want to do so with:

```
sinteractive
```

Start an adam-shell:

```
$ADAM_HOME/bin/adam-shell --master spark://hostnamefromslurmdotout:7077
```

or Run a batch job with adam-submit

```
$ADAM_HOME/bin/adam-submit --master spark://hostnamefromslurmdotout:7077
```

You should be able to connect to the Spark Web UI at `http://hostnamefromslurmdotout:4040`, however you may need to ask your system administrator to open the required ports.

1.1.14 Running ADAM's command line tools

In addition to being used as an API for *building applications*, ADAM provides a command line interface (CLI) for extracting, transforming, and loading (ETL-ing) genomics data. Our CLI is roughly divided into three sections:

- *Actions* that manipulate data using the ADAM schemas
- *Conversions* that convert data from legacy formats into Parquet
- *Printers* that provide detailed or summarized views of genomic data

ADAM's various CLI actions can be run from the command line using the `scripts/adam-submit` script. This script uses the `spark-submit` script to run an ADAM application on a Spark cluster. To use this script, either `spark-submit` must be on the `$PATH`, or the `$SPARK_HOME` environment variable must be set.

Default arguments

There are several command line options that are present across most commands. These include:

- `-h`, `-help`, `--help`, `-?`: prints the usage for this command
- `-parquet_block_size N`: sets the block size for Parquet in bytes, if writing a Parquet output file. Defaults to 128 MB (128 * 1024 * 1024).
- `-parquet_compression_codec`: The codec to use for compressing a Parquet page. Choices are:
  - `UNCOMPRESSED`: No compression.
  - `SNAPPY`: Use the Snappy compression codec.

  - `GZIP`: Use a Gzip based compression codec.
  - `LZO`: Use a LZO based compression codec. To use LZO, the LZO libraries must be installed.
- `-parquet_disable_dictionary`: Disables dictionary encoding in Parquet, and enables delta encoding.
- `-parquet_logging_level VAL`: The Log4j logging level to set for Parquet's loggers. Defaults to `severe`.
- `-parquet_page_size N`: The page size in bytes to use when writing Parquet files. Defaults to 1MB (1024 * 1024).
- `-print_metrics`: If provided, prints the instrumentation metrics to the log when the CLI operation terminates.

Legacy output options

Several tools in ADAM support saving back to legacy genomics output formats. Any tool saving to one of these formats supports the following options:

- `-single`: Merge sharded output files. If this is not provided, the output will be written as sharded files where each shard is a valid file. If this *is* provided, the shards will be written without headers as a `${OUTPUTNAME}_tail` directory, and a single header will be written to `${OUTPUTNAME}_head`. If `-single` is provided and `-defer_merging` is *not* provided, the header file and the shard directory will be merged into a single file at `${OUTPUTPATH}`.
- `-defer_merging`: If both `-defer_merging` and `-single` are provided, the output will be saved as if is a single file, but the output files will not be merged.
- `-disable_fast_concat`: If `-single` is provided and `-defer_merging` is not, this disables the use of the parallel concatenation engine. This engine is used when running on top of HDFS, and resizes the output files to match the HDFS block size before calling the Hadoop FileSystem `concat` method which concatenates the files by modifying the filesystem metadata and not the bytes on disk. This method is vastly more performant for large files, but has many invariants that may prevent it from being run (e.g., it cannot be run on an encrypted HDFS directory).

Validation stringency

Various components in ADAM support passing a validation stringency level. This is a three level scale:

- `STRICT`: If validation fails, throw an exception.
- `LENIENT`: If validation fails, ignore the data and write a warning to the log.
- `SILENT`: If validation fails, ignore the data silently.

1.1.15 Action tools

Roughly speaking, "action" tools apply some form of non-trivial transformation to data using the ADAM APIs.

countKmers and countContigKmers

Counts the *k* length substrings in either a set of reads or reference fragments. Takes three required arguments:

1. `INPUT`: The input path. A set of reads for `countKmers` or a set of reference contigs for `countContigKmers`.

2. `OUTPUT`: The path to save the output to. Saves the output as CSV containing the *k*-mer sequence and count.
3. `KMER_LENGTH`: The length *k* of substrings to count.

Beyond the *default options*, both `countKmers` and `countContigKmers` take one option:

- `-print_histogram`: If provided, prints a histogram of the *k*-mer count distribution to standard out.

transformAlignments

The `transformAlignments` CLI is the entrypoint to ADAM's read preprocessing tools. This command provides drop-in replacement commands for several commands in the Genome Analysis Toolkit "Best Practices" read preprocessing pipeline and more (DePristo et al. 2011). This CLI tool takes two required arguments:

1. `INPUT`: The input path. A file containing reads in any of the supported ADAM read input formats.
2. `OUTPUT`: The path to save the transformed reads to. Supports any of ADAM's read output formats.

Beyond the *default options* and the *legacy output options*, `transformAlignments` supports a vast range of options. These options fall into several general categories:

- General options:
  - `-cache`: If provided, the results of intermediate stages will be cached. This is necessary to avoid recomputation if running multiple transformations (e.g., Indel realignment, BQSR, etc) back to back.
  - `-storage_level`: Along with `-cache`, this can be used to set the Spark persistance level for cached data. If not provided, this defaults to `MEM_ONLY`.
  - `-stringency`: Sets the validation stringency for various operations. Defaults to `LENIENT`. See *validation stringency* for more details.
- Loading options:
  - `-repartition`: Forces a repartition on load. Useful to increase the available parallelism on a small dataset. Forces a shuffle. Takes the number of partitions to repartition to.
  - `-force_load_bam`: Forces ADAM to try to load the input as SAM/BAM/CRAM.
  - `-force_load_fastq`: Forces ADAM to try to load the input as FASTQ.
  - `-paired_fastq`: Forces `-force_load_fastq`, and passes the path of a second-of-pair FASTQ file to load.
  - `-record_group`: If loading FASTQ, sets the record group name on each read to this value.
  - `-force_load_ifastq`: Forces ADAM to try to load the input as interleaved FASTQ.
  - `-force_load_parquet`: Forces ADAM to try to load the input as Parquet encoded using the ADAM `AlignmentRecord` schema.
  - `-limit_projection`: If loading as Parquet, sets a projection that does not load the `attributes` or `origQual` fields of the `AlignmentRecord`.
  - `-aligned_read_predicate`: If loading as Parquet, only loads aligned reads.
  - `-region_predicate`: A string indicating that reads should be filtered on overlapping a genomic position or range. This argument takes a comma separated list, where each element in the list takes the form:
  - `contig:pos` for a single genomic position, or
  - `contig:start-end` for a genomic range with closed start and open end E.g., `-region_predicate 1:100,2:1000-2000` would filter all reads that overlapped either position 100 on `1` or the range from 1,000 to 2,000 on `2`.

  - `-concat`: Provides a path to an optional second file to load, which is then concatenated to the file given as the `INPUT` path.

- Duplicate marking options: Duplicate marking is run with the `-mark_duplicate_reads` option. It takes no optional parameters.
- BQSR options: BQSR is run with the `-recalibrate_base_qualities` flag. Additionally, the BQSR engine takes the following parameters:
  - `-known_snps`: Path to a VCF file/Parquet variant file containing known point variants. These point variants are used to mask read errors during recalibration. Specifically, putative read errors that are at variant sites are treated as correct observations. If BQSR is run, this option should be passed, along with a path to a known variation database (e.g., dbSNP).
- Indel realignment options: Indel realignment is run with the `-realign_indels` flag. Additionally, the Indel realignment engine takes the following options:
  - `-known_indels`: Path to a VCF file/Parquet variant file containing known Indel variants to realign against. If provided, forces the `KNOWNS_ONLY` consensus model. If not provided, forces the `CONSENSUS_FROM_READS` model. See *candidate generation and realignment*.
  - `-max_consensus_number`: The maximum number of consensus sequences to realign a single target against. If more consensus sequences are seen at a single target, we randomly downsample. Defaults to 30.
  - `-max_indel_size`: The maximum length of an Indel to realign against. Indels longer than this size are dropped before generating consensus sequences. Defaults to 500bp.
  - `-max_target_size`: The maximum length of a target to realign. Targets longer than this size are dropped before trying to realign. Defaults to 3,000bp.
  - `-max_reads_per_target`: The maximum number of reads in a target that we will try to realign. By default, this value is 20,000 reads. If we encounter a target with more than this number of reads, we skip realigning this target.
  - `-reference`: An optional path to a reference genome assembly build. If this is not provided, then we attempt to reconstruct the reference at each target from the MD tags on each read.
  - `-unclip_reads`: If provided, we will unclip reads when attempting to realign them to the reference. If not provided, we leave clipped reads clipped.
  - `-log_odds_threshold`: The log odds threshold to use for picking a consensus sequence to finalize realignments against. A consensus will not be realigned against unless the Phred weighted edit distance against the given consensus/reference pair is a sufficient improvement over the original reference realignments. This option sets that improvement weight. Defaults to 5.0.
- Base quality binning: If the `-bin_quality_scores` option is passed, the quality scores attached to the reads will be rewritten into bins. This option takes a semicolon (`;`) delimited list, where each element describes a bin. The description for a bin is three integers: the bottom of the bin, the top of the bin, and the value to assign to bases in the bin. E.g., given the description `0,20,10:20,50,30`, all quality scores between 0–19 will be rewritten to 10, and all quality scores between 20–49 will be rewritten to 30.
- `mismatchingPositions` tagging options: We can recompute the `mismatchingPositions` field of an AlignmentRecord (SAM "MD" tag) with the `-add_md_tags` flag. This flag takes a path to a reference file in either FASTA or Parquet `NucleotideContigFragment` format. Additionally, this engine takes the following options:
  - `-md_tag_fragment_size`: If loading from FASTA, sets the size of each fragment to load. Defaults to 10,000bp.
  - `-md_tag_overwrite`: If provided, recomputes and overwrites the `mismatchingPositions` field for records where this field was provided.

- Output options: `transformAlignments` supports the *legacy output* options. Additionally, there are the following options:
  - `-coalesce`: Sets the number of partitions to coalesce the output to. If `-force_shuffle_coalesce` is not provided, the Spark engine may ignore the coalesce directive.
  - `-force_shuffle_coalesce`: Forces a shuffle that leads to the output being saved with the number of partitions requested by `-coalesce`. This is necessary if the `-coalesce` would increase the number of partitions, or if it would reduce the number of partitions to fewer than the number of Spark executors. This may have a substantial performance cost, and will invalidate any sort order.
  - `-sort_reads`: Sorts reads by alignment position. Unmapped reads are placed at the end of all reads. Contigs are ordered by sequence record index.
  - `-sort_lexicographically`: Sorts reads by alignment position. Unmapped reads are placed at the end of all reads. Contigs are ordered lexicographically.
  - `-sort_fastq_output`: Ignored if not saving to FASTQ. If saving to FASTQ, sorts the output reads by read name.

transformFeatures

Loads a feature file into the ADAM `Feature` schema, and saves it back. The input and output formats are autodetected. Takes two required arguments:

1. `INPUT`: The input path. A file containing features in any of the supported ADAM feature input formats.
2. `OUTPUT`: The path to save the transformed features to. Supports any of ADAM's feature output formats.

Beyond the *default options* and the *legacy output options*, `transformFeatures` has one optional argument:

- `-num_partitions`: If loading from a textual feature format (i.e., not Parquet), sets the number of partitions to load. If not provided, this is chosen by Spark.

transformGenotypes

Loads a genotype file into the ADAM `Genotype` schema, and saves it back. The input and output formats are autodetected. Takes two required arguments:

1. `INPUT`: The input path. A file containing genotypes in any of the supported ADAM genotype input formats.
2. `OUTPUT`: The path to save the transformed genotypes to. Supports any of ADAM's genotype output formats.

Beyond the *default options* and the *legacy output options*, `transformGenotypes` has additional arguments:

- `-coalesce`: Sets the number of partitions to coalesce the output to. If `-force_shuffle_coalesce` is not provided, the Spark engine may ignore the coalesce directive.
- `-force_shuffle_coalesce`: Forces a shuffle that leads to the output being saved with the number of partitions requested by `-coalesce`. This is necessary if the `-coalesce` would increase the number of partitions, or if it would reduce the number of partitions to fewer than the number of Spark executors. This may have a substantial performance cost, and will invalidate any sort order.
- `-sort_on_save`: Sorts the genotypes when saving, where contigs are ordered by sequence index. Conflicts with `-sort_lexicographically_on_save`.
- `-sort_lexicographically_on_save`: Sorts the genotypes when saving, where contigs are ordered lexicographically. Conflicts with `-sort_on_save`.
- `-single`: Saves the VCF file as headerless shards, and then merges the sharded files into a single VCF.

- `-stringency`: Sets the validation stringency for conversion. Defaults to `LENIENT`. See *validation stringency* for more details.

In this command, the validation stringency is applied to the individual genotypes. If a genotype fails validation, the individual genotype will be dropped (for lenient or silent validation, under strict validation, conversion will fail). Header lines are not validated. Due to a constraint imposed by the htsjdk library, which we use to parse VCF files, user provided header lines that do not match the header line definitions from the VCF 4.2 spec will be overridden with the line definitions from the specification. Unfortunately, this behavior cannot be disabled. If there is a user provided vs. spec mismatch in format/info field count or type, this will likely cause validation failures during conversion.

transformVariants

Loads a variant file into the ADAM `Variant` schema, and saves it back. The input and output formats are autodetected. Takes two required arguments:

1. `INPUT`: The input path. A file containing variants in any of the supported ADAM variant input formats.
2. `OUTPUT`: The path to save the transformed variants to. Supports any of ADAM's variant output formats.

Beyond the *default options* and the *legacy output options*, `transformVariants` has additional arguments:

- `-coalesce`: Sets the number of partitions to coalesce the output to. If `-force_shuffle_coalesce` is not provided, the Spark engine may ignore the coalesce directive.
- `-force_shuffle_coalesce`: Forces a shuffle that leads to the output being saved with the number of partitions requested by `-coalesce`. This is necessary if the `-coalesce` would increase the number of partitions, or if it would reduce the number of partitions to fewer than the number of Spark executors. This may have a substantial performance cost, and will invalidate any sort order.
- `-sort_on_save`: Sorts the variants when saving, where contigs are ordered by sequence index. Conflicts with `-sort_lexicographically_on_save`.
- `-sort_lexicographically_on_save`: Sorts the variants when saving, where contigs are ordered lexicographically. Conflicts with `-sort_on_save`.
- `-single`: Saves the VCF file as headerless shards, and then merges the sharded files into a single VCF.
- `-stringency`: Sets the validation stringency for conversion. Defaults to `LENIENT`. See *validation stringency* for more details.

In this command, the validation stringency is applied to the individual variants. If a variant fails validation, the individual variant will be dropped (for lenient or silent validation, under strict validation, conversion will fail). Header lines are not validated. Due to a constraint imposed by the htsjdk library, which we use to parse VCF files, user provided header lines that do not match the header line definitions from the VCF 4.2 spec will be overridden with the line definitions from the specification. Unfortunately, this behavior cannot be disabled. If there is a user provided vs. spec mismatch in format/info field count or type, this will likely cause validation failures during conversion.

mergeShards

A CLI tool for merging a *sharded legacy file* that was written with the `-single` and `-defer_merging` flags. Runs the file merging process. Takes two required arguments:

1. `INPUT`: The input directory of sharded files to merge.
2. `OUTPUT`: The path to save the merged file at.

This command takes several optional arguments:

- `-buffer_size`: The buffer size in bytes to use for copying data on the driver. Defaults to 4MB (4 * 1024 * 1024).

- `-header_path`: The path to a header file that should be written to the start of the merged output.
- `-write_cram_eof`: Writes an empty CRAM container at the end of the merged output file. This should not be provided unless merging a sharded CRAM file.
- `-write_empty_GZIP_at_eof`: Writes an empty GZIP block at the end of the merged output file. This should be provided if merging a sharded BAM file or any other BGZIPed format.

This command does not support Parquet output, so the only *default options* that this command supports is `-print_metrics`.

reads2coverage

The `reads2coverage` command computes per-locus coverage from reads and saves the coverage counts as features. Takes two required arguments:

1. `INPUT`: The input path. A file containing reads in any of the supported ADAM read input formats.
2. `OUTPUT`: The path to save the coverage counts to. Saves in any of the ADAM supported feature file formats.

In addition to the *default options*, `reads2coverage` takes the following options:

- `-collapse`: If two (or more) neighboring sites have the same coverage, we collapse them down into a single genomic feature.
- `-only_negative_strands`: Only computes coverage for reads aligned on the negative strand. Conflicts with `-only_positive_strands`.
- `-only_positive_strands`: Only computes coverage for reads aligned on the positive strand. Conflicts with `-only_negative_strands`.
- `-sort_lexicographically`: Sorts coverage by position. Contigs are ordered lexicographically. Only applies if running with `-collapse`.

1.1.16 Conversion tools

These tools convert data between a legacy genomic file format and using ADAM's schemas to store data in Parquet.

fasta2adam and adam2fasta

These commands convert between FASTA and Parquet files storing assemblies using the NucleotideContigFragment schema.

`fasta2adam` takes two required arguments:

1. `FASTA`: The input FASTA file to convert.
2. `ADAM`: The path to save the Parquet formatted NucleotideContigFragments to.

`fasta2adam` supports the full set of *default options*, as well as the following options:

- `-fragment_length`: The fragment length to shard a given contig into. Defaults to 10,000bp.
- `-reads`: Path to a set of reads that includes sequence info. This read path is used to obtain the sequence indices for ordering the contigs from the FASTA file.
- `-repartition`: The number of partitions to save the data to. If provided, forces a shuffle.
- `-verbose`: If given, enables additional logging where the sequence dictionary is printed.

`adam2fasta` takes two required arguments:

1. ADAM: The path to a Parquet file containing NucleotideContigFragments.
2. FASTA: The path to save the FASTA file to.

adam2fasta only supports the -print_metrics option from the *default options*. Additionally, adam2fasta takes the following options:

- -line_width: The line width in characters to use for breaking FASTA lines. Defaults to 60 characters.
- -coalesce: Sets the number of partitions to coalesce the output to. If -force_shuffle_coalesce is not provided, the Spark engine may ignore the coalesce directive.
- -force_shuffle_coalesce: Forces a shuffle that leads to the output being saved with the number of partitions requested by -coalesce. This is necessary if the -coalesce would increase the number of partitions, or if it would reduce the number of partitions to fewer than the number of Spark executors. This may have a substantial performance cost, and will invalidate any sort order.

adam2fastq

While the `transformAlignments <#transformAlignments>‘__ command can export to FASTQ, the adam2fastq provides a simpler CLI with more output options. adam2fastq takes two required arguments and an optional third argument:

1. INPUT: The input read file, in any ADAM-supported read format.
2. OUTPUT: The path to save an unpaired or interleaved FASTQ file to, or the path to save the first-of-pair reads to, for paired FASTQ.
3. Optional SECOND_OUTPUT: If saving paired FASTQ, the path to save the second-of-pair reads to.

adam2fastq only supports the -print_metrics option from the *default options*. Additionally, adam2fastq takes the following options:

- -no_projection: By default, adam2fastq only projects the fields necessary for saving to FASTQ. This option disables that projection and projects all fields.
- -output_oq: Outputs the original read qualities, if available.
- -persist_level: Sets the Spark persistence level for cached data during the conversion back to FASTQ. If not provided, the intermediate RDDs are not cached.
- -repartition: The number of partitions to save the data to. If provided, forces a shuffle.
- -validation: Sets the validation stringency for checking whether reads are paired when saving paired reads. Defaults to LENIENT. See *validation stringency* for more details.

transformFragments

These two commands translate read data between the single read alignment and fragment representations.

transformFragments takes two required arguments:

1. INPUT: The input fragment file, in any ADAM-supported read or fragment format.
2. OUTPUT: The path to save reads at, in any ADAM-supported read or fragment format.

transformFragments takes the *default options*. Additionally, transformFragments takes the following options:

- -mark_duplicate_reads: Marks reads as fragment duplicates. Running mark duplicates on fragments improves performance by eliminating one groupBy (and therefore, a shuffle) versus running on reads.

- Base quality binning: If the `-bin_quality_scores` option is passed, the quality scores attached to the reads will be rewritten into bins. This option takes a semicolon (`;`) delimited list, where each element describes a bin. The description for a bin is three integers: the bottom of the bin, the top of the bin, and the value to assign to bases in the bin. E.g., given the description `0,20,10:20,50,30`, all quality scores between 0–19 will be rewritten to 10, and all quality scores between 20–49 will be rewritten to 30.
- `-load_as_reads`: Treats the input as a read file (uses `loadAlignments` instead of `loadFragments`), which behaves differently for unpaired FASTQ.
- `-save_as_reads`: Saves the output as a Parquet file of `AlignmentRecord`s, as SAM/BAM/CRAM, or as FASTQ, depending on the output file extension. If this option is specified, the output can also be sorted:
- `-sort_reads`: Sorts reads by alignment position. Unmapped reads are placed at the end of all reads. Contigs are ordered by sequence record index.
- `-sort_lexicographically`: Sorts reads by alignment position. Unmapped reads are placed at the end of all reads. Contigs are ordered lexicographically.

1.1.17 Printing tools

The printing tools provide some form of user readable view of an ADAM file. These commands are useful for both quality control and debugging.

print

Dumps a Parquet file to either the console or a text file as JSON. Takes one required argument:

1. `FILE(S)`: The file paths to load. These must be Parquet formatted files.

This command has several options:

- `-pretty`: Pretty prints the JSON output.
- `-o`: Provides a path to save the output dump to, instead of writing the output to the console.

This command does not support Parquet output, so the only *default options* that this command supports is `-print_metrics`.

flagstat

Runs the ADAM equivalent to the SAMTools `flagstat` command. Takes one required argument:

1. `INPUT`: The input path. A file containing reads in any of the supported ADAM read input formats.

This command has several options:

- `-stringency`: Sets the validation stringency for various operations. Defaults to `SILENT`. See *validation stringency* for more details.
- `-o`: Provides a path to save the output dump to, instead of writing the output to the console.

This command does not support Parquet output, so the only *default options* that this command supports is `-print_metrics`.

view

Runs the ADAM equivalent to the SAMTools `view` command. Takes one required argument:

1. `INPUT`: The input path. A file containing reads in any of the supported ADAM read input formats.

In addition to the *default options*, this command supports the following options:

- `-o`: Provides a path to save the output dump to, instead of writing the output to the console. Format is autodetected as any of the ADAM read outputs.
- `-F`/`-f`: Filters reads that either match all (`-f`) or none (`-F`) of the flag bits.
- `-G`/`-g`: Filters reads that either mismatch all (`-g`) or none (`-G`) of the flag bits.
- `-c`: Prints the number of reads that (mis)matched the filters, instead of the reads themselves. Conflicts with `-o`.

1.1.18 API Overview

The main entrypoint to ADAM is the *ADAMContext*, which allows genomic data to be loaded in to Spark as *GenomicRDD*. GenomicRDDs can be transformed using ADAM's built in *pre-processing algorithms*, *Spark's RDD primitives*, the *region join* primitive, and ADAM's *pipe* APIs. GenomicRDDs can also be interacted with as *Spark SQL tables*.

In addition to the Scala/Java API, ADAM can be used from *Python* and *R*.

Adding dependencies on ADAM libraries

ADAM libraries are available from Maven Central under the groupId `org.bdgenomics.adam`, such as the `adam-core` library:

```
<dependency>
  <groupId>org.bdgenomics.adam</groupId>
  <artifactId>adam-core${binary.version}</artifactId>
  <version>${adam.version}</version>
</dependency>
```

Scala apps should depend on `adam-core`, while Java applications should also depend on `adam-apis`:

```
<dependency>
  <groupId>org.bdgenomics.adam</groupId>
  <artifactId>adam-apis${binary.version}</artifactId>
  <version>${adam.version}</version>
</dependency>
```

For each release, we support four `${binary.version}`s:

- `_2.10`: Spark 1.6.x on Scala 2.10
- `_2.11`: Spark 1.6.x on Scala 2.11
- `-spark2_2.10`: Spark 2.x on Scala 2.10
- `-spark2_2.11`: Spark 2.x on Scala 2.11

Additionally, we push nightly SNAPSHOT releases of ADAM to the Sonatype snapshot repo, for developers who are interested in working on top of the latest changes in ADAM.

The ADAM Python API

ADAM's Python API wraps the *ADAMContext* and *GenomicRDD* APIs so they can be used from PySpark. The Python API is feature complete relative to ADAM's Java API, with the exception of the *region join* API, which is not supported.

The ADAM R API

ADAM's R API wraps the *ADAMContext* and *GenomicRDD* APIs so they can be used from SparkR. The R API is feature complete relative to ADAM's Java API, with the exception of the *region join* API, which is not supported.

1.1.19 Loading data with the ADAMContext

The ADAMContext is the main entrypoint to using ADAM. The ADAMContext wraps an existing SparkContext to provide methods for loading genomic data. In Scala, we provide an implicit conversion from a `SparkContext` to an `ADAMContext`. To use this, import the implicit, and call an `ADAMContext` method:

```
import org.apache.spark.SparkContext

// the ._ at the end imports the implicit from the ADAMContext companion object
import org.bdgenomics.adam.rdd.ADAMContext._
import org.bdgenomics.adam.rdd.read.AlignmentRecordRDD

def loadReads(filePath: String, sc: SparkContext): AlignmentRecordRDD = {
  sc.loadAlignments(filePath)
}
```

In Java, instantiate a JavaADAMContext, which wraps an ADAMContext:

```
import org.apache.spark.apis.java.JavaSparkContext;
import org.bdgenomics.adam.apis.java.JavaADAMContext;
import org.bdgenomics.adam.rdd.ADAMContext;
import org.bdgenomics.adam.rdd.read.AlignmentRecordRDD;

class LoadReads {

  public static AlignmentRecordRDD loadReads(String filePath,
                                             JavaSparkContext jsc) {
    // create an ADAMContext first
    ADAMContext ac = new ADAMContext(jsc.sc());

    // then wrap that in a JavaADAMContext
    JavaADAMContext jac = new JavaADAMContext(ac);

    return jac.loadAlignments(filePath);
  }
}
```

From Python, instantiate an ADAMContext, which wraps a SparkContext:

```
from bdgenomics.adam.adamContext import ADAMContext

ac = ADAMContext(sc)

reads = ac.loadAlignments("my/read/file.adam")
```

With an `ADAMContext`, you can load:

- Single reads as an `AlignmentRecordRDD`: - From SAM/BAM/CRAM using `loadBam` (Scala only) - Selected regions from an indexed BAM/CRAM using `loadIndexedBam`

  (Scala only)

  - From FASTQ using `loadFastq`, `loadPairedFastq`, and `loadUnpairedFastq` (Scala only)
  - From Parquet using `loadParquetAlignments` (Scala only)
  - The `loadAlignments` method will load from any of the above formats, and will autodetect the underlying format (Scala, Java, Python, and R, also supports loading reads from FASTA)
- Paired reads as a `FragmentRDD`: - From interleaved FASTQ using `loadInterleavedFastqAsFragments`

  (Scala only)

  - From Parquet using `loadParquetFragments` (Scala only)
  - The `loadFragments` method will load from either of the above formats, as well as SAM/BAM/CRAM, and will autodetect the underlying file format. If the file is a SAM/BAM/CRAM file and the file is queryname sorted, the data will be converted to fragments without performing a shuffle. (Scala, Java, Python, and R)
- VCF lines as a `VariantContextRDD` from VCF/BCF1 using `loadVcf` (Scala only)
- Selected lines from a tabix indexed VCF using `loadIndexedVcf` (Scala only)
- Genotypes as a `GenotypeRDD`: - From Parquet using `loadParquetGenotypes` (Scala only) - From either Parquet or VCF/BCF1 using `loadGenotypes` (Scala, Java,

  Python, and R)
- Variants as a `VariantRDD`: - From Parquet using `loadParquetVariants` (Scala only) - From either Parquet or VCF/BCF1 using `loadVariants` (Scala, Java,

  Python, and R)
- Genomic features as a `FeatureRDD`: - From BED using `loadBed` (Scala only) - From GFF3 using `loadGff3` (Scala only) - From GFF2/GTF using `loadGtf` (Scala only) - From NarrowPeak using `loadNarrowPeak` (Scala only) - From IntervalList using `loadIntervalList` (Scala only) - From Parquet using `loadParquetFeatures` (Scala only) - Autodetected from any of the above using `loadFeatures` (Scala,

  Java, Python, and R)
- Fragmented contig sequence as a `NucleotideContigFragmentRDD`: - From FASTA with `loadFasta` (Scala only) - From Parquet with `loadParquetContigFragments` (Scala only) - Autodetected from either of the above using `loadSequences` (Scala,

  Java, Python, and R)
- Coverage data as a `CoverageRDD`: - From Parquet using `loadParquetCoverage` (Scala only) - From Parquet or any of the feature file formats using

  `loadCoverage` (Scala only)

  - Contig sequence as a broadcastable `ReferenceFile` using `loadReferenceFile`, which supports 2bit files, FASTA, and Parquet (Scala only)

The methods labeled "Scala only" may be usable from Java, but may not be convenient to use.

The JavaADAMContext class provides Java-friendly methods that are equivalent to the ADAMContext methods. Specifically, these methods use Java types, and do not make use of default parameters. In addition to the load/save methods described above, the ADAMContext adds the implicit methods needed for using ADAM's **pipe_** API.

1.1.20 Working with genomic data using GenomicRDDs

As described in the section on using the *ADAMContext*, ADAM loads genomic data into a GenomicRDD which is specialized for each datatype. This GenomicRDD wraps Apache Spark's Resilient Distributed Dataset (RDD, (Zaharia et al. 2012)) API with genomic metadata. The RDD abstraction presents an array of data which is distributed across a cluster. RDDs are backed by a computational lineage, which allows them to be recomputed if a node fails and the results of a computation are lost. RDDs are processed by running functional [transformations]{#transforming} across the whole dataset.

Around an RDD, ADAM adds metadata which describes the genome, samples, or read group that a dataset came from. Specifically, ADAM supports the following metadata:

- GenomicRDD base: A sequence dictionary, which describes the reference assembly that data are aligned to, if it is aligned. Applies to all types.
- MultisampleGenomicRDD: Adds metadata about the samples in a dataset. Applies to GenotypeRDD.
- ReadGroupGenomicRDD: Adds metadata about the read groups attached to a dataset. Applies to AlignmentRecordRDD and FragmentRDD.

Additionally, GenotypeRDD, VariantRDD, and VariantContextRDD store the VCF header lines attached to the original file, to enable a round trip between Parquet and VCF.

GenomicRDDs can be transformed several ways. These include:

- The *core preprocessing* algorithms in ADAM:
- Reads:
  - Reads to coverage
  - *Recalibrate base qualities*
  - *INDEL realignment*
  - *Mark duplicate reads*
- Fragments:
  - *Mark duplicate fragments*
- *RDD transformations*
- *Spark SQL transformations*
- *By using ADAM to pipe out to another tool*

Transforming GenomicRDDs

Although GenomicRDDs do not extend Apache Spark's RDD class, RDD operations can be performed on them using the transform method. Currently, we only support RDD to RDD transformations that keep the same type as the base type of the GenomicRDD. To apply an RDD transform, use the transform method, which takes a function mapping one RDD of the base type into another RDD of the base type. For example, we could use transform on an AlignmentRecordRDD to filter out reads that have a low mapping quality, but we cannot use transform to translate those reads into Features showing the genomic locations covered by reads.

If we want to transform a GenomicRDD into a new GenomicRDD that contains a different datatype (e.g., reads to features), we can instead use the transmute function. The transmute function takes a function that transforms an RDD of the type of the first GenomicRDD into a new RDD that contains records of the type of the second GenomicRDD. Additionally, it takes an implicit function that maps the metadata in the first GenomicRDD into the metadata needed by the second GenomicRDD. This is akin to the implicit function required by the *pipe* API. As an example, let us use the transmute function to make features corresponding to reads containing INDELs:

```
// pick up implicits from ADAMContext
import org.bdgenomics.adam.rdd.ADAMContext._

val reads = sc.loadAlignments("path/to/my/reads.adam")

// the type of the transmuted RDD normally needs to be specified
// import the FeatureRDD, which is the output type
import org.bdgenomics.adam.rdd.feature.FeatureRDD
import org.bdgenomics.formats.avro.Feature

val features: FeatureRDD = reads.transmute(rdd => {
  rdd.filter(r => {
    // does the CIGAR for this read contain an I or a D?
    Option(r.getCigar)
      .exists(c => c.contains("I") || c.contains("D"))
  }).map(r => {
    Feature.newBuilder
      .setContigName(r.getContigName)
      .setStart(r.getStart)
      .setEnd(r.getEnd)
      .build
  })
})
```

ADAMContext provides the implicit functions needed to run the transmute function between all GenomicRDDs contained within the org.bdgenomics.adam.rdd package hierarchy. Any custom GenomicRDD can be supported by providing a user defined conversion function.

Transforming GenomicRDDs via Spark SQL

Spark SQL introduced the strongly-typed `Dataset API in Spark 1.6.0 <https://spark.apache.org/docs/1.6.0/sql-programming-guide.html#datasets>'__. This API supports seamless translation between the RDD API and a strongly typed DataFrame style API. While Spark SQL supports many types of encoders for translating data from an RDD into a Dataset, no encoders support the Avro models used by ADAM to describe our genomic schemas. In spite of this, Spark SQL is highly desirable because it has a more efficient execution engine than the Spark RDD APIs, which can lead to substantial speedups for certain queries.

To resolve this, we added an adam-codegen package that generates Spark SQL compatible classes representing the ADAM schemas. These classes are available in the org.bdgenomics.adam.sql package. All Avro-backed GenomicRDDs now support translation to Datasets via the dataset field, and transformation via the Spark SQL APIs through the transformDataset method. As an optimization, we lazily choose either the RDD or Dataset API depending on the calculation being performed. For example, if one were to load a Parquet file of reads, we would not decide to load the Parquet file as an RDD or a Dataset until we saw your query. If you were to load the reads from Parquet and then were to immediately run a transformDataset call, it would be more efficient to load the data directly using the Spark SQL APIs, instead of loading the data as an RDD, and then transforming that RDD into a SQL Dataset.

The functionality of the adam-codegen package is simple. The goal of this package is to take ADAM's Avro schemas and to remap them into classes that implement Scala's Product interface, and which have a specific style

of constructor that is expected by Spark SQL. Additionally, we define functions that translate between these Product classes and the bdg-formats Avro models. Parquet files written with either the Product classes and Spark SQL Parquet writer or the Avro classes and the RDD/ParquetAvroOutputFormat are equivalent and can be read through either API. However, to support this, we must explicitly set the requested schema on read when loading data through the RDD read path. This is because Spark SQL writes a Parquet schema that is equivalent but not strictly identical to the Parquet schema that the Avro/RDD write path writes. If the schema is not set, then schema validation on read fails. If reading data using the *ADAMContext* APIs, this is handled properly; this is an implementation note necessary only for those bypassing the ADAM APIs.

Similar to `transform`/`transformDataset`, there exists a `transmuteDataset` function that enables transformations between `GenomicRDD`s of different types.

1.1.21 Using ADAM's RegionJoin API

Another useful API implemented in ADAM is the RegionJoin API, which joins two genomic datasets that contain overlapping regions. This primitive is useful for a number of applications including variant calling (identifying all of the reads that overlap a candidate variant), coverage analysis (determining the coverage depth for each region in a reference), and INDEL realignment (identify INDELs aligned against a reference).

There are two overlap join implementations available in ADAM: BroadcastRegionJoin and ShuffleRegionJoin. The result of a ShuffleRegionJoin is identical to the BroadcastRegionJoin, however they serve different purposes depending on the content of the two datasets.

The ShuffleRegionJoin is a distributed sort-merge overlap join. To ensure that the data are appropriately colocated, we perform a copartition on the right dataset before the each node conducts the join locally. ShuffleRegionJoin should be used if the right dataset is too large to send to all nodes and both datasets have high cardinality.

The BroadcastRegionJoin performs an overlap join by broadcasting a copy of the entire left dataset to each node. The BroadcastRegionJoin should be used when the right side of your join is small enough to be collected and broadcast out, and the larger side of the join is unsorted and the data are too large to be worth shuffling, the data are sufficiently skewed that it is hard to load balance, or you can tolerate unsorted output.

Another important distinction between ShuffleRegionJoin and BroadcastRegionJoin is the join operations available in ADAM. Since the broadcast join does not co-partition the datasets and instead sends the full right table to all nodes, some joins (e.g. left/full outer joins) cannot be written as broadcast joins. See the table below for an exact list of what joins are available for each type of region join.

To perform a ShuffleRegionJoin, use the following:

```
dataset1.shuffleRegionJoin(dataset2)
```

To perform a BroadcastRegionJoin, use the following:

```
dataset1.broadcastRegionJoin(dataset2)
```

Where `dataset1` and `dataset2` are `GenomicRDD`s. If you used the ADAMContext to read a genomic dataset into memory, this condition is met.

ADAM has a variety of region join types that you can perform on your data, and all are called in a similar way:

- Joins implemented across both shuffle and broadcast
- Inner join
- Right outer join
- Shuffle-only joins
- Full outer join

- Inner join and group by left
- Left outer join
- Right outer join and group by left
- Broadcast-only joins
- Inner join and group by right
- Right outer join and group by right

A subset of these joins are depicted in Figure 2 below.

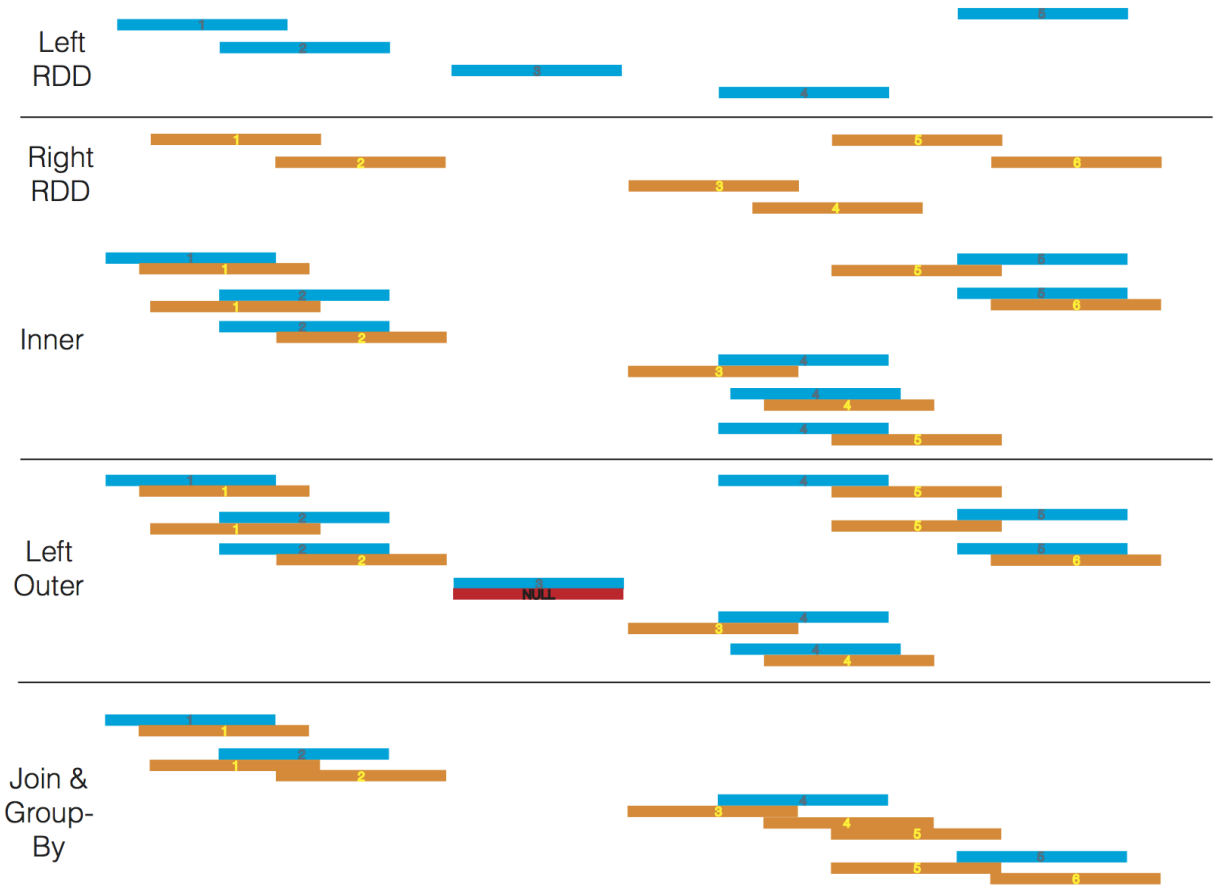

Fig. 1.11: Joins Available

One common pattern involves joining a single dataset against many datasets. An example of this is joining an RDD of features (e.g., gene/exon coordinates) against many different RDDs of reads. If the object that is being used many times (gene/exon coordinates, in this case), we can force that object to be broadcast once and reused many times with the `broadcast()` function. This pairs with the `broadcastRegionJoin` and `rightOuterBroadcastRegionJoin` functions. For example, given the following code:

```
val reads = sc.loadAlignments("my/reads.adam")
val features = sc.loadFeatures("my/features.adam")

val readsByFeature = features.broadcastRegionJoin(reads)
```

We can get a handle to the broadcast features by rewriting the code as:

```
val reads = sc.loadAlignments("my/reads.adam")
val bcastFeatures = sc.loadFeatures("my/features.adam").broadcast()

val readsByFeature = reads.broadcastRegionJoinAgainst(bcastFeatures)
```

To demonstrate how the RegionJoin APIs can be used to answer scientific questions, we will walk through three common queries that can be written using the RegionJoin API. First, we will perform a simple filter on genotypes based on a file of features. We will then demonstrate a join and group by on variants and features, providing variant data grouped by the feature they overlap. Finally, we will separate reads into those that overlap and those that do not overlap features from a feature file.

These demonstrations illustrate the difference between calling ShuffleRegionJoin and BroadcastRegionJoin and provide example code to expand from.

Filter Genotypes by Features

This query joins an RDD of Genotypes against an RDD of Features using an inner join. Because this is an inner join, records from either dataset that do not pair to the other are automatically dropped, providing the filter we are interested in. This query is useful for trying to identify genotypes that overlap features of interest. For example, if our feature file contains all the exonic regions of the genome, this query would extract all genotypes that fall in exonic regions.

```
// Inner join will filter out genotypes not covered by a feature
val genotypes = sc.loadGenotypes("my/genotypes.adam")
val features = sc.loadFeatures("my/features.adam")

// We can use ShuffleRegionJoin...
val joinedGenotypesShuffle = genotypes.shuffleRegionJoin(features)

// ...or BroadcastRegionJoin
val joinedGenotypesBcast = features.broadcastRegionJoin(genotypes)

// In the case that we only want Genotypes, we can use a simple projection
val filteredGenotypesShuffle = joinedGenotypesShuffle.rdd.map(_._1)

val filteredGenotypesBcast = joinedGenotypesBcast.rdd.map(_._2)
```

After the join, we can perform a transform function on the resulting RDD to manipulate it into providing the answer to our question. Since we were interested in the `Genotype`s that overlap a `Feature`, we map over the tuples and select just the `Genotype`.

Since a broadcast join sends the left dataset to all executors, we chose to send the `features` dataset because feature data are usually smaller in size than genotypic data.

Group overlapping variant data by the gene they overlap

This query joins an RDD of Variants against an RDD of Features, and immediately performs a group-by on the Feature. This produces an RDD whose elements are a tuple containing a Feature, and all of the Variants overlapping the Feature. This produces an RDD whose elements are tuples containing a Feature and all of the Variants overlapping the Feature.This query is useful for trying to identify annotated variants that may interact (identifying frameshift mutations within a transcript that may act as a pair to shift and then restore the reading frame) or as the start of a query that computes variant density over a set of genomic features.

```
// Inner join with a group by on the features
val features = sc.loadFeatures("my/features.adam")
val variants = sc.loadVariants("my/variants.adam")

// As a ShuffleRegionJoin, it can be implemented as follows:
val variantsByFeatureShuffle = features.shuffleRegionJoinAndGroupByLeft(variants)

// As a BroadcastRegionJoin, it can be implemented as follows:
val variantsByFeatureBcast = variants.broadcastRegionJoinAndGroupByRight(features)
```

When we switch join strategies, we swap which dataset is on the left side of the join. BroadcastRegionJoin only supports grouping by the right dataset, and ShuffleRegionJoin supports only grouping by the left dataset.

The reason BroadcastRegionJoin does not have a `joinAndGroupByLeft` implementation is due to the fact that the left dataset is broadcast to all nodes. Unlike shuffle joins, broadcast joins do not maintain a sort order invariant. Because of this, we would need to shuffle all data to a group-by on the left side of the dataset, and there is no opportunity to optimize by combining the join and group-by.

Separate reads into overlapping and non-overlapping features

This query joins an RDD of reads with an RDD of features using an outer join. The outer join will produce an RDD where each read is optionally mapped to a feature. If a given read does not overlap with any features provided, it is paired with a `None`. After we perform the join, we use a predicate to separate the reads into two RDDs. This query is useful for filtering out reads based on feature data. For example, identifying reads that overlap with ATAC-seq data to perform chromatin accessibility studies. It may be useful to separate the reads to perform distinct analyses on each resulting dataset.

```
// An outer join provides us with both overlapping and non-overlapping data
val reads = sc.loadAlignments("my/reads.adam")
val features = sc.loadFeatures("my/features.adam")

// As a ShuffleRegionJoin, we can use a LeftOuterShuffleRegionJoin:
val readsToFeatures = reads.leftOuterShuffleRegionJoin(features)

// As a BroadcastRegionJoin, we can use a RightOuterBroadcastRegionJoin:
val featuresToReads = features.rightOuterBroadcastRegionJoin(reads)

// After we have our join, we need to separate the RDD
// If we used the ShuffleRegionJoin, we filter by None in the values
val overlapsFeatures = readsToFeatures.rdd.filter(_._2.isDefined)
val notOverlapsFeatures = readsToFeatures.rdd.filter(_._2.isEmpty)

// If we used BroadcastRegionJoin, we filter by None in the keys
val overlapsFeatures = featuresToReads.rdd.filter(_._1.isDefined)
val notOverlapsFeatures = featuresToReads.rdd.filter(_._1.isEmpty)
```

Because of the difference in how ShuffleRegionJoin and BroadcastRegionJoin are called, the predicate changes between them. It is not possible to call a `leftOuterJoin` using the BroadcastRegionJoin. As previously mentioned, the BroadcastRegionJoin broadcasts the left dataset, so a left outer join would require an additional shuffle phase. For an outer join, using a ShuffleRegionJoin will be cheaper if your reads are already sorted, however if the feature dataset is small and the reads are not sorted, the BroadcastRegionJoin call would likely be more performant.

1.1.22 Using ADAM's Pipe API

ADAM's `GenomicRDD` API provides support for piping the underlying genomic data out to a single node process through the use of a `pipe` API. This builds off of Apache Spark's `RDD.pipe` API. However, `RDD.pipe` prints the objects as strings to the pipe. ADAM's pipe API adds several important functions:

- It supports on-the-fly conversions to widely used genomic file formats
- It does not require input/output type matching (i.e., you can pipe reads in and get variants back from the pipe)
- It adds the ability to set environment variables and to make local files (e.g., a reference genome) available to the run command
- If the data are aligned, we ensure that each subcommand runs over a contiguous section of the reference genome, and that data are sorted on this chunk. We provide control over the size of any flanking region that is desired.

The method signature of a pipe command is below:

```
def pipe[X, Y <: GenomicRDD[X, Y], V <: InFormatter[T, U, V]](cmd: String,
                                                              files: Seq[String] =␣
↪Seq.empty,
                                                              environment: Map[String,
↪ String] = Map.empty,
                                                              flankSize: Int =␣
↪0)(implicit tFormatterCompanion: InFormatterCompanion[T, U, V],
                                                                                  ␣
↪xFormatter: OutFormatter[X],
                                                                                  ␣
↪convFn: (U, RDD[X]) => Y,
                                                                                  ␣
↪tManifest: ClassTag[T],
                                                                                  ␣
↪xManifest: ClassTag[X]): Y
```

`X` is the type of the records that are returned (e.g., for reads, `AlignmentRecord`) and `Y` is the type of the `GenomicRDD` that is returned (e.g., for reads, `AlignmentRecordRDD`). As explicit parameters, we take:

- `cmd`: The command to run.
- `files`: Files to make available locally to each running command. These files can be referenced from `cmd` by using `$#` syntax, where `#` is the number of the file in the `files` sequence (e.g., `$0` is the head of the list, `$1` is the second file in the list, and so on).
- `environment`: Environment variable/value pairs to set locally for each running command.
- `flankSize`: The number of base pairs to flank each partition by, if piping genome aligned data.

Additionally, we take several important implicit parameters:

- `tFormatter`: The `InFormatter` that converts the data that is piped into the run command from the underlying `GenomicRDD` type.
- `xFormatter`: The `OutFormatter` that converts the data that is piped out of the run command back to objects for the output `GenomicRDD`.
- `convFn`: A function that applies any necessary metadata conversions and creates a new `GenomicRDD`.

The `tManifest` and `xManifest` implicit parameters are Scala ClassTags and will be provided by the compiler.

What are the implicit parameters used for? For each of the genomic datatypes in ADAM, we support multiple legacy genomic filetypes (e.g., reads can be saved to or read from BAM, CRAM, FASTQ, and SAM). The `InFormatter` and `OutFormatter` parameters specify the format that is being read into or out of the pipe. We support the following:

- AlignmentRecordRDD:
- InFormatters: SAMInFormatter and BAMInFormatter write SAM or BAM out to a pipe.
- OutFormatter: AnySAMOutFormatter supports reading SAM and BAM from a pipe, with the exact format autodetected from the stream.
- We do not support piping CRAM due to complexities around the reference-based compression.
- FeatureRDD:
- InFormatters: BEDInFormatter, GFF3InFormatter, GTFInFormatter, and NarrowPeakInFormatter for writing features out to a pipe in BED, GFF3, GTF/GFF2, or NarrowPeak format, respectively.
- OutFormatters: BEDOutFormatter, GFF3OutFormatter, GTFOutFormatter, and NarrowPeakInFormatter for reading features in BED, GFF3, GTF/GFF2, or NarrowPeak format in from a pipe, respectively.
- FragmentRDD:
- InFormatter: InterleavedFASTQInFormatter writes FASTQ with the reads from a paired sequencing protocol interleaved in the FASTQ stream to a pipe.
- VariantContextRDD:
- InFormatter: VCFInFormatter writes VCF to a pipe.
- OutFormatter: VCFOutFormatter reads VCF from a pipe.

The convFn implementations are provided as implicit values in the *ADAMContext*. These conversion functions are needed to adapt the metadata stored in a single GenomicRDD to the type of a different GenomicRDD (e.g., if piping an AlignmentRecordRDD through a command that returns a VariantContextRDD, we will need to convert the AlignmentRecordRDDs RecordGroupDictionary into an array of Samples for the VariantContextRDD). We provide four implementations:

- ADAMContext.sameTypeConversionFn: For piped commands that do not change the type of the GenomicRDD (e.g., AlignmentRecordRDD → AlignmentRecordRDD).
- ADAMContext.readsToVCConversionFn: For piped commands that go from an AlignmentRecordRDD to a VariantContextRDD.
- ADAMContext.fragmentsToReadsConversionFn: For piped commands that go from a FragmentRDD to an AlignmentRecordRDD.

To put everything together, here is an example command. Here, we will run a command my_variant_caller, which accepts one argument -R <reference>.fa, SAM on standard input, and outputs VCF on standard output:

```
// import RDD load functions and conversion functions
import org.bdgenomics.adam.rdd.ADAMContext._

// import functionality for piping SAM into pipe
import org.bdgenomics.adam.rdd.read.SAMInFormatter

// import functionality for reading VCF from pipe
import org.bdgenomics.adam.converters.DefaultHeaderLines
import org.bdgenomics.adam.rdd.variant.{
  VariantContextRDD,
  VCFOutFormatter
}

// load the reads
val reads = sc.loadAlignments("hdfs://mynamenode/my/read/file.bam")
```

```
// define implicit informatter for sam
implicit val tFormatter = SAMInFormatter

// define implicit outformatter for vcf
// attach all default headerlines
implicit val uFormatter = new VCFOutFormatter(DefaultHeaderLines.allHeaderLines)

// run the piped command
// providing the explicit return type (VariantContextRDD) will ensure that
// the correct implicit convFn is selected
val variantContexts: VariantContextRDD = reads.pipe("my_variant_caller -R $0",
  files = Seq("hdfs://mynamenode/my/reference/genome.fa"))

// save to vcf
variantContexts.saveAsVcf("hdfs://mynamenode/my/variants.vcf")
```

In this example, we assume that `my_variant_caller` is on the PATH on each machine in our cluster. We suggest several different approaches:

- Install the executable on the local filesystem of each machine on your cluster.
- Install the executable on a shared file system (e.g., NFS) that is accessible from every machine in your cluster, and make sure that necessary prerequisites (e.g., python, dynamically linked libraries) are installed across each node on your cluster.
- Run the command using a container system such as Docker or Singularity.

Using the Pipe API from Java

The pipe API example above uses Scala's implicit system and type inference to make it easier to use the pipe API. However, we also provide a Java equivalent. There are several changes:

- The out-formatter is provided explicitly.
- Instead of implicitly providing the companion object for the in-formatter, you provide the class of the in-formatter. This allows us to access the companion object via reflection.
- For the conversion function, you can provide any function that implements the `org.apache.spark.api.java.Function2` interface. We provide common functions equivalent to those in `ADAMContext` in `org.bdgenomics.adam.api.java.GenomicRDDConverters`.

To run the Scala example code above using Java, we would write:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.bdgenomics.adam.models.VariantContext
import org.bdgenomics.adam.rdd.read.AlignmentRecordRDD;
import org.bdgenomics.adam.rdd.read.SAMInFormatter;
import org.bdgenomics.adam.rdd.variant.VariantContextRDD;
import org.bdgenomics.adam.rdd.variant.VCFOutFormatter;
import org.bdgenomics.adam.api.java.AlignmentRecordToVariantContextConverter;

class PipeRunner {

  VariantContextRDD runPipe(AlignmentRecordRDD reads) {
```

```
    List<String> files = new ArrayList<String>();
    files.add("hdfs://mynamenode/my/reference/genome.fa")

    Map<String, String> env = new HashMap<String, String>();

    return reads.pipe<VariantContext,
                      VariantContextRDD,
                      SAMInFormatter>("my_variant_caller -R $0",
                                      files,
                                      env,
                                      0,
                                      SAMInFormatter.class,
                                      new VCFOutFormatter,
                                      new AlignmentRecordToVariantContextConverter);
  }
}
```

Using the Pipe API from Python/R

Python and R follow the same calling style as the *Java pipe API*, but the in/out-formatter and conversion functions are passed by name. We then use the classnames that are passed to the function to create the objects via reflection. To run the example code from above in Python, we would write:

```
from bigdatagenomics.adam.adamContext import ADAMContext

ac = ADAMContext(self.sc)
reads = ac.loadAlignments("hdfs://mynamenode/my/read/file.bam")

variants = reads.pipe("my_variant_caller -R $0",
                      "org.bdgenomics.adam.rdd.read.SAMInFormatter",
                      "org.bdgenomics.adam.rdd.variant.VCFOutFormatter",
                      "org.bdgenomics.adam.api.java.
↪AlignmentRecordToVariantContextConverter",
                      files=[ "hdfs://mynamenode/my/reference/genome.fa" ])
```

In R, we would write:

```
library(bdg.adam)

ac <- ADAMContext(sc)

reads <- loadAlignments(ac, "hdfs://mynamenode/my/read/file.bam")

files <- list("hdfs://mynamenode/my/reference/genome.fa")

variants <- pipe(reads,
                 "my_variant_caller -R $0",
                 "org.bdgenomics.adam.rdd.read.SAMInFormatter",
                 "org.bdgenomics.adam.rdd.variant.VCFOutFormatter",
                 "org.bdgenomics.adam.api.java.
↪AlignmentRecordToVariantContextConverter",
                 files=files)
```

1.1.23 Building Downstream Applications

ADAM is packaged so that it can be used interactively via the ADAM shell, called from the command line interface (CLI), or included as a library when building downstream applications.

This document covers three patterns for building applications downstream of ADAM:

- Extend the ADAM CLI by *adding new commands*
- Extend the ADAM CLI by *adding new commands in an external repository*
- Use ADAM as a *library in new applications*

1.1.24 Extend the ADAM CLI by adding new commands

ADAM's CLI is implemented in the adam-cli Apache Maven module of the bdgenomics/adam repository, one .scala source file for each CLI action (e.g. TransformAlignments.scala for the *transformAlignments* action), and a main class (ADAMMain.scala) that assembles and delegates to the various CLI actions.

To add a new command:

1. *Extend Args4jBase to specify arguments*
2. *Extend BDGCommandCompanion*
3. *Build ADAM and run the new command*

Extend Args4jBase to specify arguments

Extend `Args4jBase` class to specify arguments to your new command. Arguments are defined using the args4j library. If reading from or writing to Parquet, consider including Parquet arguments via `with ParquetArgs`.

```
class MyCommandArgs extends Args4jBase with ParquetArgs {
  @Argument(required = true, metaVar = "INPUT", usage = "Input to my command", index
↪= 0)
  var inputPath: String = null
}
```

Extend BDGCommandCompanion

Extend `BDGCommandCompanion` object to specify the command name and description. The `apply` method associates `MyCommandArgs` defined above with `MyCommand`.

```
object MyCommand extends BDGCommandCompanion {
  val commandName = "myCommand"
  val commandDescription = "My command example."

  def apply(cmdLine: Array[String]) = {
    new MyCommand(Args4j[MyCommandArgs](cmdLine))
  }
}
```

Extend `BDGSparkCommand` class and implement the `run(SparkContext)` method. The `MyCommandArgs` class defined above is provided in the constructor and specifies the CLI command for `BDGSparkCommand`. We must define a companion object because the command cannot run without being added to the list of accepted commands described below. For access to an slf4j Logger via the `log` field, mix in the org.bdgenomics.utils.misc.Logging trait by adding `with Logging` to the class definition.

```
class MyCommand(protected val args: MyCommandArgs) extends␣
↪BDGSparkCommand[MyCommandArgs] with Logging {
  val companion = MyCommand

  def run(sc: SparkContext) {
    log.info("Doing something...")
    // do something
  }
}
```

Add the new command to the default list of commands in org.bdgenomics.adam.cli.ADAMMain.

```
...
  val defaultCommandGroups =
    List(
      CommandGroup(
        "ADAM ACTIONS",
        List(
          MyCommand,
          CountReadKmers,
          CountContigKmers,
...
```

Build ADAM and run the new command

Build ADAM and run the new command via `adam-submit`.

```
$ mvn install
$ ./bin/adam-submit --help
Using ADAM_MAIN=org.bdgenomics.adam.cli.ADAMMain
Using SPARK_SUBMIT=/usr/local/bin/spark-submit

       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

Usage: adam-submit [<spark-args> --] <adam-args>

Choose one of the following commands:

ADAM ACTIONS
           myCommand : My command example.
          countKmers : Counts the k-mers/q-mers from a read dataset.
    countContigKmers : Counts the k-mers/q-mers from a read dataset.
...

$ ./bin/adam-submit myCommand input.foo
```

Then consider creating an issue to start the process toward including the new command in ADAM! Please see CONTRIBUTING.md before opening a new pull request.

1.1.25 Extend the ADAM CLI by adding new commands in an external repository

To extend the ADAM CLI by adding new commands in an external repository, create a new object with a `main(args: Array[String])` method that delegates to `ADAMMain` and provides additional command(s) via its constructor.

```
import org.bdgenomics.adam.cli.{ ADAMMain, CommandGroup }
import org.bdgenomics.adam.cli.ADAMMain.defaultCommandGroups

object MyCommandsMain {
  def main(args: Array[String]) {
    val commandGroup = List(CommandGroup("MY COMMANDS", List(MyCommand1, MyCommand2)))
    new ADAMMain(defaultCommandGroups.union(commandGroup))(args)
  }
}
```

Build the project and run the new external commands via `adam-submit`, specifying `ADAM_MAIN` environment variable as the new main class, and providing the jar file in the Apache Spark `--jars` argument.

Note the `--` argument separator between Apache Spark arguments and ADAM arguments.

```
$ ADAM_MAIN=MyCommandsMain \
  adam-submit \
  --jars my-commands.jar \
  -- \
  --help

Using ADAM_MAIN=MyCommandsMain
Using SPARK_SUBMIT=/usr/local/bin/spark-submit

       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

Usage: adam-submit [<spark-args> --] <adam-args>

Choose one of the following commands:
...

MY COMMANDS
          myCommand1 : My command example 1.
          myCommand2 : My command example 2.

$ ADAM_MAIN=MyCommandsMain \
  adam-submit \
  --jars my-commands.jar \
  -- \
  myCommand1 input.foo
```

A complete example of this pattern can be found in the heuermh/adam-commands repository.

1.1.26 Use ADAM as a library in new applications

To use ADAM as a library in new applications:

Create an object with a `main(args: Array[String])` method and handle command line arguments. Feel free to use the args4j library or any other argument parsing library.

```
object MyExample {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("at least one argument required, e.g. input.foo")
      System.exit(1)
    }
  }
}
```

Create an Apache Spark configuration `SparkConf` and use it to create a new `SparkContext`. The following serialization configuration needs to be present to register ADAM classes. If any additional Kyro serializers need to be registered, *create a registrator that delegates to the ADAM registrator*. You might want to provide your own serializer registrator if you need custom serializers for a class in your code that either has a complex structure that Kryo fails to serialize properly via Kryo's serializer inference, or if you want to require registration of all classes in your application to improve performance.

```
val conf = new SparkConf()
  .setAppName("MyCommand")
  .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  .set("spark.kryo.registrator", "org.bdgenomics.adam.serialization.
↪ADAMKryoRegistrator")
  .set("spark.kryo.referenceTracking", "true")

val sc = new SparkContext(conf)
// do something
```

Configure the new application build to create a fat jar artifact with ADAM and its transitive dependencies included. For example, this `maven-shade-plugin` configuration would work for an Apache Maven build.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <configuration>
    <createDependencyReducedPom>false</createDependencyReducedPom>
    <filters>
      <filter>
        <artifact>*:*</artifact>
        <excludes>
          <exclude>META-INF/*.SF</exclude>
          <exclude>META-INF/*.DSA</exclude>
          <exclude>META-INF/*.RSA</exclude>
        </excludes>
      </filter>
    </filters>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.resource.
↪ServicesResourceTransformer" />
```

```
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Build the new application and run via `spark-submit`.

```
spark-submit \
  --class MyCommand \
  target/my-command.jar \
  input.foo
```

A complete example of this pattern can be found in the heuermh/adam-examples repository.

Writing your own registrator that calls the ADAM registrator

As we do in ADAM, an application may want to provide its own Kryo serializer registrator. The custom registrator may be needed in order to register custom serializers, or because the application's configuration requires all serializers to be registered. In either case, the application will need to provide its own Kryo registrator. While this registrator can manually register ADAM's serializers, it is simpler to call to the ADAM registrator from within the registrator. As an example, this pattern looks like the following code:

```
import com.esotericsoftware.kryo.Kryo
import org.apache.spark.serializer.KryoRegistrator
import org.bdgenomics.adam.serialization.ADAMKryoRegistrator

class MyCommandKryoRegistrator extends KryoRegistrator {

  private val akr = new ADAMKryoRegistrator()

  override def registerClasses(kryo: Kryo) {

    // register adam's requirements
    akr.registerClasses(kryo)

    // ... register any other classes I need ...
  }
}
```

1.1.27 Core Algorithms

Read Preprocessing Algorithms

In ADAM, we have implemented the three most-commonly used pre-processing stages from the GATK pipeline (DePristo et al. 2011). In this section, we describe the stages that we have implemented, and the techniques we have used to improve performance and accuracy when running on a distributed system. These pre-processing stages include:

- *Duplicate Removal:* During the process of preparing DNA for sequencing, reads are duplicated by errors during the sample preparation and polymerase chain reaction stages. Detection of duplicate reads requires matching all reads by their position and orientation after read alignment. Reads with identical position and orientation are assumed to be duplicates. When a group of duplicate reads is found, each read is scored, and all but the highest quality read are marked as duplicates. We have validated our duplicate removal code against Picard (The Broad Institute of Harvard and MIT 2014), 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.[2] Specifically, because Picard's traversal engine is restricted to processing linearly sorted alignments, Picard mishandles these alignments. Since our engine is not constrained by the underlying layout of data on disk, we are able to properly handle chimeric read pairs.

- *Local Realignment:* In local realignment, we correct areas where variant alleles cause reads to be locally misaligned from the reference genome.[3] In this algorithm, we first identify regions as targets for realignment. In the GATK, this identification is done by traversing sorted read alignments. In our implementation, we fold over partitions where we generate targets, and then we merge the tree of targets. This process allows us to eliminate the data shuffle needed to achieve the sorted ordering. As part of this fold, we must compute the convex hull of overlapping regions in parallel. We discuss this in more detail later in this section. 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.
- *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, we label each base that we have sequenced with an error covariate. For each covariate, we count the total number of bases that we saw, as well as the total number of bases within the covariate that do not match the reference genome. From this data, we apply a correction by estimating the error probability for each set of covariates under a beta-binomial model with uniform prior. We have validated the concordance of our BQSR implementation against the GATK. Across both tools, only 5000 of the 180B bases ($< 0.0001\%$) in the high-coverage NA12878 genome dataset differ. After investigating this discrepancy, we have determined that this is due to an error in the GATK, where paired-end reads are mishandled if the two reads in the pair overlap.
- *ShuffleRegionJoin Load Balancing:* Because of the non-uniform distribution of regions in mapped reads, joining two genomic datasets can be difficult or impossible when neither dataset fits completely on a single node. To reduce the impact of data skew on the runtime of joins, we implemented a load balancing engine in ADAM's ShuffleRegionJoin core. This load balancing is a preprocessing step to the ShuffleRegionJoin and improves performance by 10–100x. The first phase of the load balancer is to sort and repartition the left dataset evenly across all partitions, regardless of the mapped region. This offers significantly better distribution of the data than the standard binning approach. After rebalancing the data, we copartition the right dataset with the left based on the region bounds of each partition. Once the data has been copartitioned, it is sorted locally and the join is performed.

In the rest of this section, we discuss the high level implementations of these algorithms.

1.1.28 BQSR Implementation

Base quality score recalibration seeks to identify and correct correlated errors in base quality score estimates. At a high level, this is done by associating sequenced bases with possible error covariates, and estimating the true error rate of this covariate. Once the true error rate of all covariates has been estimated, we then apply the corrected covariate.

Our system is generic and places no limitation on the number or type of covariates that can be applied. A covariate describes a parameter space where variation in the covariate parameter may be correlated with a sequencing error. We provide two common covariates that map to common sequencing errors (Nakamura et al. 2011):

- *CycleCovariate:* This covariate expresses which cycle the base was sequenced in. Read errors are known to occur most frequently at the start or end of reads.
- *DinucCovariate:* This covariate covers biases due to the sequence context surrounding a site. The two-mer ending at the sequenced base is used as the covariate parameter value.

[2] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al (Li and Durbin 2010).
[3] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al (DePristo et al. 2011).

To generate the covariate observation table, we aggregate together the number of observed and error bases per covariate. The two algorithms below demonstrate this process.

$read \leftarrow$ the read to observe
$covariates \leftarrow$ covariates to use for recalibration
$sites \leftarrow$ sites of known variation
$observations \leftarrow \emptyset$
for $base \in read$
  $covariate \leftarrow identifyCovariate(base)$
  if isUnknownSNP$(base, sites)$
  $observation \leftarrow Observation(1, 1)$
  else
  $observation \leftarrow Observation(1, 0)$
$observations$.append$((covariate, observation))$
return$observations$

$reads \leftarrow$ input dataset
$covariates \leftarrow$ covariates to use for recalibration
$sites \leftarrow$ known variant sites
$sites$.broadcast()
$observations \leftarrow reads$.map$(read \Rightarrow$ emitObservations$(read, covariates, sites))$
$table \leftarrow observations$.aggregate(CovariateTable(), mergeCovariates)
return$table$

The `Observation` class stores the number of bases seen and the number of errors seen. For example, `Observation(1, 1)` creates an `Observation` object that has seen one base, which was an erroneous base.

Once we have computed the observations that correspond to each covariate, we estimate the observed base quality using the below equation. This represents a Bayesian model of the mismatch probability with Binomial likelihood and a Beta(1, 1) prior.

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

After these probabilities are estimated, we go back across the input read dataset and reconstruct the quality scores of the read by using the covariate assigned to the read to look into the covariate table.

1.1.29 Indel Realignment Implementation

Although global alignment will frequently succeed at aligning reads to the proper region of the genome, the local alignment of the read may be incorrect. Specifically, the error models used by aligners may penalize local alignments containing INDELs more than a local alignment that converts the alignment to a series of mismatches. To correct for this, we perform local realignment of the reads against consensus sequences in a three step process. In the first step, we identify candidate sites that have evidence of an insertion or deletion. We then compute the convex hull of these candidate sites, to determine the windows we need to realign over. After these regions are identified, we generate candidate haplotype sequences, and realign reads to minimize the overall quantity of mismatches in the region.

Realignment Target Identification

To identify target regions for realignment, we simply map across all the reads. If a read contains INDEL evidence, we then emit a region corresponding to the region covered by that read.

Convex-Hull Finding

Once we have identified the target realignment regions, we must then find the maximal convex hulls across the set 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:

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i$$
$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R}$$
$$\hat{R} \subset R$$

In our problem, we seek to find all of the maximal convex hulls, given a set of regions. For genomics, the convexity constraint is trivial to check: specifically, the genome is assembled out of reference contigs that define disparate 1-D coordinate spaces. If two regions exist on different contigs, they are known not to overlap. If two regions are on a single contig, we simply check to see if they overlap on that contig's 1-D coordinate plane.

Given this realization, we can define the convex hull Algorithm, which is a data parallel algorithm for finding the maximal convex hulls that describe a genomic dataset.

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

The `generateTarget` function projects each datapoint into a Red-Black tree that 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 that is described in the hull set merging algorithm.

$first \leftarrow$ first target set to merge
$second \leftarrow$ second target set to merge
if $first = \emptyset \wedge second = \emptyset$
  return $\emptyset$
else if $first = \emptyset$
  return $second$
else if $second = \emptyset$
  return $first$
return
  if last($first$) $\cap$ head($second$) $= \emptyset$
    return $first + second$
  else
    $mergeItem \leftarrow$ (last($first$) $\cup$ head($second$))
    $mergeSet \leftarrow$ allButLast($first$) $\cup\ mergeItem$
    $trimSecond \leftarrow$ allButFirst($second$)
    return mergeTargetSets($mergeSet, trimSecond$)

The set returned by this function is used as an index for mapping reads directly to realignment targets.

Candidate Generation and Realignment

Once we have generated the target set, we map across all the reads and check to see if the read overlaps a realignment target. We then group together all reads that map to a given realignment target; reads that do not map to a target are randomly assigned to a "null" target. We do not attempt realignment for reads mapped to null targets.

To process non-null targets, we must first generate candidate haplotypes to realign against. We support several processes for generating these consensus sequences:

- *Use known INDELs:* Here, we use known variants that were provided by the user to generate consensus sequences. These are typically derived from a source of common variants such as dbSNP (Sherry et al. 2001).
- *Generate consensuses from reads:* In this process, we take all INDELs that are contained in the alignment of a read in this target region.
- *Generate consensuses using Smith-Waterman:* With this method, we take all reads that were aligned in the region and perform an exact Smith-Waterman alignment (Smith and Waterman 1981) against the reference in this site. We then take the INDELs that were observed in these realignments as possible consensuses.

From these consensuses, we generate new haplotypes by inserting the INDEL consensus into the reference sequence of the region. Per haplotype, we then take each read and compute the quality score weighted Hamming edit distance of the read placed at each site in the consensus sequence. We then take the minimum quality score weighted edit versus the consensus sequence and the reference genome. We aggregate these scores together for all reads against this consensus sequence. Given a consensus sequence *c*, a reference sequence *R*, and a set of reads *mathbf{r}*, we calculate this score using the equation below.

$$
\begin{aligned}
q_{i,j} &= \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j \in \{0, \ldots, l_c - l_{r_i}\} \\
q_{i,R} &= \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j = \text{pos}(r_i|R) \\
q_i &= \min(q_{i,R}, \min_{j \in \{0,\ldots,l_c - l_{r_i}\}} q_{i,j}) \\
q_c &= \sum_{r_i \in \mathbf{r}} q_i
\end{aligned}
$$

In the above equation, *s(i)* denotes the base at position *i* of sequence *s*, and *l_s* denotes the length of sequence *s*. We pick the consensus sequence that minimizes the *q_c* value. If the chosen consensus has a log-odds ratio (LOD) that is greater than *5.0* with respect to the reference, we realign the reads. This is done by recomputing the CIGAR and MDTag for each new alignment. Realigned reads have their mapping quality score increased by 10 in the Phred scale.

1.1.30 Duplicate Marking Implementation

Reads may be duplicated during sequencing, either due to clonal duplication via PCR before sequencing, or due to optical duplication while on the sequencer. To identify duplicated reads, we apply a heuristic algorithm that looks at read fragments that have a consistent mapping signature. First, we bucket together reads that are from the same sequenced fragment by grouping reads together on the basis of read name and record group. Per read bucket, we then identify the 5' mapping positions of the primarily aligned reads. We mark as duplicates all read pairs that have the same pair alignment locations, and all unpaired reads that map to the same sites. Only the highest scoring read/read pair is kept, where the score is the sum of all quality scores in the read that are greater than 15.

1.1.31 ShuffleRegionJoin Load Balancing

ShuffleRegionJoins perform a sort-merge join on distributed genomic data. The current standard for distributing genomic data are to use a binning approach where ranges of genomic data are assigned to a particular partition. This approach has a significant limitation that we aim to solve: no matter how fine-grained the bins created, they can never resolve extremely skewed data. ShuffleRegionJoin also requires that the data be sorted, so we keep track of the fact that knowledge of sort through the join so we can reuse this knowledge downstream.

The first step in ShuffleRegionJoin is to sort and balance the data. This is done with a sampling method and the data are sorted if it was not previously. When we shuffle the data, we also store the region ranges for all the data on this

partition. Storing these partition bounds allows us to copartition the right dataset by assigning all records to a partition if the record falls within the partition bounds. After the right data are colocated with the correct records in the left dataset, we perform the join locally on each partition.

Maintaining the sorted knowledge and partition bounds are extremely useful for downstream applications that can take advantage of sorted data. Subsequent joins, for example, will be much faster because the data are already relatively balanced and sorted. Additional set theory and aggregation primitives, such as counting nearby regions, grouping and clustering nearby regions, and finding the set difference will all benefit from the sorted knowledge because each of these primitives requires that the data be sorted first.

- genindex
- search

CHAPTER 2

References

Armbrust, Michael, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, et al. 2015. "Spark SQL: Relational Data Processing in Spark." In *Proceedings of the International Conference on Management of Data (SIGMOD '15)*.

DePristo, Mark A, Eric Banks, Ryan Poplin, Kiran V Garimella, Jared R Maguire, Christopher Hartl, Anthony A Philippakis, et al. 2011. "A Framework for Variation Discovery and Genotyping Using Next-Generation DNA Sequencing Data." *Nature Genetics* 43 (5). Nature Publishing Group: 491–98.

Langmead, Ben, Michael C Schatz, Jimmy Lin, Mihai Pop, and Steven L Salzberg. 2009. "Searching for SNPs with Cloud Computing." *Genome Biology* 10 (11). BioMed Central: R134.

Li, Heng, and Richard Durbin. 2010. "Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform." *Bioinformatics* 26 (5). Oxford Univ Press: 589–95.

Massie, Matt, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. 2013. "ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing." UCB/EECS-2013-207, EECS Department, University of California, Berkeley.

McKenna, Aaron, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, et al. 2010. "The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data." *Genome Research* 20 (9). Cold Spring Harbor Lab: 1297–1303.

Melnik, Sergey, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. "Dremel: Interactive Analysis of Web-Scale Datasets." *Proceedings of the VLDB Endowment* 3 (1-2). VLDB Endowment: 330–39.

Nakamura, Kensuke, Taku Oshima, Takuya Morimoto, Shun Ikeda, Hirofumi Yoshikawa, Yuh Shiwa, Shu Ishikawa, et al. 2011. "Sequence-Specific Error Profile of Illumina Sequencers." *Nucleic Acids Research*. Oxford Univ Press, gkr344.

Nothaft, Frank A, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, et al. 2015. "Rethinking Data-Intensive Science Using Scalable Analytics Systems." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM.

Sandberg, Russel, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. 1985. "Design and Implementation of the Sun Network Filesystem." In *Proceedings of the USENIX Conference*, 119–30.

Schadt, Eric E, Michael D Linderman, Jon Sorenson, Lawrence Lee, and Garry P Nolan. 2010. "Computational Solutions to Large-Scale Data Management and Analysis." *Nature Reviews Genetics* 11 (9). Nature Publishing Group: 647–57.

Schatz, Michael C. 2009. "CloudBurst: Highly Sensitive Read Mapping with MapReduce." *Bioinformatics* 25 (11). Oxford Univ Press: 1363–69.

Sherry, Stephen T, M-H Ward, M Kholodov, J Baker, Lon Phan, Elizabeth M Smigielski, and Karl Sirotkin. 2001. "dbSNP: The NCBI Database of Genetic Variation." *Nucleic Acids Research* 29 (1). Oxford Univ Press: 308–11.

Smith, Temple F, and Michael S Waterman. 1981. "Identification of Common Molecular Subsequences." *Journal of Molecular Biology* 147 (1). Elsevier: 195–97.

The Broad Institute of Harvard and MIT. 2014. "Picard." http://broadinstitute.github.io/picard/.

Vavilapalli, Vinod Kumar, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. 2013. "Apache Hadoop YARN: Yet Another Resource Negotiator." In *Proceedings of the Symposium on Cloud Computing (SoCC '13)*, 5. ACM.

Vivian, John, Arjun Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, et al. 2016. "Rapid and Efficient Analysis of 20,000 RNA-Seq Samples with Toil." *BioRxiv*. Cold Spring Harbor Labs Journals.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing." In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI '12)*, 2. USENIX Association.

Zimmermann, Hubert. 1980. "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications* 28 (4). IEEE: 425–32.