Evaluating Cloud Frameworks on Genomic Applications

Michele Bertoni*, Stefano Ceri*, Abdulrahman Kaitoua* and Pietro Pinoli*
**Department of Electronics, Information and Bioengineering*
Politecnico di Milano, Milano, Italia
Email: firstname.lastname@polimi.it

***Abstract*—We are developing a new, holistic data management system for genomics, which uses cloud-based computing for querying thousands of heterogeneous genomic datasets. In our project, it is essential to leverage upon a modern cloud computing framework, so as to encode our query expressions into high-level operations provided by the framework.**

After releasing our first implementation using Pig and Hadoop 1, we are currently targeting Spark and Flink, two emerging frameworks for general-purpose big data analytics. While Spark appears to have a stronger critical mass, Flink supports high-level optimization for data management operations; both systems appear suited to support our domain-specific data management operations.

In this paper, we focus on a comparison of the two frameworks at work based upon three typical genomic applications, stemming from our data management requirements and needs; we describe the coding of the genomic applications using Flink and Spark, discuss their common aspects and differences, and comparatively evaluate the performance and scalability of the implementations over datasets consisting of billions of genomic regions.

***Keywords*-Cloud frameworks for big data management; encoding genomic applications using Flink and Spark; comparative performance evaluation of Flink and Spark.**

I. Introduction

Next Generation Sequencing (NGS) is a technology for reading the DNA that is changing biological research and will change medical practice; thanks to the availability of millions of whole genome sequences, genomic data management may soon become the biggest and most important "big data" problem of mankind, and bringing genomics to the cloud is becoming more and more essential [21]. In this context, we are currently developing a new, holistic approach to genomic data modelling and querying that uses cloud-based computing to manage heterogeneous data produced by NGS technology [15]. Our approach is based on a new, high-level query language, called GenoMetric Query Language (GMQL), which enables building new datasets from a repository of existing datasets, using algebraic operations.

Data management in genomics applies to three different phases. *Primary analysis* is concerned with producing raw data in the form of short reads of DNA or RNA sequences; *secondary analysis* is concerned with extracting the DNA or RNA sequences from the reads (alignment) or evaluating (or *calling*) specific features from aligned files (e.g., mutations or peaks of expression); this processing is performed by a large number of bio-informatic tools, some developed by using Pig [19] or Spark [26]. Data management systems developed so far concentrate on secondary analysis (e.g., [14], [22]); Adam [16], an offset of Spark dedicated to genomics, is also focused on secondary analysis. GMQL is concerned with *ternary analysis*, i.e. querying thousands of samples of processed data, which are being assembled by international consortia (such as ENCODE [10], TCGA [24], and 1000 Genomes Project [1]). The spin-off company *Paradigm4*, founded by this year's Turing award Mike Stonebraker, also focuses on tertiary analysis, by developing genomic adds-on to SciDB [2]; they provide access to data from TCGA and 1000 Genomes Project, but they advocate the use of specialized databases rather than cloud computing.

In our initial release, available for download[1], we translated GMQL to PIG [5]; we are currently working towards a new GMQL release, that will be available in 2016, and will support two parallel implementations, respectively using Flink [4] and Spark [6], two emerging data frameworks. Architecture design is driven by strong platform portability requirements: the two implementations differ only in the encoding of about twenty GMQL language components, while the compiler, logical optimizer, and APIs/UIs are independent from the adoption of either framework.

In this paper, we focus on the comparison of Flink and Spark at work on genomic queries. We describe the encoding of three components of our query language; we discuss the many common aspects and few differences of their implementations, and evaluate their performance and scalability. A thorough comparison of Spark and Flink has recently appeared [20], but without a focus on genomics; it uses classic algorithms of wordcount, pagerank, K-Means, and a relational query, which are prototypical applications for the two frameworks. We instead consider operations upon genomic regions, such as building a region histogram, mapping regions to known annotations (e.g., genes), and joining overlapping regions.

We built a big data benchmark with a large dataset of regions and samples (the *very large* configuration consists of 2.5 billion regions scattered over 5 thousand samples.) Indirectly, this paper demonstrates that both Flink and Spark are capable of managing such huge workload, and therefore they

[1]http://www.bioinformatics.deib.polimi.it/genomic_computing/

qualify as relevant candidates for hosting ternary genomic data analysis.

The organization of this paper is as follows. Section 2 briefly introduces Flink and Spark; Section 3 explains three genomic applications; and Section 4 provides their benchmark. Conclusions summarize our findings and compare them to those of [20].

II. Platform Features

Flink and Spark are both motivated by providing high-level data processing operators and making a more efficient use of resilient memory as compared with low-level map-reduce programming. They are both Apache projects; in the open source data processing landscape they are located at the same level, together with other data processing engines (like Storm, Tez, and the standard Hadoop Map-Reduce framework.) All these frameworks rely on lower-level distributed resource manager (like YARN and HDFS) and they offer APIs to higher-level applications (like Hive or Pig.) Thus we can say that they are in part integrated with and in part alternative to the Hadoop framework. They both can be executed in a variety of ways (standalone, on Hadoop, Mesos) and/or access several data sources (including Cassandra and HBase).

A. Flink Foundations

Flink was developed as a cooperative project within Technical University (TU), Humboldt University (HU) and Asso Plattner Institute (HPI) [3], [11], [23]. It is now developed as an open-source Apache project, with more than 125 contributors, with most action concentrated within the Data Artisans spinoff. Its programming model is based on the notion of DataSet, that can be constructed from collections (lists, sets, arrays) or from external sources (files, databases). DataSets are transformed by operators, which apply to DataSets and return one DataSet, currently: *Map, flatMap, mapPartition, sortPartition, hashPartition, partitionCustom, Reduce, Rebalance, Filter, Union, Cross, coGroup, combineGroup, reduceGroup, firstN, project, aggregate, deltaIteration, bulkIteration*. Their names clearly recall algebraic data manipulations, and indeed each operation performs a high-level transformation upon DataSets. The distinguishing aspects of Flink are:

- Transparent use of persistent memory management: Flink starts by operating in memory, and splits data to disk based on need, with custom object serializer for Flink operations.
- Use of high-level optimization, based upon equivalence transformations applicable to job graphs (derived from program operators). Transformations produce an optimal join graph based on a cost model; as a consequence, the Flink programmer should not be concerned about low-level implementation of operators.
- Use of two kinds of iterators within program workflows. The bulk iterator applies to complete DataSets, the delta iterator applies to the new items added to a DataSet during the last iteration. Iteration allows optimizing flows, in particular to use suitable data formats and pipelining between two consecutive graph operations, omitting useless data transformations.
- Use of streaming processes as true streams, by means of pipelines, which apply to streams and move incoming data to operators as soon as they arrive, thereby allowing flexible window operations on streams.

Flink is presented as a *streaming engine* because it is able to send data from one operation to the next tuple by tuple, without executiong computations on batches as atomic units of work. This feature is used also in the batch processing, since batches are considered as a finite sets of streaming data. Iteration is particularly useful for implementing machine-learning algorithms [20], like K-Means, where the same block of instructions that calculates the centroids is executed many times over the same dataset of points. Iteration is also used by several join and cross methods.

B. Spark Foundations

Spark was initially developed at Berkeley University as part of the AMP (Algorithms, Machines, People) [2] and became an open-source project in 2009; it is now a much larger Apache project comparatively to Flink, with more than 400 developers from over 50 companies [25], [27], [28]; development of commercial products for deploying Spark-based solutions is carried over by the Databricks spinoff.

The programming model of Spark is based on an abstraction called *resilient distributed datasets* (RDDs); each RDD holds the data objects in memory, whereas conventional MapReduce systems read data from stable storage (e.g. the distributed file system) and write it back to stable storage, incurring significant cost for loading the data and writing it back at each stage. Internally, the Spark engine receives an operator DAG of RDD objects, then the *DAG Scheduler* takes care of partitioning them so as to support parallelism, and the *Task Scheduler* launches tasks and manages task failures in a way that is agnostic to the content of tasks: finally, *Workers* execute individual tasks. Optimizations of operations consist in selecting algorithms based on the partitioning option that minimizes data transfer between workers.

Spark includes a richer set of operators compared to Flink, but we find many operations with the same name[3], including *Map, flatMap, mapPartition, Reduce, Repartition, Filter, Union, cartesian, coGroup, SortByKey, CountByKey*; The above operations are also denoted as *transformations*,

[2]https://amplab.cs.berkeley.edu/

[3]Obviously, operations with the same name may show subtle semantic differences

as they produce RDDs from either RDDs or input files, whereas other operations are denoted as *actions*, as they do not produce RDDs, but instead they either pass a result set to the embedding program or write data to the disk. The distinguishing aspects of Spark are:

- Support of declarative, SQL-like queries through the *Spark SQL* version, that supports structured queries over distributed dataset (RDD), with integrated APIs in Python, Scala, Java and R. The tight integration allows injecting SQL queries within complex analytic algorithms.
- Support of a rich set of operations based on key-value pairs (e.g. sortByKey, reduceByKey, countByKey, aggregateByKey).
- Support of checkpointing of operations [27] that provides the ability to rebuild lost data on failure using lineage: each RDD remembers how it was built from other datasets and can recompute its values from the last checkpoint.

It is important to note that Spark lacks of explicit iteration operators, while it dedicates several operators to key-based computations, including sorting, counting, grouping and reducing; this makes Spark particularly suited to implement classic key-based map-reduce algorithms, such as WordCount. In our project, we make little use of key-based and iterator-based computations; hence the project is ideal for a fair comparison of the two systems. In the next section, we show three applications that are programmed in Flink and Spark by making use of the same (or very similar) operators.

III. Genomic Applications

Genomic applications discussed in this section share a common, simple model. Our algorithms deal with DNA regions, i.e. portions of the genome placed within one chromosome and further characterized by a start and stop position. Regions are aligned with respect to a reference genome (e.g. reference `h19` for the human species), therefore regions are related to each other by a single system of coordinates. We can think of a region as defined by the triple `<chromosome,start,stop>`, e.g. `<7,145677,678999>`, with `start<stop`, and it is possible to define operations among them, such as taking two regions and computing their union, difference, and intersection; these operations produce a result when two regions overlap, i.e. if they are both on the same chromosome and one end of either regions is enclosed within the two ends of the other region. Every region may have a value describing the properties of the region, called the region's *features*; the value can be missing, or as simple as a single real number (e.g., if the region represents a portion of the DNA reacting to a given treatment using an antibody, the value may be an indication of the significance of the reaction), or as complex as the sequence of amino acids which are present in the region (i.e, a string of the letters `A`, `C`, `G`, `T` with a length equal to `stop-start`; each position - or basis - can also be associated with an encoded indication of the confidence of that specific position). Thus, the value of a region can be complex, and has to be associated with a given signature, which depends on the type of experiment producing it.

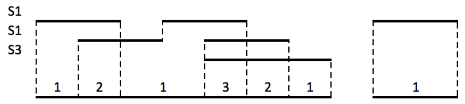

Figure 1. Histogram problem in genomics: accumulation index computation.

We collect the regions which are produced by each experimental condition (e.g. a specific cell culture) within a file or *sample*; we then collect the samples produced by the same experiment type (hence with the same signature) within the same *dataset*; each sample within a dataset has a unique identifier. In summary, each region is represented as a quintuple of values:

```
Sample = 1837
Chromosome = 1
Start = 15834
Stop = 16135
Values = []
```

We next present three classical examples of genomic applications. We present in greater details the first one, which is simpler and whose logics can be explained step by step.

A. Histogram

A classic operation in genomics is to compute the *accumulation index*, i.e. for each position in the genome the number of regions which overlap with that position; the operation applies to all the samples of a dataset. Programming this algorithm requires to partition the genome into intervals such that each interval ranges between the start and stop positions of the regions and it has a given accumulation index; see Fig. 1 with three input samples `S1, S2, S3`; the output is a sequence of regions on the whole genome with an associated accumulation index. In this case, the six input regions of the three samples have several overlaps, and the accumulation index ranges between 1 and 3.

A sequential algorithm for solving this problem consists of scanning the genome from left to right and maintain the accumulation count. Every time we meet the start of a region, we increment the count by one; conversely, every time the stop of a region is met, we decrease it. The result is made of all the consecutive couples of region ends (either starts or stops) between which the accumulation count is positive and does not change. In the following, we propose a parallel and distributed version of this algorithm, which relies on partitioning the genome into segments of identical length, called *bins*. For each chromosome, the *i-th* bin spans from $i * BIN\_SIZE$ to $(i+1) * BIN\_SIZE$. Binning

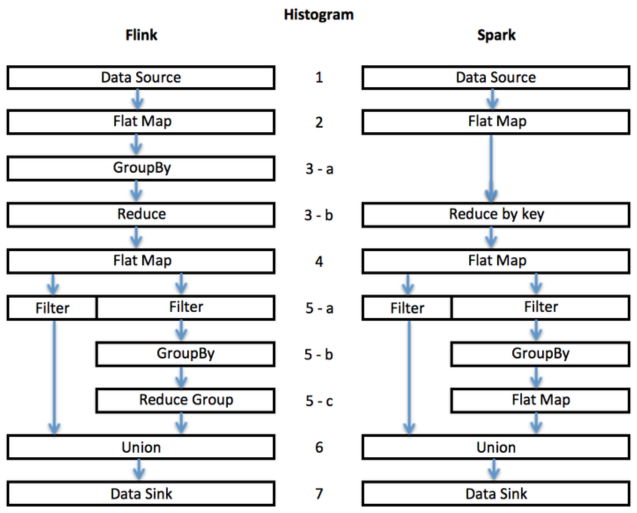

Figure 2. Operators for encoding the Histogram algorithm in Flink and Spark.

the genome has been introduced within the UCSC Genome Browser [13] in order to speed up the search for portions of the genome that must be loaded within the same browser window.

This algorithm can be programmed in Spark and Flink using very similar workflows of operators, hence it is an excellent benchmark; we start discussing the Flink implementation, supposing $BIN\_SIZE = 500$. Its high-level operation flow is shown in Fig. 2 on the left.

- Block 1 (Data Source) is responsible for reading the sample files, parsing them line-by-line, cast each value according to the desired type, and generate a unique dataset of regions. We show only three regions from sample `1` and chromosomes `1` and `2`, with the peculiarity that the first region stops where the second region starts and that the third region intersects with two bins.

```
id, chromosome, start, stop
1 chr1 154 237
1 chr1 237 450
1 chr2 460 600
```

- Block 2 (FlatMap) is responsible for the binning. For each region, it emits a new tuple for each bin it intersects. The output tuple contains the chromosome, the bin and a hash-map; in the hash map, we associate every region start with +1 and every region stop with -1. In the case a region crosses the border between two bins, we split it into two contiguous regions; one from the start to the border and one from the border to the stop (if the regions spans for more than two bins, the same procedure is repeated).

```
chromosome, bin, HashMap[Int,Int]
chr1 0 {154->+1, 237->-1}
chr1 0 {237->+1, 450->-1}
chr2 0 {460->+1, 500->-1}
chr2 1 {500->+1, 600->-1}
```

- Block 3 (GroupBy) is responsible of grouping the output dataset of the previous block by chromosome and bin. Then, on each partition an associative function is applied by the Reduce, which builds a single tuple for each chromosome and bin containing a hash-map with all the starts and stops of the regions in the bin; notice that in the worst case, the size of this hash-map is the same as the the length of the bin, therefore it fits in memory.

```
chromosome, bin, HashMap[Int,Int]
chr1 0 {154->+1, 237->0, 450->-1}
chr2 0 {460->+1, 500->-1}
chr2 1 {500->+1, 600->-1}
```

- Block 4 (FlatMap) returns the list of produced regions, along with their accumulation value, with each region placed within a bin, thus creating a raw histogram:

```
chromosome, start, stop, count
chr1 154  450  1
chr2 460  500  1
chr2 500  600  1
```

- Block 5 (Filter) starts with two filters that separate the regions properly contained in the bins (left filter) from the regions overlapping with bins (right filter). The latter regions must be merged when they are adjacent and with the same count. This processing requires a GroupBy and a ReduceGroup. In the specific example, the right filter is applied to the regions of chromosome 2, producing the region:

```
chromosome, start, stop, count
chr2 460 600 1
```

- Finally, Block 6 (Union) performs the union of the regions separately produced, and Block 7 (DataSink) writes them to the disk; it generates:

```
chromosome, start, stop, count
chr1 154 450 1
chr2 460 600 1
```

The Spark implementation slightly differs from the Flink implementation because it supports a ReduceByKey operation at step 3-b that makes the GroupBy operation at step 3-a unnecessary, and executes a FlatMap at step 5-c instead of a ReduceGroup (compare the left and right sides of Fig. 2.)

B. Mapping to a Reference

We next consider a similar problem to Histogram, which however compares two different datasets, that are called Reference and Experiment. Although the problem formulation is generic, one can think to Reference region as known annotations (e.g. genes, exons, introns, enhancers) and to Experiment as regions produced by NGS processing (e.g. peaks of expressions or mutations). This operation performs

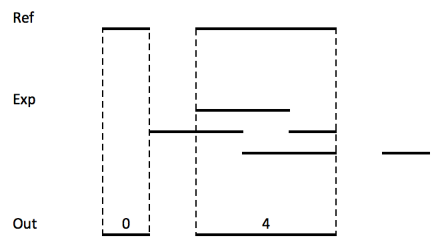

Figure 3. Mapping experiments to references in genomics.

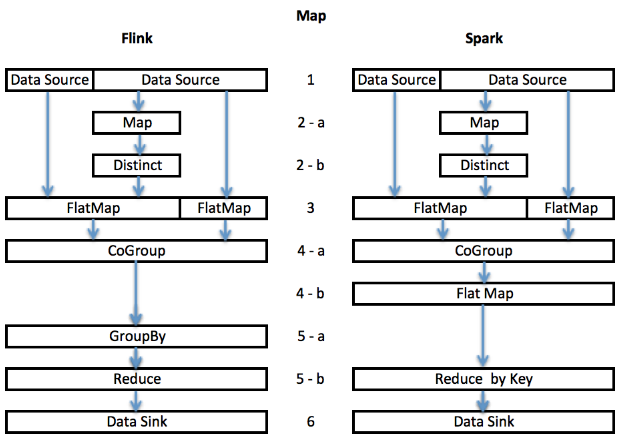

Figure 4. Operators for mapping experiments to references in Flink and Spark.

the intersection of Experiment samples over the Reference and then computes an aggregate over such intersection (e.g., counts for each reference region how many Experiments intersect with it). Intuitively, this operation allows one to observe the Experiment from the Reference point of view, e.g. counting how many mutations or how many peaks of expressions occur on given genes.

This behavior is explained in Figure 3, where we show a simple case consisting of one sample of Reference and one sample of Experiment, with overlapping regions, where we count the number of Experiment regions which intersect with each Reference region (e.g., the second region of the Reference intersects with 4 regions of Experiment and therefore its count is 4).

The encoding of this problem as a sequence of operations for Spark and Flink is shown in Fig. 4. The algorithm requires to bin the two datasets, to group them by sample pair, chromosome and binning, to compute intersections within the bins, to count them, and output the results for each sample pair. The complexity of this problem grows quadratically with the sizes of Experiments and References; in our benchmark, the reference is a single sample.

C. Join of Overlapping Regions

Finally, we compare Spark and Flink on the join of regions of different samples. We consider three datasets, filter two of them by simple predicates (e.g. on the region's `SCORE`), join the overlapping regions of the first two datasets and produce as result the union of those regions; then we join the resulting regions in the same way with the regions of a third dataset. Two regions satisfy the join predicate when they overlap, i.e. when the starting point of one region falls between the start and end point of the other one. Note that in most genomic applications joins have complex join conditions (they are *theta-joins*) and resulting tuples are assembled through region-based computations (such as the union of join operands).

Also in this case we use binning so as to parallelize the join operations along the genome. The binning procedure has some inherent difficulty when two joined regions spread over many bins. In such cases, the resulting region should be produced only by one of the bins, specifically the first one where the two regions overlap. This condition generalizes a binning method presented in [7].

In Flink, it is possible to apply a selection function while reading the input, and at the same time to assign a region to all the bins with which it overlaps; the code of the Map function is:

```
ds.flatMap {
(r: FlinkRegionTypeReduced,
 out: Collector[(Long, String, Long,
   Long, Array[Double], Int, Int)]) =>
 if (selection.fun(r._5(selection.index)))
  {val binStart = (r._3 / BIN_SIZE).toInt
   val binEnd = (r._4 / BIN_SIZE).toInt
   for (i <- binStart to binEnd) {
      out.collect((r._1, r._2, r._3, r._4,
          r._5, binStart, i)) }}}
```

Then, Flink supports only equi-join in the where clause (equal chromosome, equal bin), but it allows to put more conditions for theta-join (overlap) as much as construction of the result (by taking the union of regions) as internal predicates and constructors which are applied to matching regions, with a very compact code shown below.

```
leftDs.join(rightDs).where(1,6)
   .equalTo(1,6){
 (l, r, out : Collector[(Long, String,
   Long, Long, Array[Double])]) => {
   if((l._6.equals(l._7) ||
       r._6.equals(r._7)) &&
       (l._3 < r._4 && r._3 < l._4))
       {out.collect((l._1, l._2,
          Math.min(l._3, r._3),
          Math.max(l._4, r._4),
          l._5 ++ r._5)) }}}
```

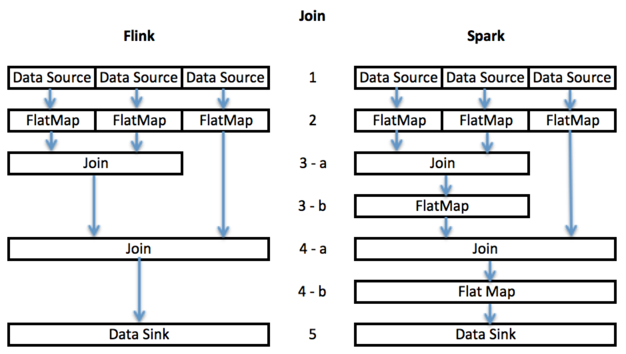

Figure 5. Operators for encoding the join algorithm in Flink and Spark

The resulting workflow is quite simple, and consists just of the cascading of the above operations for each pair of datasets, as shown in Fig. 5.

Spark supports joining on keys without additional internal predicates or constructor; therefore, each join of the application requires a couple of operations, a simple join on keys (chromosome and bin) followed by the application of a FlatMap operator to filter the results and keeping only overlapping regions, thus producing the output with two passes on the input rather than one; see Fig. 5. Besides this aspect, the Flink and Spark implementations are quite similar.

The Spark engine supports also *SQL Spark*, a more declarative and SQL-like dialect of Spark. In SQL Spark, variables are read to a data frame and can be given a schema; specifically, after reading the input data and binning, each dataset is independently defined as a table with the operation illustrated below:

```
binRegions(inputDS).toDF("ID","Chr",
   "Start","Stop", "Values","binstart",
   "bin").registerTempTable("ds")
```

At this point, SQL Spark supports SQL-like select-project-join operations, as follows:

```
val result =
  sqlContext.sql("SELECT * " +
   "FROM ds1 JOIN ds2 " +
   "ON ds1.Chr1 = ds2.Chr2
      AND ds1.bin1 = ds2.bin2 " +
   "WHERE ds1.Start1 < ds2.Stop2 " +
   "AND ds2.Start2 < ds1.Stop1 " +
   "AND (ds1.binstart1 = ds1.bin1
      OR ds2.binstart2 = ds1.bin1)")
```

This operation includes the selection and join but does not include the construction of resulting regions, which is produced by a FlapMap operation (as before); therefore, the operator flow of this second encoding is still the one represented in Fig. 5. In our benchmark, we found no significant difference in performance between the two encodings, most likely because are they internally mapped to the same operators.

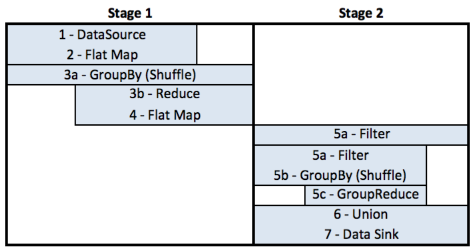

Figure 6. Stages of histogram computation in Flink.

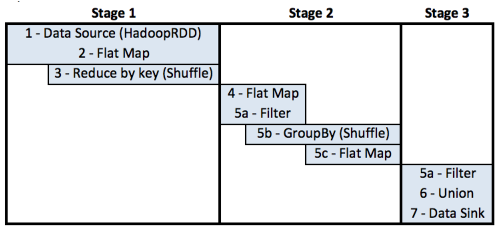

Figure 7. Stages of histogram computation in Spark.

IV. Benchmark

We performed our experiments on the Amazon Web Services (AWS) cloud, using a configuration with m3.2xlarge machines, each with 8 virtual CPUs, 30GB of memory, and 2 x80 GB of SSD storage. The testing setup contained one driver node and three configurations of slave nodes, set at 10, 15, and 19 nodes respectively. With 15 slave nodes, we set the number of executors to 120 giving $120/15 = 8$ executors per node; considering that the OS and Hadoop consume about 6GB of the node's memory, each executor had about 3GByte of memory. We used 80 executors in the case of 10 nodes and 152 executors in the case of 19 nodes, so that also in these cases we had 8 executors per node, with the same amount of available memory. With this setting, we observed (by using the Ganglia resource monitor[4]) that servers were fully used in terms of their CPU. We used Flink 0.9.0 and Spark 1.3.1.

A. Histogram execution

We start by comparing how the Flink and Spark engine manage the blocks of operations discussed in Section III-A. We observe that:

- Flink groups the blocks within two stages, that are sequentially executed. In particular, Blocks 1-4 belong to Stage 1, and Blocks 5-7 belong to Stage 2, as illustrated in Fig. 6.
- Spark groups the blocks within three stages, that are sequentially executed. In particular, Blocks 1-3 belong

[4]http://ganglia.sourceforge.net/

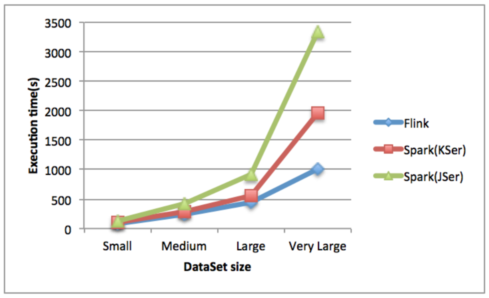

Figure 8. Execution time of the Histogram application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

to Stage 1, Blocks 4-5 belong to Stage 2, and Blocks 5a-6-7 belong to Stage 3, as illustrated in Fig. 7.

Note that Flink produces less stages, each with more operations; this is in general an advantage, because the end of stages typically require a synchronization, while inside stages operations run in parallel, yielding to greater parallelism.

Case	Size (GByte)	Regions	Samples
Small	4.1GB	100,947.792	200
Medium	21GB	509,237,187	1000
Large	43GB	1,034,186,018	2000
Very Large	105GB	2,556,236,090	5000

Table I
FEATURES OF THE DATASETS USED IN THE HISTOGRAM APPLICATION.

Next we discuss the experiments in terms of data sizes. We used the same experimental data for both the Histogram and Map application, and we designed four cases, respectively named *small*, *medium*, *large* and *very large*, whose dimensions are summarized in Table I. Regions are extracted from samples of Encode repository [10] but they are then redistributed to artificial samples so as to guarantee the availability of enough experimental data. Note that the *very large* setting includes 2.5 billions of regions, subdivided within 5000 datasets.

Engine	Small	Medium	Large	Very Large
Flink	78	250	446	1020
Spark (KSer)	101	277	554	1957
Spark (JSer)	122	420	916	3332

Table II
EXECUTION TIMES (IN SECONDS) FOR THE HISTOGRAM APPLICATION.

Execution times of the application in Flink and Spark with 15 slave nodes are reported in Table II, and graphically compared in Fig. 8. We note that Flink outperforms Spark, especially in the *very large* setting. In Spark, it is possible to change the data serializer, which can either be adapted to the data format or be generic. In our benchmark, we used both

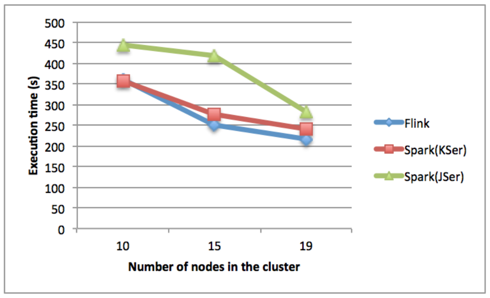

Figure 9. Execution time of the Histogram application in Flink and Spark, medium setting, with increasing nodes.

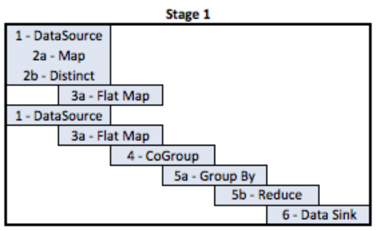

Figure 10. Stages of Map computation in Flink.

the default Java serializer and the Kryo general serializer[5]; performance was best with the latter choice, as shown in Fig. 8; in the *very large* setting, Flink outperforms Spark with Kryo serializer by a factor 2, and the Java serializer by a factor 3.

We next considered the *medium* setting and considered different cluster sizes, ranging from 10 to 19[6], see Fig. 9. In this setting, the two frameworks have very similar performance when Spark uses the Kryo serializer.

B. *Map execution*

We compare how the Flink and Spark engine manage the blocks of operations discussed in Section 3. We observe that Flink groups all the blocks within one stage (see Fig. 10), while Spark requires four stages (see Fig. 11). This is again an advantage for Flink in terms of less need for synchronization and greater parallelism.

[5]We will write serializers specifically suited to genomic data formats, but the general benchmark is best served by generic serializers.

[6]Our AWS configuration, covered by a grant, is limited to 20 nodes.

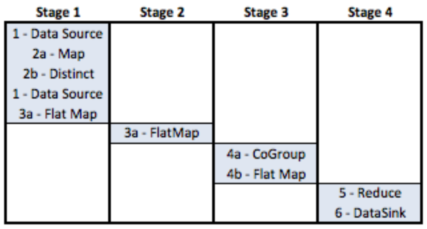

Figure 11. Stages of Map computation in Spark.

Engine	Small	Medium	Large	Very Large
Flink	53	178	281	652
Spark (KSer)	136	377	935	2154
Spark (JSer)	175	583	1049	2710

Table III
EXECUTION TIMES (IN SECONDS) FOR THE MAP APPLICATION.

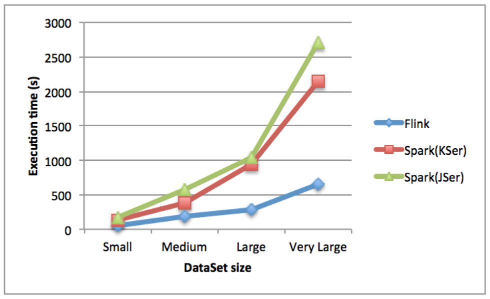

Figure 12. Execution time of the Map application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

For what concerns the reference file, we used the RefSeq genes, which amounts to 30,692 unique regions. Mapping thousands of experiments to the set of known genes is a biologically relevant query, allowing to quantitatively compare the genes in terms of their overall overlap with available peaks of expression. Execution times of the application in Flink and Spark are reported in Table III, and graphically compared in Fig. 12. In this application Flink outperforms Spark, showing execution times that are three to four times faster in all settings; Kryo serialization slightly outperforms the Java serialization.

We next considered the *medium* setting and considered again different cluster sizes, ranging from 10 to 19, see Fig. 13. Note that the difference in performances further increases with 10 nodes.

C. Join execution

We finally consider the Join application of Section III-C; we observe that Flink groups all the blocks within one stage (see Fig. 14), while Spark requires four stages, with stages 1 and 3 dedicated to loading data from the sources, stages 2 and 4 dedicated to joins, and with stage 3 in parallel with the sequence of stages 1 and 2 (see Fig. 15).

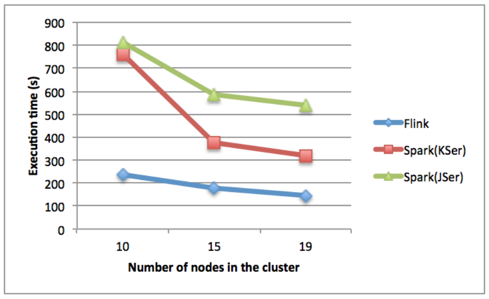

Figure 13. Execution time of the map application in Flink and Spark, medium setting, with increasing nodes.

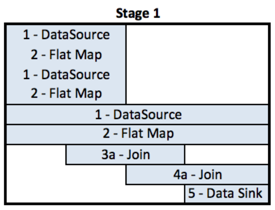

Figure 14. Stages of join computation in Flink.

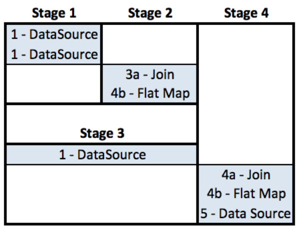

Figure 15. Stages of join computation in Spark.

Case	Size (GByte)	Regions	Samples
Small	1x3	39,424,000x3	1x3
Medium	2.5x3	98,560,000x3	1x3
Large	5x3	197,120,000x3	1x3

Table IV
FEATURES OF THE DATASETS USED IN THE JOIN APPLICATION.

For what concerns data sizes, we designed three cases, respectively named *small*, *medium*, and *large*, whose dimensions are summarized in Table IV; we generated three tables with very close numbers of regions. Regions have an attribute SCORE, used for the selection condition.

Engine	Small	Medium	Large
Flink	81	361	867
Spark	80	115	204

Table V
EXECUTION TIMES (IN SECONDS) FOR THE JOIN APPLICATION.

Execution times of the application in Flink and Spark with 15 slave nodes are reported in Table V, and graphically compared in Fig. 16; we used the Kryo serialization. In this application, Spark is faster than Flink, especially with the *large* setting (where it is 4 times faster). The better performances of Spark over Flink are confirmed by considering also the diagram showing how execution time decreases with increasing number of the nodes, fixing the input data to the medium case (see Fig. 17).

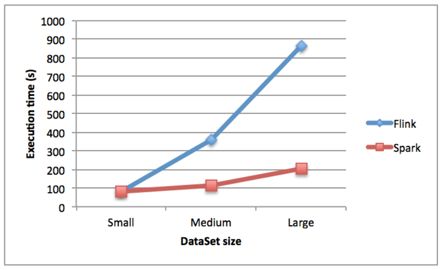

Figure 16. Execution time of the Join application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

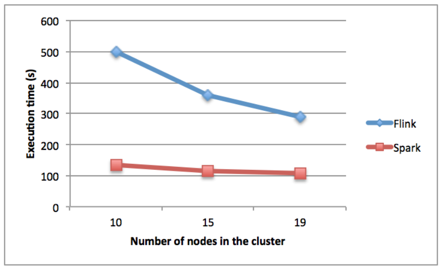

Figure 17. Execution time of the Join application in Flink and Spark, medium setting, with increasing nodes.

V. Conclusions

The result of the benchmark indicates that Flink was superior in performance in the Histogram and Map applications, featuring faster execution by a factor between 2 and 3 in the *very large* setting. We believe that this result is due to a better concurrent execution, that results in less sequential stages, and to a very efficient pipelining: more operators are chained within each stage, and data flows tuple by tuple

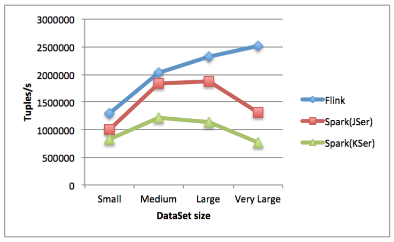

Figure 18. Rate of output tuples per second, Histogram implementation.

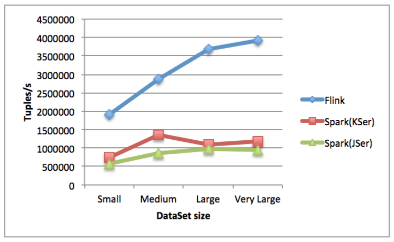

Figure 19. Rate of output tuples per second, Map implementation.

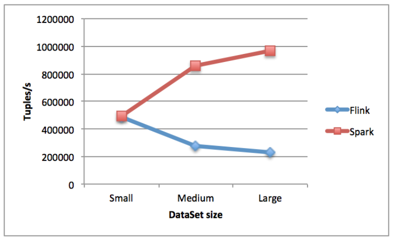

Figure 20. Rate of output tuples per second, Join implementation.

from one operator to the next, thanks to the Flink core that is optimized for streaming operation. Spark requires more stages, with extra data shuffling between them; pipelining is done block by block and not tuple by tuple. Figures 18 and 19 show the rate of result tuples produced per unit time and they show that such rate is constantly increasing for Flink and is instead decreasing with Spark, suggesting a reduction of efficiency possibly due to heavier pipelining with the growth of the data size.

We instead found that in the genomic Join application Spark outperformed Flink. This result was not expected after reading [20], where Flink was found superior than Spark on an equi-join query of two tables, but our result was confirmed also with changes to the execution contexts and datasets; the two systems have identical performance on the *small* setting, but Spark is faster by a factor 4 on the *large* setting. In genomics we expect queries to grow in the number of regions or of joined samples, but we do not expect queries with a large number of joins, hence our Join application benchmark is appropriate. Note also that Spark produces result tuples at a faster rate with increasing data sizes, whereas Flink rate reduces with size, as illustrated in Fig. 20.

We also note that with Spark it is possible to change the data serializer, while Flink does not support this option. Spark developers suggest using Kryo serializer that is faster and consuming less memory than the default Java serializer; this was confirmed in our tests, as shown in Fig. 8 and 12.

We complement our benchmark with a discussion of the other results presented in [20]. In that work, authors do not mention the version of Flink and Spark that was used for experiments. Spark was found superior on the WordCount application, which is essentially a simple, batch map/reduce sequence over a very large text body. Flink was instead found more efficient than Spark on graph processing algorithms (k-means clustering and pagerank), which both require a variable number of iterations prior to convergence. The discussion on the influence of such number of iterations was not conclusive, because the gap between the two systems as a function of the number of iterations was increasing in pagerank and decreasing in k-means clustering.

There are many aspects in which Flink shows to be at an earlier stage of its development. Flink versions change quite often and its documentation is at times incomplete, whereas Spark documentation appears well organized, with clear indications of how to embed Spark within Scala, Java, R, and Python. Currently Spark appears superior in terms of market penetration and partners credibility; the public interest on Spark on Internet is at much higher levels than the interest on Flink, as visually very clear by inspecting Google trends[7]. Thus, we believe that this performance-centred benchmark covers just one aspect of the confrontation among these two frameworks, and it is not the final word. Indeed, as the first benchmark between the two frameworks became available on June 25, 2015 [20], we expect more works of this kind to become available soon, and we also expect that the benchmarks will stimulate internal development, yielding to important performance improvements for both frameworks.

Our project's architectural choice, which includes a portable GMQL implementation to both Flink and Spark, appears even more strongly motivated after this benchmark; we believe that in the long run it will be a key feature of our genomic data management project.

Acknowledgment

The authors would like to thank Vahid Jalili, Marco Masseroli, Heiko Muller and Francesco Venco, who contributed to the design of GMQL. This work is supported by the PRIN Project GenData 2020. We also acknowledge a donation of 10.000 US $ from AWS, used for our benchmark.

References

[1] The 1000 Genomes Consortium, An integrated map of genetic variation from 1,092 human genomes. *Nature* 491, 5665, November 2012.

[2] Anonymous paper, Accelerating bioinformatics research with new software for big data to knowledge (BD2K), Paradigm4, April 2015 (downloaded from: www.paradigm4.com, June 2015.)

[3] A. Alexandrov et al. The Strathosphere platform for big data analytics. *VLDB Journal* 23(6), 939-964, 2014.

[4] Apache Flink. http://flink.apache.org/

[5] Apache Pig. http://pig.apache.org/

[6] Apache Spark. http://spark.apache.org/

[7] B. Chawda et al. Processing Interval Joins On Map-Reduce *Proc. EDBT*, 463-474, March 2014.

[8] J. Ekanayake et al. MapReduce for data intensive scientific analyses. In *Proc. IEEE eScience*, 277-284, 2008.

[7]https://www.google.com/trends/explore#q=Apache Spark %2C Apache Flink

[9] B. Elser et al., An evaluation study of bigdata frameworks for graph processing, *IEEE Big Data Conference*, 2013, 60-67.

[10] ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57-74, 2012.

[11] S. Ewen et al., Spinning fast iterative data flows. *PVLDB 2012*, 1268-1279.

[12] Hadoop 2. http://hadoop.apache.org/docs/stable/

[13] W.J. Kent, The human genome browser at UCSC. *Genome Res.*, 2002 Jun;12(6):996-1006.

[14] C. Kozanitis et al. Using Genome Query Language to uncover genetic variation. *Bioinformatics* 30(1):1-8, 2014.

[15] M. Masseroli et al. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*, 2015, doi: 10.1093/bioinformatics/btv048.

[16] M. Messie et al. ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing, Technical Report No. UCB/EECS-2013-207, Berkeley Univ., December 2013.

[17] C. Olston et al. Pig Latin: A not-so-foreign language for data processing. *ACM-SIGMOD*, 1099-1110, 2008.

[18] U. Röhm and J. Blakeley. Data management for high-throughput genomics. In *Proc. CDIR*, 1-10, 2009.

[19] A. Schumacher et al. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119-120, 2014.

[20] N. Spangelberg et al. Evaluating New Approaches for Big Data Analytics Frameworks, *BIS 2015*, LNBIP 18, Springer-Verlag, June 25, 2015.

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

[22] S. Tata et al. Periscope/SQL: Interactive exploration of biological sequence databases. In *Proc. VLDB*, 1406-1409, 2007.

[23] F. Hueske et al. Opening the black boxes in dataflow optimization. *PVLDB 2012*, 1256-1267.

[24] J. N. Weinstein et al. The Cancer Genome Atlas Pan-Cancer analysis project. *Nat Genet.*, 45(10):1113-1120, 2013.

[25] R. Xin et al. Shark: SQL and Rich Analytics at Scale. In *Proc. ACM-SIGMOD*, June 2013.

[26] M. S. Weiwiorka et al. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652-2653, 2014.

[27] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 15-28, 2012.

[28] M. Zaharia et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. SOSP*, November 2013.