Scalable Genomic Data Management System on the Cloud

Abdulrahman Kaitoua*, Andrea Gulino, Marco Masseroli*, Pietro Pinoli*, Stefano Ceri*
Dipartimento di Elettronica, Informazione e Bio-Ingegneria
Politecnico di Milano, Piazza L. da Vinci 32, Milano Italy
Email: andrea.gulino@mail.polimi.it , first.last@polimi.it (*)

***Abstract*—Thanks to the huge amount of sequenced data that is becoming available, building scalable solutions for supporting query processing and data analysis over genomics datasets is increasingly important. This paper presents GDMS, a scalable Genomic Data Management System for querying region-based genomic datasets; the focus of the paper is on the deployment of the system on a cluster hosted by CINECA.**

***Keywords*—Genomic computing, big data processing, data management on the cloud, system architecture.**

I. Introduction

Thanks to Next Generation Sequencing, a recent technological revolution to read the DNA, a huge number of genomic datasets have become available within large reserach centers and hospitals; several collections of biological data, generated by large consortia, are available for public use [1], [2], [3]. Many laboratories worldwide are currently focused on building computationally expensive pipelines to extract genomic signals (e.g. revealing DNA mutations, expressions or regulation) from raw data at the end of sequencing machines; this is regarded as *primary and secondary data analysis for genomics*. In the GeCo Project[1] we are instead interested in the integration of heterogeneous signals involving thousands of biological conditions and millions of regions, to answer relevant biological and clinical questions. This problem is called *tertiary data analysis for genomics* and is becoming hot, with few other systems being developed very recently; among them, FireCloud [18], SciDB [9] and DeepBlue [4].

So far, we have defined GenoMetric Query Language (GMQL), a language for querying and analysing genomic datasets [8]. We have also developed two engines that support GMQL implementations; the first engine, described in [6], is based on Hadoop 1 and targeted to the Pig language [14], the second engine, described in [7], is based on Hadoop 2 [16] and targeted to Flink [13] and Spark [17]. This paper is focused on the description of the ongoing deployment of the latter engine to a cluster architecture hosted at CINECA [2] which uses the Spark implementation. Prior to do so, we provide minimal information about what is a query within our system and how a query is translated to code that is executed on a cloud; we also describe our public and private repository structure.

[1]Data-Driven Genomic Computing, ERC Adv. Grant, 2016-2021

[2]CINECA is a not-for-profit Consortium whose members include 70 Italian Universities, 6 national research consortia, and the Ministry of Education, Universities and Research of Italy (MIUR).

A. GMQL Query

GenoMetric Query Language (GMQL) is a bioinformaticians-focused query language enabling queries over heterogeneous datasets, each consisting of thousands of samples; we provide several interfaces to submit queries, including a Web interface and a command-line interface (we are currently working on provisioning Python and R interfaces to GMQL). We next show a simple GMQL query:

```
GENES = SELECT() ANNOTATIONS;
PEAKS = SELECT() BED_PEAKS;
MAPPED = MAP() PEAKS GENES;
SELECTED = SELECT(Count_PEAKS_GENES>0) MAPPED;
RELEVANT = COVER(1,2) SELECTED;
MATERIALIZE RELEVANT INTO OUTPUT;
```

The query consists of 5 operations which returns into `Output` regions responding to a specific biological problem. The query performs both classic relational operations and domain-specific ones.

B. Query Translation

In the example above, `BED_PEAKS` used in the first operation is a dataset which includes thousands of samples, each one representing a signal extracted from the reading of DNA in a specific biological or clinical condition; each GMQL operation is implicitly mapped to all the samples of a dataset. Each sample, in turn, includes both genomic regions (typically in the orders of thousands to millions) and metadata (typically in the order of hundreds to thousands); therefore, each GMQL operation applies to two distinct data structures, one for metadata and one for regions. A syntax-directed translator produces a description of the query as **operator DAG**, i.e. a directed acyclic graph whose nodes represent operations to be performed on the data and arcs represent precedences among the operations; the operator DAG corresponding to the above query is show in Fig. 1. The process of translation is discussed elsewhere [7]; for the purpose of this paper, it is important to understand that GMQL programs are mapped to low-level operations over two kinds of data structures, called *regions* (RD, represented in red) and *metadata* (MD, represented in blue). Low-level operations are similar to relational operations (e.g. select, join, combine, collapse) or instead are domain-specific (e.g. map, cover).

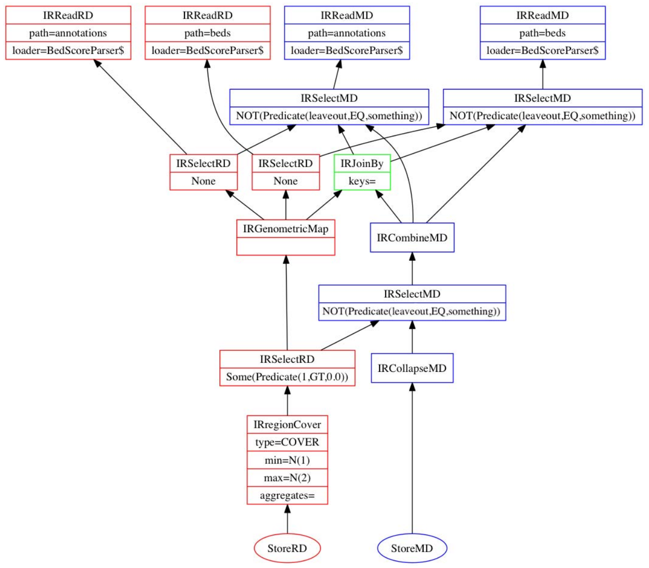

Fig. 1. Example of operator DAG for a GMQL query

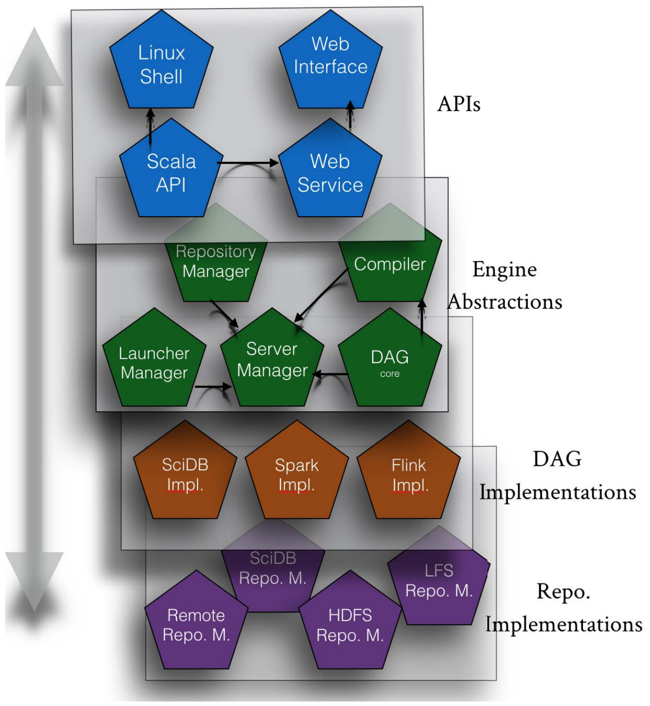

Fig. 2. GDMS Architecture.

II. Software Architecture

We next briefly describe the software architecture of **Genomic Data Management System (GDMS)**, a scalable data manager for genomic data; the architecture, described as well in [7], includes the engine and the repository. GeCo software is available at https://github.com/DEIB-GECO/GMQL.

A. Engine

The engine is organized according to a four layer architecture, as shown in figure 2:

- The **Access Layer**, supporting:
  - Intermediate Representation APIs.
  - Shell command line interface.
  - Web Services.
  - A user-friendly Web Interface.
- The **Engine Components**, including:
  - GMQL Compiler, for compiling a GMQL query into a DAG (which embodies execution plans).
  - DAG Manager, for supporting the creation and dispatching of DAG operations to other components.
  - Server Manager, for managing multi-user execution and their access capabilities.
  - Repository Manager, for managing the access to the repository.
  - Launches Manager, for launching the executions of implementations. Currently, we have five launchers; Local Launcher, Yarn Launcher, Remote Launcher, and SciDB Launcher.
- The **Implementation Components** (or executors), including the three implementations currently supported. Each package contains the implementation of the operations (abstract classes) of the DAG nodes.
  - Spark Implementation (the default option, and the most stable of the current implementations, based on [17]).
  - Flink Implementation (based on [13])
  - SciDB Implementation (based on [9]).
- The **Repository Implementations**, including:
  - Local File System (LFS) repository, used when the installation is for a single machine.
  - HDFS repository, used when Hadoop Distributed File System [16] is selected as the storage.
  - Remote File System (RFS), used in the cluster-based architecture as discussed next.

B. Repository

The Repository Manager is the system component in charge of storing and managing the datasets imported from external repositories or generated by an user as result of a query execution. We support a private repository for each user and a public, read-only repository shared by all the users, which contains datasets from open public collections, such as ENCODE [2], TCGA [3] and others.

Table I lists some of our public datasets and their size in GB. The complexity of the operations performed by GMQL on this data and the size of intermediate results that are computed during processing is remarkable. Therefore, it is fundamental to design an efficient and scalable solution for our engine, capable of distributing data and parallelizing computations appropriately; a solution adopting a cluster of machines is

TABLE I. PUBLIC DATASETS SIZE

Dataset	Samples	Size[GB]
HG19_TCGA_cnv	22632	756.8
HG19_ENCODE_CHIPSEQ_BEDGRAPH	159	571.3
HG19_ENCODE_BEDGRAPH	812	435.9
HG19_TCGA_dnaseq	6914	271,6
HG19_TCGA_dnamethylation	12552	207,6
...	...	...
Total	138,725	2,843

highly required, as discussed also in [11]. Such deployment is discussed in Section 4.

III. GDMS ON A SINGLE NODE

We briefly describe GDMS deployment on a single node; this was our first installation at CINECA, and has been operational for one year (April 2016 - April 2017). The system uses a one-machine configuration, deployed as a Virtual Machine consisting of 40 cores and 128 GB of RAM. The software architecture of GDMS on a single machine is shown in Fig. 3. We include the following modules from the conceptual architecture mentioned in Figure 2, starting from top to bottom:

- Web interface.
- Web services.
- Sever manager.
- Launcher manager (Local Launcher implementation).
- Spark Implementation.
- HDFS repository implementation.

The Server Manager in this case includes the compiler and the DAG abstractions. The Spark code is executed either from a Spark Shell Submit command - in this case a ready JAR (Java Archive) is launched - or by invoking the programmatic API of Spark. In the latter case, there is no need for Spark installation, and the execution is performed as multi-thread system, using the Spark Local mode. GDMS Local Launcher's implementation does not use Spark submit for execution; we use instead the Spark programmatic API to run Spark, using its native parallelism.

IV. GDMS DEPLOYMENT ON THE CINECA CLUSTER

We next describe the deployment of our engine to the CINECA BigInsights cluster; it consists of four nodes in which we use 16 cores per node and 46 GB of RAM per core. Thus, the distributed computing environment consists of a total of 64 cores and 184 GB of RAM.

We opted for a deployment strategy based on an application server and a cluster of machines for execution. Fig. 4 shows how the GMQL Engine deployment is split between the application server and the cluster of machines. The GMQL engine receives a GMQL script from the web service/web interface and compiles the script producing a DAG of the operations. The DAG is then serialized and sent from the GMQL application server to the cluster for processing. The application runs on a Virtual Machine equipped with 16 cores and 128 GB of RAM.

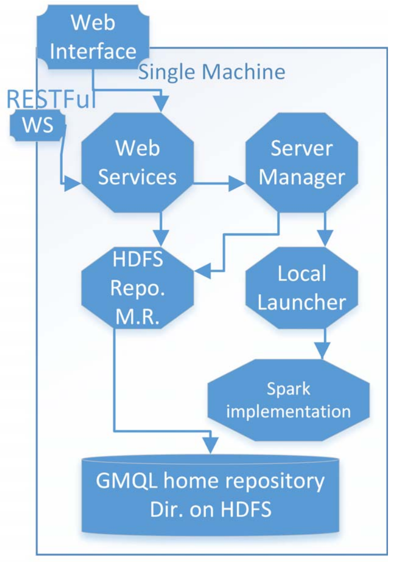

Fig. 3. GDMS single node installation.

A. Supporting Technologies

We use Knox [15] and Livy [19] to control GMQL on the cluster from the remote application server.

Apache Knox provides a REST API for the Apache Hadoop ecosystem. Apache Knox Gateway is designed as a reverse proxy to manage the authentication and the integration to several Hadoop related services. In the scope of this paper, we used Apache Knox to access the Hadoop Distributed File System (HDFS [16], [10]) through an authenticated RESTful web service.

The Repository Manager uses a local representation of the distributed repository containing references to the real dataset location. Moreover, compact representations of meta files and of the dataset schemas are replicated locally to provide fast access to the web interface. The Apache Knox package [15] is used to connect to the remote HDFS. This is suitable when we have Application server for GMQL web interface, and a remote Hadoop cluster for execution and data storage.

Livy [19], an open source project, is a REST interface that allows to interact with Spark, submitting batch jobs or running interactive shells. Livy also provides a mechanism to track the execution of submitted jobs through a log that describes the state of their execution.

B. GMQL Application Server

The GMQL Application server is responsible for:

- Opening GMQL to the world by exposing a set of RESTful web services and a web interface. Web Services allow to submit new GMQL jobs, track their executions and provide a set of operations for datasets management. The total number of web services is 43. The web interface uses the web services for performing all GMQL operations and to allow users to manage their datasets.

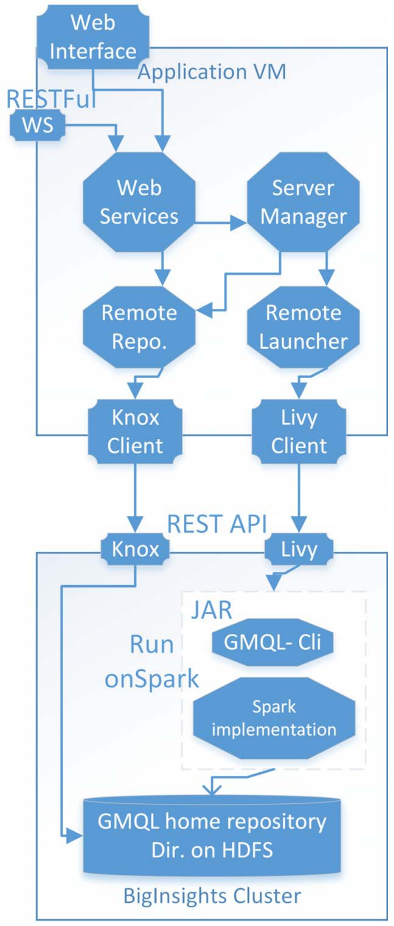

Fig. 4. CINECA installation.

- Compiling GMQL scripts. The GMQL compiler, which is imported in the server manager, compiles the provided GMQL script and generates a DAG representation both for region and meta operations defined in the script.
- Serializing the DAG and using the Livy Client to connect to the Livy web services and submit the job, passing the serialized DAG as parameter. Livy Client is a GMQL component used to communicate with Livy web services, providing the following functionalities:
  - Run a job.
  - Kill a job.
  - Get job status.
  - Get application name, that is the job application ID running on Yarn.
  - Retrieve Livy log and Yarn log.
- Managing users' authentication. GMQL is a multi-user system , supporting a local repository where each user can download her datasets and a read-only, global repository storing information from open data (TCGA, ENCODE, RoadMap Epigenomics).
- Managing the GMQL repository. Metadata are stored on the application machine (for web interface access) while the dataset data (region, meta and schema) are stored on the remote cluster for processing. The repository manager allows the addition, deletion, and update of datasets and samples and supports the uploading and downloading between the HDFS and the user's local system, by streaming data to/from the remote HDFS from/to the user. The repository manager controls the quota for each user and limits the access to public datasets.
- Keeping history of the execution. In particular, when the execution is invoked from the Web interface, logging information is returned to the interactive window, tracing the progression of the execution.

C. GMQL submission

The server manager, shown in Figure 4, is responsible for the GMQL script execution cycle. The script is submitted by a web service to the server manager specifying the user name; the associated GMQL job is created and registered in the server manager. Then, the server manager compiles the script producing the DAG and uses Livy client to submit the GMQL job to the remote server, performing a Livy batch submission operation with the following parameters:

- Spark parameters:
  - The GMQL Spark Implementation JAR (Java ARchive) location on the remote cluster.
  - The name of the main class of the JAR.
  - Number of Spark executors to run this job.
  - Memory size for each executor.
  - Memory size of the Spark Driver instance.
- GMQL JAR parameters:
  - The job serialized DAG.
  - Logging mode (verbose or info).
  - JobID, that is the GMQL job id for this job being executed.

The server manager monitors the execution of GMQL jobs through the Livy client and passively reports the execution status to the web services. When a job finishes its execution the server manager alerts the repository Manager to create a local dataset description of the new datasets in the repository. The repository manager connects to the HDFS on the remote cluster using Apache Knox and lists all the generated files to add their names to the new dataset description.

D. GMQL Cluster

Two sets of operations are performed on CINECA Cluster.

1) GMQL Execution: GMQL Jobs are submitted by Livy server. Livy executes the fat JAR mentioned in the parameters of the Livy call, which contains the library of GMQL Spark implementation along with all the dependencies. Livy uses the Command Line Interface of GMQL to submit the DAG and the job configurations to the job (GMQL parameters).

2) Data Storage: Input and result datasets are stored in the GMQL repository structure. GMQL repository is structured in a directory structure on Hadoop Distributed File System. A folder is added for each user of GMQL under the repository home directory.

E. Spark Implementation

In the Spark implementation, every operation of the DAG is mapped to several operations on Resilient Distributed Datasets (RDDs) [12], as discussed in [7]. This results in highly complex Spark dataflows. For example, the *map* and *cover* operations are mapped to the Spark dataflows illustrated in Figures 5 and 6.

Spark provides efficient parallel processing and the ability to run on a cluster on top of the Hadoop Distributed File System. Parallelism is achieved by running the same application in several independent processes (called executors), each running on a separate Java Virtual Machine (JVM) and using multiple threads.

In cluster mode, executors are allocated to different nodes and coordinated by a coordinator program (called driver program). Spark allows also to set the level of parallelism for each submitted application (e.g. setting the number of executors and the memory of both the driver and the executors).

F. Efficient Resource Allocation

Running on a cluster introduces additional latency that may become remarkable in some situations. For example, many Spark tasks, such as the join task, require data shuffling across the cluster. Depending on the network speed and on the amount of data that has to be moved, shuffling time may become the execution bottleneck, resulting in a loss of efficiency that hampers the increased efficiency due to parallelism. Trade-offs between data shuffling and parallelism relative to the Amazon Web Services (AWS) cloud are discussed in [7].

Resources must be allocated to each executor depending on the needs of each computation step, and such needs can be computed statically but in a query-specific way. By dimensioning resource allocation depending on the submitted job, we may allow more jobs in the cluster to be executed in parallel without lowering the cluster performance. For next releases of our GDMS, we are studying a way to provide a dynamic query-dependent Spark resource allocation. We will define a set of heuristics capable to predict the computational effort and the volume of data that the cluster must tolerate, starting from the GMQL query submitted by the user to the system.

V. Performance testing

We are currently completing the cluster installation at CINECA, so we have not yet a full set of performance figures. However, in [5] we have deployed the architecture discussed in this section 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.

Table II summarizes the dimension of four datasets that we used in the testing. Note that DS4 includes 2.5 billion regions over 5000 samples.

Dataset	Size (GByte)	Regions	Samples
DS1	4.1GB	100,947.792	200
DS2	21GB	509,237,187	1000
DS3	43GB	1,034,186,018	2000
DS4	105GB	2,556,236,090	5000

TABLE II. Sizes of the datasets used in the performance evaluation.

Tables III and IV show the performances of the system with a configuration using 15 nodes. Note that the map operation (similar to a join) over 1 billion regions and 2000 samples requires requires about 30 minutes; a similar execution time is required by the cover operation over 2.5 billion regions and 5000 samples.

Time(sec)	DS1	DS2	DS3	DS4
Spark exec.	136	377	935	2154

TABLE III. Execution times (in seconds) for the Map operation over 15 AWS nodes.

Time(sec)	DS1	DS2	DS3	DS3
Spark exec.	101	277	554	1957

TABLE IV. Execution times (in seconds) for the Cover operation on 15 AWS Nodes.

Table V applies to the cover operation and the DS2 dataset, and shows the scale-up of the system, whose performance improves with the rising of execution nodes, from 10 to 15 to 19; execution time reduces from 6 minutes to 3.5 minutes.

AWS Nodes	10	15	19
Spark Exec.	360	277	210

TABLE V. Scaling of execution times (in seconds) for the Cover operation and DS2 dataset.

VI. Conclusion

A single node based GMQL installation has been in use for about one year on a virtual machine (since April 2016); we used it mainly for research projects conducted by GeCo researchers in cooperation with their partners, including European Institute of Oncology (IEO), Harvard University (IACS) and National University of Singapore (NUS). This installation is tuned at best to support coordinated use of the server: when heavy computations are required, e.g. complex queries involving the integration of thousands of samples, users do not interfere with each other.

We are in these days completing the installation of our system in the cluster, where we will have access to more resources and we will also have control over them. By assigning query-specific resources we will be able to support multiple users at best, giving them adequate resources, so as to enable more substantial use of our integrated repository of open data, that is already available on CINECA server. This interaction will be based on a careful and cooperative management of computing resources (number of nodes and limitations on execution time for each user.)

Acknowledgment

This work is supported by the ERC Advanced Grant GeCo (Data-Driven Genomic Computing, 2016-2021). We like to thank CINECA for hosting our system since April 2016, and specifically Giorgio Pedrazzi for careful assistance.

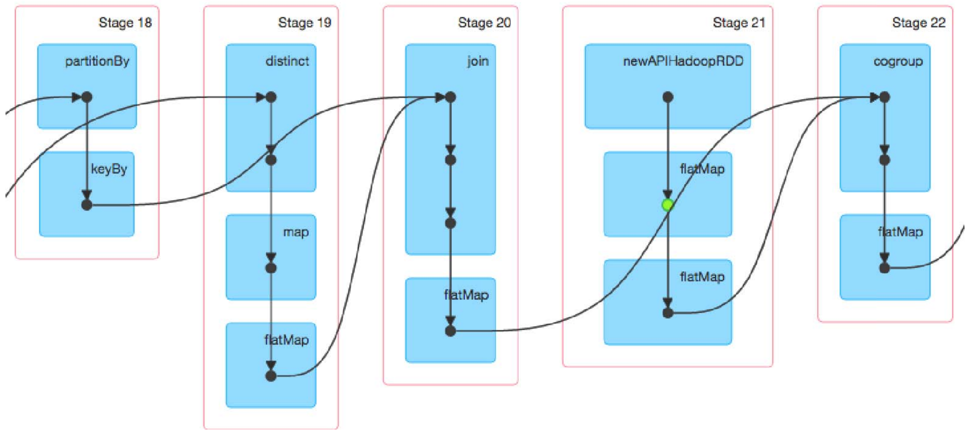

Fig. 5. Map operation in Spark.

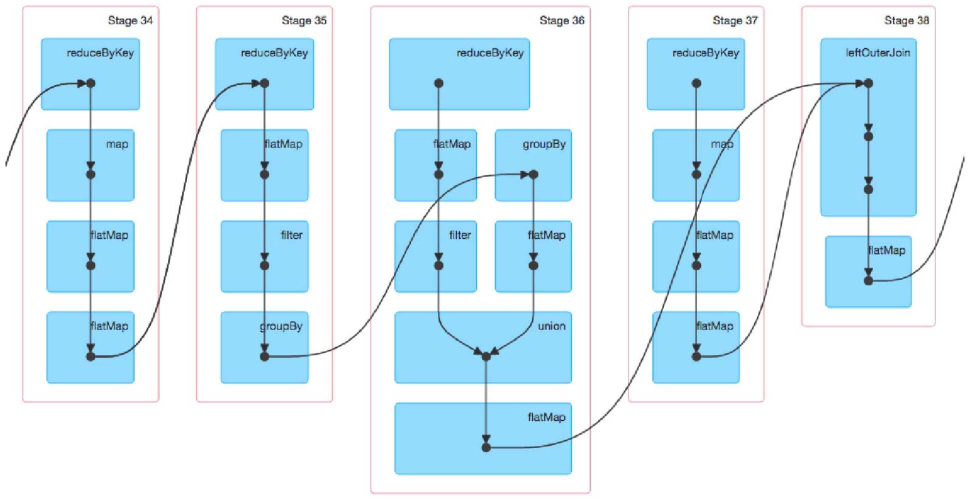

Fig. 6. Cover operation in Spark.

REFERENCES

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

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

[3] Cancer Genome Atlas Research Network. The Cancer Genome Atlas pan-cancer analysis project, *Nat Genet*., 45(10): 1113-1120, 2013.

[4] F. Albrecht, M. List, C. Bock, T. Lengauer. DeepBlue epigenomic data server: programmatic data retrieval and analysis of epigenome region sets, Nucleic acids research, Oxford Univ. Press, 44:W1, W581–W586, 2016.

[5] M. Bertoni, S. Ceri, A. Kaitoua, P. Pinoli Evaluating cloud frameworks on genomic applications. In *IEEE-Big Data Conference*.193-202, 2015.

[6] S. Ceri, A. Kaitoua, M. Masseroli, P. Pinoli, F. Venco. Big Data Management for Genomic Datasets *IEEE-Transactions on Computational Biology and Bioinformatics*, 13(2):233-247, 2016.

[7] A. Kaitoua, S. Ceri, M. Bertoni, P. Pinoli. Framework for Supporting Genomic Operations *IEEE-TC*, 2017, DOI 10.1109/TC.2016.2603980.

[8] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Paluzzi, H. Muller, S. Ceri. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*, 31(12):1881-1888, 2015., doi: 10.1093/bioinformatics/btv048.

[9] Anonymous paper, Accelerating Bioinformatics Research with New Software for Big Data to Knowledge (BD2K), Paradigm4, April 2015 (downloaded from: www.paradigm4.com, April 2017.)

[10] K. Shvachko, H. Kuang, S. Radia, R. Chansler. The Hadoop distributed file system. In *Proc. MSST*, 1-10, 2010.

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

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX*, 15-28, 2012.

[13] Apache Flink: http://flink.apache.org/, Online; accessed 10-April-2017

[14] Apache Pig: http://pig.apache.org/, Online; accessed 10-April-2017

[15] Apache Knox: https://knox.apache.org/, Online; accessed 10-April-2017

[16] Apache Hadoop 2: http://hadoop.apache.org/, Online; accessed 10-April-2017

[17] Apache Spark: http://spark.apache.org/, Online; accessed 10-April-2017

[18] FireCloud: https://software.broadinstitute.org/firecloud/ Online; accessed 10-April-2017

[19] Livy: http://livy.io/ Online; accessed 10-April-2017